Curve Fitting Toolbox™

User's Guide

MATLAB®

MathWorks®

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

Contents

Getting Started

1

Interactive Fitting

2

Programmatic Curve and Surface Fitting

3

Linear and Nonlinear Regression

4

Custom Linear and Nonlinear Regression

5

Interpolation and Smoothing

6

7

Spline Fitting

Advanced Spline Examples

11

Examples

12

13

Functions

A

Bibliography

Getting Started

Curve Fitting Toolbox Product Description

Fit curves and surfaces to data using regression, interpolation, and smoothing

Curve Fitting Toolbox provides an app and functions for fitting curves and surfaces to data. The toolbox lets you perform exploratory data analysis, preprocess and post-process data, compare candidate models, and remove outliers. You can conduct regression analysis using the library of linear and nonlinear models provided or specify your own custom equations. The library provides optimized solver parameters and starting conditions to improve the quality of your fits. The toolbox also supports nonparametric modeling techniques, such as splines, interpolation, and smoothing.

After creating a fit, you can apply a variety of post-processing methods for plotting, interpolation, and extrapolation; estimating confidence intervals; and calculating integrals and derivatives.

Key Features

- Curve Fitting app for curve and surface fitting
- Linear and nonlinear regression with custom equations
- Library of regression models with optimized starting points and solver parameters
- Interpolation methods, including B-splines, thin plate splines, and tensor-product splines
- Smoothing techniques, including smoothing splines, localized regression, Savitzky-Golay filters, and moving averages
- Preprocessing routines, including outlier removal and sectioning, scaling, and weighting data
- Post-processing routines, including interpolation, extrapolation, confidence intervals, integrals and derivatives

Curve Fitting Tools

Curve Fitting Toolbox software allows you to work in two different environments:

- An interactive environment, with the Curve Fitting app and the Spline Tool
- A programmatic environment that allows you to write object-oriented MATLAB® code using curve and surface fitting methods

To open the Curve Fitting app or Spline Tool, enter one of the following:

- `cftool`. See **Curve Fitting**.
- `splinetool`

To list the Curve Fitting Toolbox functions for use in MATLAB programming, type

```
help curvefit
```

The code for any function can be opened in the MATLAB Editor by typing

```
edit function_name
```

Brief, command line help for any function is available by typing

```
help function_name
```

Complete documentation for any function is available by typing

```
doc function_name
```

You can change the way any toolbox function works by copying and renaming its file, examining your copy in the editor, and then modifying it.

You can also extend the toolbox by adding your own files, or by using your code in combination with functions from other toolboxes, such as Statistics and Machine Learning Toolbox or Optimization Toolbox software.

Curve Fitting

Interactive Curve Fitting

To interactively fit a curve, follow the steps in this simple example:

1   Load some data at the MATLAB command line.

```
load hahn1
```

2   Open the Curve Fitting app. Enter:

```
cftool
```

3   In the Curve Fitting app, select **X Data** and **Y Data**.

Curve Fitting app creates a default interpolation fit to the data.

4   Choose a different model type using the fit category drop-down list, e.g., select **Polynomial**.

5   Try different fit options for your chosen model type.

6   Select **File > Generate Code**.

Curve Fitting app creates a file in the Editor containing MATLAB code to recreate all fits and plots in your interactive session.

For more information about fitting curves in the Curve Fitting app, see "Interactive Curve and Surface Fitting" on page 2-2.

For details and examples of specific model types and fit analysis, see the following sections:

1   "Linear and Nonlinear Regression"

2   "Interpolation"

3   "Smoothing"

4   "Fit Postprocessing"

Programmatic Curve Fitting

To programmatically fit a curve, follow the steps in this simple example:

1   Load some data.

```
load hahn1
```

Create a fit using the `fit` function, specifying the variables and a model type (in this case `rat23` is the model type).

```
f = fit( temp, thermex, 'rat23' )
```

Plot your fit and the data.

```
plot( f, temp, thermex )
f( 600 )
```

To learn what functions you can use to create and work with fits, see: "Curve and Surface Fitting" on page 3-2.

For details and examples of specific model types and fit analysis, see the following sections:

1   "Linear and Nonlinear Regression"
2   "Interpolation"
3   "Smoothing"
4   "Fit Postprocessing"

See Also
`fit`

Related Examples
•   "Interactive Curve and Surface Fitting" on page 2-2

Surface Fitting

Interactive Surface Fitting

To interactively fit a surface, follow the steps in this simple example:

1 Load some data at the MATLAB command line.

```
load franke
```

2 Open the Curve Fitting app. Enter:

```
cftool
```

3 In the Curve Fitting app, select **X Data**, **Y Data** and **Z Data**.

Curve Fitting app creates a default interpolation fit to the data.

4 Choose a different model type using the fit category drop-down list, e.g., select **Polynomial**.

5 Try different fit options for your chosen model type.

6 Select **File > Generate Code**.

Curve Fitting app creates a file in the Editor containing MATLAB code to recreate all fits and plots in your interactive session.

For more information about fitting surfaces in the Curve Fitting app, see "Interactive Curve and Surface Fitting" on page 2-2.

For details and examples of specific model types and fit analysis, see the following sections:

1 "Linear and Nonlinear Regression"

2 "Interpolation"

3 "Smoothing"

4 "Fit Postprocessing"

Programmatic Surface Fitting

To programmatically fit a surface, follow the steps in this simple example:

1 Load some data.

```
load franke
```

2 Create a fit using the `fit` function, specifying the variables and a model type (in this case `poly23` is the model type).

```
 f = fit( [x, y], z, 'poly23' )
```

3 Plot your fit and the data.

```
plot(f, [x,y], z)
```

To learn what functions you can use to create and work with fits, see: "Curve and Surface Fitting" on page 3-2.

For details and examples of specific model types and fit analysis, see the following sections:

1 "Linear and Nonlinear Regression"

2 "Interpolation"

3 "Smoothing"

4 "Fit Postprocessing"

See Also

`fit`

Related Examples

-

Spline Fitting

In this section...
"About Splines in Curve Fitting Toolbox" on page 1-8
"Interactive Spline Fitting" on page 1-8
"Programmatic Spline Fitting" on page 1-8

About Splines in Curve Fitting Toolbox

You can work with splines in Curve Fitting Toolbox in several ways.

Using the Curve Fitting app or the `fit` function you can:

- Fit cubic spline interpolants to curves or surfaces
- Fit smoothing splines and shape-preserving cubic spline interpolants to curves (but not surfaces)
- Fit thin-plate splines to surfaces (but not curves)

The toolbox also contains specific splines functions to allow greater control over what you can create. For example, you can use the `csapi` function for cubic spline interpolation. Why would you use `csapi` instead of the `fit` function `'cubicinterp'` option? You might require greater flexibility to work with splines for the following reasons:

- You want to combine the results with other splines, e.g., by addition.
- You want vector-valued splines. You can use `csapi` with scalars, vectors, matrices, and ND-arrays. The `fit` function only allows scalar-valued splines.
- You want other types of splines such as ppform, B-form, tensor-product, rational, and stform thin-plate splines.
- You want to create splines without data.
- You want to specify breaks, optimize knot placement, and use specialized functions for spline manipulation such as differentiation and integration.

If you require specialized spline functions, see the following sections for interactive and programmatic spline fitting.

Interactive Spline Fitting

You can access all spline functions from the `splinetool` GUI.

See "Introducing Spline Fitting" on page 8-2.

Programmatic Spline Fitting

To programmatically fit splines, see "Spline Construction" for descriptions of types of splines and numerous examples.

Interactive Fitting

Interactive Curve and Surface Fitting

In this section...
"Introducing the Curve Fitting App" on page 2-2
"Fit a Curve" on page 2-2
"Fit a Surface" on page 2-3
"Model Types for Curves and Surfaces" on page 2-4
"Selecting Data to Fit in Curve Fitting App" on page 2-5
"Save and Reload Sessions" on page 2-6

Introducing the Curve Fitting App

You can fit curves and surfaces to data and view plots with the Curve Fitting app.

- Create, plot, and compare multiple fits.
- Use linear or nonlinear regression, interpolation, smoothing, and custom equations.
- View goodness-of-fit statistics, display confidence intervals and residuals, remove outliers, and assess fits with validation data.
- Automatically generate code to fit and plot curves and surfaces, or export fits to the workspace for further analysis.

Fit a Curve

1   Load some example data at the MATLAB command line:

```
load census
```

2   Open the Curve Fitting app by entering:

```
cftool
```

Alternatively, click **Curve Fitting** on the **Apps** tab.

3   Select **X data** and **Y data**. For details, see "Selecting Data to Fit in Curve Fitting App" on page 2-5.

The Curve Fitting app creates a default polynomial fit to the data.

4   Try different fit options. For example, change the polynomial **Degree** to 3 to fit a cubic polynomial.
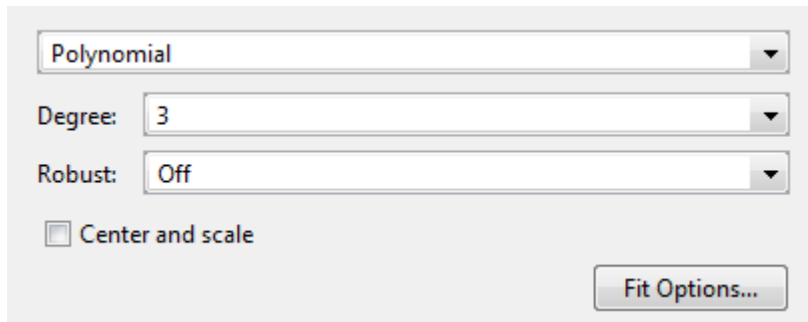
5 Select a different model type from the fit category list, e.g., **Smoothing Spline**. For information about models you can fit, see "Model Types for Curves and Surfaces" on page 2-4.
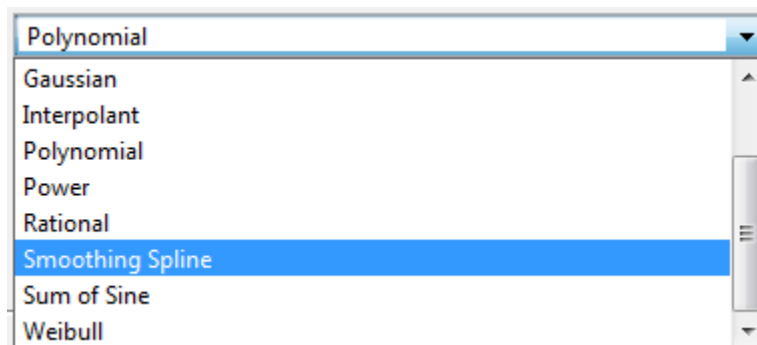
6 Select **File > Generate Code**.

The Curve Fitting app creates a file in the Editor containing MATLAB code to recreate all fits and plots in your interactive session.

Tip For a detailed workflow example, see "Compare Fits in Curve Fitting App" on page 2-16.

To create multiple fits and compare them, see "Create Multiple Fits in Curve Fitting App" on page 2-11.

Fit a Surface

1 Load some example data at the MATLAB command line:

```
load franke
```

2 Open the Curve Fitting app:

```
cftool
```

3 Select **X data**, **Y data**, and **Z data**. For more information, see "Selecting Data to Fit in Curve Fitting App" on page 2-5.
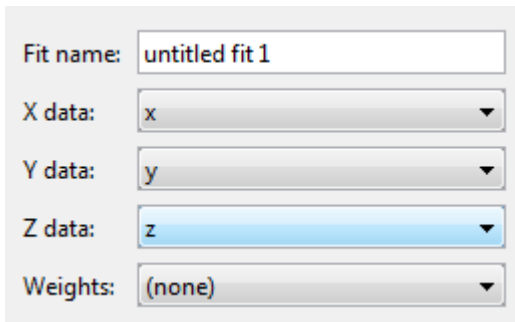
The Curve Fitting app creates a default interpolation fit to the data.

4 Select a different model type from the fit category list, e.g., **Polynomial**.

For information about models you can fit, see "Model Types for Curves and Surfaces" on page 2-4.
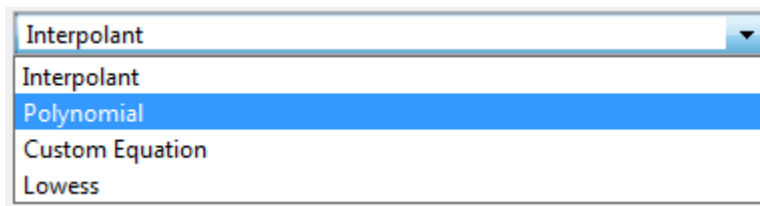
5 Try different fit options for your chosen model type.

6 Select **File > Generate Code**.

The Curve Fitting app creates a file in the Editor containing MATLAB code to recreate all fits and plots in your interactive session.

Tip For a detailed example, see "Surface Fitting to Franke Data" on page 2-26.

To create multiple fits and compare them, see "Create Multiple Fits in Curve Fitting App" on page 2-11.

Model Types for Curves and Surfaces

Based on your selected data, the fit category list shows either curve or surface fit categories. The following table describes the options for curves and surfaces.

Fit Category	Curves	Surfaces
Regression Models		
Polynomial on page 4-14	Yes (up to degree 9)	Yes (up to degree 5)
Exponential on page 4-28	Yes	
Fourier on page 4-34	Yes	
Gaussian on page 4-42	Yes	
Power on page 4-45	Yes	

Fit Category	Curves	Surfaces
Rational on page 4-48	Yes	
Sum of Sine on page 4-54	Yes	
Weibull on page 4-56	Yes	
Interpolation		
Interpolant on page 6-3	Yes Methods: Nearest neighbor Linear Cubic Shape-preserving (PCHIP)	Yes Methods: Nearest neighbor Linear Cubic Biharmonic (v4) Thin-plate spline
Smoothing		
Smoothing Spline on page 6-9	Yes	
Lowess on page 6-16		Yes
Custom		
Custom Equation on page 5-2	Yes	Yes
Linear Fitting on page 5-6	Yes	

For information about these fit types, see:

- "Linear and Nonlinear Regression"
- "Custom Models" on page 5-2
- "Interpolation"
- "Smoothing"

Selecting Data to Fit in Curve Fitting App

To select data to fit, use the drop-down lists in the Curve Fitting app to select variables in your MATLAB workspace.

- To fit curves:

  - Select **X data** and **Y data**.
  - Select only **Y data** to plot Y against index (`x=1:length( y )`).

- To fit surfaces, select **X data**, **Y data**, and **Z data**.

You can use the Curve Fitting app drop-down lists to select any numeric variables (with more than one element) in your MATLAB workspace.

Similarly, you can select any numeric data in your workspace to use as **Weights**.

For curves, X, Y, and Weights must be matrices with the same number of elements.

For surfaces, X, Y, and Z must be either:

- Matrices with the same number of elements
- Data in the form of a table

For surfaces, weights must have the same number of elements as Z.

For more information see "Selecting Compatible Size Surface Data" on page 2-9.

When you select variables, the Curve Fitting app immediately creates a curve or surface fit with the default settings. If you want to avoid time-consuming refitting for large data sets, you can turn off **Auto fit** by clearing the check box.

Note The Curve Fitting app uses a snapshot of the data you select. Subsequent workspace changes to the data have no effect on your fits. To update your fit data from the workspace, first change the variable selection, and then reselect the variable with the drop-down controls.

If there are problems with the data you select, you see messages in the **Results** pane. For example, the Curve Fitting app ignores `Infs`, `NaNs`, and imaginary components of complex numbers in the data, and you see messages in the **Results** pane in these cases.

If you see warnings about reshaping your data or incompatible sizes, read "Selecting Compatible Size Surface Data" on page 2-9 and "Troubleshooting Data Problems" on page 2-10 for information.

Save and Reload Sessions

- "Overview" on page 2-6
- "Saving Sessions" on page 2-7
- "Reloading Sessions" on page 2-7
- "Removing Sessions" on page 2-7

Overview

You can save and reload sessions for easy access to multiple fits. The session file contains all the fits and variables in your session and remembers your layout.

Saving Sessions

To save your session, first select **File > Save Session** to open your file browser. Next, select a name and location for your session file (with file extension `.sfit`).

After you save your session once, you can use **File > Save Session** to overwrite that session for subsequent saves.

To save the current session under a different name, select **File > Save Session As**.

Reloading Sessions

Use **File > Load Session** to open a file browser where you can select a saved curve fitting session file to load.

Removing Sessions

Use **File > Clear Session** to remove all fits from the current Curve Fitting app session.

See Also

Related Examples
- "Compare Fits in Curve Fitting App" on page 2-16
- "Create Multiple Fits in Curve Fitting App" on page 2-11

Data Selection

In this section...
"Selecting Data to Fit in Curve Fitting App" on page 2-8
"Selecting Compatible Size Surface Data" on page 2-9
"Troubleshooting Data Problems" on page 2-10

Selecting Data to Fit in Curve Fitting App

To select data to fit, use the drop-down lists in the Curve Fitting app to select variables in your MATLAB workspace.

- To fit curves:

    - Select **X data** and **Y data**.
    - Select only **Y data** to plot Y against index (x=1:length(y)).

- To fit surfaces, select **X data**, **Y data** and **Z data**.

You can use the Curve Fitting app drop-down lists to select any numeric variables (with more than one element) in your MATLAB workspace.

Similarly, you can select any numeric data in your workspace to use as **Weights**.

For curves, X, Y, and Weights must be matrices with the same number of elements.

For surfaces, X, Y, and Z must be either:

- Matrices with the same number of elements
- Data in the form of a table

For surfaces, weights must have the same number of elements as Z.

For more information see "Selecting Compatible Size Surface Data" on page 2-9.

When you select variables, the Curve Fitting app immediately creates a curve or surface fit with the default settings. If you want to avoid time-consuming refitting for large data sets, you can turn off **Auto fit** by clearing the check box.

Note The Curve Fitting app uses a snapshot of the data you select. Subsequent workspace changes to the data have no effect on your fits. To update your fit data from the workspace, first change the variable selection, and then reselect the variable with the drop-down controls.

Selecting Compatible Size Surface Data

For surface data, in Curve Fitting app you can select either "Matrices of the Same Size" on page 2-9 or "Table Data" on page 2-9.

Matrices of the Same Size

Curve Fitting app expects inputs to be the same size. If the sizes are different but the number of elements are the same, then the tool reshapes the inputs to create a fit and displays a warning in the **Results** pane. The warning indicates a possible problem with your selected data.

Table Data

Table data means that X and Y represent the row and column headers of a table (sometimes called breakpoints) and the values in the table are the values of the Z output.

Sizes are compatible if:

*   **X** is a vector of length `n`.
*   **Y** is a vector of length `m`.
*   **Z** is a 2D matrix of size `[m,n]`.

The following table shows an example of data in the form of a table with `n = 4` and `m = 3`.

	x(1)	x(2)	x(3)	x(4)
y(1)	z(1,1)	z(1,2)	z(1,3)	z(1,4)
y(2)	z(2,1)	z(2,2)	z(2,3)	z(2,4)
y(3)	z(3,1)	z(3,2)	z(3,3)	z(3,4)

Like the `surf` function, the Curve Fitting app expects inputs where `length(X) = n`, `length(Y) = m` and `size(Z) = [m,n]`. If the size of Z is `[n,m]`, the tool creates a fit but first transposes Z and warns about transforming your data. You see a warning in the **Results** pane like the following example:

```
Using X Input for rows and Y Input for columns
to match Z Output matrix.
```

For suitable example table data, run the following code:

```
x = linspace( 0, 1, 7 );
y = linspace( 0, 1, 9 ).';
z = bsxfun( @franke, x, y );
```

For surface fitting at the command line with the `fit` function, use the `prepareSurfaceData` function if your data is in table form.

Weights

If you specify surface Weights, assign an input the same size as Z. If the sizes are different but the number of elements is the same, Curve Fitting app reshapes the weights and displays a warning.

Troubleshooting Data Problems

If there are problems with the data you select, you see messages in the **Results** pane. For example, the Curve Fitting app ignores `Infs`, `NaNs`, and imaginary components of complex numbers in the data, and you see messages in the **Results** pane in these cases.

If you see warnings about reshaping your data or incompatible sizes, read "Selecting Compatible Size Surface Data" on page 2-9 for information.

If you see the following warning: `Duplicate x-y data points detected: using average of the z values`, this means that there are two or more data points where the input values (x, y) are the same or very close together. The default interpolant fit type needs to calculate a unique value at that point. You do not need do anything to fix the problem, this warning is just for your information. The Curve Fitting app automatically takes the average z value of any group of points with the same x-y values.

Other problems with your selected data can produce the following error:

```
Error computing Delaunay triangulation. Please try again with
different data.
```

Some arrangements of data make it impossible for Curve Fitting app to compute a Delaunay triangulation. Three out of the four surface interpolation methods (linear, cubic, and nearest) require a Delaunay triangulation of the data. An example of data that can cause this error is a case where all the data lies on a straight line in x-y. In this case, Curve Fitting app cannot fit a surface to the data. You need to provide more data in order to fit a surface.

Note Data selection is disabled if you are in debug mode. Exit debug mode to change data selections.

Create Multiple Fits in Curve Fitting App

In this section...
"Refining Your Fit" on page 2-11
"Creating Multiple Fits" on page 2-11
"Duplicating a Fit" on page 2-11
"Deleting a Fit" on page 2-12
"Displaying Multiple Fits Simultaneously" on page 2-12
"Using the Statistics in the Table of Fits" on page 2-13

Refining Your Fit

After you create a single fit, you can refine your fit, using any of the following optional steps:

- Change fit type and settings. Select GUI settings to use the Curve Fitting app built-in fit types or create custom equations. For fit settings for each model type, see "Linear and Nonlinear Regression", "Interpolation", and "Smoothing".
- Exclude data by removing outliers in the Curve Fitting app. See "Remove Outliers" on page 7-8.
- Select weights. See "Data Selection" on page 2-8.
- Select validation data. See "Select Validation Data" on page 7-12
- Create multiple fits and you can compare different fit types and settings side by side in the Curve Fitting app. See "Creating Multiple Fits" on page 2-11.

Creating Multiple Fits

After you create a single fit, it can be useful to create multiple fits to compare. When you create multiple fits you can compare different fit types and settings side-by-side in the Curve Fitting app.

After creating a fit, you can add an additional fit using any of these methods:

- Click the New Fit button next to your fit figure tabs in the Document Bar.
- Right-click the Document Bar and select **New Fit**.
- Select **Fit > New Fit**.

Each additional fit appears as a new tab in the Curve Fitting app and a new row in the **Table of Fits**. See "Create Multiple Fits in Curve Fitting App" on page 2-11 for information about displaying and analyzing multiple fits.

Optionally, after you create an additional fit, you can copy your data selections from a previous fit by selecting **Fit > Use Data From > *Other Fit Name***. This copies your selections for x, y, and z from the previous fit, and any selected validation data. No fit options are changed.

Use sessions to save and reload your fits. See "Save and Reload Sessions" on page 2-6.

Duplicating a Fit

To create a copy of the current fit tab, select **Fit > Duplicate "*Current Fit Name*"**. You also can right-click a fit in the **Table of Fits** and select **Duplicate**

Each additional fit appears as a new tab in the Curve Fitting app.

Deleting a Fit

Delete a fit from your session using one of these methods:

- Select the fit tab display and select **Fit > Delete** *Current Fit Name*.
- Select the fit in the **Table of Fits** and press **Delete**.
- Right-click the fit in the table and select **Delete** *Current Fit Name*.

Displaying Multiple Fits Simultaneously

When you have created multiple fits you can compare different fit types and settings side by side in the Curve Fitting app. You can view plots simultaneously and you can examine the goodness-of-fit statistics to compare your fits. This section describes how to compare multiple fits.

To compare plots and see multiple fits simultaneously, use the layout controls at the top right of the Curve Fitting app. Alternatively, you can click **Window** on the menu bar to select the number and position of tiles you want to display. A fit figure displays the fit settings, results pane and plots for a single fit. The following example shows two fit figures displayed side by side. You can see multiple fits in the session listed in the **Table of Fits**.
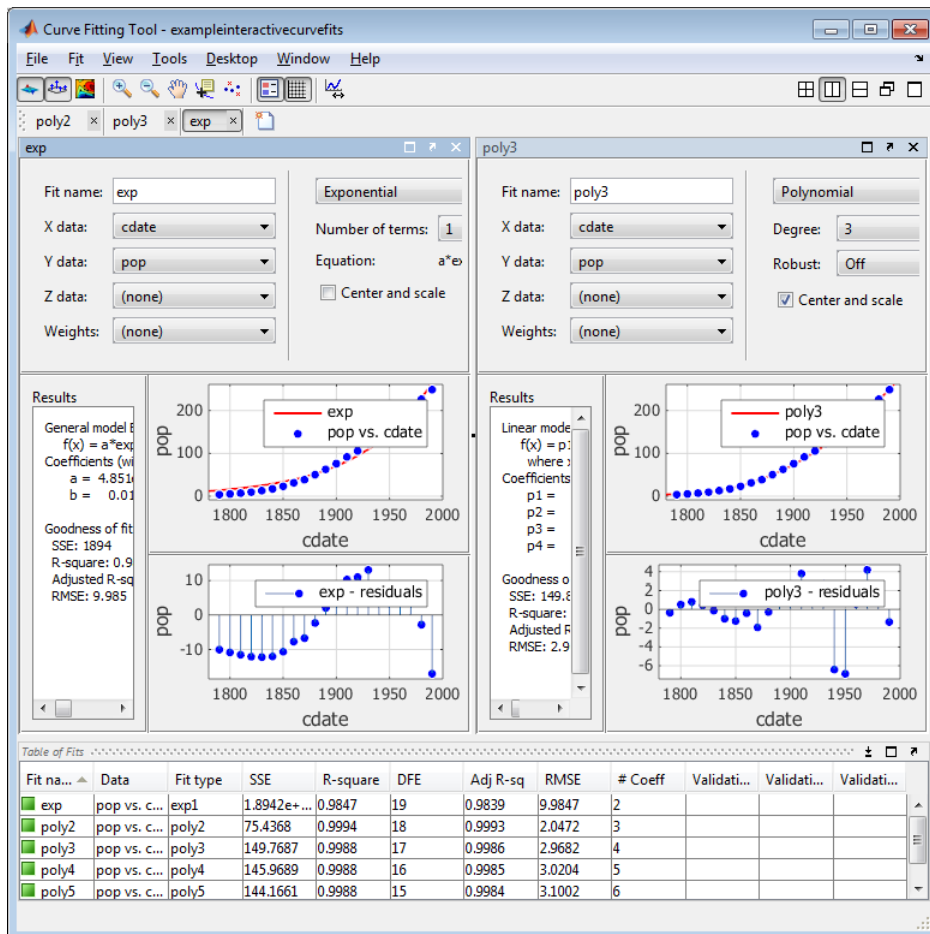
You can close fit figures displays (with the Close button, **Fit** menu, or context menu), but they remain in your session. The **Table of Fits** displays all your fits (open and closed). Double-click a fit in the **Table of Fits** to open (or focus if already open) the fit figure. To remove a fit, see "Deleting a Fit" on page 2-12.

Tip If you want more space to view and compare plots, as shown next, use the **View** menu to hide or show the **Fit Settings**, **Fit Results**, or **Table of Fits** panes.
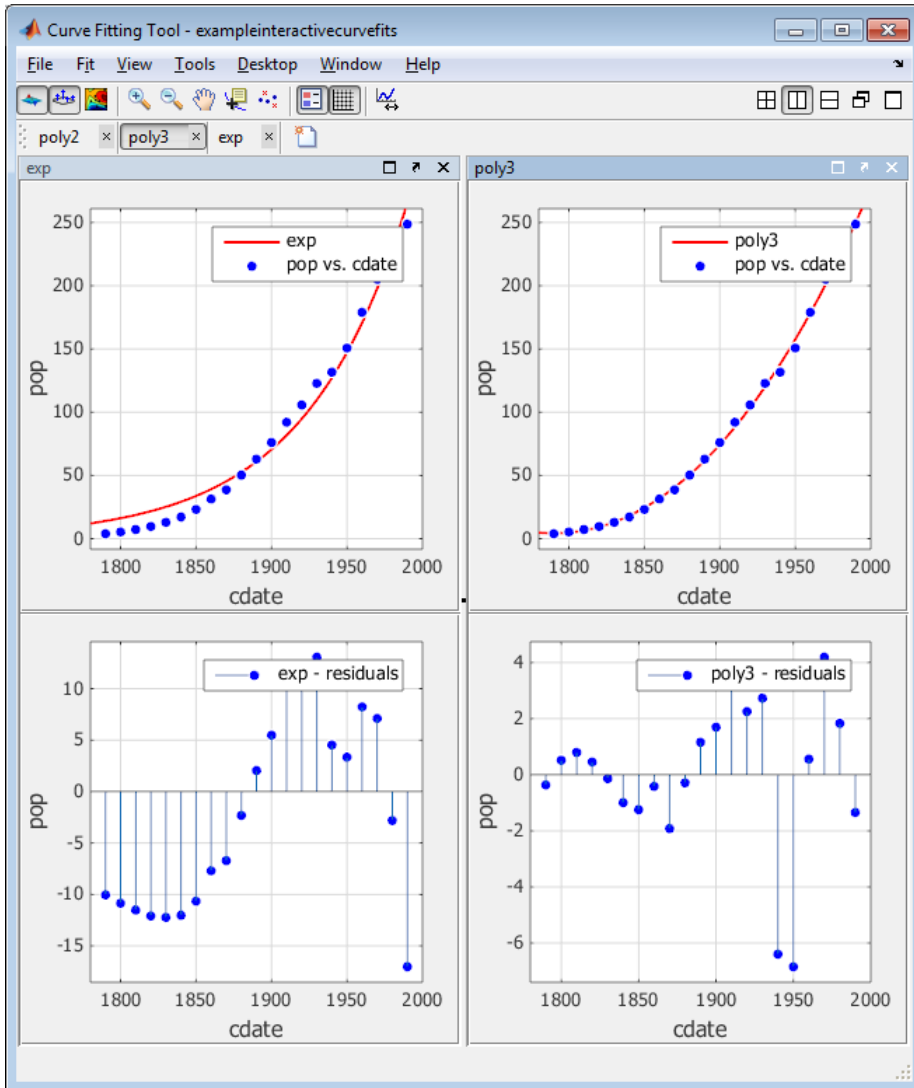
You can dock and undock individual fits and navigate between them using the standard MATLAB **Desktop** and **Window** menus in the Curve Fitting app. For more information, see "Optimize Desktop Layout".

Using the Statistics in the Table of Fits

The **Table of Fits** list pane shows all fits in the current session.

Fit name	Data	Fit type	SSE ▲	R-square	DFE	Adj R-sq	RMSE	# Coeff
■ poly6	pop vs. cd...	poly6	106.9276	0.9991	14	0.9988	2.7636	7
■ poly5	pop vs. cd...	poly5	144.1661	0.9988	15	0.9984	3.1002	6
■ poly4	pop vs. cd...	poly4	145.9689	0.9988	16	0.9985	3.0204	5
■ poly3	pop vs. cd...	poly3	149.7687	0.9988	17	0.9986	2.9682	4
■ poly2	pop vs. cd...	poly2	159.0293	0.9987	18	0.9986	2.9724	3
■ exp	pop vs. cd...	exp1	1.8942e+03	0.9847	19	0.9839	9.9847	2

After using graphical methods to evaluate the goodness of fit, you can examine the goodness-of-fit statistics shown in the table to compare your fits. The goodness-of-fit statistics help you determine how well the model fits the data. Click the table column headers to sort by statistics, name, fit type, and so on.

The following guidelines help you use the statistics to determine the best fit:

- **SSE** is the sum of squares due to error of the fit. A value closer to zero indicates a fit that is more useful for prediction.
- **R-square** is the square of the correlation between the response values and the predicted response values. A value closer to 1 indicates that a greater proportion of variance is accounted for by the model.
- **DFE** is the degree of freedom in the error.
- **Adj R-sq** is the degrees of freedom adjusted R-square. A value closer to 1 indicates a better fit.
- **RMSE** is the root mean squared error or standard error. A value closer to 0 indicates a fit that is more useful for prediction.
- **# Coeff** is the number of coefficients in the model. When you have several fits with similar goodness-of-fit statistics, look for the smallest number of coefficients to help decide which fit is best. You must trade off the number of coefficients against the goodness of fit indicated by the statistics to avoid overfitting.

For a more detailed explanation of the Curve Fitting Toolbox statistics, see "Goodness-of-Fit Statistics" on page 7-45.

To compare the statistics for different fits and decide which fit is the best tradeoff between over- and under-fitting, use a similar process to that described in "Compare Fits in Curve Fitting App" on page 2-16.

See Also

Related Examples

- "Compare Fits in Curve Fitting App" on page 2-16
- "Compare Fits Programmatically" on page 7-32

Generating MATLAB Code and Exporting Fits

Interactive Code Generation and Programmatic Fitting

Curve Fitting app makes it easy to plot and analyze fits at the command line. You can export individual fits to the workspace for further analysis, or you can generate MATLAB code to recreate all fits and plots in your session. By generating code you can use your interactive curve fitting session to quickly assemble code for curve and surface fits and plots into useful programs.

1 Select **File > Generate Code**.

The Curve Fitting app generates code from your session and displays the file in the MATLAB Editor. The file includes all fits and plots in your current session. The file captures the following information:

- Names of fits and their variables
- Fit settings and options
- Plots
- Curve or surface fitting objects and methods used to create the fits:

  - A cell-array of `cfit` or `sfit` objects representing the fits
  - A structure array with goodness-of fit information.

2 Save the file.

For more information on working with your generated code, exporting fits to the workspace, and recreating your fits and plots at the command line, see:

- "Generating Code from the Curve Fitting App" on page 7-13
- "Exporting a Fit to the Workspace" on page 7-14

Compare Fits in Curve Fitting App

In this section...

Interactive Curve Fitting Workflow

The next topics fit some census data using polynomial equations up to the sixth degree, and a single-term exponential equation. The steps demonstrate how to:

- Load data and explore various fits using different library models.
- Search for the best fit by:

  - Comparing graphical fit results
  - Comparing numerical fit results including the fitted coefficients and goodness-of-fit statistics

- Export your best fit results to the MATLAB workspace to analyze the model at the command line.
- Save the session and generate MATLAB code for all fits and plots.

Loading Data and Creating Fits

You must load the data variables into the MATLAB workspace before you can fit data using the Curve Fitting app. For this example, the data is stored in the MATLAB file `census.mat`.

1   Load the data:

    ```
    load census
    ```

    The workspace contains two new variables:

    - `cdate` is a column vector containing the years 1790 to 1990 in 10-year increments.
    - `pop` is a column vector with the U.S. population figures that correspond to the years in `cdate`.

2   Open the Curve Fitting app:

    ```
    cftool
    ```

3   Select the variable names `cdate` and `pop` from the **X data** and **Y data** lists.

    The Curve Fitting app creates and plots a default fit to X input (or predictor data) and Y output (or response data). The default fit is a linear polynomial fit type. Observe the fit settings display `Polynomial`, of **Degree** 1.

4   Change the fit to a second degree polynomial by selecting 2 from the **Degree** list.

    The Curve Fitting app plots the new fit. The Curve Fitting app calculates a new fit when you change fit settings because **Auto fit** is selected by default. If refitting is time consuming, e.g., for large data sets, you can turn off **Auto fit** by clearing the check box.

The Curve Fitting app displays results of fitting the census data with a quadratic polynomial in the **Results** pane, where you can view the library model, fitted coefficients, and goodness-of-fit statistics.

5 Change the **Fit name** to `poly2`.

6 Display the residuals by selecting **View > Residuals Plot**.

The residuals indicate that a better fit might be possible. Therefore, continue exploring various fits to the census data set.
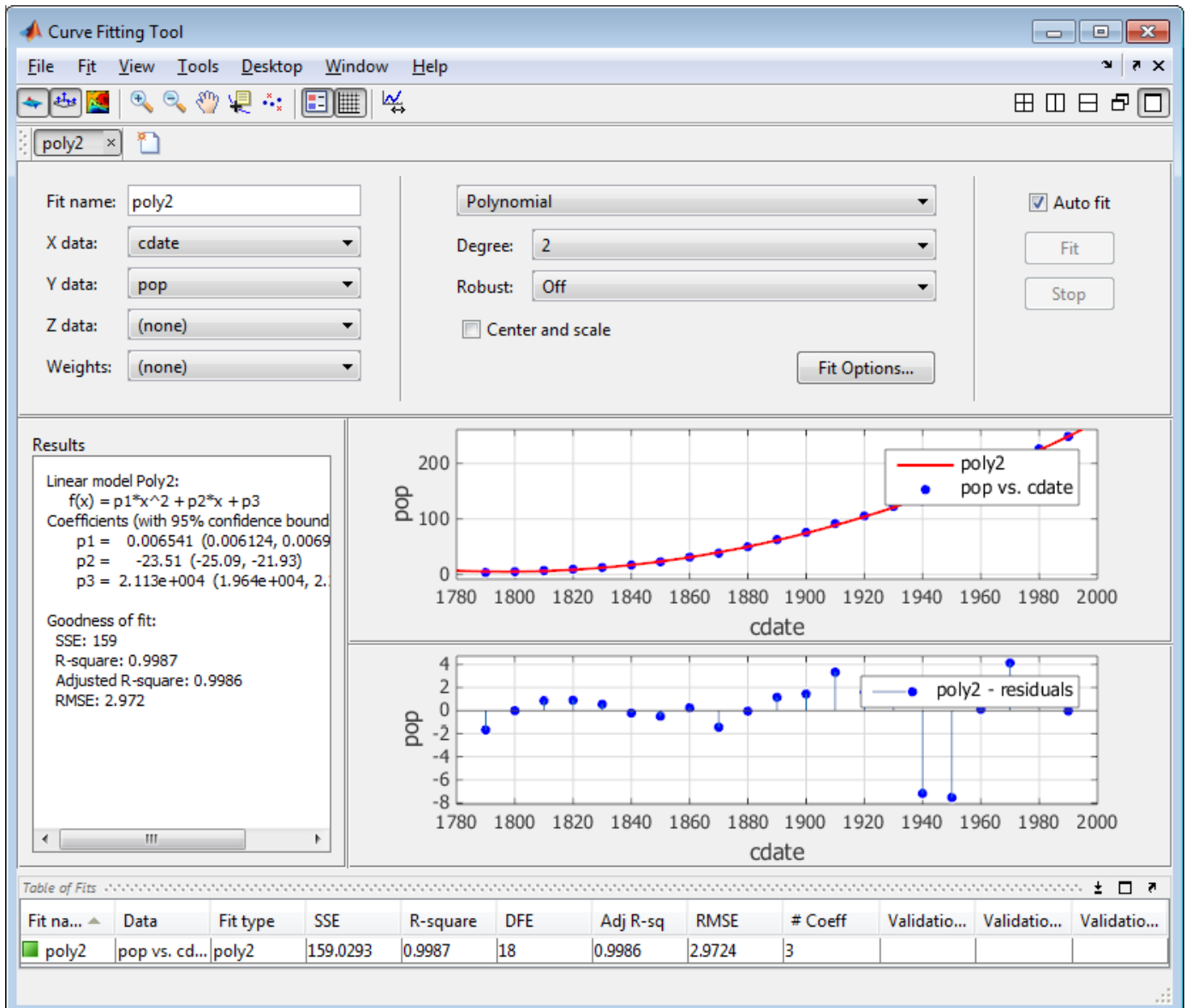
7 Add new fits to try the other library equations.

    a Right-click the fit in the **Table of Fits** and select **Duplicate "poly2"** (or use the **Fit** menu).

Tip For fits of a given type (for example, polynomials), use **Duplicate "*fitname*"** instead of a new fit because copying a fit requires fewer steps. The duplicated fit contains the same data selections and fit settings.

---

b Change the polynomial **Degree** to 3 and rename the fit `poly3`.

c When you fit higher degree polynomials, the **Results** pane displays this warning:

```
Equation is badly conditioned. Remove repeated data points
or try centering and scaling.
```

Normalize the data by selecting the **Center and scale** check box.

d Repeat steps a and b to add polynomial fits up to the sixth degree, and then add an exponential fit.

e For each new fit, look at the **Results** pane information, and the residuals plot in the Curve Fitting app.

The residuals from a good fit should look random with no apparent pattern. A pattern, such as a tendency for consecutive residuals to have the same sign, can be an indication that a better model exists.

About Scaling

The warning about scaling arises because the fitting procedure uses the `cdate` values as the basis for a matrix with very large values. The spread of the `cdate` values results in a scaling problem. To address this problem, you can normalize the `cdate` data. Normalization scales the predictor data to improve the accuracy of the subsequent numeric computations. A way to normalize `cdate` is to center it at zero mean and scale it to unit standard deviation. The equivalent code is:

```
(cdate - mean(cdate))./std(cdate)
```

Note Because the predictor data changes after normalizing, the values of the fitted coefficients also change when compared to the original data. However, the functional form of the data and the resulting goodness-of-fit statistics do not change. Additionally, the data is displayed in the Curve Fitting app plots using the original scale.

Determining the Best Fit

To determine the best fit, you should examine both the graphical and numerical fit results.

Examine the Graphical Fit Results

1 Determine the best fit by examining the graphs of the fits and residuals. To view plots for each fit in turn, double-click the fit in the Table of Fits. The graphical fit results indicate that:

- The fits and residuals for the polynomial equations are all similar, making it difficult to choose the best one.
- The fit and residuals for the single-term exponential equation indicate it is a poor fit overall. Therefore, it is a poor choice and you can remove the exponential fit from the candidates for best fit.

2 Examine the behavior of the fits up to the year 2050. The goal of fitting the census data is to extrapolate the best fit to predict future population values.

   a   Double-click the sixth-degree polynomial fit in the Table of Fits to view the plots for this fit.

   b   Change the axes limits of the plots by selecting **Tools > Axes Limits**.

   c   Alter the **X (cdate) Maximum** to 2050, and increase the **Main Y (pop) Maximum** to 400, and press **Enter**.

   d   Examine the fit plot. The behavior of the sixth-degree polynomial fit beyond the data range makes it a poor choice for extrapolation and you can reject this fit.
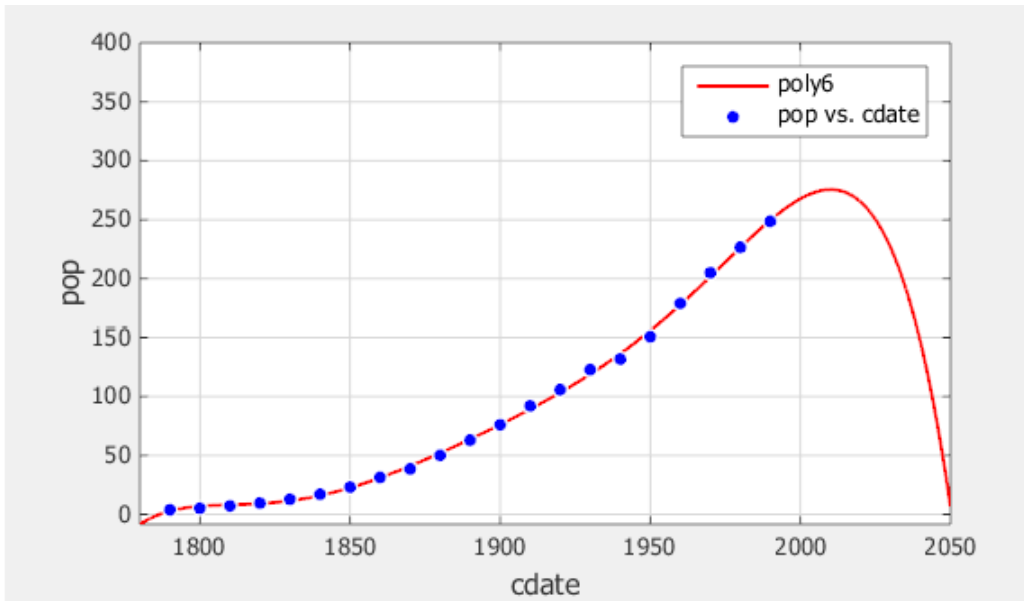
Evaluate the Numerical Fit Results

When you can no longer eliminate fits by examining them graphically, you should examine the numerical fit results. The Curve Fitting app displays two types of numerical fit results:

- Goodness-of-fit statistics
- Confidence bounds on the fitted coefficients

The goodness-of-fit statistics help you determine how well the curve fits the data. The confidence bounds on the coefficients determine their accuracy.

Examine the numerical fit results:

1   For each fit, view the goodness-of-fit statistics in the **Results** pane.

Results

Linear model Poly3:
$$f(x) = p1*x^\wedge 3 + p2*x^\wedge 2 + p3*x + p4$$
where x is normalized by mean 1890 and std 62.05
Coefficients (with 95% confidence bounds):
p1 =   0.921 (-0.9743, 2.816)
p2 =   25.18 (23.57, 26.79)
p3 =   73.86 (70.33, 77.39)
p4 =   61.74 (59.69, 63.8)

Goodness of fit:
SSE: 149.8
R-square: 0.9988
Adjusted R-square: 0.9986
RMSE: 2.968
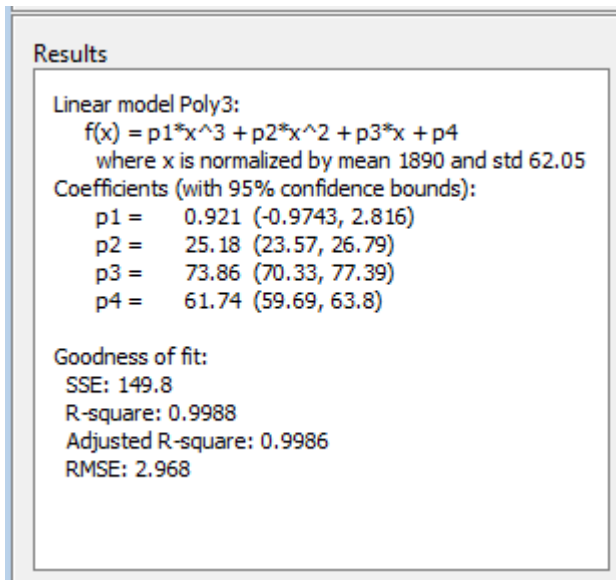
2   Compare all fits simultaneously in the **Table of Fits**. Click the column headings to sort by statistics results.

Table of Fits

Fit name	Data	Fit type	SSE ▲	R-square	DFE	Adj R-sq	RMSE	# Coeff
poly6	pop vs. cd...	poly6	106.9276	0.9991	14	0.9988	2.7636	7
poly5	pop vs. cd...	poly5	144.1661	0.9988	15	0.9984	3.1002	6
poly4	pop vs. cd...	poly4	145.9689	0.9988	16	0.9985	3.0204	5
poly3	pop vs. cd...	poly3	149.7687	0.9988	17	0.9986	2.9682	4
poly2	pop vs. cd...	poly2	159.0293	0.9987	18	0.9986	2.9724	3
exp	pop vs. cd...	exp1	1.8942e+03	0.9847	19	0.9839	9.9847	2

3   Examine the sum of squares due to error (SSE) and the adjusted *R*-square statistics to help determine the best fit. The SSE statistic is the least-squares error of the fit, with a value closer to zero indicating a better fit. The adjusted *R*-square statistic is generally the best indicator of the fit quality when you add additional coefficients to your model.

The largest SSE for `exp1` indicates it is a poor fit, which you already determined by examining the fit and residuals. The lowest SSE value is associated with `poly6`. However, the behavior of this fit beyond the data range makes it a poor choice for extrapolation, so you already rejected this fit by examining the plots with new axis limits.

The next best SSE value is associated with the fifth-degree polynomial fit, `poly5`, suggesting it might be the best fit. However, the SSE and adjusted *R*-square values for the remaining polynomial fits are all very close to each other. Which one should you choose?

4   Resolve the best fit issue by examining the confidence bounds for the remaining fits in the Results pane. Double-click a fit in the **Table of Fits** to open (or focus if already open) the fit figure and view the Results pane. A fit figure displays the fit settings, results pane and plots for a single fit.

Display the fifth-degree polynomial and the `poly2` fit figures side by side. Examining results side by side can help you assess fits.

a   To show two fit figures simultaneously, use the layout controls at the top right of the Curve Fitting app or select **Window > Left/Right Tile** or **Top/Bottom Tile**.

b To change the displayed fits, click to select a fit figure and then double-click the fit to display in the **Table of Fits**.

c Compare the coefficients and bounds (p1, p2, and so on) in the Results pane for both fits, `poly5` and `poly2`. The toolbox calculates 95% confidence bounds on coefficients. The confidence bounds on the coefficients determine their accuracy. Check the equations in the Results pane (f(x)=p1*x+p2*x...) to see the model terms for each coefficient. Note that p2 refers to the p2*x term in `Poly2` and the p2*x^4 term in `Poly5`. Do not compare normalized coefficients directly with non-normalized coefficients.

Tip Use the **View** menu to hide the **Fit Settings** or **Table of Fits** if you want more space to view and compare plots and results, as shown next. You can also hide the **Results** pane to show only plots.
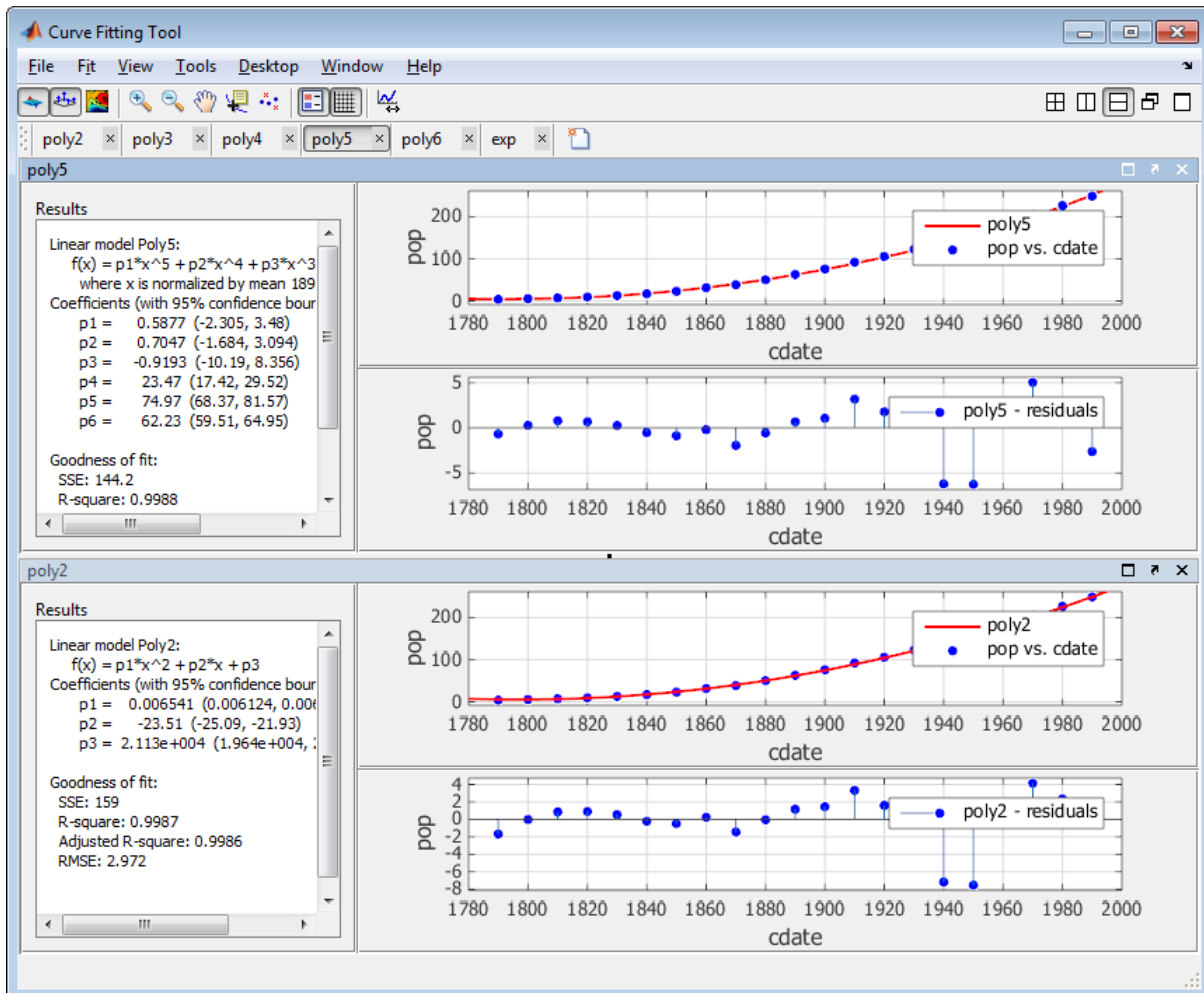
The bounds cross zero on the p1, p2, and p3 coefficients for the fifth-degree polynomial. This means you cannot be sure that these coefficients differ from zero. If the higher order model terms may have coefficients of zero, they are not helping with the fit, which suggests that this model overfits the census data.

```
Results

Linear model Poly5:
    f(x) = p1*x^5 + p2*x^4 + p3*x^3 + p4*x^2 + p5*x + p6
    where x is normalized by mean 1890 and std 62.05
Coefficients (with 95% confidence bounds):
    p1 =    0.5877 (-2.305, 3.48)
    p2 =    0.7047 (-1.684, 3.094)
    p3 =   -0.9193 (-10.19, 8.356)
    p4 =    23.47 (17.42, 29.52)
    p5 =    74.97 (68.37, 81.57)
    p6 =    62.23 (59.51, 64.95)

Goodness of fit:
  SSE: 144.2
  R-square: 0.9988
  Adjusted R-square: 0.9984
  RMSE: 3.1
```

However, the small confidence bounds do not cross zero on p1, p2, and p3 for the quadratic fit, `poly2` indicate that the fitted coefficients are known fairly accurately.

```
Results

Linear model Poly2:
    f(x) = p1*x^2 + p2*x + p3
Coefficients (with 95% confidence bounds):
    p1 =   0.006541 (0.006124, 0.006958)
    p2 =    -23.51 (-25.09, -21.93)
    p3 = 2.113e+004 (1.964e+004, 2.262e+004)

Goodness of fit:
  SSE: 159
  R-square: 0.9987
  Adjusted R-square: 0.9986
  RMSE: 2.972
```

Therefore, after examining both the graphical and numerical fit results, you should select `poly2` as the best fit to extrapolate the census data.

Note The fitted coefficients associated with the constant, linear, and quadratic terms are nearly identical for each normalized polynomial equation. However, as the polynomial degree increases, the coefficient bounds associated with the higher degree terms cross zero, which suggests overfitting.

Analyzing Your Best Fit in the Workspace

You can use **Save to Workspace** to export the selected fit and the associated fit results to the MATLAB workspace. The fit is saved as a MATLAB object and the associated fit results are saved as structures.

1   Right-click the `poly2` fit in the **Table of Fits** and select **Save "poly2" to Workspace** (or use the **Fit** menu).
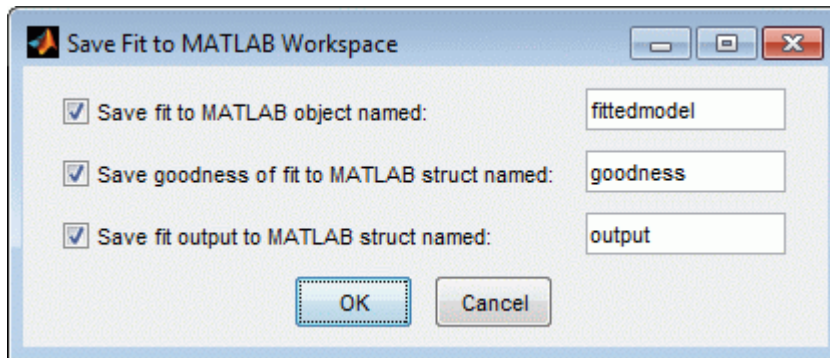
2 Click **OK** to save with the default names.

The `fittedmodel` is saved as a Curve Fitting Toolbox `cfit` object.

```
>> whos fittedmodel
  Name                Size              Bytes  Class

   fittedmodel        1x1                 822  cfit
```

Examine the `fittedmodel cfit` object to display the model, the fitted coefficients, and the confidence bounds for the fitted coefficients:

`fittedmodel`

```
fittedmodel =
     Linear model Poly2:
       fittedmodel(x) = p1*x^2 + p2*x + p3
     Coefficients (with 95% confidence bounds):
       p1 =    0.006541  (0.006124, 0.006958)
       p2 =       -23.51  (-25.09, -21.93)
       p3 =   2.113e+004  (1.964e+004, 2.262e+004)
```

Examine the `goodness` structure to display goodness-of-fit results:

`goodness`

```
goodness =
           sse: 159.0293
       rsquare: 0.9987
           dfe: 18
     adjrsquare: 0.9986
          rmse: 2.9724
```

Examine the `output` structure to display additional information associated with the fit, such as the residuals:

`output`

```
output =
        numobs: 21
      numparam: 3
     residuals: [21x1 double]
      Jacobian: [21x3 double]
      exitflag: 1
     algorithm: 'QR factorization and solve'
    iterations: 1
```

You can evaluate (interpolate or extrapolate), differentiate, or integrate a fit over a specified data range with various postprocessing functions.

For example, to evaluate the `fittedmodel` at a vector of values to extrapolate to the year 2050, enter:

```
y = fittedmodel(2000:10:2050)
y =

  274.6221
  301.8240
  330.3341
  360.1524
  391.2790
  423.7137
```

Plot the fit to the census data and the extrapolated fit values:

```
plot(fittedmodel, cdate, pop)
hold on
plot(fittedmodel, 2000:10:2050, y)
hold off
```

For more examples and instructions for interactive and command-line fit analysis, and a list of all postprocessing functions, see "Fit Postprocessing".

For an example reproducing this interactive census data analysis using the command line, see "Polynomial Curve Fitting" on page 12-2.

Saving Your Work

The toolbox provides several options for saving your work. You can save one or more fits and the associated fit results as variables to the MATLAB workspace. You can then use this saved information for documentation purposes, or to extend your data exploration and analysis. In addition to saving your work to MATLAB workspace variables, you can:

- Save the current curve fitting session by selecting **File > Save Session**. The session file contains all the fits and variables in your session and remembers your layout. See "Save and Reload Sessions" on page 2-6.

- Generate MATLAB code to recreate all fits and plots in your session by selecting **File > Generate Code**. The Curve Fitting app generates code from your session and displays the file in the MATLAB Editor.

  You can recreate your fits and plots by calling the file at the command line with your original data as input arguments. You can also call the file with new data, and automate the process of fitting multiple data sets. For more information, see "Generating Code from the Curve Fitting App" on page 7-13.

See Also

Related Examples
- "Create Multiple Fits in Curve Fitting App" on page 2-11

- "Evaluate a Curve Fit" on page 7-16

Surface Fitting to Franke Data

The Curve Fitting app provides some example data generated from Franke's bivariate test function. This data is suitable for trying various fit settings in Curve Fitting app.

To load the example data and create, compare, and export surface fits, follow these steps:

1   To load example data to use in the Curve Fitting app, enter `load franke` at the MATLAB command line. The variables x, y, and z appear in your workspace.

The example data is generated from Franke's bivariate test function, with added noise and scaling, to create suitable data for trying various fit settings in Curve Fitting app. For details on the Franke function, see the following paper:

Franke, R., Scattered Data Interpolation: Tests of Some Methods, Mathematics of Computation 38 (1982), pp. 181–200.

2   To divide the data into fitting and validation data, enter the following syntax:

```
xv = x(200:293);
yv = y(200:293);
zv = z(200:293);
x = x(1:199);
y = y(1:199);
z = z(1:199);
```

3   To fit a surface using this example data:

    a   Open Curve Fitting app. Enter `cftool`, or select **Curve Fitting** on the **Apps** tab.

    b   Select the variables x, y, and z interactively in the Curve Fitting app.
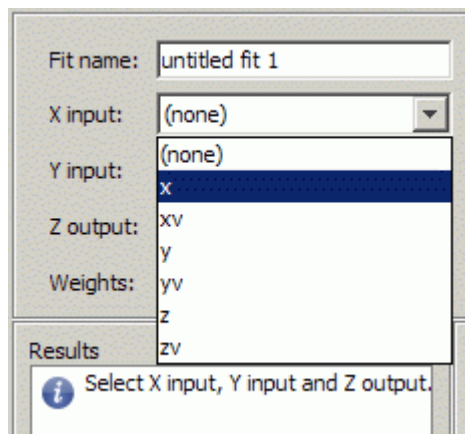
Alternatively, you can specify the variables when you enter `cftool(x,y,z)` to open Curve Fitting app (if necessary) and create a default fit.

The Curve Fitting app plots the data points as you select variables. When you select x, y, and z, the tool automatically creates a default surface fit. The default fit is an interpolating surface that passes through the data points.
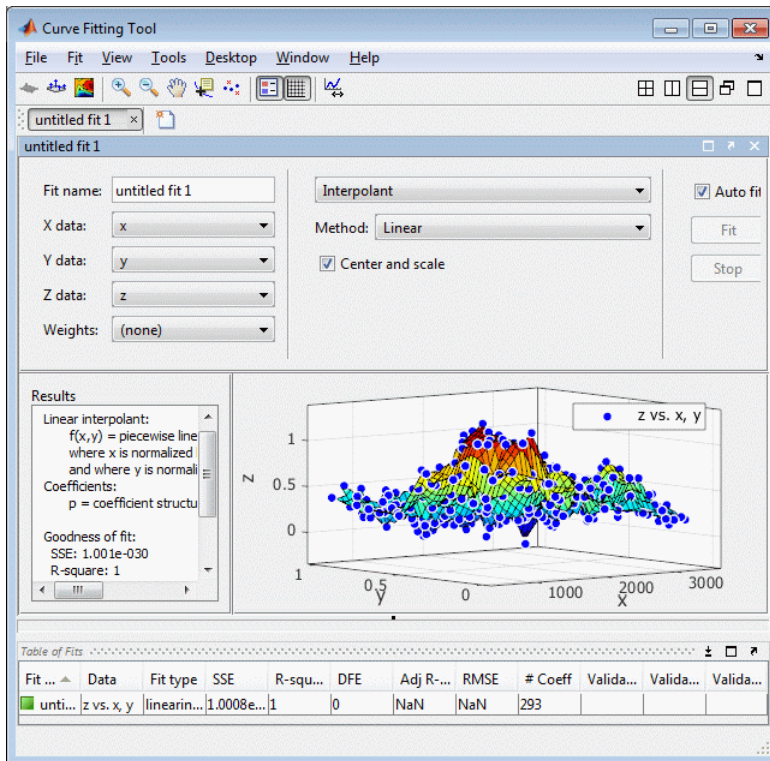
4   Try a Lowess fit type. Select the `Lowess` fit type from the drop-down list in the Curve Fitting app.
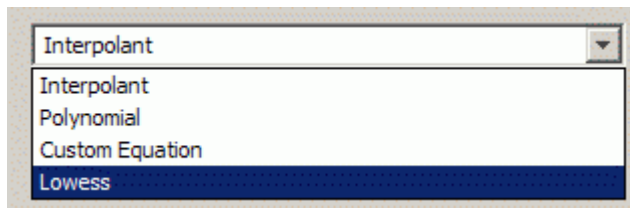
The Curve Fitting app creates a local smoothing regression fit.

5   Try altering the fit settings. Enter `10` in the **Span** edit box.

By reducing the span from the default to 10% of the total number of data points you produce a surface that follows the data more closely. The span defines the neighboring data points the toolbox uses to determine each smoothed value.
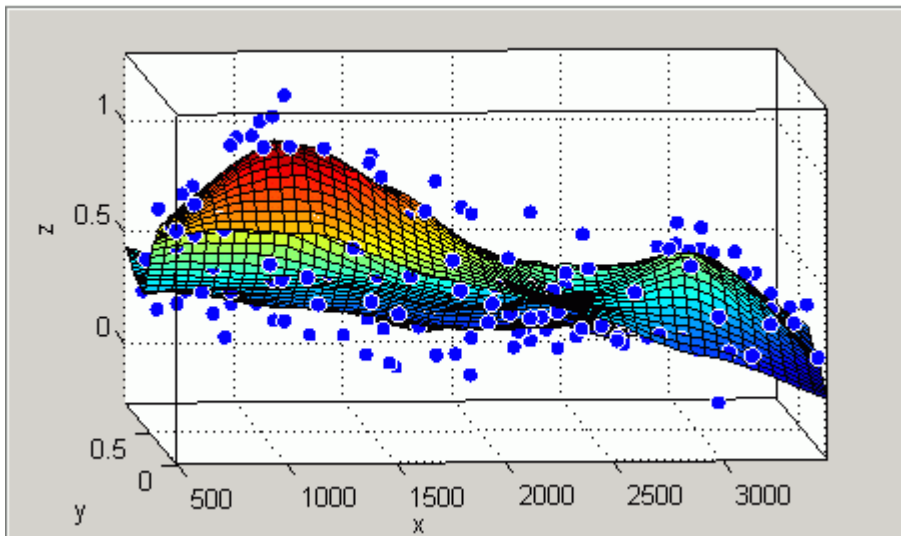
6 Edit the **Fit name** to `Smoothing regression`.

7 If you divided your data into fitting and validation data in step 2, select this validation data. Use the validation data to help you check that your surface is a good model, by comparing it against some other data not used for fitting.

    a Select **Fit > Specify Validation Data**. The Specify Validation Data dialog box opens.

    b Select the validation variables in the drop-down lists for **X input**, **Y input**, and **Z output**: `xv`, `yv`, and `zv`.

Review your selected validation data in the plots and the validation statistics (SSE and RMSE) in the **Results** pane and the **Table of Fits**.
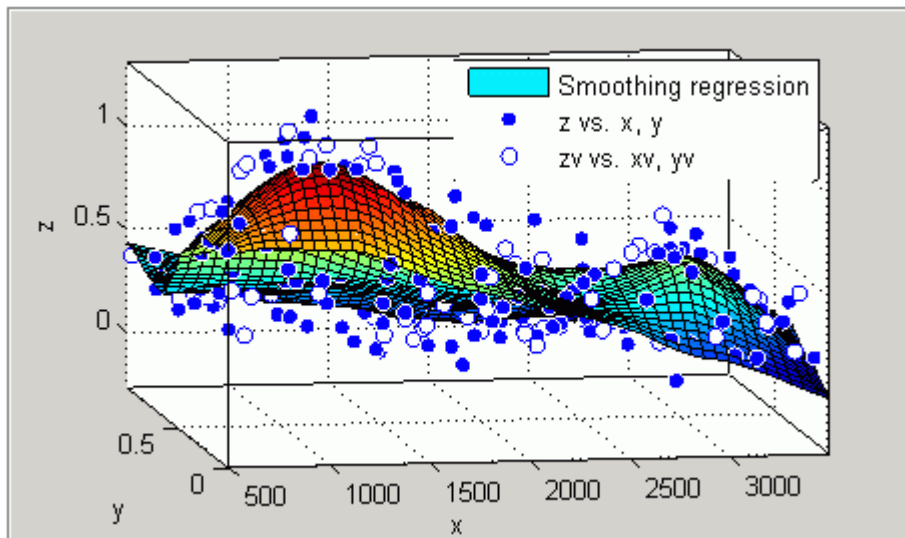
8 Create another fit to compare by making a copy of the current surface fit. Either select **Fit > Duplicate "Smoothing regression"**, or right-click the fit in the **Table of Fits**, and select **Duplicate**

The tool creates a new fit figure with the same fit settings, data, and validation data. It also adds a new row to the table of fits at the bottom.

9   Change the fit type to `Polynomial` and edit the fit name to `Polynomial`.
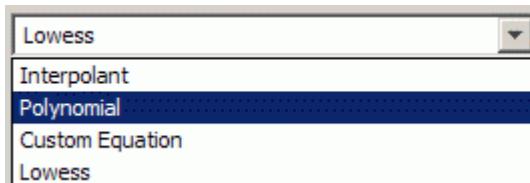
10   Change the **Degrees** of **x** and **y** to 3, to fit a cubic polynomial in both dimensions.

11   Look at the scales on the x and y axes, and read the warning message in the **Results** pane:

```
Equation is badly conditioned. Remove repeated data points
        or try centering and scaling.
```

Select the **Center and scale** check box to normalize and correct for the large difference in scales in x and y.
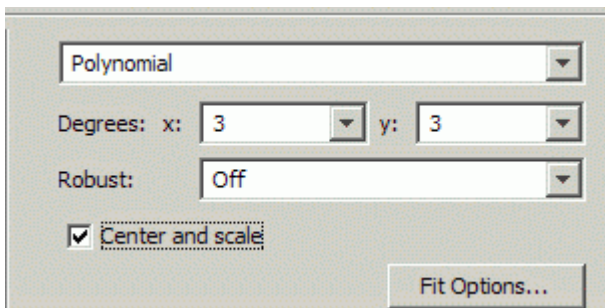
Normalizing the surface fit removes the warning message from the **Results** pane.

12   Look at the **Results** pane. You can view (and copy if desired):

- The model equation
- The values of the estimated coefficients
- The goodness-of-fit statistics
- The goodness of validation statistics

```
Linear model Poly33:
     f(x,y) = p00 + p10*x + p01*y + p20*x^2 + p11*x*y...
                    + p02*y^2 + p30*x^3 + p21*x^2*y
                    + p12*x*y^2 + p03*y^3
     where x is normalized by mean 1977 and std 866.5
     and where y is normalized by mean 0.4932 and std 0.29
Coefficients (with 95% confidence bounds):
       p00 =      0.4359  (0.3974, 0.4743)
       p10 =     -0.1375  (-0.194, -0.08104)
       p01 =     -0.4274  (-0.4843, -0.3706)
       p20 =      0.0161  (-0.007035, 0.03923)
       p11 =     0.07158  (0.05091, 0.09225)
       p02 =    -0.03668  (-0.06005, -0.01332)
       p30 =     0.02081  (-0.005475, 0.04709)
       p21 =     0.02432  (0.0012, 0.04745)
       p12 =    -0.03949  (-0.06287, -0.01611)
       p03 =      0.1185  (0.09164, 0.1453)
```

2-29

```
Goodness of fit:
  SSE: 4.125
  R-square: 0.776
  Adjusted R-square: 0.7653
  RMSE: 0.1477

Goodness of validation:
  SSE : 2.26745
  RMSE : 0.155312
```

13 To export this fit information to the workspace, select **Fit > Save to Workspace**. Executing this command also exports other information such as the numbers of observations and parameters, residuals, and the fitted model.

You can treat the fitted model as a function to make predictions or evaluate the surface at values of X and Y. For details see "Exporting a Fit to the Workspace" on page 7-14.

14 Display the residuals plot to check the distribution of points relative to the surface. Click the toolbar button or select **View > Residuals Plot**.
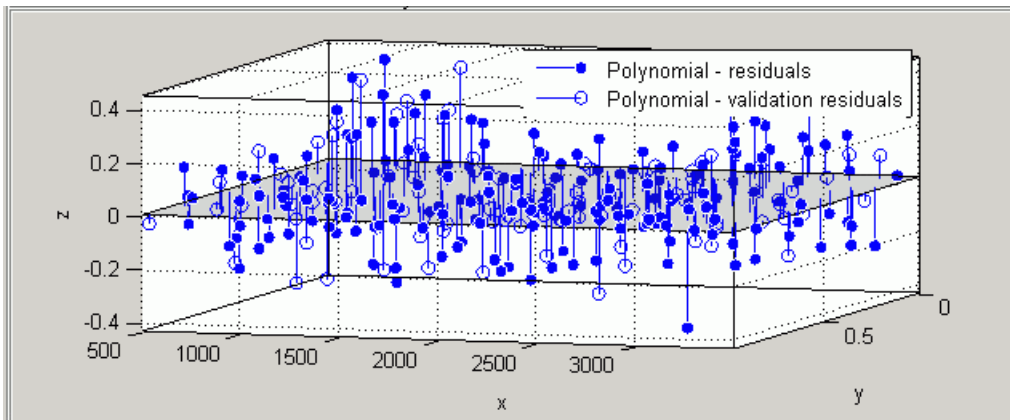
15 Right-click the residuals plot to select the **Go to X-Z view**. The X-Z view is not required, but the view makes it easier to see to remove outliers.

16 To remove outliers, click the toolbar button or select **Tools > Exclude Outliers**.

When you move the mouse cursor to the plot, it changes to a cross-hair to show you are in outlier selection mode.

a Click a point that you want to exclude in the surface plot or residuals plot. Alternatively, click and drag to define a rectangle and remove all enclosed points.

A removed plot point displays as a red star in the plots.
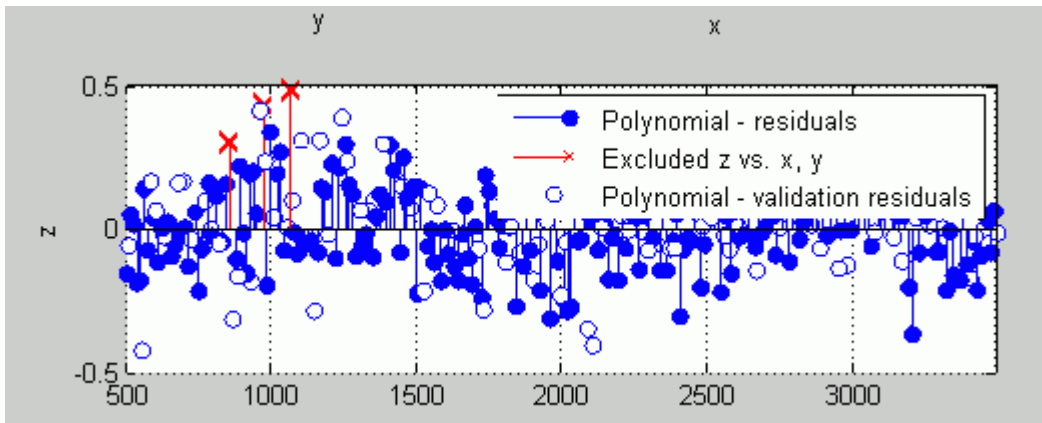
b If you have **Auto-fit** selected, the Curve Fitting app refits the surface without the point. Otherwise, you can click **Fit** to refit the surface.

c To return to rotation mode, click the toolbar button ![icon] again to switch off **Exclude Outliers** mode.

17 To compare your fits side-by-side, use the tile tools. Select **Window > Left/Right Tile**, or use the toolbar buttons.
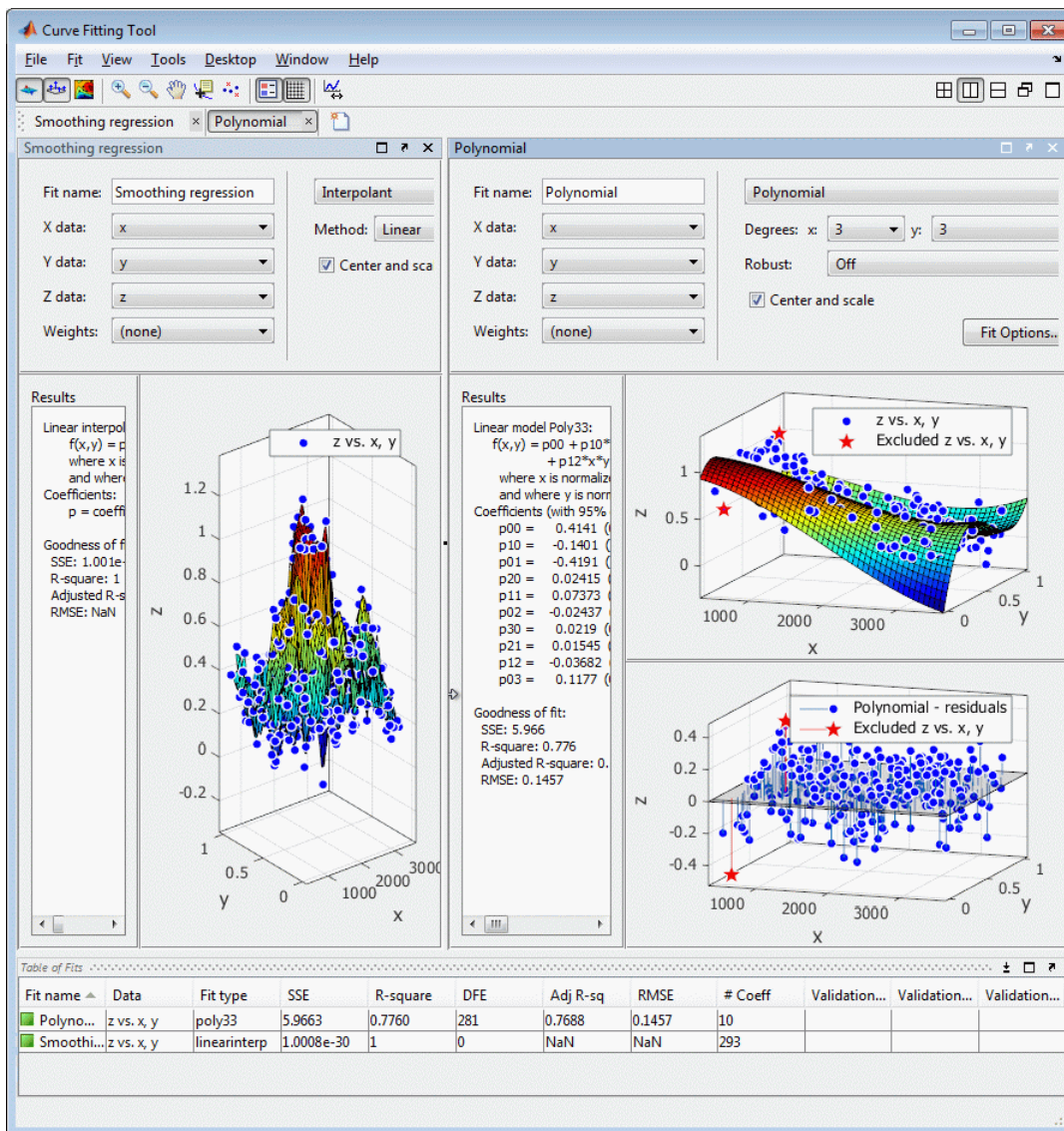
18 Review the information in the **Table of Fits**. Compare goodness-of-fit statistics for all fits in your session to determine which is best.

19 To save your interactive surface fitting session, select **File > Save Session**. You can save and reload sessions to access multiple fits. The session file contains all the fits and variables in your session and remembers your layout.

20 After interactively creating and comparing fits, you can generate code for all fits and plots in your Curve Fitting app session. Select **File > Generate Code**.

The Curve Fitting app generates code from your session and displays the file in the MATLAB Editor. The file includes all fits and plots in your current session.

21 Save the file with the default name, `createFits.m`.

22 You can recreate your fits and plots by calling the file from the command line (with your original data or new data as input arguments). In this case, your original variables still appear in the workspace.

- Highlight and evaluate the first line of the file (excluding the word `function`). Either right-click and select **Evaluate**, press **F9**, or copy and paste the following to the command line:

  ```
  [fitresult, gof] = createFits(x, y, z, xv, yv, zv)
  ```

- The function creates a figure window for each fit you had in your session. Observe that the polynomial fit figure shows both the surface and residuals plots that you created interactively in the Curve Fitting app.

- If you want you can use the generated code as a starting point to change the surface fits and plots to fit your needs. For a list of methods you can use, see `sfit`.

For more information on all fit settings and tools for comparing fits, see:

- "Create Multiple Fits in Curve Fitting App" on page 2-11
- "Linear and Nonlinear Regression"
- "Interpolation"
- "Smoothing"
- "Fit Postprocessing"

3

Programmatic Curve and Surface Fitting

- "Curve and Surface Fitting" on page 3-2
- "Curve and Surface Fitting Objects and Methods" on page 3-6

Curve and Surface Fitting

In this section...
"Fitting a Curve" on page 3-2
"Fitting a Surface" on page 3-2
"Model Types and Fit Analysis" on page 3-2
"Workflow for Command Line Fitting" on page 3-3

Fitting a Curve

To programmatically fit a curve, follow the steps in this simple example:

1 Load some data.

```
load hahn1
```

Create a fit using the `fit` function, specifying the variables and a model type (in this case `rat23` is the model type).

```
f = fit( temp, thermex, 'rat23' )
```

Plot your fit and the data.

```
plot( f, temp, thermex )
f( 600 )
```

For an example comparing various polynomial fits, see "Polynomial Curve Fitting" on page 12-2.

Fitting a Surface

To programmatically fit a surface, follow the steps in this simple example:

1 Load some data.

```
load franke
```

2 Create a fit using the `fit` function, specifying the variables and a model type (in this case `poly23` is the model type).

```
 f = fit( [x, y], z, 'poly23' )
```

3 Plot your fit and the data.

```
plot(f, [x,y], z)
```

For an example fitting custom equations, see "Surface Fitting With Custom Equations to Biopharmaceutical Data" on page 12-14.

Model Types and Fit Analysis

For details and examples of specific model types and fit analysis, see the following sections:

1 "Linear and Nonlinear Regression"

2 "Interpolation"

3 "Smoothing"

4 "Fit Postprocessing"

Workflow for Command Line Fitting

Curve Fitting Toolbox software provides a variety of methods for data analysis and modeling.

Tip To quickly assemble MATLAB code for curve and surface fits and plots, use Curve Fitting app and then generate code. You can transform your interactive analysis of a single data set into a reusable function for command-line analysis or for batch processing of multiple data sets. See "Generate Code and Export Fits to the Workspace" on page 7-13.

To use curve fitting functions for programmatic fitting and analysis, follow this workflow:

1 Import your data into the MATLAB workspace using the `load` command (if your data has previously been stored in MATLAB variables) or any of the MATLAB functions for reading data from particular file types. You might need to reshape your data: see `prepareCurveData` or `prepareSurfaceData`.

2 (Optional) If your data is noisy, you might want to smooth it using the `smooth` function. Smoothing is used to identify major trends in the data that can assist you in choosing an appropriate family of parametric models. If a parametric model is not evident or appropriate, smoothing can be an end in itself, providing a nonparametric fit of the data.

Note Smoothing estimates the center of the distribution of the response at each predictor. It invalidates the assumption that errors in the data are independent, and so also invalidates the methods used to compute confidence and prediction intervals. Accordingly, once a parametric model is identified through smoothing, the *original* data should be passed to the `fit` function.

---

3 Specify a parametric model for the data—either a Curve Fitting Toolbox library model or a custom model that you define. You specify the model by passing a string or expression to the `fit` function or (optional) with a `fittype` object you create with the `fittype` function.

To view available library models, see "List of Library Models for Curve and Surface Fitting" on page 4-10.

4 (Optional) You can create a fit options structure for the fit using the `fitoptions` function. Fit options specify things like weights for the data, fitting methods, and low-level options for the fitting algorithm.

5 (Optional) You can create an exclusion rule for the fit using the `excludedata` function. Exclusion rules indicate which data values will be treated as outliers and excluded from the fit.

6 Specify the x and y (and z, if surface fitting) data, a model (string, expression or `fittype` object), and (optionally) a fit options structure and an exclusion rule, with the `fit` function to perform the fit.

The `fit` function returns a `cfit` (for curves) or `sfit` (for surfaces) object that encapsulates the computed coefficients and the fit statistics. If you want to learn more about the fit objects, see "Curve and Surface Fitting Objects and Methods" on page 3-6.

7   You can postprocess the fit objects returned by the `fit` function, by passing them to a variety of functions, such as `feval`, `differentiate`, `integrate`, `plot`, `coeffvalues`, `probvalues`, `confint`, and `predint`.

Use the following functions to work with curve and surface fits.

Curve or Surface Fit Method	Description
`argnames`	Get input argument names
`category`	Get fit category
`coeffnames`	Get coefficient names
`coeffvalues`	Get coefficient values
`confint`	Get confidence intervals for fit coefficients
`dependnames`	Get dependent variable name
`differentiate`	Differentiate fit
`excludedata`	Exclude data from fit
`feval`	Evaluate model at specified predictors
`fittype`	Construct `fittype` object
`formula`	Get formula string
`indepnames`	Get independent variable name
`integrate`	Integrate curve fit
`islinear`	Determine if model is linear
`numargs`	Get number of input arguments
`numcoeffs`	Get number of coefficients
`plot`	Plot fit
`predint`	Get prediction intervals
`probnames`	Get problem-dependent parameter names
`probvalues`	Get problem-dependent parameter values
`quad2d`	Numerically integrate surface fit (`sfit` object)
`setoptions`	Set model fit options
`type`	Get name of model

See Also
`excludedata` | `fit` | `fitoptions` | `fittype` | `prepareCurveData` | `prepareSurfaceData`

Related Examples

- "Generating MATLAB Code and Exporting Fits" on page 2-15
- "List of Library Models for Curve and Surface Fitting" on page 4-10
- "Polynomial Curve Fitting" on page 12-2
- "Custom Nonlinear Census Fitting" on page 5-16
- "Surface Fitting With Custom Equations to Biopharmaceutical Data" on page 12-14

- "Evaluate a Curve Fit" on page 7-16
- "Evaluate a Surface Fit" on page 7-25
- "Fit Postprocessing"

Curve and Surface Fitting Objects and Methods

In this section...
"Curve Fitting Objects" on page 3-6
"Curve Fitting Methods" on page 3-7
"Surface Fitting Objects and Methods" on page 3-9

This section describes how to use Curve Fitting Toolbox functions from the command-line or to write programs for curve and surface fitting applications.

The Curve Fitting app allows convenient, interactive use of Curve Fitting Toolbox functions, without programming. You can, however, access Curve Fitting Toolbox functions directly, and write programs that combine curve fitting functions with MATLAB functions and functions from other toolboxes. This allows you to create a curve fitting environment that is precisely suited to your needs.

Models and fits in the Curve Fitting app are managed internally as curve fitting *objects*. Objects are manipulated through a variety of functions called *methods*. You can create curve fitting objects, and apply curve fitting methods, outside of the Curve Fitting app.

Curve Fitting Objects

In MATLAB programming, all workspace variables are *objects* of a particular *class*. Familiar examples of MATLAB classes are `double`, `char`, and `function_handle`. You can also create custom MATLAB classes, using object-oriented programming.

Methods are functions that operate exclusively on objects of a particular class. *Data types* package together objects and methods so that the methods operate exclusively on objects of their own type, and not on objects of other types. A clearly defined encapsulation of objects and methods is the goal of object-oriented programming.

Curve Fitting Toolbox software provides you with new MATLAB data types for performing curve fitting:

- `fittype` — Objects allow you to encapsulate information describing a parametric model for your data. Methods allow you to access and modify that information.
- `cfit` and `sfit` — Two subtypes of `fittype`, for curves and surfaces. Objects capture information from a particular fit by assigning values to coefficients, confidence intervals, fit statistics, etc. Methods allow you to post-process the fit through plotting, extrapolation, integration, etc.
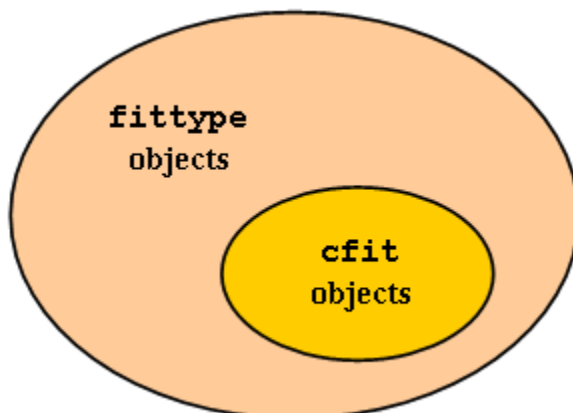
Because `cfit` is a subtype of `fittype`, `cfit` inherits all `fittype` methods. In other words, you can apply `fittype` methods to both `fittype` and `cfit` objects, but `cfit` methods are used exclusively with `cfit` objects. Similarly for `sfit` objects.

As an example, the `fittype` method `islinear`, which determines if a model is linear or nonlinear, would apply equally well before or after a fit; that is, to both `fittype` and `cfit` objects. On the other hand, the `cfit` methods `coeffvalues` and `confint`, which, respectively, return fit coefficients and their confidence intervals, would make no sense if applied to a general `fittype` object which describes a parametric model with undetermined coefficients.

Curve fitting objects have properties that depend on their type, and also on the particulars of the model or the fit that they encapsulate. For example, the following code uses the constructor methods for the two curve fitting types to create a `fittype` object `f` and a `cfit` object `c`:

```
f = fittype('a*x^2+b*exp(n*x)')
f =
     General model:
       f(a,b,n,x) = a*x^2+b*exp(n*x)
c = cfit(f,1,10.3,-1e2)
c =
     General model:
       c(x) = a*x^2+b*exp(n*x)
     Coefficients:
       a =            1
       b =         10.3
       n =         -100
```

Note that the display method for `fittype` objects returns only basic information, piecing together outputs from `formula` and `indepnames`.

`cfit` and `fittype` objects are evaluated at predictor values `x` using `feval`. You can call `feval` indirectly using the following functional syntax:

```
y = cfun(x) % cfit objects;
y = ffun(coef1,coef2,...,x) % fittype objects;
```

Curve Fitting Methods

Curve fitting methods allow you to create, access, and modify curve fitting objects. They also allow you, through methods like `plot` and `integrate`, to perform operations that uniformly process the entirety of information encapsulated in a curve fitting object.

The methods listed in the following table are available for all `fittype` objects, including `cfit` objects.

Fit Type Method	Description
argnames	Get input argument names
category	Get fit category
coeffnames	Get coefficient names
dependnames	Get dependent variable name
feval	Evaluate model at specified predictors
fittype	Construct `fittype` object

Fit Type Method	Description
`formula`	Get formula string
`indepnames`	Get independent variable name
`islinear`	Determine if model is linear
`numargs`	Get number of input arguments
`numcoeffs`	Get number of coefficients
`probnames`	Get problem-dependent parameter names
`setoptions`	Set model fit options
`type`	Get name of model

The methods listed in the following table are available exclusively for `cfit` objects.

Curve Fit Method	Description
`cfit`	Construct `cfit` object
`coeffvalues`	Get coefficient values
`confint`	Get confidence intervals for fit coefficients
`differentiate`	Differentiate fit
`integrate`	Integrate fit
`plot`	Plot fit
`predint`	Get prediction intervals
`probvalues`	Get problem-dependent parameter values

A complete list of methods for a curve fitting object can be obtained with the MATLAB `methods` command. For example,

```
f = fittype('a*x^2+b*exp(n*x)');
methods(f)

Methods for class fittype:

argnames      dependnames   fittype       islinear      probnames
category      feval         formula       numargs       setoptions
coeffnames    fitoptions    indepnames    numcoeffs     type
```

Note that some of the methods listed by `methods` do not appear in the tables above, and do not have reference pages in the Curve Fitting Toolbox documentation. These additional methods are generally low-level operations used by the Curve Fitting app, and not of general interest when writing curve fitting applications.

There are no global accessor methods, comparable to `getfield` and `setfield`, available for `fittype` objects. Access is limited to the methods listed above. This is because many of the properties of `fittype` objects are derived from other properties, for which you do have access. For example,

```
f = fittype('a*cos( b*x-c )')
f =
     General model:
       f(a,b,c,x) = a*cos( b*x-c )
```

```
formula(f)
ans =
a*cos( b*x-c )

argnames(f)
ans =
    'a'
    'b'
    'c'
    'x'
```

You construct the `fittype` object `f` by giving the formula, so you do have write access to that basic property of the object. You have read access to that property through the `formula` method. You also have read access to the argument names of the object, through the `argnames` method. You don't, however, have direct write access to the argument names, which are derived from the formula. If you want to set the argument names, set the formula.

Surface Fitting Objects and Methods

Surface Fitting Objects and Methods

The surface fit object (`sfit`) stores the results from a surface fitting operation, making it easy to plot and analyze fits at the command line.

Like `cfit` objects, `sfit` objects are a subclass of `fittype` objects, so they inherit all the same methods of `fittype` listed in "Curve Fitting Methods" on page 3-7.

`sfit` objects also provide methods exclusively for `sfit` objects. See `sfit`.

One way to quickly assemble code for surface fits and plots into useful programs is to generate a file from a session in the Curve Fitting app. In this way, you can transform your interactive analysis of a single data set into a reusable function for command-line analysis or for batch processing of multiple data sets. You can use the generated file without modification, or edit and customize the code as needed. See "Generate Code and Export Fits to the Workspace" on page 7-13.

Linear and Nonlinear Regression

Parametric Fitting

In this section...
"Parametric Fitting with Library Models" on page 4-2
"Selecting a Model Type Interactively" on page 4-3
"Selecting Model Type Programmatically" on page 4-4
"Using Normalize or Center and Scale" on page 4-4
"Specifying Fit Options and Optimized Starting Points" on page 4-5

Parametric Fitting with Library Models

Parametric fitting involves finding coefficients (parameters) for one or more models that you fit to data. The data is assumed to be statistical in nature and is divided into two components:

data = deterministic component + random component

The deterministic component is given by a parametric model and the random component is often described as error associated with the data:

data = parametric model + error

The model is a function of the independent (predictor) variable and one or more coefficients. The error represents random variations in the data that follow a specific probability distribution (usually Gaussian). The variations can come from many different sources, but are always present at some level when you are dealing with measured data. Systematic variations can also exist, but they can lead to a fitted model that does not represent the data well.

The model coefficients often have physical significance. For example, suppose you collected data that corresponds to a single decay mode of a radioactive nuclide, and you want to estimate the half-life ($T_{1/2}$) of the decay. The law of radioactive decay states that the activity of a radioactive substance decays exponentially in time. Therefore, the model to use in the fit is given by

$$y = y_0 e^{-\lambda t}$$

where y_0 is the number of nuclei at time $t = 0$, and λ is the decay constant. The data can be described by

$$\text{data} = y_0 e^{-\lambda t} + \text{error}$$

Both y_0 and λ are coefficients that are estimated by the fit. Because $T_{1/2} = \ln(2)/\lambda$, the fitted value of the decay constant yields the fitted half-life. However, because the data contains some error, the deterministic component of the equation cannot be determined exactly from the data. Therefore, the coefficients and half-life calculation will have some uncertainty associated with them. If the uncertainty is acceptable, then you are done fitting the data. If the uncertainty is not acceptable, then you might have to take steps to reduce it either by collecting more data or by reducing measurement error and collecting new data and repeating the model fit.

With other problems where there is no theory to dictate a model, you might also modify the model by adding or removing terms, or substitute an entirely different model.

The Curve Fitting Toolbox parametric library models are described in the following sections.

Selecting a Model Type Interactively

Select a model type to fit from the drop-down list in the Curve Fitting app.

What fit types can you use for curves or surfaces? Based on your selected data, the fit category list shows either curve or surface categories. The following table describes the options for curves and surfaces.

Fit Category	Curves	Surfaces
Regression Models		
Polynomial on page 4-14	Yes (up to degree 9)	Yes (up to degree 5)
Exponential on page 4-28	Yes	
Fourier on page 4-34	Yes	
Gaussian on page 4-42	Yes	
Power on page 4-45	Yes	
Rational on page 4-48	Yes	
Sum of Sine on page 4-54	Yes	
Weibull on page 4-56	Yes	
Interpolation		
Interpolant on page 6-3	Yes Methods: Nearest neighbor Linear Cubic Shape-preserving (PCHIP)	Yes Methods: Nearest neighbor Linear Cubic Biharmonic Thin-plate spline
Smoothing		
Smoothing Spline on page 6-9	Yes	
Lowess on page 6-16		Yes
Custom		
Custom Equation on page 5-2	Yes	Yes
"Custom Linear Fitting" on page 5-6		Yes

For all fit categories, look in the **Results** pane to see the model terms, the values of the coefficients, and the goodness-of-fit statistics.

Tip   If your fit has problems, messages in the **Results** pane help you identify better settings.

Selecting Fit Settings

The Curve Fitting app provides a selection of fit types and settings that you can change to try to improve your fit. Try the defaults first, then experiment with other settings.

For an overview of how to use the available fit options, see "Specifying Fit Options and Optimized Starting Points" on page 4-5.

You can try a variety of settings within a single fit figure, and you can also create multiple fits to compare. When you create multiple fits you can compare different fit types and settings side by side in the Curve Fitting app. See "Create Multiple Fits in Curve Fitting App" on page 2-11.

Selecting Model Type Programmatically

You can specify a library model name as a string when you call the `fit` function. For example, to specify a quadratic `poly2`:

```
f = fit( x, y, 'poly2' )
```

See "List of Library Models for Curve and Surface Fitting" on page 4-10 to view all available library model names.

You can also use the `fittype` function to construct a `fittype` object for a library model, and use the `fittype` as an input to the `fit` function.

Use the `fitoptions` function to find out what parameters you can set, for example:

```
fitoptions(poly2)
```

For examples, see the sections for each model type, listed in the table in "Selecting a Model Type Interactively" on page 4-3. For details on all the functions for creating and analysing models, see "Curve and Surface Fitting" on page 3-2.

Using Normalize or Center and Scale

Most model types in the Curve Fitting app share the **Center and scale** option. When you select this option, the tool refits with the data centered and scaled, by applying the `Normalize` setting to the variables. At the command line, you can use `Normalize` as an input argument to the `fitoptions` function. See the `fitoptions` reference page.

Generally, it is a good idea to normalize inputs (also known as *predictor data*), which can alleviate numerical problems with variables of different scales. For example, suppose your surface fit inputs are engine speed with a range of 500–4500 r/min and engine load percentage with a range of 0–1. Then, **Center and scale** generally improves the fit because of the great difference in scale between the two inputs. However, if your inputs are in the same units or similar scale (e.g., eastings and northings for geographic data), then **Center and scale** is less useful. When you normalize inputs with this option, the values of the fitted coefficients change when compared to the original data.

If you are fitting a curve or surface to estimate coefficients, or the coefficients have physical significance, clear the **Center and scale** check box. The Curve Fitting app plots use the original scale with or without the **Center and scale** option.

At the command line, to set the option to center and scale the data before fitting, create the default fit options structure, set `Normalize` to `on`, then fit with the options:

```
options = fitoptions;
options.Normal = 'on';
options
options =
    Normalize: 'on'
     Exclude: [1x0 double]
     Weights: [1x0 double]
      Method: 'None'

load census
f1 = fit(cdate,pop,'poly3',options)
```

Specifying Fit Options and Optimized Starting Points

- "About Fit Options" on page 4-5
- "Fitting Method and Algorithm" on page 4-6
- "Finite Differencing Parameters" on page 4-7
- "Fit Convergence Criteria" on page 4-7
- "Coefficient Parameters" on page 4-7
- "Optimized Starting Points and Default Constraints" on page 4-8
- "Specifying Fit Options at the Command Line" on page 4-8

About Fit Options

Interactive fit options are described in the following sections. To specify the same fit options programmatically, see "Specifying Fit Options at the Command Line" on page 4-8.

To specify fit options interactively in the Curve Fitting app, click the **Fit Options** button to open the Fit Options dialog box. All fit categories except interpolants and smoothing splines have configurable fit options.

The available options depend on whether you are fitting your data using a linear model, a nonlinear model, or a nonparametric fit type:

- All the options described next are available for nonlinear models.
- **Lower** and **Upper** coefficient constraints are the only fit options available in the dialog box for polynomial linear models. For polynomials you can set **Robust** in the Curve Fitting app, without opening the Fit Options dialog box.
- Nonparametric fit types have no additional fit options dialog box (interpolant, smoothing spline, and lowess).

The fit options for the single-term exponential are shown next. The coefficient starting values and constraints are for the census data.
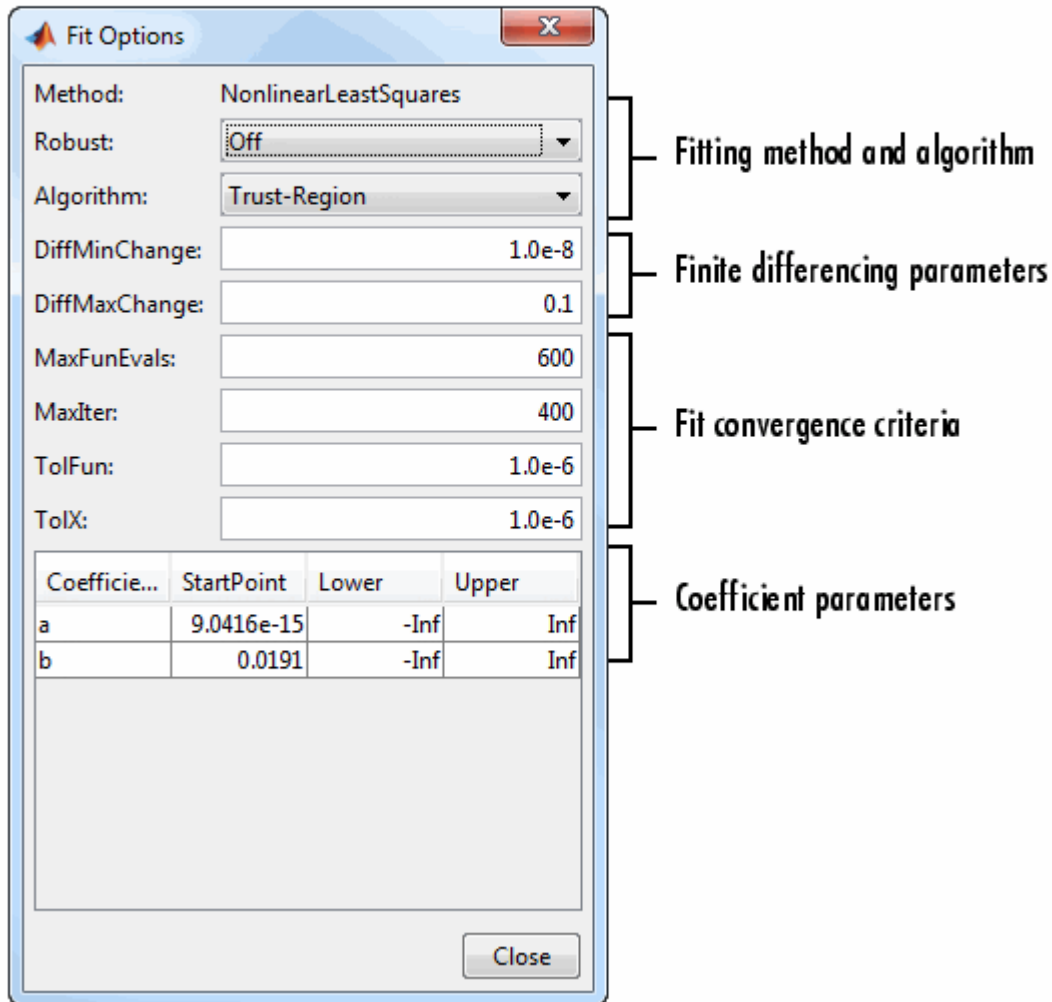
Fitting Method and Algorithm

- **Method** — The fitting method.

  The method is automatically selected based on the library or custom model you use. For linear models, the method is **LinearLeastSquares**. For nonlinear models, the method is **NonlinearLeastSquares**.

- **Robust** — Specify whether to use the robust least-squares fitting method.

  - **Off** — Do not use robust fitting (default).
  - **On** — Fit with the default robust method (bisquare weights).
  - **LAR** — Fit by minimizing the least absolute residuals (LAR).

- **Bisquare** — Fit by minimizing the summed square of the residuals, and reduce the weight of outliers using bisquare weights. In most cases, this is the best choice for robust fitting.
- **Algorithm** — Algorithm used for the fitting procedure:

  - **Trust-Region** — This is the default algorithm and must be used if you specify **Lower** or **Upper** coefficient constraints.
  - **Levenberg-Marquardt** — If the trust-region algorithm does not produce a reasonable fit, and you do not have coefficient constraints, try the Levenberg-Marquardt algorithm.

Finite Differencing Parameters

- **DiffMinChange** — Minimum change in coefficients for finite difference Jacobians. The default value is 10^{-8}.
- **DiffMaxChange** — Maximum change in coefficients for finite difference Jacobians. The default value is 0.1.

Note that **DiffMinChange** and **DiffMaxChange** apply to:

- Any nonlinear custom equation, that is, a nonlinear equation that you write
- Some of the nonlinear equations provided with Curve Fitting Toolbox software

However, **DiffMinChange** and **DiffMaxChange** do not apply to any linear equations.

Fit Convergence Criteria

- **MaxFunEvals** — Maximum number of function (model) evaluations allowed. The default value is 600.
- **MaxIter** — Maximum number of fit iterations allowed. The default value is 400.
- **TolFun** — Termination tolerance used on stopping conditions involving the function (model) value. The default value is 10^{-6}.
- **TolX** — Termination tolerance used on stopping conditions involving the coefficients. The default value is 10^{-6}.

Coefficient Parameters

- **Coefficients** — Symbols for the unknown coefficients to be fitted.
- **StartPoint** — The coefficient starting values. The default values depend on the model. For rational, Weibull, and custom models, default values are randomly selected within the range [0,1]. For all other nonlinear library models, the starting values depend on the data set and are calculated heuristically. See optimized starting points below.
- **Lower** — Lower bounds on the fitted coefficients. The tool only uses the bounds with the trust region fitting algorithm. The default lower bounds for most library models are `-Inf`, which indicates that the coefficients are unconstrained. However, a few models have finite default lower bounds. For example, Gaussians have the width parameter constrained so that it cannot be less than 0. See default constraints below.
- **Upper** — Upper bounds on the fitted coefficients. The tool only uses the bounds with the trust region fitting algorithm. The default upper bounds for all library models are `Inf`, which indicates that the coefficients are unconstrained.

For more information about these fit options, see the `lsqcurvefit` function in the Optimization Toolbox documentation.

Optimized Starting Points and Default Constraints

The default coefficient starting points and constraints for library and custom models are shown in the next table. If the starting points are optimized, then they are calculated heuristically based on the current data set. Random starting points are defined on the interval [0,1] and linear models do not require starting points.

If a model does not have constraints, the coefficients have neither a lower bound nor an upper bound. You can override the default starting points and constraints by providing your own values using the Fit Options dialog box.

Default Starting Points and Constraints

Model	Starting Points	Constraints
Custom linear	N/A	None
Custom nonlinear	Random	None
Exponential	Optimized	None
Fourier	Optimized	None
Gaussian	Optimized	$c_i > 0$
Polynomial	N/A	None
Power	Optimized	None
Rational	Random	None
Sum of Sine	Optimized	$b_i > 0$
Weibull	Random	$a, b > 0$

Note that the sum of sines and Fourier series models are particularly sensitive to starting points, and the optimized values might be accurate for only a few terms in the associated equations.

Specifying Fit Options at the Command Line

Create the default fit options structure and set the option to center and scale the data before fitting:

```
options = fitoptions;
options.Normal = 'on';
options
options =
    Normalize: 'on'
    Exclude: [1x0 double]
    Weights: [1x0 double]
    Method: 'None'
```

Modifying the default fit options structure is useful when you want to set the `Normalize`, `Exclude`, or `Weights` fields, and then fit your data using the same options with different fitting methods. For example:

```
load census
f1 = fit(cdate,pop,'poly3',options);
f2 = fit(cdate,pop,'exp1',options);
f3 = fit(cdate,pop,'cubicsp',options);
```

Data-dependent fit options are returned in the third output argument of the `fit` function. For example, the smoothing parameter for smoothing spline is data-dependent:

```
[f,gof,out] = fit(cdate,pop,'smooth');
smoothparam = out.p
smoothparam =
     0.0089
```

Use fit options to modify the default smoothing parameter for a new fit:

```
options = fitoptions('Method','Smooth','SmoothingParam',0.0098);
[f,gof,out] = fit(cdate,pop,'smooth',options);
```

For more details on using fit options, see the `fitoptions` reference page.

List of Library Models for Curve and Surface Fitting

In this section...
"Use Library Models to Fit Data" on page 4-10
"Library Model Types" on page 4-10
"Model Names and Equations" on page 4-11

Use Library Models to Fit Data

You can use the Curve Fitting Toolbox library of models for data fitting with the `fit` function. You use library model names as input arguments in the `fit`, `fitoptions`, and `fittype` functions.

Library Model Types

The following tables describe the library model types for curves and surfaces.

- Use the links in the table for examples and detailed information on each library type.
- If you want a quick reference of model names for input arguments to the `fit` function, see "Model Names and Equations" on page 4-11.

Library Model Types for Curves	**Description**
`distribution`	Distribution models such as Weibull. See "Weibull Distributions" on page 4-56.
`exponential`	Exponential function and sum of two exponential functions. See "Exponential Models" on page 4-28.
`fourier`	Up to eight terms of Fourier series. See "Fourier Series" on page 4-34.
`gaussian`	Sum of up to eight Gaussian models. See "Gaussian Models" on page 4-42.
`interpolant`	Interpolating models, including linear, nearest neighbor, cubic spline, and shape-preserving cubic spline. See "Nonparametric Fitting" on page 6-2.
`polynomial`	Polynomial models, up to degree nine. See "Polynomial Models" on page 4-14.
`power`	Power function and sum of two power functions. See "Power Series" on page 4-45.
`rational`	Rational equation models, up to 5th degree/5th degree (i.e., up to degree 5 in both the numerator and the denominator). See "Rational Polynomials" on page 4-48.
`sin`	Sum of up to eight sin functions. See "Sum of Sines Models" on page 4-54.
`spline`	Cubic spline and smoothing spline models. See "Nonparametric Fitting" on page 6-2.

Library Model Types for Surfaces	Description
`interpolant`	Interpolating models, including linear, nearest neighbor, cubic spline, biharmonic, and thin-plate spline interpolation. See "Interpolation Methods" on page 6-3.
`lowess`	Lowess smoothing models. See "Lowess Smoothing" on page 6-16.
`polynomial`	Polynomial models, up to degree five. See "Polynomial Models" on page 4-14.

Model Names and Equations

To specify the model you want to fit, consult the following tables for a model name to use as an input argument to the `fit` function. For example, to specify a quadratic curve with model name "`poly2`":

```
f = fit(x, y,  'poly2')
```

Polynomial Model Names and Equations

Examples of Polynomial Model Names for Curves	Equations
`poly1`	`Y = p1*x+p2`
`poly2`	`Y = p1*x^2+p2*x+p3`
`poly3`	`Y = p1*x^3+p2*x^2+...+p4`
...etc., up to `poly9`	`Y = p1*x^9+p2*x^8+...+p10`

For polynomial surfaces, model names are '`poly`ij', where i is the degree in x and j is the degree in y. The maximum for both i and j is five. The degree of the polynomial is the maximum of i and j. The degree of x in each term will be less than or equal to i, and the degree of y in each term will be less than or equal to j. See the following table for some example model names and equations, of many potential examples.

Examples of Polynomial Model Names for Surfaces	Equations
`poly21`	`Z = p00 + p10*x + p01*y + p20*x^2 + p11*x*y`
`poly13`	`Z = p00 + p10*x + p01*y + p11*x*y + p02*y^2 + p12*x*y^2 + p03*y^3`
`poly55`	`Z = p00 + p10*x + p01*y +...+ p14*x*y^4 + p05*y^5`

Distribution Model Name and Equation

Distribution Model Names	Equations
`weibull`	`Y = a*b*x^(b-1)*exp(-a*x^b)`

Exponential Model Names and Equations

Exponential Model Names	Equations
exp1	Y = a*exp(b*x)
exp2	Y = a*exp(b*x)+c*exp(d*x)

Fourier Series Model Names and Equations

Fourier Series Model Names	Equations
fourier1	Y = a0+a1*cos(x*p)+b1*sin(x*p)
fourier2	Y = a0+a1*cos(x*p)+b1*sin(x*p) +a2*cos(2*x*p)+b2*sin(2*x*p)
fourier3	Y = a0+a1*cos(x*p)+b1*sin(x*p) +...+a3*cos(3*x*p)+b3*sin(3*x*p)
...etc., up to fourier8	Y = a0+a1*cos(x*p)+b1*sin(x*p) +...+a8*cos(8*x*p)+b8*sin(8*x*p)

Where p = 2*pi/(max(xdata)-min(xdata)).

Gaussian Model Names and Equations

Gaussian Model Names	Equations
gauss1	Y = a1*exp(-((x-b1)/c1)^2)
gauss2	Y = a1*exp(-((x-b1)/c1)^2)+a2*exp(-((x-b2)/c2)^2)
gauss3	Y = a1*exp(-((x-b1)/c1)^2)+...+a3*exp(-((x-b3)/c3)^2)
...etc., up to gauss8	Y = a1*exp(-((x-b1)/c1)^2)+...+a8*exp(-((x-b8)/c8)^2)

Power Model Names and Equations

Power Model Names	Equations
power1	Y = a*x^b
power2	Y = a*x^b+c

Rational Model Names and Equations

Rational models are polynomials over polynomials with the leading coefficient of the denominator set to 1. Model names are rat*ij*, where *i* is the degree of the numerator and *j* is the degree of the denominator. The degrees go up to five for both the numerator and the denominator.

Examples of Rational Model Names	Equations
rat02	Y = (p1)/(x^2+q1*x+q2)
rat21	Y = (p1*x^2+p2*x+p3)/(x+q1)
rat55	Y = (p1*x^5+...+p6)/(x^5+...+q5)

Sum of Sine Model Names and Equations

Sum of Sine Model Names	Equations
sin1	Y = a1*sin(b1*x+c1)
sin2	Y = a1*sin(b1*x+c1)+a2*sin(b2*x+c2)
sin3	Y = a1*sin(b1*x+c1)+...+a3*sin(b3*x+c3)
...etc., up to sin8	Y = a1*sin(b1*x+c1)+...+a8*sin(b8*x+c8)

Spline Model Names

Spline models are supported for curve fitting, not for surface fitting.

Spline Model Names	Description
cubicspline	Cubic interpolating spline
smoothingspline	Smoothing spline

Interpolant Model Names

Type	Interpolant Model Names	Description
Curves and Surfaces	linearinterp	Linear interpolation
	nearestinterp	Nearest neighbor interpolation
	cubicinterp	Cubic spline interpolation
Curves only	pchipinterp	Shape-preserving piecewise cubic Hermite (pchip) interpolation
Surfaces only	biharmonicinterp	Biharmonic (MATLAB griddata) interpolation
	thinplateinterp	Thin-plate spline interpolation

Lowess Model Names

Lowess models are supported for surface fitting, not for curve fitting.

Lowess Model Names	Description
lowess	Local linear regression
loess	Local quadratic regression

Polynomial Models

In this section...
"About Polynomial Models" on page 4-14
"Fit Polynomial Models Interactively" on page 4-15
"Fit Polynomials Using the Fit Function" on page 4-16
"Polynomial Model Fit Options" on page 4-25
"Defining Polynomial Terms for Polynomial Surface Fits" on page 4-26

About Polynomial Models

Polynomial models for curves are given by

$$y = \sum_{i=1}^{n+1} p_i x^{n+1-i}$$

where $n + 1$ is the *order* of the polynomial, n is the *degree* of the polynomial, and $1 \le n \le 9$. The order gives the number of coefficients to be fit, and the degree gives the highest power of the predictor variable.

In this guide, polynomials are described in terms of their degree. For example, a third-degree (cubic) polynomial is given by

$$y = p_1 x^3 + p_2 x^2 + p_3 x + p_4$$

Polynomials are often used when a simple empirical model is required. You can use the polynomial model for interpolation or extrapolation, or to characterize data using a global fit. For example, the temperature-to-voltage conversion for a Type J thermocouple in the 0 to 760º temperature range is described by a seventh-degree polynomial.

Note If you do not require a global parametric fit and want to maximize the flexibility of the fit, piecewise polynomials might provide the best approach. Refer to "Nonparametric Fitting" on page 6-2 for more information.

The main advantages of polynomial fits include reasonable flexibility for data that is not too complicated, and they are linear, which means the fitting process is simple. The main disadvantage is that high-degree fits can become unstable. Additionally, polynomials of any degree can provide a good fit within the data range, but can diverge wildly outside that range. Therefore, exercise caution when extrapolating with polynomials.

When you fit with high-degree polynomials, the fitting procedure uses the predictor values as the basis for a matrix with very large values, which can result in scaling problems. To handle this, you should normalize the data by centering it at zero mean and scaling it to unit standard deviation. Normalize data by selecting the **Center and scale** check box in the Curve Fitting app.

Fit Polynomial Models Interactively

1 Open the Curve Fitting app by entering `cftool`. Alternatively, click Curve Fitting on the Apps tab.

2 In the Curve Fitting app, select curve or surface data.

- If you select curve data (**X data** and **Y data**, or just **Y data** against index), Curve Fitting app creates the default curve fit, `Polynomial`.

- If you select surface data (**X data**, **Y data**, and **Z data**), Curve Fitting app creates the default surface fit, `Interpolant`. Change the model type from `Interpolant` to `Polynomial`.

For *curves*, the `Polynomial` model fits a polynomial in **x**.

For *surfaces*, the `Polynomial` model fits a polynomial in **x** and **y**.

You can specify the following options:

- The degree for the **x** and **y** inputs:

  - For curves, degree of **x** can be up to 9.
  - For surfaces, degree of **x** and **y** can be up to 5.

    The degree of the polynomial is the maximum of **x** and **y** degrees. See "Defining Polynomial Terms for Polynomial Surface Fits" on page 4-26.

- The robust linear least-squares fitting method to use (`Off`, `LAR`, or `Bisquare`). For details, see `Robust` on the `fitoptions` reference page.

- Set bounds or exclude terms by clicking **Fit Options**. You can exclude any term by setting its bounds to 0.

  Look in the **Results** pane to see the model terms, the values of the coefficients, and the goodness-of-fit statistics.

Tip If your input variables have very different scales, select and clear the **Center and scale** check box to see the difference in the fit. Messages in the **Results** pane prompt you when scaling might improve your fit.

For an example comparing various polynomial fits, see "Compare Fits in Curve Fitting App" on page 2-16.

Fit Polynomials Using the Fit Function

This example shows how to use the `fit` function to fit polynomials to data. The steps fit and plot polynomial curves and a surface, specify fit options, return goodness of fit statistics, calculate predictions, and show confidence intervals.

The polynomial library model is an input argument to the fit and fittype functions. Specify the model type `poly` followed by the degree in x (up to 9), or x and y (up to 5). For example, you specify a quadratic curve with `'poly2'`, or a cubic surface with `'poly33'`.

Create and Plot a Quadratic Polynomial Curve

Load some data and fit a quadratic polynomial. Specify a quadratic, or second-degree polynomial, with the string `'poly2'`.

```
load census;
fitpoly2=fit(cdate,pop,'poly2')
% Plot the fit with the plot method.
plot(fitpoly2,cdate,pop)
% Move the legend to the top left corner.
legend('Location','NorthWest' );


fitpoly2 =

     Linear model Poly2:
     fitpoly2(x) = p1*x^2 + p2*x + p3
     Coefficients (with 95% confidence bounds):
       p1 =    0.006541  (0.006124, 0.006958)
       p2 =       -23.51  (-25.09, -21.93)
       p3 =    2.113e+04  (1.964e+04, 2.262e+04)
```
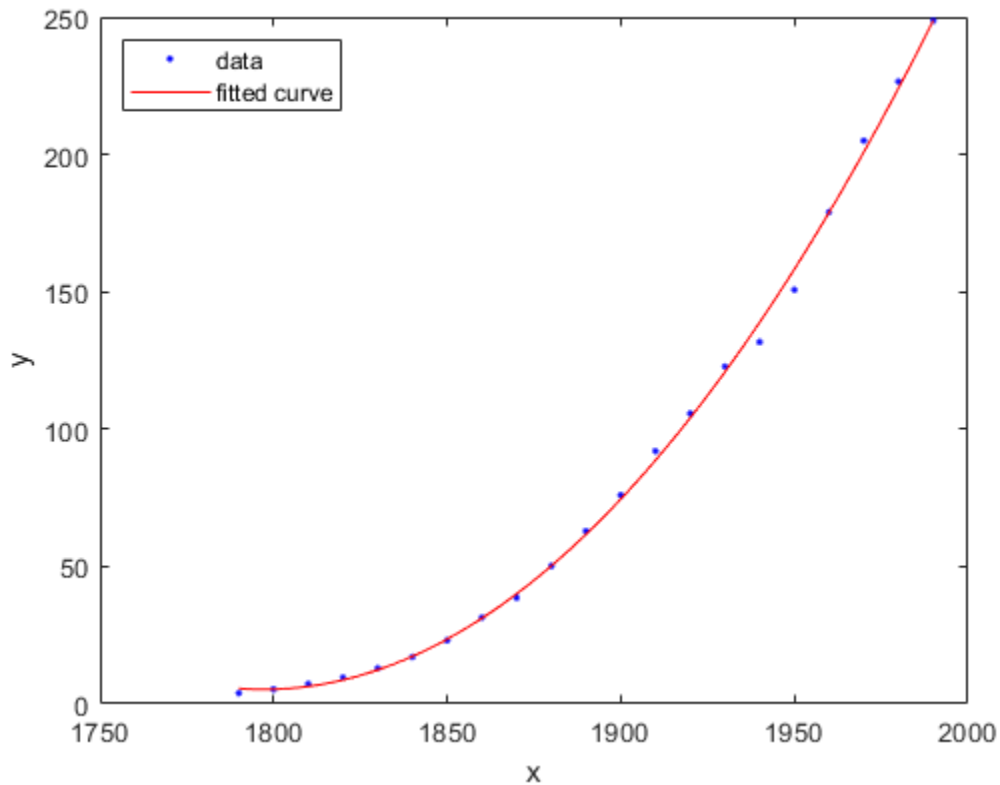
Create a Cubic Curve

Fit a cubic polynomial `'poly3'`.

```
fitpoly3=fit(cdate,pop,'poly3')
plot(fitpoly3,cdate,pop)
```

```
Warning: Equation is badly conditioned. Remove repeated data points or try
centering and scaling.
```

```
fitpoly3 =

     Linear model Poly3:
     fitpoly3(x) = p1*x^3 + p2*x^2 + p3*x + p4
     Coefficients (with 95% confidence bounds):
       p1 =   3.855e-06  (-4.078e-06, 1.179e-05)
       p2 =     -0.01532  (-0.06031, 0.02967)
       p3 =        17.78  (-67.2, 102.8)
       p4 =        -4852  (-5.834e+04, 4.863e+04)
```
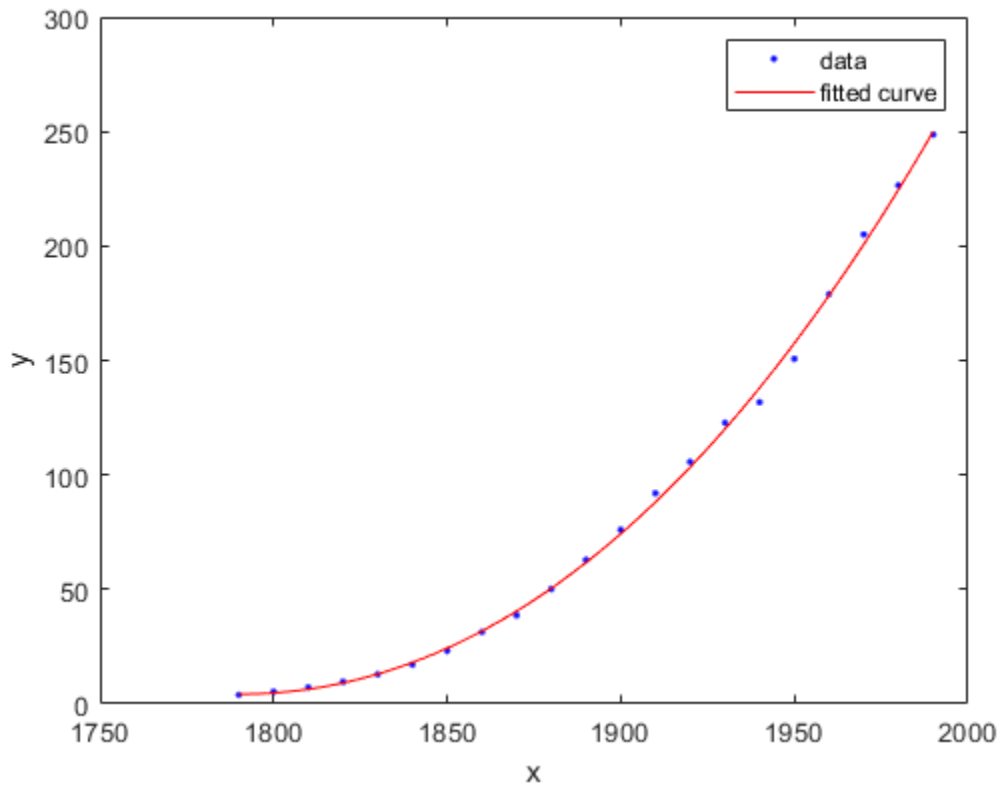
Specify Fit Options

The cubic fit warns that the equation is badly conditioned, so you should try centering and scaling by specifying the `'Normalize'` option. Fit the cubic polynomial with both center and scale and robust fitting options. Robust `'on'` is a shortcut equivalent to `'Bisquare'`, the default method for robust linear least-squares fitting method.

```
fit3=fit(cdate, pop,'poly3','Normalize','on','Robust','on')
plot(fit3,cdate,pop)
```

```
fit3 =

     Linear model Poly3:
     fit3(x) = p1*x^3 + p2*x^2 + p3*x + p4
       where x is normalized by mean 1890 and std 62.05
     Coefficients (with 95% confidence bounds):
       p1 =     -0.4619  (-1.895, 0.9707)
       p2 =       25.01  (23.79, 26.22)
       p3 =       77.03  (74.37, 79.7)
       p4 =       62.81  (61.26, 64.37)
```
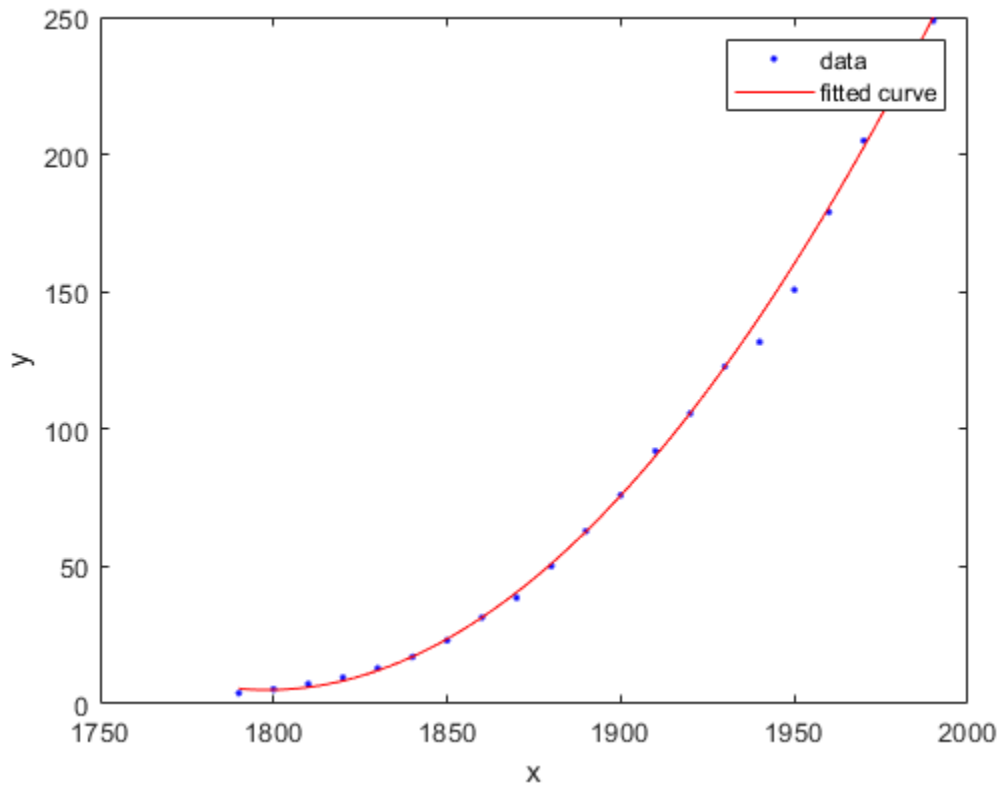
To find out what parameters you can set for the library model `'poly3'`, use the fitoptions function.

```
fitoptions poly3
```

```
ans =

    Normalize: 'off'
      Exclude: []
      Weights: []
       Method: 'LinearLeastSquares'
       Robust: 'Off'
        Lower: [1x0 double]
        Upper: [1x0 double]
```

Get Goodness of Fit Statistics

Specify the `'gof'` output argument to get the goodness-of-fit statistics for the cubic polynomial fit.

```
[fit4, gof]=fit(cdate, pop,'poly3','Normalize','on');
gof
```

```
gof =

  struct with fields:
```

```
        sse: 149.7687
    rsquare: 0.9988
        dfe: 17
 adjrsquare: 0.9986
       rmse: 2.9682
```

Plot the Residuals to Evaluate the Fit

To plot residuals, specify `'residuals'` as the plot type in the plot method.

```
plot(fit4,cdate, pop,'residuals');
```
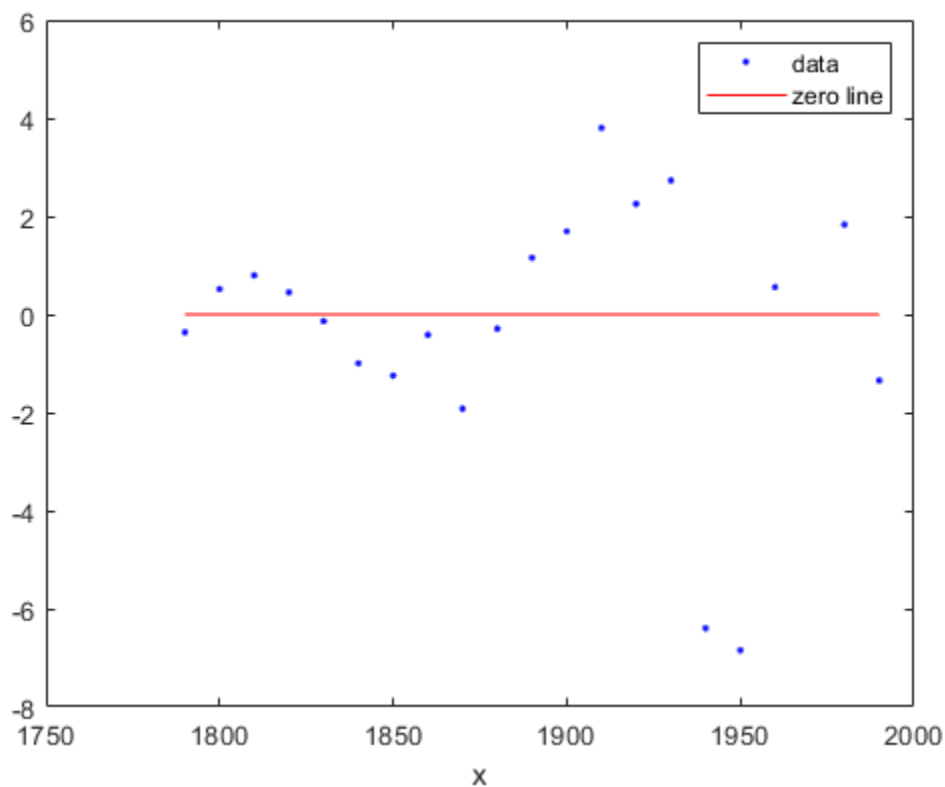

Examine a Fit Beyond the Data Range

By default, the fit is plotted over the range of the data. To plot a fit over a different range, set the x-limits of the axes before plotting the fit. For example, to see values extrapolated from the fit, set the upper x-limit to 2050.

```
plot( cdate, pop, 'o' );
xlim( [1900, 2050] );
hold on
plot( fit4 );
hold off
```
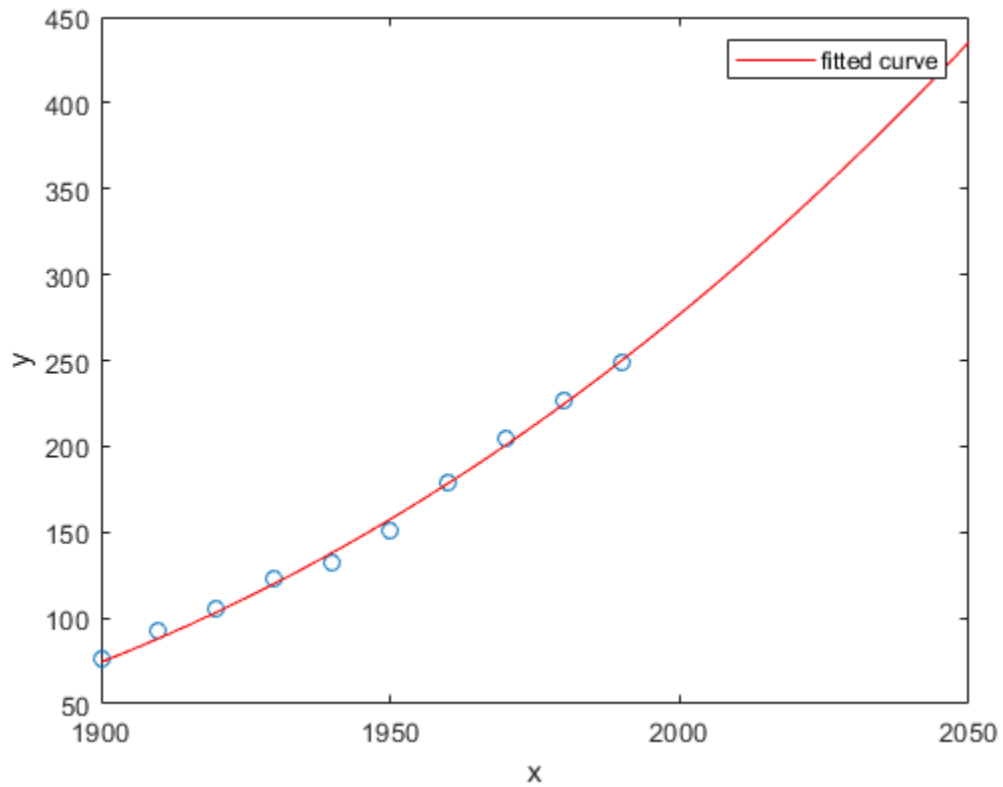
Plot Prediction Bounds

To plot prediction bounds, use `'predobs'` or `'predfun'` as the plot type.

```
plot(fit4,cdate,pop,'predobs')
```
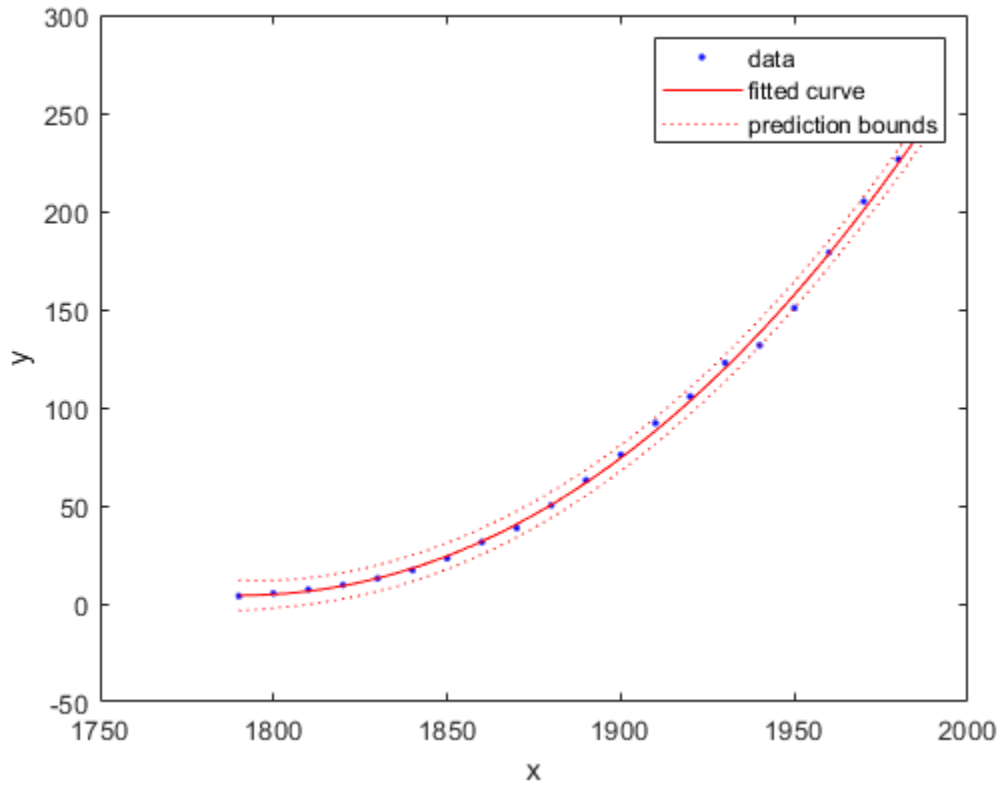
Plot prediction bounds for the cubic polynomial up to year 2050.

```
plot( cdate, pop, 'o' );
xlim( [1900, 2050] )
hold on
plot( fit4, 'predobs' );
hold off
```
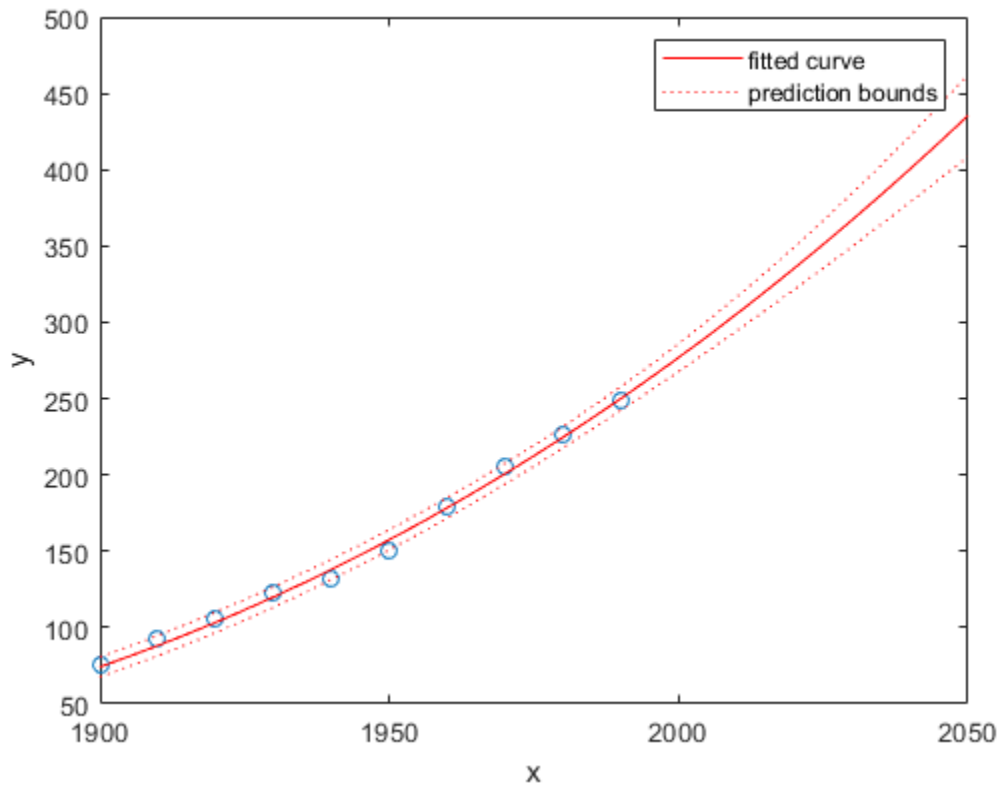
Get Confidence Bounds at New Query Points

Evaluate the fit for some new query points.

```
cdateFuture = (2000:10:2020).';
popFuture = fit4( cdateFuture )
```

```
popFuture =

  276.9632
  305.4420
  335.5066
```

Compute 95% confidence bounds on the prediction for the population in the future, using the predint method.

```
ci = predint( fit4, cdateFuture, 0.95, 'observation' )
```

```
ci =

  267.8589   286.0674
  294.3070   316.5770
  321.5924   349.4208
```

Plot the predicted future population, with confidence intervals, against the fit and data.

```
plot(cdate, pop, 'o');
xlim([1900, 2040])
hold on
plot(fit4)
h = errorbar(cdateFuture,popFuture,popFuture-ci(:,1),ci(:,2)-popFuture,'.');
hold off
legend('cdate v pop','poly3','prediction','Location','NorthWest')
```
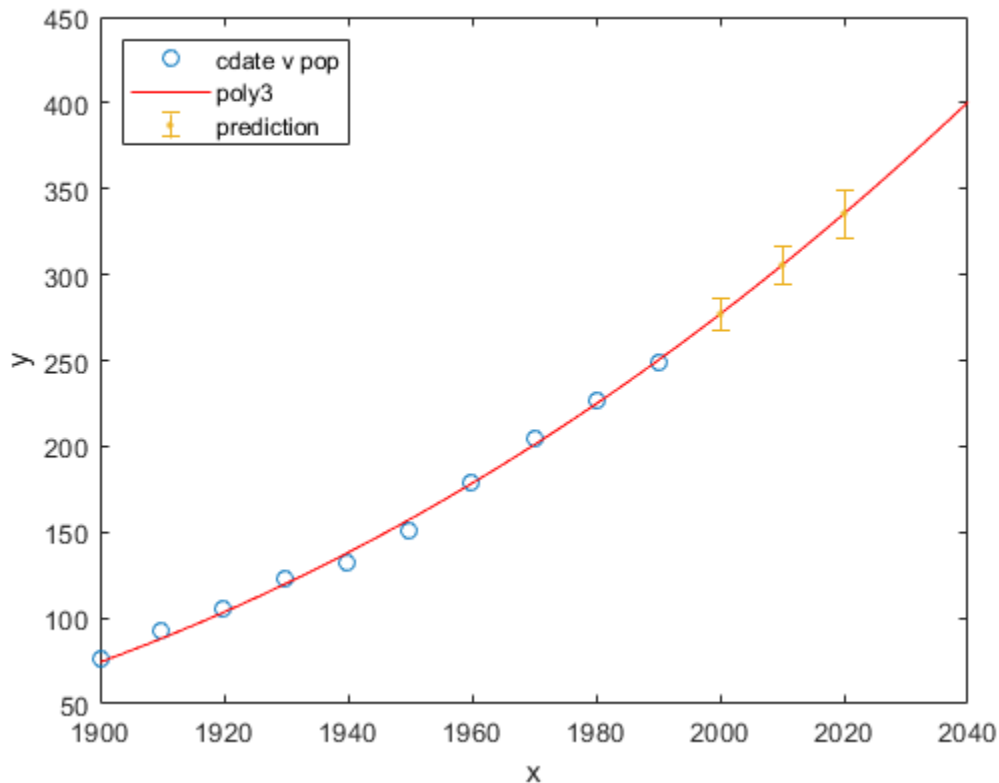

Fit and Plot a Polynomial Surface

Load some surface data and fit a fourth-degree polynomial in x and y.

```
load franke;
fitsurface=fit([x,y],z, 'poly44','Normalize','on')
plot(fitsurface, [x,y],z)
```

```
Linear model Poly44:
fitsurface(x,y) = p00 + p10*x + p01*y + p20*x^2 + p11*x*y + p02*y^2 + p30*x^3
               + p21*x^2*y + p12*x*y^2 + p03*y^3 + p40*x^4 + p31*x^3*y
               + p22*x^2*y^2 + p13*x*y^3 + p04*y^4
   where x is normalized by mean 1982 and std 868.6
   and where y is normalized by mean 0.4972 and std 0.2897
Coefficients (with 95% confidence bounds):
   p00 =      0.3471  (0.3033, 0.3909)
   p10 =     -0.1502  (-0.1935, -0.107)
   p01 =     -0.4203  (-0.4637, -0.377)
   p20 =      0.2165  (0.1514, 0.2815)
```

```
p11 =        0.1717  (0.1175, 0.2259)
p02 =       0.03189  (-0.03351, 0.09729)
p30 =       0.02778  (0.00749, 0.04806)
p21 =       0.01501  (-0.002807, 0.03283)
p12 =      -0.03659  (-0.05439, -0.01879)
p03 =        0.1184  (0.09812, 0.1387)
p40 =      -0.07661  (-0.09984, -0.05338)
p31 =      -0.02487  (-0.04512, -0.004624)
p22 =     0.0007464  (-0.01948, 0.02098)
p13 =      -0.02962  (-0.04987, -0.009366)
p04 =      -0.02399  (-0.0474, -0.0005797)
```
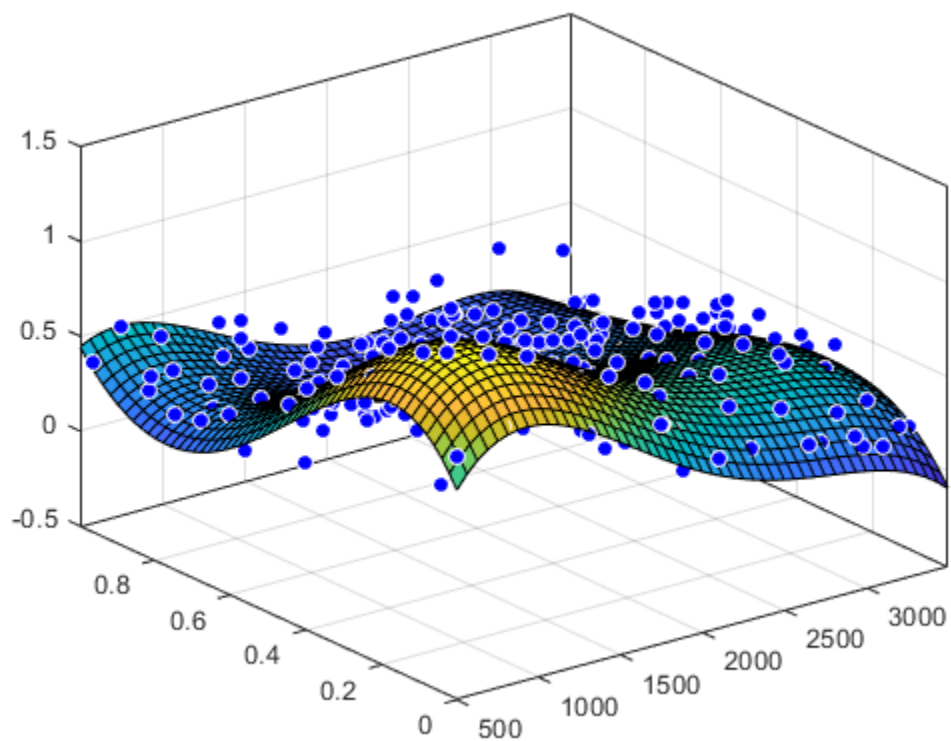

Polynomial Model Fit Options

All fitting methods have the default properties `Normalize`, `Exclude`, `Weights`, and `Method`. For an example, see "Specifying Fit Options at the Command Line" on page 4-8.

Polynomial models have the `Method` property value `LinearLeastSquares`, and the additional fit options properties shown in the next table. For details on all fit options, see the `fitoptions` reference page.

Property	Description
`Robust`	Specifies the robust linear least-squares fitting method to use. Values are `'on'`, `'off'`, `'LAR'`, or `'Bisquare'`. The default is `'off'`. `'LAR'` specifies the least absolute residual method and `'Bisquare'` specifies the bisquare weights method. `'on'` is equivalent to `'Bisquare'`, the default method.
`Lower`	A vector of lower bounds on the coefficients to be fitted. The default value is an empty vector, indicating that the fit is unconstrained by lower bounds. If bounds are specified, the vector length must equal the number of coefficients. Individual unconstrained lower bounds can be specified by `-Inf`.
`Upper`	A vector of upper bounds on the coefficients to be fitted. The default value is an empty vector, indicating that the fit is unconstrained by upper bounds. If bounds are specified, the vector length must equal the number of coefficients. Individual unconstrained upper bounds can be specified by `Inf`.

Defining Polynomial Terms for Polynomial Surface Fits

You can control the terms to include in the polynomial surface model by specifying the degrees for the x and y inputs. If i is the degree in x and j is the degree in y, the total degree of the polynomial is the maximum of i and j. The degree of x in each term is less than or equal to i, and the degree of y in each term is less than or equal to j. The maximum for both i and j is five.

For example:

```
poly21  Z = p00 + p10*x + p01*y + p20*x^2 + p11*x*y

poly13  Z = p00 + p10*x + p01*y + p11*x*y + p02*y^2
                  + p12*x*y^2 + p03*y^3

poly55  Z = p00 + p10*x + p01*y +...+ p14*x*y^4
                  + p05*y^5
```

For example, if you specify an x degree of 3 and a y degree of 2, the model name is `poly32`. The model terms follow the form in this table.

Degree of Term	0	1	2
0	1	y	y^2
1	x	xy	xy^2
2	x^2	x^2y	N/A
3	x^3	N/A	N/A

The total degree of the polynomial cannot exceed the maximum of i and j. In this example, terms such as x^3y and x^2y^2 are excluded because their degrees sum to more than 3. In both cases, the total degree is 4.

See Also

"Polynomial Model Names and Equations" on page 4-11 | `fit` | `fitoptions` | `fittype`

Related Examples

- Compare Polynomial Fits Interactively on page 2-16
- "Polynomial Curve Fitting" on page 12-2
- Polynomial Curve Fitting in MATLAB

Exponential Models

In this section...
"About Exponential Models" on page 4-28
"Fit Exponential Models Interactively" on page 4-28
"Fit Exponential Models Using the fit Function" on page 4-30

About Exponential Models

The toolbox provides a one-term and a two-term exponential model as given by

$$y = ae^{bx}$$
$$y = ae^{bx} + ce^{dx}$$

Exponentials are often used when the rate of change of a quantity is proportional to the initial amount of the quantity. If the coefficient associated with b and/or d is negative, y represents exponential decay. If the coefficient is positive, y represents exponential growth.

For example, a single radioactive decay mode of a nuclide is described by a one-term exponential. a is interpreted as the initial number of nuclei, b is the decay constant, x is time, and y is the number of remaining nuclei after a specific amount of time passes. If two decay modes exist, then you must use the two-term exponential model. For the second decay mode, you add another exponential term to the model.

Examples of exponential growth include contagious diseases for which a cure is unavailable, and biological populations whose growth is uninhibited by predation, environmental factors, and so on.

Fit Exponential Models Interactively

1. Open the Curve Fitting app by entering `cftool`. Alternatively, click Curve Fitting on the Apps tab.

2. In the Curve Fitting app, select curve data (**X data** and **Y data**, or just **Y data** against index).

   Curve Fitting app creates the default curve fit, `Polynomial`.

3. Change the model type from `Polynomial` to `Exponential`.

You can specify the following options:

- Choose one or two terms to fit `exp1` or `exp2`.

Look in the **Results** pane to see the model terms, the values of the coefficients, and the goodness-of-fit statistics.

- (Optional) Click **Fit Options** to specify coefficient starting values and constraint bounds appropriate for your data, or change algorithm settings.

The toolbox calculates optimized start points for exponential fits, based on the current data set. You can override the start points and specify your own values in the Fit Options dialog box.

The fit options for the single-term exponential are shown next. The coefficient starting values and constraints are for the census data.
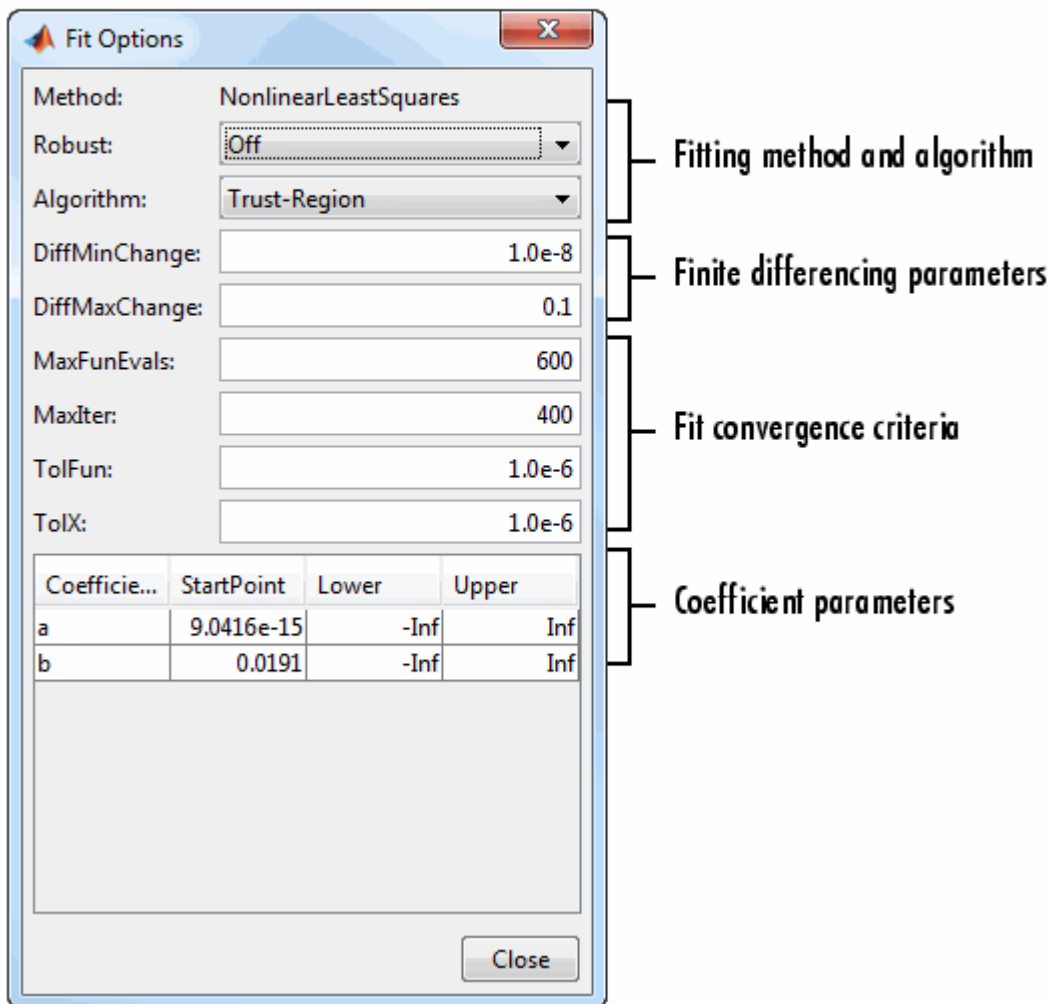
For an example specifying starting values appropriate to the data, see "Gaussian Fitting with an Exponential Background" on page 5-27.

For more information on the settings, see "Specifying Fit Options and Optimized Starting Points" on page 4-5.

Fit Exponential Models Using the fit Function

This example shows how to fit an exponential model to data using the `fit` function.

The exponential library model is an input argument to the `fit` and `fittype` functions. Specify the model type `'exp1'` or `'exp2'`.

Fit a Single-Term Exponential Model

Generate data with an exponential trend and then fit the data using a single-term exponential. Plot the fit and data.

```
x = (0:0.2:5)';
y = 2*exp(-0.2*x) + 0.1*randn(size(x));
f = fit(x,y,'exp1')
```

```
f =
     General model Exp1:
     f(x) = a*exp(b*x)
     Coefficients (with 95% confidence bounds):
       a =        2.021  (1.89, 2.151)
       b =      -0.1812  (-0.2104, -0.152)
```

```
plot(f,x,y)
```
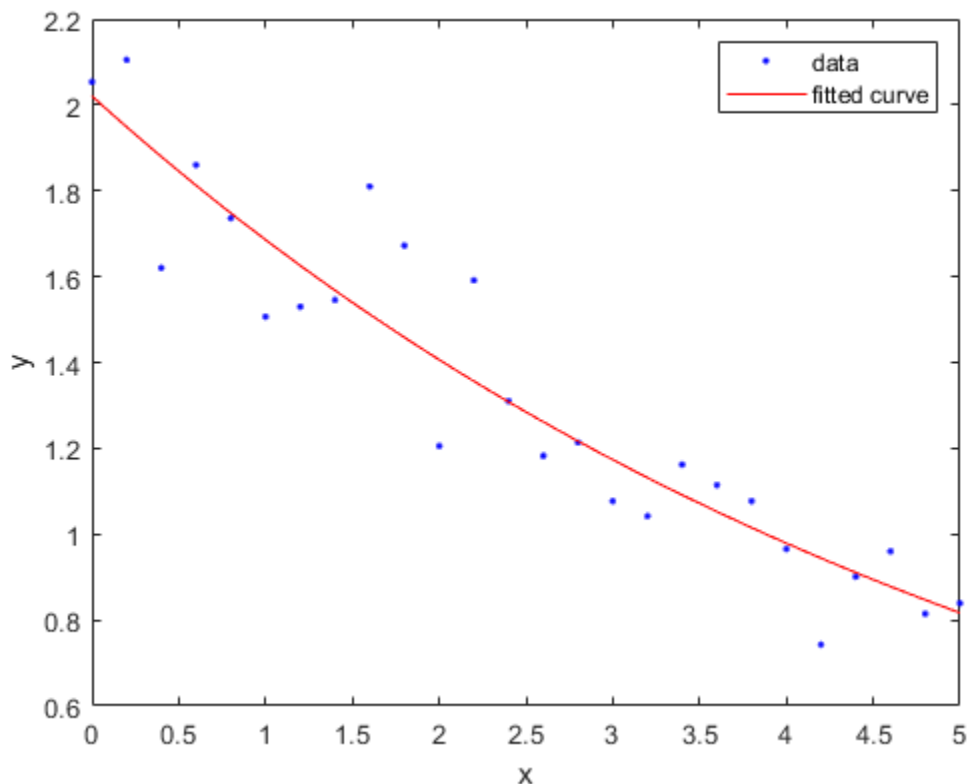
Fit a Two-Term Exponential Model

```
f2 = fit(x,y,'exp2')

f2 =
     General model Exp2:
     f2(x) = a*exp(b*x) + c*exp(d*x)
     Coefficients (with 95% confidence bounds):
       a =        537.7   (-1.307e+10, 1.307e+10)
       b =      -0.2573   (-4112, 4112)
       c =       -535.7   (-1.307e+10, 1.307e+10)
       d =      -0.2576   (-4131, 4130)
```

```
plot(f2,x,y)
```
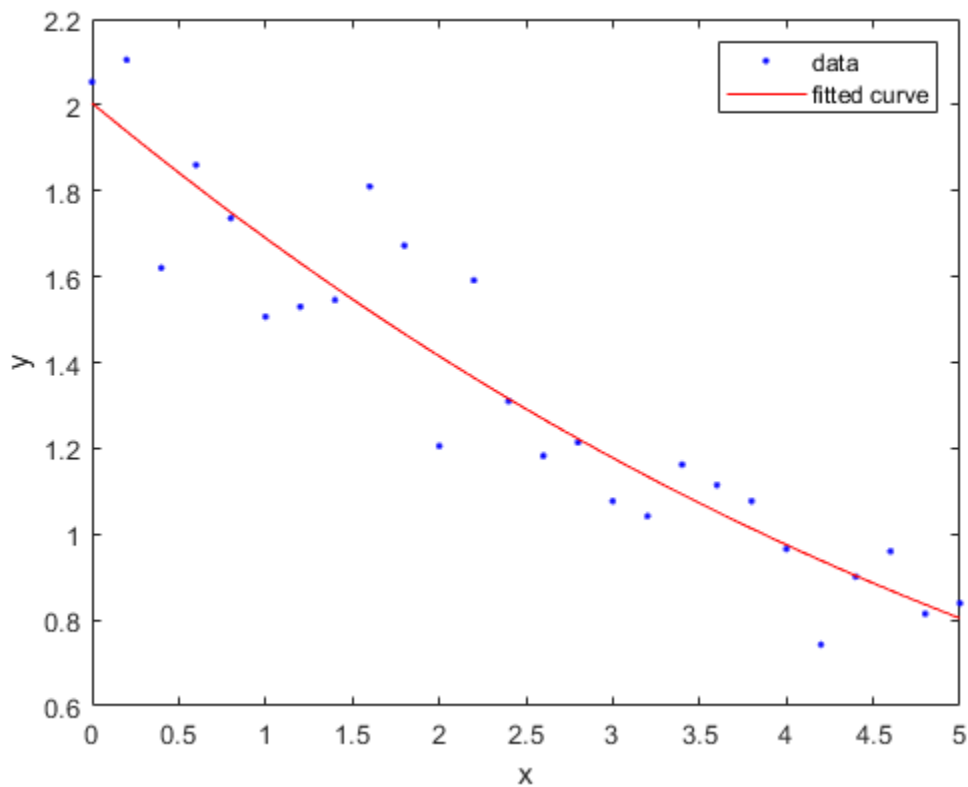

Set Start Points

The toolbox calculates optimized start points for exponential fits based on the current data set. You can override the start points and specify your own values.

Find the order of the entries for coefficients in the first model (f) by using the `coeffnames` function.

```
coeffnames(f)
```

```
ans = 2x1 cell
    {'a'}
    {'b'}
```

If you specify start points, choose values appropriate to your data. Set arbitrary start points for coefficients a and b for example purposes.

```
f = fit(x,y,'exp1','StartPoint',[1,2])

f =
     General model Exp1:
     f(x) = a*exp(b*x)
     Coefficients (with 95% confidence bounds):
       a =       2.021  (1.89, 2.151)
       b =      -0.1812  (-0.2104, -0.152)
```

```
plot(f,x,y)
```
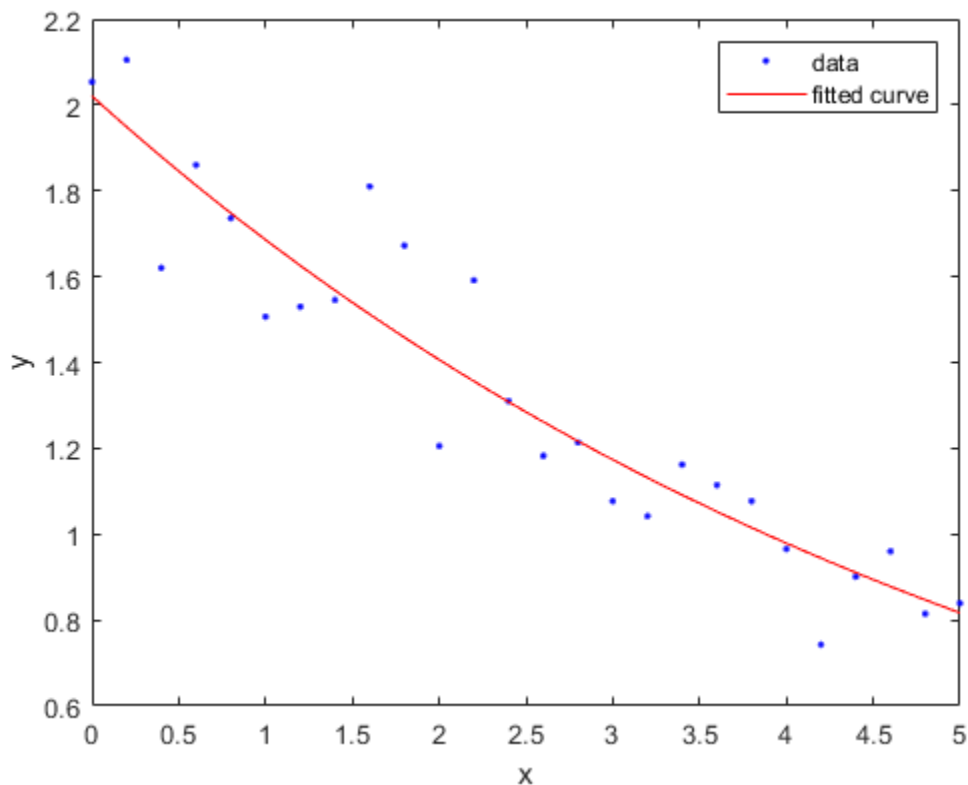

Examine Exponential Fit Options

Examine the fit options if you want to modify fit options such as coefficient starting values and constraint bounds appropriate for your data, or change algorithm settings. For details on these options, see the table of properties for NonlinearLeastSquares on the `fitoptions` reference page.

```
fitoptions('exp1')

ans =

        Normalize: 'off'
          Exclude: []
          Weights: []
           Method: 'NonlinearLeastSquares'
```

```
        Robust: 'Off'
    StartPoint: [1x0 double]
         Lower: [1x0 double]
         Upper: [1x0 double]
     Algorithm: 'Trust-Region'
 DiffMinChange: 1.0000e-08
 DiffMaxChange: 0.1000
       Display: 'Notify'
    MaxFunEvals: 600
       MaxIter: 400
        TolFun: 1.0000e-06
          TolX: 1.0000e-06
```

See Also
fit | fitoptions | fittype

Related Examples

- "Gaussian Fitting with an Exponential Background" on page 5-27
- "Specifying Fit Options and Optimized Starting Points" on page 4-5

Fourier Series

In this section...
"About Fourier Series Models" on page 4-34
"Fit Fourier Models Interactively" on page 4-34
"Fit Fourier Models Using the fit Function" on page 4-35

About Fourier Series Models

The Fourier series is a sum of sine and cosine functions that describes a periodic signal. It is represented in either the trigonometric form or the exponential form. The toolbox provides this trigonometric Fourier series form

$$y = a_0 + \sum_{i=1}^{n} a_i\cos(iwx) + b_i\sin(iwx)$$

where a_0 models a constant (intercept) term in the data and is associated with the $i = 0$ cosine term, w is the fundamental frequency of the signal, n is the number of terms (harmonics) in the series, and $1 \le n \le 8$.

For more information about the Fourier series, refer to "Fourier Analysis and Filtering".

Fit Fourier Models Interactively

1 Open the Curve Fitting app by entering `cftool`. Alternatively, click Curve Fitting on the Apps tab.

2 In the Curve Fitting app, select curve data (**X data** and **Y data**, or just **Y data** against index).

Curve Fitting app creates the default curve fit, `Polynomial`.

3 Change the model type from `Polynomial` to `Fourier`.
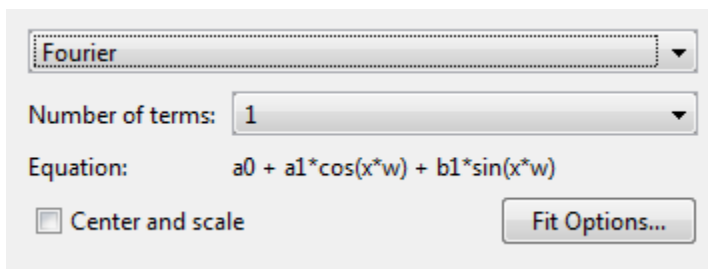
You can specify the following options:

- Choose the number of terms: `1` to `8`.

  Look in the **Results** pane to see the model terms, the values of the coefficients, and the goodness-of-fit statistics.

- (Optional) Click **Fit Options** to specify coefficient starting values and constraint bounds, or change algorithm settings.

The toolbox calculates optimized start points for Fourier series models, based on the current data set. You can override the start points and specify your own values in the Fit Options dialog box.

For more information on the settings, see "Specifying Fit Options and Optimized Starting Points" on page 4-5.

For an example comparing the library Fourier fit with custom equations, see "Custom Nonlinear ENSO Data Analysis" on page 5-19.

Fit Fourier Models Using the fit Function

This example shows how to use the `fit` function to fit a Fourier model to data.

The Fourier library model is an input argument to the `fit` and `fittype` functions. Specify the model type `fourier` followed by the number of terms, e.g., `'fourier1'` to `'fourier8'` .

This example fits the El Nino-Southern Oscillation (ENSO) data. The ENSO data consists of monthly averaged atmospheric pressure differences between Easter Island and Darwin, Australia. This difference drives the trade winds in the southern hemisphere.

The ENSO data is clearly periodic, which suggests it can be described by a Fourier series. Use Fourier series models to look for periodicity.

Fit a Two-Term Fourier Model

Load some data and fit an two-term Fourier model.

```
load enso;
f = fit(month,pressure,'fourier2')

f =
     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =        10.63  (10.23, 11.03)
       a1 =        2.923  (2.27, 3.576)
       b1 =        1.059  (0.01593, 2.101)
       a2 =      -0.5052  (-1.086, 0.07532)
       b2 =       0.2187  (-0.4202, 0.8576)
       w =        0.5258  (0.5222, 0.5294)

plot(f,month,pressure)
```
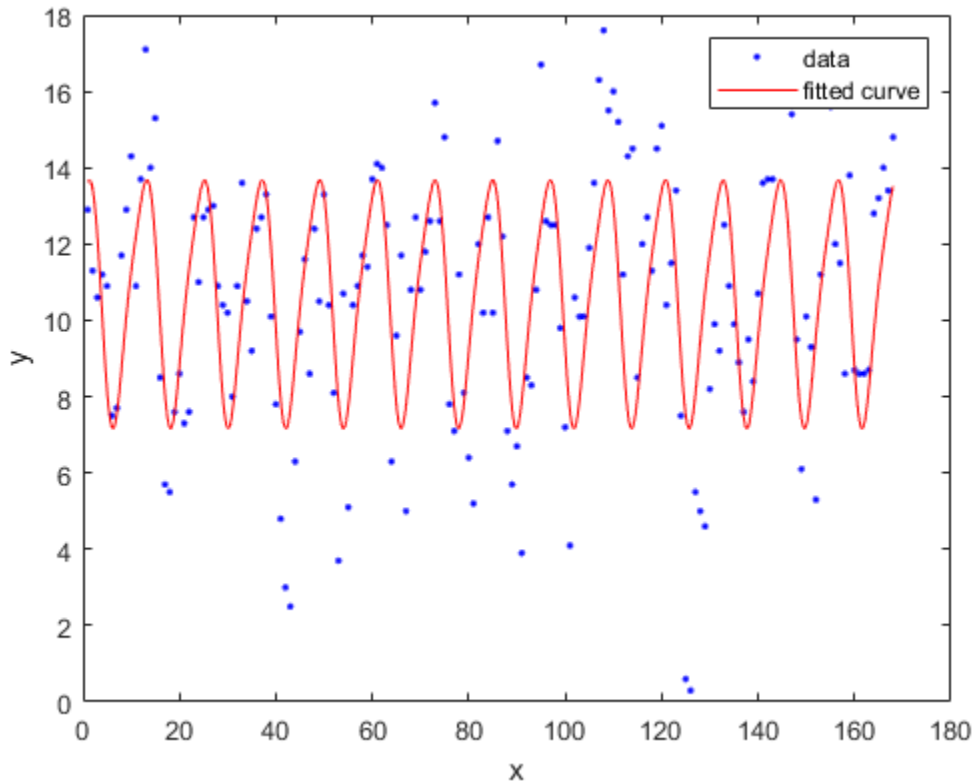
The confidence bounds on `a2` and `b2` cross zero. For linear terms, you cannot be sure that these coefficients differ from zero, so they are not helping with the fit. This means that this two term model is probably no better than a one term model.

Measure Period

The `w` term is a measure of period. `2*pi/w` converts to the period in months, because the period of `sin()` and `cos()` is `2*pi`.

```
w = f.w
```

```
w = 0.5258
```

```
2*pi/w
```

```
ans = 11.9497
```

`w` is very close to 12 months, indicating a yearly period. Observe this looks correct on the plot, with peaks approximately 12 months apart.

Fit an Eight-Term Fourier Model

```
f2 = fit(month,pressure,'fourier8')
```

```
f2 =
     General model Fourier8:
     f2(x) =
              a0 + a1*cos(x*w) + b1*sin(x*w) +
```

```
            a2*cos(2*x*w) + b2*sin(2*x*w) + a3*cos(3*x*w) + b3*sin(3*x*w) +
            a4*cos(4*x*w) + b4*sin(4*x*w) + a5*cos(5*x*w) + b5*sin(5*x*w) +
            a6*cos(6*x*w) + b6*sin(6*x*w) + a7*cos(7*x*w) + b7*sin(7*x*w) +
            a8*cos(8*x*w) + b8*sin(8*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =        10.63  (10.28, 10.97)
       a1 =       0.5668  (0.07981, 1.054)
       b1 =       0.1969  (-0.2929, 0.6867)
       a2 =       -1.203  (-1.69, -0.7161)
       b2 =      -0.8087  (-1.311, -0.3065)
       a3 =       0.9321  (0.4277, 1.436)
       b3 =       0.7602  (0.2587, 1.262)
       a4 =      -0.6653  (-1.152, -0.1788)
       b4 =      -0.2038  (-0.703, 0.2954)
       a5 =     -0.02919  (-0.5158, 0.4575)
       b5 =      -0.3701  (-0.8594, 0.1192)
       a6 =     -0.04856  (-0.5482, 0.4511)
       b6 =      -0.1368  (-0.6317, 0.3581)
       a7 =        2.811  (2.174, 3.449)
       b7 =        1.334  (0.3686, 2.3)
       a8 =      0.07979  (-0.4329, 0.5925)
       b8 =      -0.1076  (-0.6037, 0.3885)
       w =      0.07527  (0.07476, 0.07578)
```

```
plot(f2,month,pressure)
```
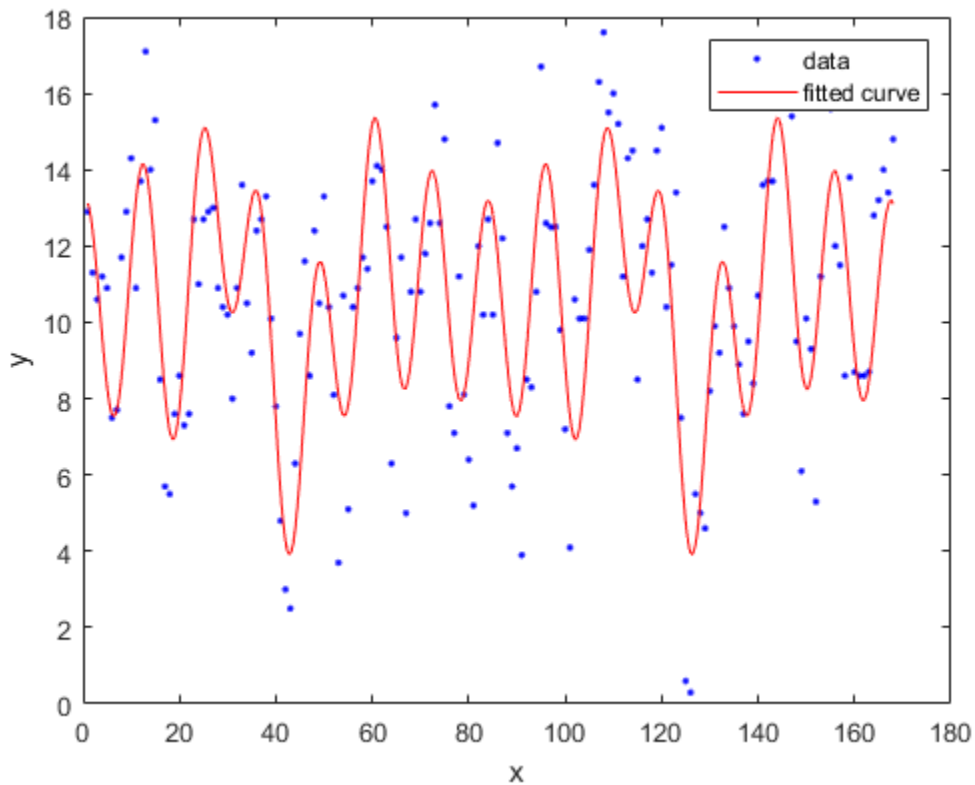
Measure Period

```
w = f2.w
```

```
w = 0.0753
```

```
(2*pi)/w
```

```
ans = 83.4736
```

With the `f2` model, the period `w` is approximately 7 years.

Examine Terms

Look for the coefficients with the largest magnitude to find the most important terms.

- `a7` and `b7` are the largest. Look at the `a7` term in the model equation: `a7*cos(7*x*w)`. `7*w ==` 7/7 = 1 year cycle. `a7` and `b7` indicate the annual cycle is the strongest.
- Similarly, `a1` and `b1` terms give 7/1, indicating a seven year cycle.
- `a2` and `b2` terms are a 3.5 year cycle (7/2). This is stronger than the 7 year cycle because the `a2` and `b2` coefficients have larger magnitude than a1 and b1.
- `a3` and `b3` are quite strong terms indicating a 7/3 or 2.3 year cycle.
- Smaller terms are less important for the fit, such as `a6`, `b6`, `a5`, and `b5`.

Typically, the El Nino warming happens at irregular intervals of two to seven years, and lasts nine months to two years. The average period length is five years. The model results reflect some of these periods.

Set Start Points

The toolbox calculates optimized start points for Fourier fits, based on the current data set. Fourier series models are particularly sensitive to starting points, and the optimized values might be accurate for only a few terms in the associated equations. You can override the start points and specify your own values.

After examining the terms and plots, it looks like a 4 year cycle might be present. Try to confirm this by setting `w`. Get a value for `w`, where 8 years = 96 months.

```
w = (2*pi)/96
```

```
w = 0.0654
```

Find the order of the entries for coefficients in the model ('f2') by using the `coeffnames` function.

```
coeffnames(f2)
```

```
ans = 18x1 cell
    {'a0'}
    {'a1'}
    {'b1'}
    {'a2'}
    {'b2'}
    {'a3'}
    {'b3'}
    {'a4'}
    {'b4'}
```

```
{'a5'}
{'b5'}
{'a6'}
{'b6'}
{'a7'}
{'b7'}
{'a8'}
{'b8'}
{'w' }
```

Get the current coefficient values.

```
coeffs = coeffvalues(f2)
```

coeffs = 1×18

```
   10.6261    0.5668    0.1969   -1.2031   -0.8087    0.9321    0.7602   -0.6653   -0.2038   -0.0
```

Set the last coefficient, w, to 0.065.

```
coeffs(:,18) = w
```

coeffs = 1×18

```
   10.6261    0.5668    0.1969   -1.2031   -0.8087    0.9321    0.7602   -0.6653   -0.2038   -0.0
```

Set the start points for coefficients using the new value for w.

```
f3 = fit(month,pressure,'fourier8', 'StartPoint', coeffs);
```

Plot both fits to see that the new value for w in f3 does not produce a better fit than f2 .

```
plot(f3,month,pressure)
hold on
plot(f2, 'b')
hold off
legend( 'Data', 'f3', 'f2')
```
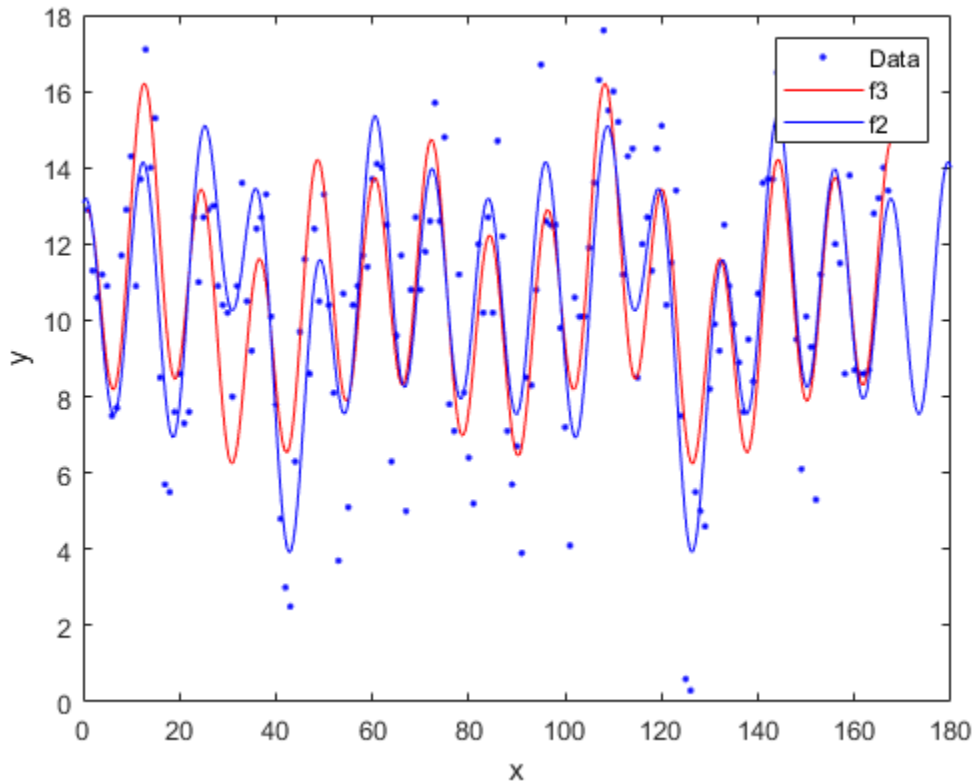
Find Fourier Fit Options

Find available fit options using `fitoptions(modelname)`, where `modelname` is the model type `fourier` followed by the number of terms, e.g., `'fourier1'` to `'fourier8'`.

```
fitoptions('fourier8')
```

```
ans =

        Normalize: 'off'
          Exclude: []
          Weights: []
           Method: 'NonlinearLeastSquares'
           Robust: 'Off'
       StartPoint: [1x0 double]
            Lower: [1x0 double]
            Upper: [1x0 double]
        Algorithm: 'Trust-Region'
    DiffMinChange: 1.0000e-08
    DiffMaxChange: 0.1000
          Display: 'Notify'
       MaxFunEvals: 600
          MaxIter: 400
           TolFun: 1.0000e-06
             TolX: 1.0000e-06
```

If you want to modify fit options such as coefficient starting values and constraint bounds appropriate for your data, or change algorithm settings, see the options for NonlinearLeastSquares on the `fitoptions` reference page.

See Also
`fit` | `fitoptions` | `fittype`

Related Examples
- "Custom Nonlinear ENSO Data Analysis" on page 5-19
- "Specifying Fit Options and Optimized Starting Points" on page 4-5

Gaussian Models

In this section...

About Gaussian Models

The Gaussian model fits peaks, and is given by

$$y = \sum_{i=1}^{n} a_i e^{\left[-\left(\frac{x-b_i}{c_i}\right)^2\right]}$$

where a is the amplitude, b is the centroid (location), c is related to the peak width, n is the number of peaks to fit, and $1 \le n \le 8$.

Gaussian peaks are encountered in many areas of science and engineering. For example, Gaussian peaks can describe line emission spectra and chemical concentration assays.

Fit Gaussian Models Interactively

1   Open the Curve Fitting app by entering `cftool`. Alternatively, click Curve Fitting on the Apps tab.

2   In the Curve Fitting app, select curve data (**X data** and **Y data**, or just **Y data** against index).

   Curve Fitting app creates the default curve fit, `Polynomial`.

3   Change the model type from `Polynomial` to `Gaussian`.
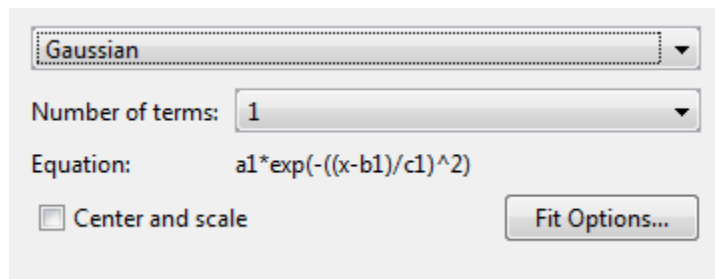
You can specify the following options:

* Choose the number of terms: 1 to 8.

   Look in the **Results** pane to see the model terms, the values of the coefficients, and the goodness-of-fit statistics.

* (Optional) Click **Fit Options** to specify coefficient starting values and constraint bounds, or change algorithm settings.

   The toolbox calculates optimized start points for Gaussian models, based on the current data set. You can override the start points and specify your own values in the Fit Options dialog box.

Gaussians have the width parameter c_1 constrained with a lower bound of 0. The default lower bounds for most library models are $-Inf$, which indicates that the coefficients are unconstrained.

For more information on the settings, see "Specifying Fit Options and Optimized Starting Points" on page 4-5.

Fit Gaussian Models Using the fit Function

This example shows how to use the `fit` function to fit a Gaussian model to data.

The Gaussian library model is an input argument to the `fit` and `fittype` functions. Specify the model type `gauss` followed by the number of terms, e.g., `'gauss1'` through `'gauss8'`.

Fit a Two-Term Gaussian Model

Load some data and fit a two-term Gaussian model.

```
[x,y] = titanium;
f = fit(x.',y.','gauss2')

f =
     General model Gauss2:
     f(x) =  a1*exp(-((x-b1)/c1)^2) + a2*exp(-((x-b2)/c2)^2)
     Coefficients (with 95% confidence bounds):
       a1 =         1.47  (1.426, 1.515)
       b1 =        897.7  (897, 898.3)
       c1 =        27.08  (26.08, 28.08)
       a2 =       0.6994  (0.6821, 0.7167)
       b2 =        810.8  (790, 831.7)
       c2 =        592.9  (500.1, 685.7)

plot(f,x,y)
```
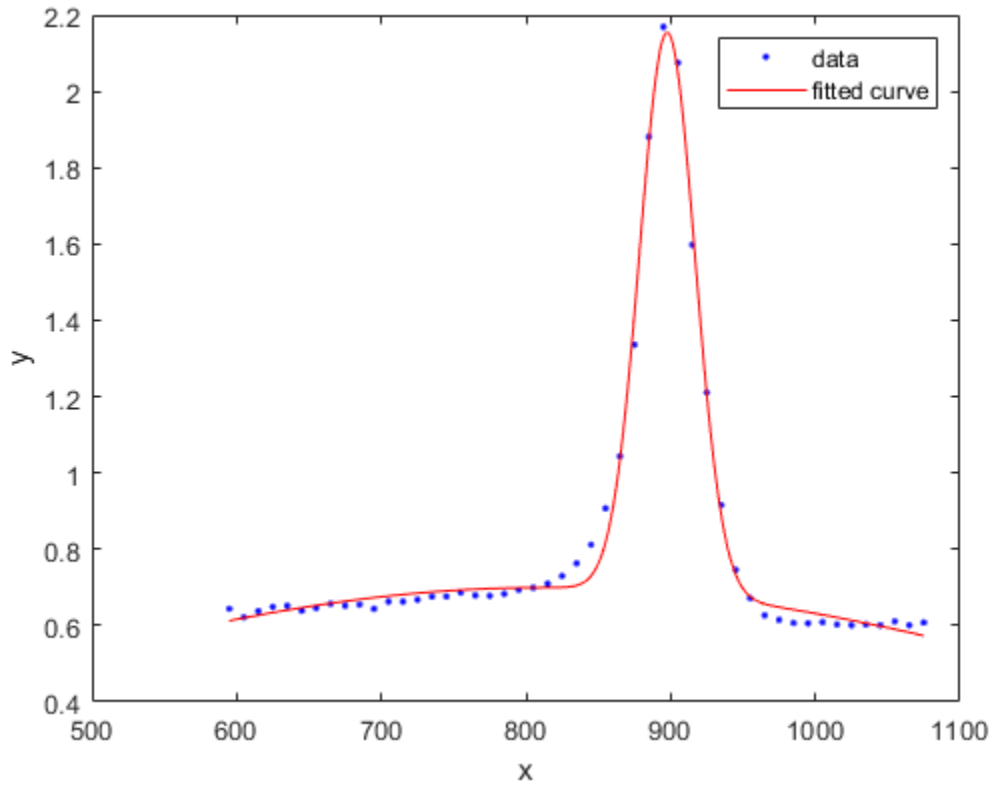
See Also
`fit` | `fitoptions` | `fittype`

Related Examples

- "Gaussian Fitting with an Exponential Background" on page 5-27
- "Specifying Fit Options and Optimized Starting Points" on page 4-5

Power Series

In this section...

About Power Series Models

The toolbox provides a one-term and a two-term power series model as given by

$$y = ax^b$$
$$y = ax^b + c$$

Power series models describe a variety of data. For example, the rate at which reactants are consumed in a chemical reaction is generally proportional to the concentration of the reactant raised to some power.

Fit Power Series Models Interactively

1   Open the Curve Fitting app by entering `cftool`. Alternatively, click Curve Fitting on the Apps tab.

2   In the Curve Fitting app, select curve data (**X data** and **Y data**, or just **Y data** against index).

   Curve Fitting app creates the default curve fit, `Polynomial`.

3   Change the model type from `Polynomial` to `Power`.
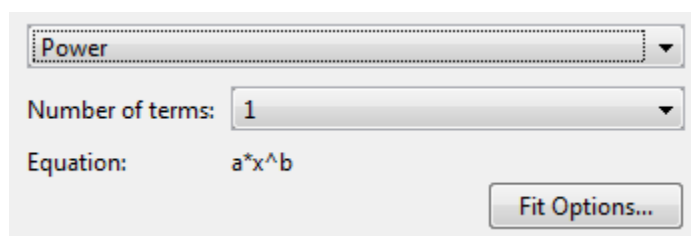
You can specify the following options:

• Choose the number of terms: 1 to 2.

   Look in the **Results** pane to see the model terms, the values of the coefficients, and the goodness-of-fit statistics.

• (Optional) Click **Fit Options** to specify coefficient starting values and constraint bounds, or change algorithm settings.

   The toolbox calculates optimized start points for power series models, based on the current data set. You can override the start points and specify your own values in the Fit Options dialog box.

   For more information on the settings, see "Specifying Fit Options and Optimized Starting Points" on page 4-5.

Fit Power Series Models Using the fit Function

This example shows how to use the `fit` function to fit power series models to data.

The power series library model is an input argument to the `fit` and `fittype` functions. Specify the model type `'power1'` or `'power2'`.

Fit a Single-Term Power Series Model

```
load hahn1;
f = fit(temp,thermex,'power1')

f =
     General model Power1:
     f(x) = a*x^b
     Coefficients (with 95% confidence bounds):
       a =         1.46  (1.224, 1.695)
       b =       0.4094  (0.3825, 0.4363)

plot(f,temp,thermex)
```
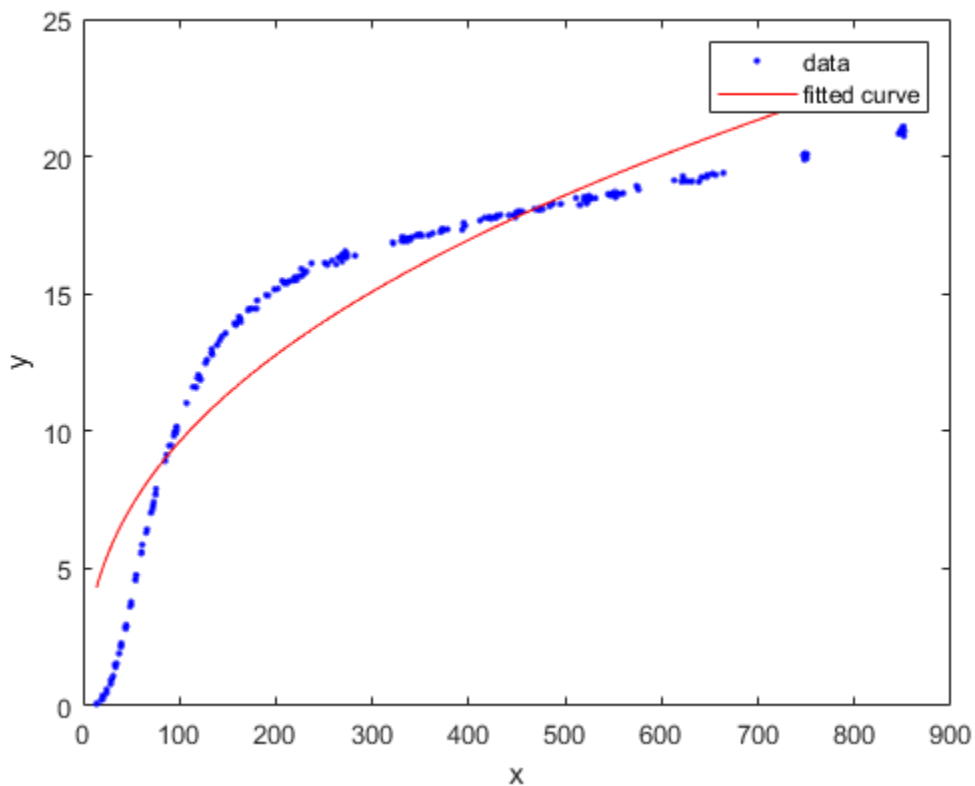

Fit a Two-Term Power Series Model

```
f = fit(temp,thermex,'power2')

f =
     General model Power2:
     f(x) = a*x^b+c
```

```
Coefficients (with 95% confidence bounds):
  a =       -78.61  (-80.74, -76.48)
  b =      -0.2349  (-0.271, -0.1989)
  c =         36.9  (33.09, 40.71)
```

plot(f,temp,thermex)
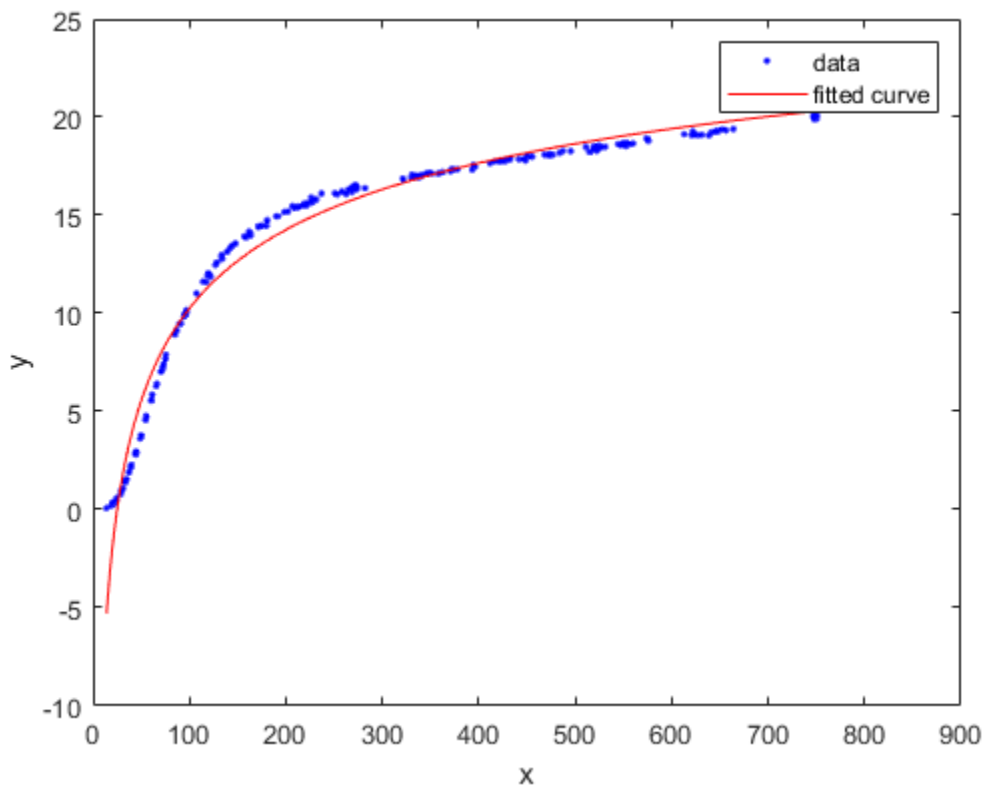
See Also
fit | fitoptions | fittype

Related Examples
- "Specifying Fit Options and Optimized Starting Points" on page 4-5

Rational Polynomials

In this section...
"About Rational Models" on page 4-48
"Fit Rational Models Interactively" on page 4-48
"Selecting a Rational Fit at the Command Line" on page 4-49
"Example: Rational Fit" on page 4-49

About Rational Models

Rational models are defined as ratios of polynomials and are given by

$$y = \frac{\sum\limits_{i=1}^{n+1} p_i x^{n+1-i}}{x^m + \sum\limits_{i=1}^{m} q_i x^{m-1}}$$

where n is the degree of the numerator polynomial and $0 \le n \le 5$, while m is the degree of the denominator polynomial and $1 \le m \le 5$. Note that the coefficient associated with x^m is always 1. This makes the numerator and denominator unique when the polynomial degrees are the same.

In this guide, rationals are described in terms of the degree of the numerator/the degree of the denominator. For example, a quadratic/cubic rational equation is given by

$$y = \frac{p_1 x^2 + p_2 x + p_3}{x^3 + q_1 x^2 + q_2 x + q_3}$$

Like polynomials, rationals are often used when a simple empirical model is required. The main advantage of rationals is their flexibility with data that has a complicated structure. The main disadvantage is that they become unstable when the denominator is around 0. For an example that uses rational polynomials of various degrees, see "Example: Rational Fit" on page 4-49.

Fit Rational Models Interactively

1   Open the Curve Fitting app by entering `cftool`. Alternatively, click Curve Fitting on the Apps tab.

2   In the Curve Fitting app, select curve data (**X data** and **Y data**, or just **Y data** against index).

    Curve Fitting app creates the default curve fit, `Polynomial`.

3   Change the model type from `Polynomial` to `Rational`.

You can specify the following options:

- Choose the degree of the numerator and denominator polynomials. The numerator can have degree 0 to 5, and the denominator from 1 to 5.

  Look in the **Results** pane to see the model terms, the values of the coefficients, and the goodness-of-fit statistics.

- (Optional) Click **Fit Options** to specify coefficient starting values and constraint bounds, or change algorithm settings.

  The toolbox calculates random start points for rational models, defined on the interval [0,1]. You can override the start points and specify your own values in the Fit Options dialog box.

  For more information on the settings, see "Specifying Fit Options and Optimized Starting Points" on page 4-5.

Selecting a Rational Fit at the Command Line

Specify the model type `ratij`, where *i* is the degree of the numerator polynomial and *j* is the degree of the denominator polynomial. For example, `'rat02'`, `'rat21'` or `'rat55'`.

For example, to load some data and fit a rational model:

```
load hahn1;
f = fit( temp, thermex, 'rat32')
plot(f,temp,thermex)
```

See "Example: Rational Fit" on page 4-49 to fit this example interactively with various rational models.

If you want to modify fit options such as coefficient starting values and constraint bounds appropriate for your data, or change algorithm settings, see the table of additional properties with `NonlinearLeastSquares` on the `fitoptions` reference page.

Example: Rational Fit

This example fits measured data using a rational model. The data describes the coefficient of thermal expansion for copper as a function of temperature in degrees kelvin.

For this data set, you will find the rational equation that produces the best fit. Rational models are defined as a ratio of polynomials as given by:

$$y = \frac{p_1 x^n + p_2 x^{n-1} + ... + p_{n+1}}{x^m + q_1 x^{m-1} + ... + q_m}$$

where *n* is the degree of the numerator polynomial and *m* is the degree of the denominator polynomial. Note that the rational equations are not associated with physical parameters of the data. Instead, they provide a simple and flexible empirical model that you can use for interpolation and extrapolation.

1 Load the thermal expansion data from the file `hahn1.mat`, which is provided with the toolbox.

```
load hahn1
```

The workspace contains two new variables:

- `temp` is a vector of temperatures in degrees kelvin.
- `thermex` is a vector of thermal expansion coefficients for copper.

2   Open the Curve Fitting app by entering:

`cftool`

3   Select `temp` and `thermex` from the **X data** and **Y data** lists.

The Curve Fitting app fits and plots the data.

4   Select `Rational` in the fit category list.

5   Try an initial choice for the rational model of quadratic/quadratic. Select 2 for both **Numerator degree** and **Denominator degree**.

Rational	▼
Numerator degree:	2 ▼
Denominator degree:	2 ▼
☐ Center and scale	Fit Options...
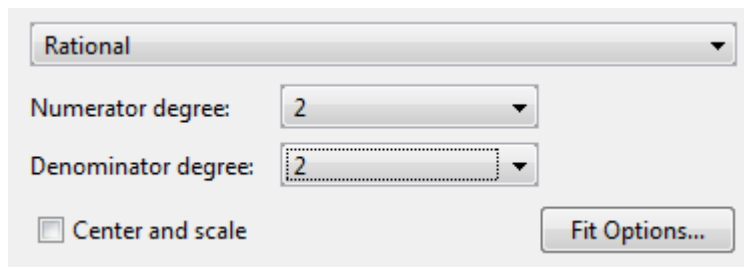
The Curve Fitting app fits a quadratic/quadratic rational.

6   Examine the residuals. Select **View > Residuals Plot** or click the toolbar button.

Examine the data, fit, and residuals. Observe that the fit misses the data for the smallest and largest predictor values. Additionally, the residuals show a strong pattern throughout the entire data set, indicating that a better fit is possible.
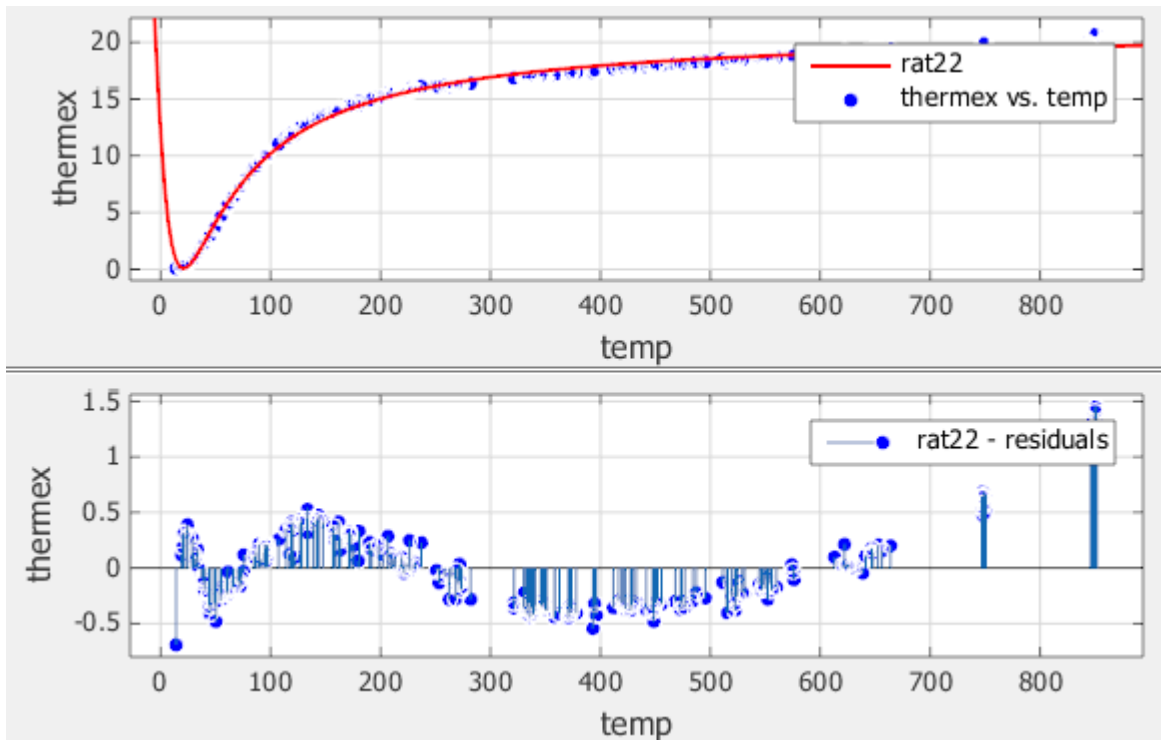
7   For the next fit, try a cubic/cubic equation. Select 3 for both **Numerator degree** and **Denominator degree**.

Examine the data, fit, and residuals. The fit exhibits several discontinuities around the zeros of the denominator.

Note Your results depend on random start points and may vary from those shown.
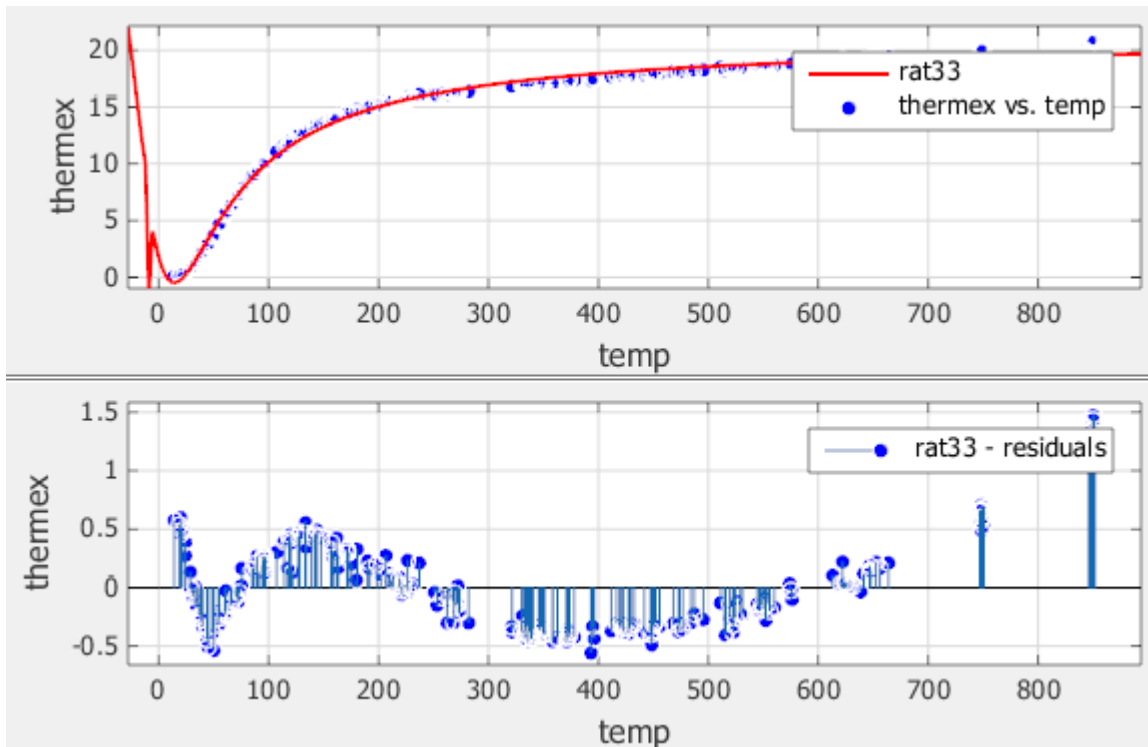
8   Look in the **Results** pane. The message and numerical results indicate that the fit did not converge.

```
Fit computation did not converge:
Fitting stopped because the number of iterations
 or function evaluations exceeded the specified maximum.
```

Although the message in the **Results** pane indicates that you might improve the fit if you increase the maximum number of iterations, a better choice at this stage of the fitting process is to use a different rational equation because the current fit contains several discontinuities. These discontinuities are due to the function blowing up at predictor values that correspond to the zeros of the denominator.

9   Try fitting the data using a cubic/quadratic equation. Select 2 for the **Denominator degree** and leave the **Numerator degree** set to 3.

10 The input variables have very different scales, so select the **Center and scale** option.

The data, fit, and residuals are shown below.
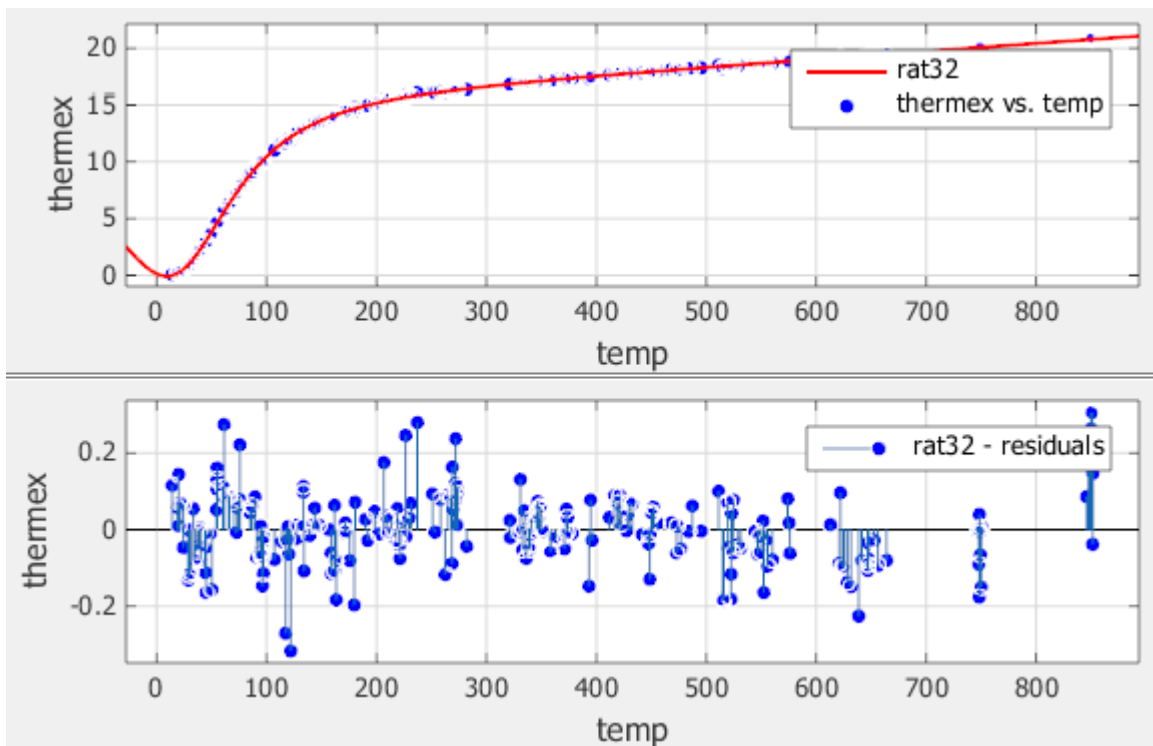
The fit is well behaved over the entire data range, and the residuals are randomly scattered about zero. Therefore, you can confidently use this fit for further analysis.

See Also

`fit` | `fitoptions` | `fittype`

Related Examples

- "Specifying Fit Options and Optimized Starting Points" on page 4-5

Sum of Sines Models

In this section...

About Sum of Sines Models

The sum of sines model fits periodic functions, and is given by

$$y = \sum_{i=1}^{n} a_i \sin(b_i x + c_i)$$

where a is the amplitude, b is the frequency, and c is the phase constant for each sine wave term. n is the number of terms in the series and $1 \le n \le 8$. This equation is closely related to the Fourier series described in "Fourier Series" on page 4-34. The main difference is that the sum of sines equation includes the phase constant, and does not include a constant (intercept) term.

Fit Sum of Sine Models Interactively

1 Open the Curve Fitting app by entering `cftool`. Alternatively, click Curve Fitting on the Apps tab.

2 In the Curve Fitting app, select curve data (**X data** and **Y data**, or just **Y data** against index).

   Curve Fitting app creates the default curve fit, `Polynomial`.

3 Change the model type from `Polynomial` to `Sum of Sine`.
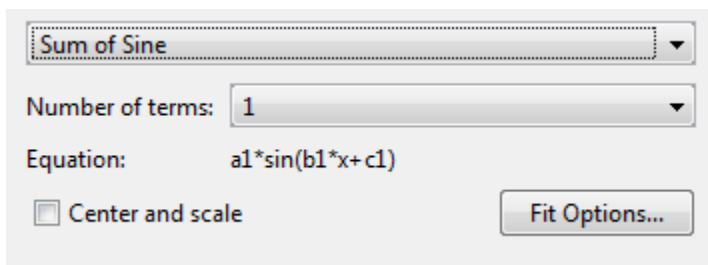
You can specify the following options:

• Choose the number of terms: 1 to 8.

   Look in the **Results** pane to see the model terms, the values of the coefficients, and the goodness-of-fit statistics.

• (Optional) Click **Fit Options** to specify coefficient starting values and constraint bounds, or change algorithm settings.

   The toolbox calculates optimized start points for sum of sine models, based on the current data set. You can override the start points and specify your own values in the Fit Options dialog box.

The sum of sine model has a lower bound constraint on c_i of 0. The default lower bounds for most library models are `-Inf`.

For more information on the settings, see "Specifying Fit Options and Optimized Starting Points" on page 4-5.

Selecting a Sum of Sine Fit at the Command Line

Specify the model type `sin` followed by the number of terms, e.g., `'sin1'` to `'sin8'`.

For example, to load some periodic data and fit a six-term sum of sine model:

```
load enso;
f = fit( month, pressure, 'sin6')
plot(f,month,pressure)
```

If you want to modify fit options such as coefficient starting values and constraint bounds appropriate for your data, or change algorithm settings, see the table of additional properties with `NonlinearLeastSquares` on the `fitoptions` reference page.

See Also
`fit` | `fitoptions` | `fittype`

Related Examples
- "Specifying Fit Options and Optimized Starting Points" on page 4-5

Weibull Distributions

In this section...
"About Weibull Distribution Models" on page 4-56
"Fit Weibull Models Interactively" on page 4-56
"Selecting a Weibull Fit at the Command Line" on page 4-57

About Weibull Distribution Models

The Weibull distribution is widely used in reliability and life (failure rate) data analysis. The toolbox provides the two-parameter Weibull distribution

$$y = abx^{b-1}e^{-ax^b}$$

where a is the scale parameter and b is the shape parameter.

Note that there are other Weibull distributions but you must create a custom equation to use these distributions:

- A three-parameter Weibull distribution with x replaced by $x - c$ where c is the location parameter
- A one-parameter Weibull distribution where the shape parameter is fixed and only the scale parameter is fitted.

Curve Fitting Toolbox does not fit Weibull probability distributions to a sample of data. Instead, it fits curves to response and predictor data such that the curve has the same shape as a Weibull distribution.

Fit Weibull Models Interactively

1   Open the Curve Fitting app by entering `cftool`. Alternatively, click Curve Fitting on the Apps tab.

2   In the Curve Fitting app, select curve data (**X data** and **Y data**, or just **Y data** against index).

    Curve Fitting app creates the default curve fit, `Polynomial`.

3   Change the model type from `Polynomial` to `Weibull`.
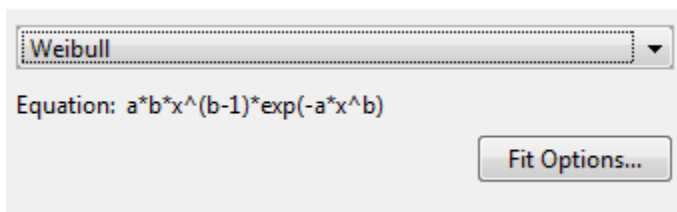
There are no fit settings to configure.

(Optional) Click **Fit Options** to specify coefficient starting values and constraint bounds, or change algorithm settings.

The toolbox calculates random start points for Weibull models, defined on the interval [0,1]. You can override the start points and specify your own values in the Fit Options dialog box.

For more information on the settings, see "Specifying Fit Options and Optimized Starting Points" on page 4-5.

Selecting a Weibull Fit at the Command Line

Specify the model type `weibull`.

For example, to load some example data measuring blood concentration of a compound against time, and fit and plot a Weibull model specifying a start point:

```
time = [ 0.1;   0.1;   0.3;   0.3;   1.3;   1.7;   2.1;...
   2.6;   3.9;   3.9; ...
         5.1;   5.6;   6.2;   6.4;   7.7;   8.1;   8.2;...
   8.9;   9.0;   9.5; ...
         9.6;  10.2;  10.3;  10.8;  11.2;  11.2;  11.2;...
  11.7;  12.1;  12.3; ...
        12.3;  13.1;  13.2;  13.4;  13.7;  14.0;  14.3;...
  15.4;  16.1;  16.1; ...
        16.4;  16.4;  16.7;  16.7;  17.5;  17.6;  18.1;...
  18.5;  19.3;  19.7;];
conc = [0.01;  0.08;  0.13;  0.16;  0.55;  0.90;  1.11;...
  1.62;  1.79;  1.59; ...
        1.83;  1.68;  2.09;  2.17;  2.66;  2.08;  2.26;...
  1.65;  1.70;  2.39; ...
        2.08;  2.02;  1.65;  1.96;  1.91;  1.30;  1.62;...
  1.57;  1.32;  1.56; ...
        1.36;  1.05;  1.29;  1.32;  1.20;  1.10;  0.88;...
  0.63;  0.69;  0.69; ...
        0.49;  0.53;  0.42;  0.48;  0.41;  0.27;  0.36;...
  0.33;  0.17;  0.20;];

f=fit(time, conc/25, 'Weibull', ...
'StartPoint', [0.01, 2] )
plot(f,time,conc/25, 'o');
```

If you want to modify fit options such as coefficient starting values and constraint bounds appropriate for your data, or change algorithm settings, see the table of additional properties with `NonlinearLeastSquares` on the `fitoptions` reference page.

Appropriate start point values and scaling `conc/25` for the two-parameter Weibull model were calculated by fitting a 3 parameter Weibull model using this custom equation:

```
f=fit(time, conc, ' c*a*b*x^(b-1)*exp(-a*x^b)', 'StartPoint', [0.01, 2, 5] )

f =
    General model:
    f(x) = c*a*b*x^(b-1)*exp(-a*x^b)
    Coefficients (with 95% confidence bounds):
      a =    0.009854  (0.007465, 0.01224)
      b =       2.003  (1.895, 2.11)
      c =       25.65  (24.42, 26.89)
```

This Weibull model is defined with three parameters: the first scales the curve along the horizontal axis, the second defines the shape of the curve, and the third scales the curve along the vertical axis.

Notice that while this curve has almost the same form as the Weibull probability density function, it is not a density because it includes the parameter c, which is necessary to allow the curve's height to adjust to data.

See Also
`fit` | `fitoptions` | `fittype`

Related Examples
*   "Specifying Fit Options and Optimized Starting Points" on page 4-5

Least-Squares Fitting

Introduction

Curve Fitting Toolbox software uses the method of least squares when fitting data. Fitting requires a parametric model that relates the response data to the predictor data with one or more coefficients. The result of the fitting process is an estimate of the model coefficients.

To obtain the coefficient estimates, the least-squares method minimizes the summed square of residuals. The residual for the ith data point r_i is defined as the difference between the observed response value y_i and the fitted response value $\hat{y}_i$, and is identified as the error associated with the data.

$$r_i = y_i - \widehat{y}_i$$
$$\text{residual} = \text{data} - \text{fit}$$

The summed square of residuals is given by

$$S = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} \left(y_i - \widehat{y}_i \right)^2$$

where n is the number of data points included in the fit and S is the sum of squares error estimate. The supported types of least-squares fitting include:

- Linear least squares
- Weighted linear least squares
- Robust least squares
- Nonlinear least squares

Error Distributions

When fitting data that contains random variations, there are two important assumptions that are usually made about the error:

- The error exists only in the response data, and not in the predictor data.
- The errors are random and follow a normal (Gaussian) distribution with zero mean and constant variance, σ^2.

The second assumption is often expressed as

$$error \sim N\left(0, \sigma^2\right)$$

The errors are assumed to be normally distributed because the normal distribution often provides an adequate approximation to the distribution of many measured quantities. Although the least-squares fitting method does not assume normally distributed errors when calculating parameter estimates, the method works best for data that does not contain a large number of random errors with extreme values. The normal distribution is one of the probability distributions in which extreme random errors are uncommon. However, statistical results such as confidence and prediction bounds do require normally distributed errors for their validity.

If the mean of the errors is zero, then the errors are purely random. If the mean is not zero, then it might be that the model is not the right choice for your data, or the errors are not purely random and contain systematic errors.

A constant variance in the data implies that the "spread" of errors is constant. Data that has the same variance is sometimes said to be of equal quality.

The assumption that the random errors have constant variance is not implicit to weighted least-squares regression. Instead, it is assumed that the weights provided in the fitting procedure correctly indicate the differing levels of quality present in the data. The weights are then used to adjust the amount of influence each data point has on the estimates of the fitted coefficients to an appropriate level.

Linear Least Squares

Curve Fitting Toolbox software uses the linear least-squares method to fit a linear model to data. A *linear* model is defined as an equation that is linear in the coefficients. For example, polynomials are linear but Gaussians are not. To illustrate the linear least-squares fitting process, suppose you have n data points that can be modeled by a first-degree polynomial.

$$y = p_1 x + p_2$$

To solve this equation for the unknown coefficients p_1 and p_2, you write S as a system of n simultaneous linear equations in two unknowns. If n is greater than the number of unknowns, then the system of equations is *overdetermined*.

$$S = \sum_{i=1}^{n} \left(y_i - (p_1 x_i + p_2)\right)^2$$

Because the least-squares fitting process minimizes the summed square of the residuals, the coefficients are determined by differentiating S with respect to each parameter, and setting the result equal to zero.

$$\frac{\partial S}{\partial p_1} = -2 \sum_{i=1}^{n} x_i(y_i - (p_1 x_i + p_2)) = 0$$

$$\frac{\partial S}{\partial p_2} = -2 \sum_{i=1}^{n} (y_i - (p_1 x_i + p_2)) = 0$$

The estimates of the true parameters are usually represented by b. Substituting b_1 and b_2 for p_1 and p_2, the previous equations become

$$\sum x_i(y_i - (b_1 x_i + b_2)) = 0$$

$$\sum (y_i - (b_1 x_i + b_2)) = 0$$

where the summations run from $i = 1$ to n. The *normal equations* are defined as

$$b_1 \sum x_i^2 + b_2 \sum x_i = \sum x_i y_i$$

$$b_1 \sum x_i + n b_2 = \sum y_i$$

Solving for b_1

$$b_1 = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - \left(\sum x_i\right)^2}$$

Solving for b_2 using the b_1 value

$$b_2 = \frac{1}{n}\left(\sum y_i - b_1 \sum x_i\right)$$

As you can see, estimating the coefficients p_1 and p_2 requires only a few simple calculations. Extending this example to a higher degree polynomial is straightforward although a bit tedious. All that is required is an additional normal equation for each linear term added to the model.

In matrix form, linear models are given by the formula

$y = X\beta + \varepsilon$

where

- y is an n-by-1 vector of responses.
- β is a m-by-1 vector of coefficients.
- X is the n-by-m design matrix for the model.
- ε is an n-by-1 vector of errors.

For the first-degree polynomial, the n equations in two unknowns are expressed in terms of y, X, and β as

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ . \\ . \\ . \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 1 \\ x_2 1 \\ x_3 1 \\ . \\ . \\ . \\ x_n 1 \end{bmatrix} \times \begin{bmatrix} p_1 \\ p_2 \end{bmatrix}$$

The least-squares solution to the problem is a vector b, which estimates the unknown vector of coefficients β. The normal equations are given by

$(X^T X)b = X^T y$

where X^T is the transpose of the design matrix X. Solving for b,

$b = (X^TX)^{-1} X^Ty$

Use the MATLAB backslash operator (`mldivide`) to solve a system of simultaneous linear equations for unknown coefficients. Because inverting X^TX can lead to unacceptable rounding errors, the backslash operator uses *QR* decomposition with pivoting, which is a very stable algorithm numerically. Refer to "Arithmetic Operations" for more information about the backslash operator and *QR* decomposition.

You can plug *b* back into the model formula to get the predicted response values, $\hat{y}$.

$\hat{y} = Xb = Hy$

$H = X(X^TX)^{-1} X^T$

A hat (circumflex) over a letter denotes an estimate of a parameter or a prediction from a model. The projection matrix *H* is called the hat matrix, because it puts the hat on *y*.

The residuals are given by

$r = y - \hat{y} = (1-H)y$

Weighted Least Squares

It is usually assumed that the response data is of equal quality and, therefore, has constant variance. If this assumption is violated, your fit might be unduly influenced by data of poor quality. To improve the fit, you can use weighted least-squares regression where an additional scale factor (the weight) is included in the fitting process. Weighted least-squares regression minimizes the error estimate

$$s = \sum_{i=1}^{n} w_i(y_i - \widehat{y}_i)^2$$

where w_i are the weights. The weights determine how much each response value influences the final parameter estimates. A high-quality data point influences the fit more than a low-quality data point. Weighting your data is recommended if the weights are known, or if there is justification that they follow a particular form.

The weights modify the expression for the parameter estimates *b* in the following way,

$$b = \widehat{\beta} = (X^TWX)^{-1}X^TWy$$

where *W* is given by the diagonal elements of the weight matrix *w*.

You can often determine whether the variances are not constant by fitting the data and plotting the residuals. In the plot shown below, the data contains replicate data of various quality and the fit is assumed to be correct. The poor quality data is revealed in the plot of residuals, which has a "funnel" shape where small predictor values yield a bigger scatter in the response values than large predictor values.
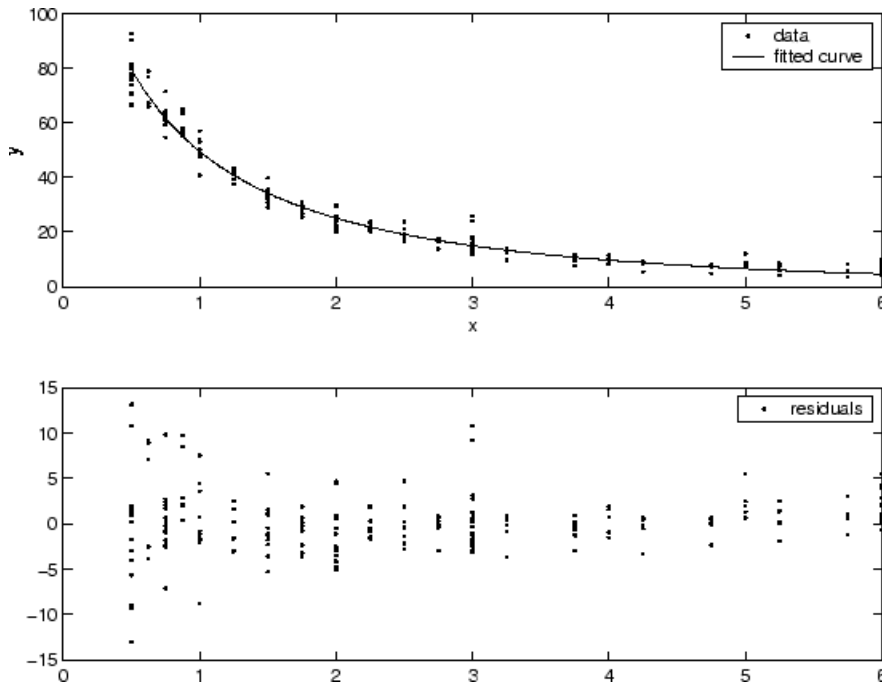
The weights you supply should transform the response variances to a constant value. If you know the variances of the measurement errors in your data, then the weights are given by

$$w_i = 1/\sigma_i^2$$

Or, if you only have estimates of the error variable for each data point, it usually suffices to use those estimates in place of the true variance. If you do not know the variances, it suffices to specify weights on a relative scale. Note that an overall variance term is estimated even when weights have been specified. In this instance, the weights define the relative weight to each point in the fit, but are not taken to specify the exact variance of each point.

For example, if each data point is the mean of several independent measurements, it might make sense to use those numbers of measurements as weights.

Robust Least Squares

It is usually assumed that the response errors follow a normal distribution, and that extreme values are rare. Still, extreme values called *outliers* do occur.

The main disadvantage of least-squares fitting is its sensitivity to outliers. Outliers have a large influence on the fit because squaring the residuals magnifies the effects of these extreme data points. To minimize the influence of outliers, you can fit your data using robust least-squares regression. The toolbox provides these two robust regression methods:

- Least absolute residuals (LAR) — The LAR method finds a curve that minimizes the absolute difference of the residuals, rather than the squared differences. Therefore, extreme values have a lesser influence on the fit.

- Bisquare weights — This method minimizes a weighted sum of squares, where the weight given to each data point depends on how far the point is from the fitted line. Points near the line get full

weight. Points farther from the line get reduced weight. Points that are farther from the line than would be expected by random chance get zero weight.

For most cases, the bisquare weight method is preferred over LAR because it simultaneously seeks to find a curve that fits the bulk of the data using the usual least-squares approach, and it minimizes the effect of outliers.

Robust fitting with bisquare weights uses an iteratively reweighted least-squares algorithm, and follows this procedure:

1 Fit the model by weighted least squares.

2 Compute the *adjusted residuals* and standardize them. The adjusted residuals are given by

$$r_{adj} = \frac{r_i}{\sqrt{1 - h_i}}$$

r_i are the usual least-squares residuals and h_i are *leverages* that adjust the residuals by reducing the weight of high-leverage data points, which have a large effect on the least-squares fit. The standardized adjusted residuals are given by

$$u = \frac{r_{adj}}{Ks}$$

K is a tuning constant equal to 4.685, and s is the robust standard deviation given by $MAD/0.6745$ where MAD is the median absolute deviation of the residuals.

3 Compute the robust weights as a function of u. The bisquare weights are given by

$$w_i = \begin{cases} \left(1 - (u_i)^2\right)^2 & |u_i| < 1 \\ 0 & |u_i| \geq 1 \end{cases}$$

Note that if you supply your own regression weight vector, the final weight is the product of the robust weight and the regression weight.

4 If the fit converges, then you are done. Otherwise, perform the next iteration of the fitting procedure by returning to the first step.

The plot shown below compares a regular linear fit with a robust fit using bisquare weights. Notice that the robust fit follows the bulk of the data and is not strongly influenced by the outliers.
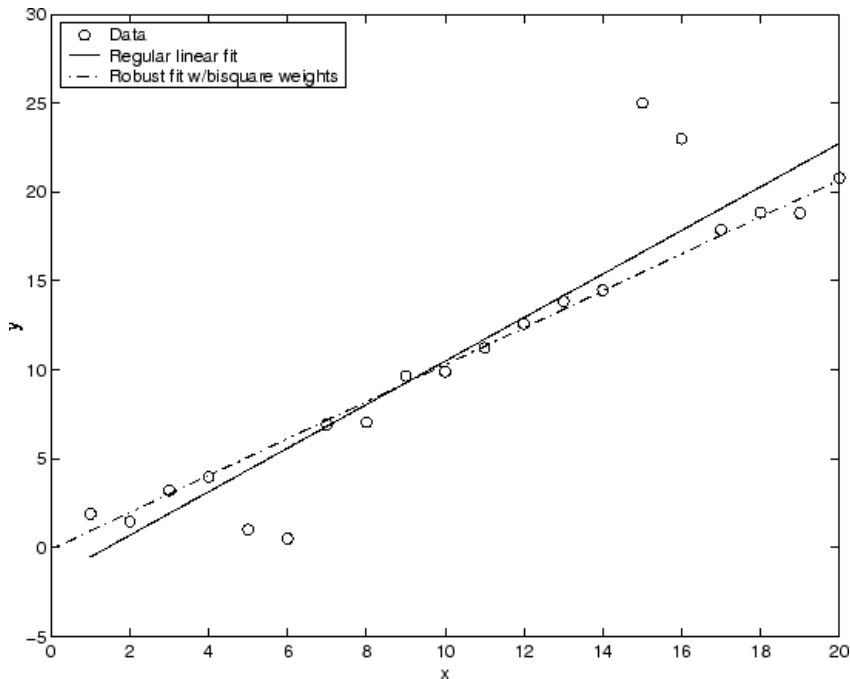
Instead of minimizing the effects of outliers by using robust regression, you can mark data points to be excluded from the fit. Refer to "Remove Outliers" on page 7-8 for more information.

Nonlinear Least Squares

Curve Fitting Toolbox software uses the nonlinear least-squares formulation to fit a nonlinear model to data. A nonlinear model is defined as an equation that is nonlinear in the coefficients, or a combination of linear and nonlinear in the coefficients. For example, Gaussians, ratios of polynomials, and power functions are all nonlinear.

In matrix form, nonlinear models are given by the formula

$y = f(X,\beta) + \varepsilon$

where

- y is an n-by-1 vector of responses.
- f is a function of β and X.
- β is a m-by-1 vector of coefficients.
- X is the n-by-m design matrix for the model.
- ε is an n-by-1 vector of errors.

Nonlinear models are more difficult to fit than linear models because the coefficients cannot be estimated using simple matrix techniques. Instead, an iterative approach is required that follows these steps:

1   Start with an initial estimate for each coefficient. For some nonlinear models, a heuristic approach is provided that produces reasonable starting values. For other models, random values on the interval [0,1] are provided.

2   Produce the fitted curve for the current set of coefficients. The fitted response value $\hat{y}$ is given by

$\hat{y} = f(X,b)$

and involves the calculation of the *Jacobian* of $f(X,b)$, which is defined as a matrix of partial derivatives taken with respect to the coefficients.

3   Adjust the coefficients and determine whether the fit improves. The direction and magnitude of the adjustment depend on the fitting algorithm. The toolbox provides these algorithms:

- Trust-region — This is the default algorithm and must be used if you specify coefficient constraints. It can solve difficult nonlinear problems more efficiently than the other algorithms and it represents an improvement over the popular Levenberg-Marquardt algorithm.

- Levenberg-Marquardt — This algorithm has been used for many years and has proved to work most of the time for a wide range of nonlinear models and starting values. If the trust-region algorithm does not produce a reasonable fit, and you do not have coefficient constraints, you should try the Levenberg-Marquardt algorithm.

4   Iterate the process by returning to step 2 until the fit reaches the specified convergence criteria.

You can use weights and robust fitting for nonlinear models, and the fitting process is modified accordingly.

Because of the nature of the approximation process, no algorithm is foolproof for all nonlinear models, data sets, and starting points. Therefore, if you do not achieve a reasonable fit using the default starting points, algorithm, and convergence criteria, you should experiment with different options. Refer to "Specifying Fit Options and Optimized Starting Points" on page 4-5 for a description of how to modify the default options. Because nonlinear models can be particularly sensitive to the starting points, this should be the first fit option you modify.

Robust Fitting

This example shows how to compare the effects of excluding outliers and robust fitting. The example shows how to exclude outliers at an arbitrary distance greater than 1.5 standard deviations from the model. The steps then compare removing outliers with specifying a robust fit which gives lower weight to outliers.

Create a baseline sinusoidal signal:

```
xdata = (0:0.1:2*pi)';
y0 = sin(xdata);
```

Add noise to the signal with nonconstant variance.

```
% Response-dependent Gaussian noise
gnoise = y0.*randn(size(y0));

% Salt-and-pepper noise
spnoise = zeros(size(y0));
p = randperm(length(y0));
sppoints = p(1:round(length(p)/5));
spnoise(sppoints) = 5*sign(y0(sppoints));

ydata = y0 + gnoise + spnoise;
```

Fit the noisy data with a baseline sinusoidal model, and specify 3 output arguments to get fitting information including residuals.

```
f = fittype('a*sin(b*x)');
[fit1,gof,fitinfo] = fit(xdata,ydata,f,'StartPoint',[1 1]);
```

Examine the information in the fitinfo structure.

```
fitinfo
```

```
fitinfo = struct with fields:
          numobs: 63
        numparam: 2
       residuals: [63x1 double]
        Jacobian: [63x2 double]
        exitflag: 3
    firstorderopt: 0.0883
      iterations: 5
       funcCount: 18
     cgiterations: 0
       algorithm: 'trust-region-reflective'
        stepsize: 4.1539e-04
         message: 'Success, but fitting stopped because change in residuals less than tolerance
```

Get the residuals from the fitinfo structure.

```
residuals = fitinfo.residuals;
```

Identify "outliers" as points at an arbitrary distance greater than 1.5 standard deviations from the baseline model, and refit the data with the outliers excluded.

```
I = abs( residuals) > 1.5 * std( residuals );
outliers = excludedata(xdata,ydata,'indices',I);

fit2 = fit(xdata,ydata,f,'StartPoint',[1 1],...
           'Exclude',outliers);
```

Compare the effect of excluding the outliers with the effect of giving them lower bisquare weight in a robust fit.

```
fit3 = fit(xdata,ydata,f,'StartPoint',[1 1],'Robust','on');
```

Plot the data, the outliers, and the results of the fits. Specify an informative legend.

```
plot(fit1,'r-',xdata,ydata,'k.',outliers,'m*')
hold on
plot(fit2,'c--')
plot(fit3,'b:')
xlim([0 2*pi])
legend( 'Data', 'Data excluded from second fit', 'Original fit',...
    'Fit with points excluded', 'Robust fit' )
hold off
```

Plot the residuals for the two fits considering outliers:

```
figure
plot(fit2,xdata,ydata,'co','residuals')
hold on
plot(fit3,xdata,ydata,'bx','residuals')
hold off
```

5
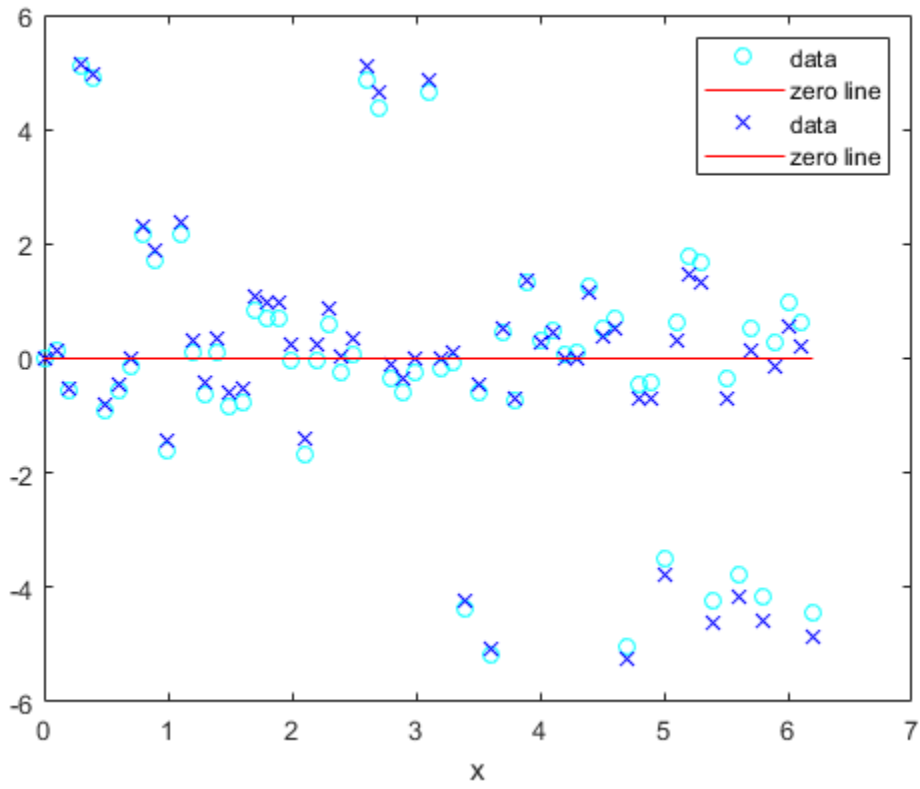
Custom Linear and Nonlinear Regression

Custom Models

In this section...
"Custom Models vs. Library Models" on page 5-2
"Selecting a Custom Equation Fit Interactively" on page 5-2
"Selecting a Custom Equation Fit at the Command Line" on page 5-4

Custom Models vs. Library Models

If the toolbox library does not contain a desired parametric equation, you can create your own custom equation. Library models, however, offer the best chance for rapid convergence. This is because:

- For most library models, the toolbox calculates optimal default coefficient starting points. For custom models, the toolbox chooses random default starting points on the interval [0,1]. You need to find suitable start points for custom models.
- Library models use an analytic Jacobian. Custom models use finite differencing.

Linear and Nonlinear Fitting

You can create custom general equations with the Custom Equation fit type. General models are nonlinear combinations of (perhaps nonlinear) terms. They are defined by equations that might be nonlinear in the parameters. The custom equation fit uses the nonlinear least-squares fitting procedure.

You can define a custom linear equation using the Custom Equation fit type, though the nonlinear fitting is less efficient and usually slower than linear least-squares fitting.

- If you don't know if your equation can be expressed as a set of linear functions, then select `Custom Equation`. You might need to search for suitable start points.
- If you need linear least-squares fitting for custom equations, select the `Linear Fitting` model type instead. See "Custom Linear Fitting" on page 5-6.

Selecting a Custom Equation Fit Interactively

In the Curve Fitting app, select `Custom Equation` from the model type list.

Use the custom equation fit to define your own equations. An example custom equation appears when you select **Custom Equation** from the list, as shown here for curve data.
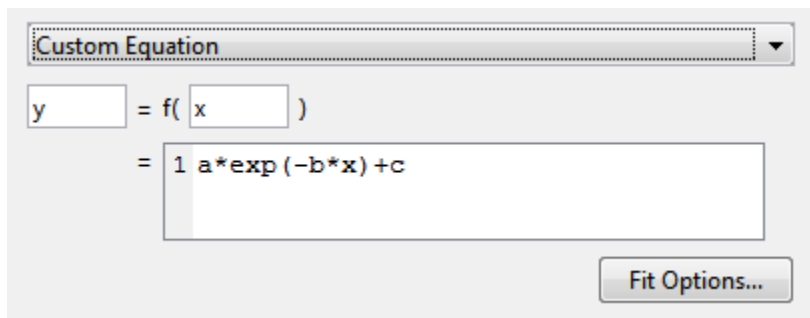
If you have surface data, the example custom equation uses both **x** and **y**.
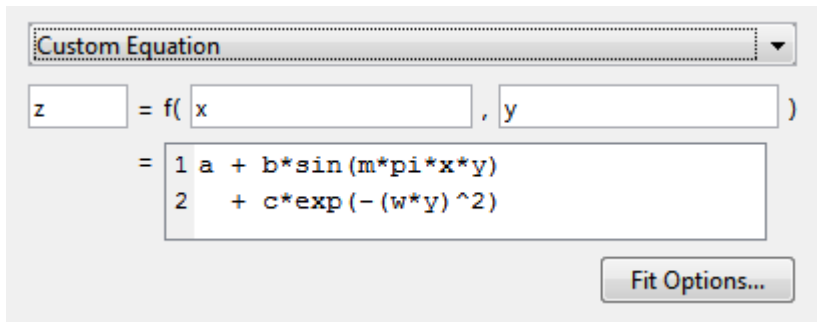
1. You can edit x, y, and z to any valid variable names.

2. In the lower box, edit the example to define your own custom equation. You can enter any valid MATLAB expression in terms of your variable names. You can specify a function or script name (see "Fitting a Curve Defined by a File in the Curve Fitting App" on page 5-3).

3. Click **Fit Options** if you want to specify start points or bounds. By default, the starting values are randomly selected on the interval [0,1] and are unconstrained. You might need to search for suitable start points and bounds. For an example, see "Custom Nonlinear ENSO Data Analysis" on page 5-19.

   If you set fit options and then alter other fit settings, the app remembers your choices for lower and upper bounds and start points, if possible. For custom equations Curve Fitting app always remembers user values, but for many library models if you change fit settings then the app automatically calculates new best values for start points or lower bounds.

You can save your custom equations as part of your saved Curve Fitting app sessions.

Your function can execute a number of times, both during fitting and during preprocessing before fitting. Be aware this may be time-consuming if you are using functions with side effects such as writing data to a file, or displaying diagnostic information to the Command Window.

For examples, see:

- "Custom Nonlinear ENSO Data Analysis" on page 5-19
- "Gaussian Fitting with an Exponential Background" on page 5-27
- "Surface Fitting to Biopharmaceutical Data" on page 5-30
- "Custom Linear Fitting" on page 5-6

Fitting a Curve Defined by a File in the Curve Fitting App

This example shows how to provide a function or script name as the fitting model in the Curve Fitting app. Define a function in a file and use it to fit a curve.

1. Define a function in a MATLAB file.

   ```
   function y = piecewiseLine(x,a,b,c,d,k)
   % PIECEWISELINE   A line made of two pieces
   % that is not continuous.

   y = zeros(size(x));

   % This example includes a for-loop and if statement
   % purely for example purposes.
   ```

```
for i = 1:length(x)
    if x(i) < k,
        y(i) = a + b.* x(i);
    else
        y(i) = c + d.* x(i);
    end
end
end
```

Save the file on the MATLAB path.

2   Define some data and open the Curve Fitting app.

```
x = [0.81;0.91;0.13;0.91;0.63;0.098;0.28;0.55;...
    0.96;0.96;0.16;0.97;0.96];
y = [0.17;0.12;0.16;0.0035;0.37;0.082;0.34;0.56;...
    0.15;-0.046;0.17;-0.091;-0.071];
cftool
```

3   In the Curve Fitting app, select x and y in the **X data** and **Y data** lists.

4   Use your `piecewiseLine` function in the Curve Fitting app by selecting the Custom Equation fit type, and then entering your function expression in the custom equation text box. The function takes x data and some parameters for fitting.

```
piecewiseLine( x, a, b, c, d, k )
```

The Curve Fitting app creates a fit using your function.
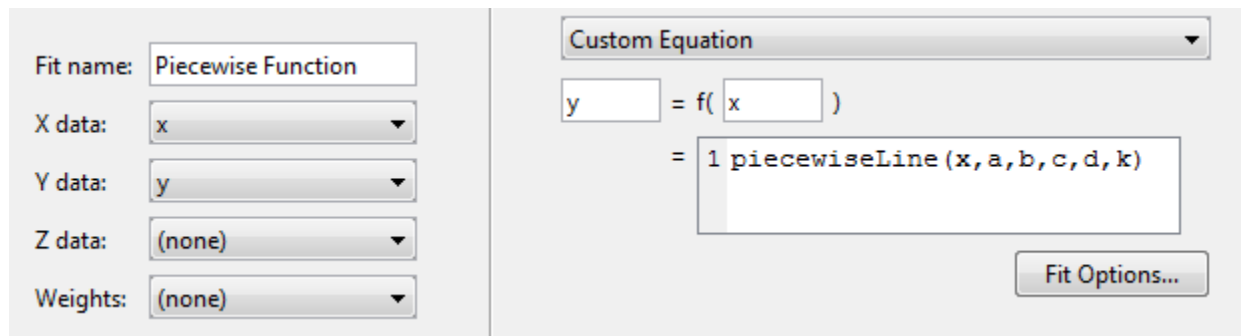
Tip If you want to use the same function for fitting at the command line, use the same expression as an input to `fittype`, and then use the `fittype` as an input to `fit`:

```
ft = fittype('piecewiseLine( x, a, b, c, d, k )');
f = fit( x, y, ft)
```

For more examples, see the `fit` function.

Selecting a Custom Equation Fit at the Command Line

To fit custom models, either:

- Supply a custom model to the `fit` function in the `fitType` input argument. You can use a MATLAB expression (including any `.m` file), a cell array of linear model terms, or an anonymous function.

- Create a `fittype` object with the `fittype` function to use as an input argument for the `fit` function.

This example loads some data and uses a custom equation defining a Weibull model as an input to the `fit` function:

```
time = [ 0.1;   0.1;   0.3;   0.3;   1.3;   1.7;   2.1;   2.6;   3.9;   3.9; ...
         5.1;   5.6;   6.2;   6.4;   7.7;   8.1;   8.2;   8.9;   9.0;   9.5; ...
         9.6;  10.2;  10.3;  10.8;  11.2;  11.2;  11.2;  11.7;  12.1;  12.3; ...
        12.3;  13.1;  13.2;  13.4;  13.7;  14.0;  14.3;  15.4;  16.1;  16.1; ...
        16.4;  16.4;  16.7;  16.7;  17.5;  17.6;  18.1;  18.5;  19.3;  19.7;];
conc = [0.01;  0.08;  0.13;  0.16;  0.55;  0.90;  1.11;  1.62;  1.79;  1.59; ...
        1.83;  1.68;  2.09;  2.17;  2.66;  2.08;  2.26;  1.65;  1.70;  2.39; ...
        2.08;  2.02;  1.65;  1.96;  1.91;  1.30;  1.62;  1.57;  1.32;  1.56; ...
        1.36;  1.05;  1.29;  1.32;  1.20;  1.10;  0.88;  0.63;  0.69;  0.69; ...
        0.49;  0.53;  0.42;  0.48;  0.41;  0.27;  0.36;  0.33;  0.17;  0.20;]);

f = fit( time, conc, 'c*a*b*x^(b-1)*exp(-a*x^b)', 'StartPoint', [0.01, 2, 5] )
plot( f, time, conc )
```

To define a custom model using `fittype`, use the form:

```
f = fittype(expr)
```

which constructs a custom model `fittype` object for the MATLAB expression contained in the string, cell array, or anonymous function *expr*.

See the `fittype` reference page for details on:

- Specifying dependent and independent variables, problem parameters, and coefficients using `fittype`.
- Specifying a cell array of terms to use a linear fitting algorithm for your custom equation. If *expr* is a string or anonymous function, then the toolbox uses a nonlinear fitting algorithm.

  For more details on linear fitting, see "Selecting Linear Fitting at the Command Line" on page 5-7.
- Examples of linear and nonlinear custom models.

For a step-by-step example, see "Custom Nonlinear Census Fitting" on page 5-16.

Custom Linear Fitting

In this section...
"About Custom Linear Models" on page 5-6
"Selecting a Linear Fitting Custom Fit Interactively" on page 5-6
"Selecting Linear Fitting at the Command Line" on page 5-7
"Fit Custom Linear Legendre Polynomials" on page 5-8

About Custom Linear Models

In the Curve Fitting app, you can use the `Custom Equation` fit to define your own linear or nonlinear equations. The custom equation fit uses the nonlinear least-squares fitting procedure.

You can define a custom linear equation in `Custom Equation`, but the nonlinear fitting is less efficient and usually slower than linear least-squares fitting. If you need linear least-squares fitting for custom equations, select `Linear Fitting` instead. Linear models are linear combinations of (perhaps nonlinear) terms. They are defined by equations that are linear in the parameters.

Tip If you need linear least-squares fitting for custom equations, select `Linear Fitting`. If you don't know if your equation can be expressed as a set of linear functions, then select `Custom Equation` instead. See "Selecting a Custom Equation Fit Interactively" on page 5-2.

Selecting a Linear Fitting Custom Fit Interactively

1 In the Curve Fitting app, select some curve data in the **X data** and **Y data** lists. You can only see `Linear Fitting` in the model type list after you select some curve data, because `Linear Fitting` is for curves, not surfaces.

Curve Fitting app creates a default polynomial fit.

2 Change the model type from `Polynomial` to `Linear Fitting` in the model type list.

An example equation appears when you select `Linear Fitting` from the list.

3 You can change x and y to any valid variable names.

4 The lower box displays the example equation. Click **Edit** to change the example terms in the Edit Custom Linear Terms dialog box and define your own equation.

For an example, see "Fit Custom Linear Legendre Polynomials in Curve Fitting App" on page 5-8.

Selecting Linear Fitting at the Command Line

To use a linear fitting algorithm, specify a cell array of model terms as an input to the `fit` or `fittype` functions. Do not include coefficients in the expressions for the terms. If there is a constant term, use '1' as the corresponding expression in the cell array.

To specify a linear model of the following form:

```
coeff1 * term1 + coeff2 * term2 + coeff3 * term3 + ...
```

where no coefficient appears within any of `term1`, `term2`, etc., use a cell array where each term, without coefficients, is specified in a cell array of strings, as follows:

```
LinearModelTerms = {'term1', 'term2', 'term3', ... }
```

1 Identify the linear model terms you need to input to `fittype`. For example, the model

```
a*log(x) + b*x + c
```

is linear in `a`, `b`, and `c`. It has three terms `log(x)`, `x`, and `1` (because `c=c*1`). To specify this model you use this cell array of terms: `LinearModelTerms = {'log(x)','x','1'}`.

2 Use the cell array of linear model terms as the input to the `fittype` function:

```
linearfittype = fittype({'log(x)','x','1'})

linearfittype =

    Linear model:
    linearfittype(a,b,c,x) = a*log(x) + b*x + c
```

3 Load some data and use the `fittype` as an input to the `fit` function.

```
load census
f = fit(cdate,pop,linearfittype)

f =

    Linear model:
    f(x) = a*log(x) + b*x + c
    Coefficients (with 95% confidence bounds):
      a =  -4.663e+04  (-4.973e+04, -4.352e+04)
      b =        25.9  (24.26, 27.55)
      c =   3.029e+05  (2.826e+05, 3.232e+05)
```

Alternatively, you can specify the cell array of linear model terms as an input to the `fit` function:

```
f = fit(x,z,{'log(x)','x','1'})
```

4 Plot the fit and data.

```
plot(f,cdate,pop)
```

For an example, see "Fit Custom Linear Legendre Polynomials at the Command Line" on page 5-14.

Fit Custom Linear Legendre Polynomials

Fit Custom Linear Legendre Polynomials in Curve Fitting App

This example shows how to fit data using several custom linear equations. The data is generated, and is based on the nuclear reaction $^{12}C(e,e'\alpha)^8Be$. The equations use sums of Legendre polynomial terms.

Consider an experiment in which 124 MeV electrons are scattered from ^{12}C nuclei. In the subsequent reaction, alpha particles are emitted and produce the residual nuclei ^8Be. By analyzing the number of alpha particles emitted as a function of angle, you can deduce certain information regarding the nuclear dynamics of ^{12}C. The reaction kinematics are shown next.
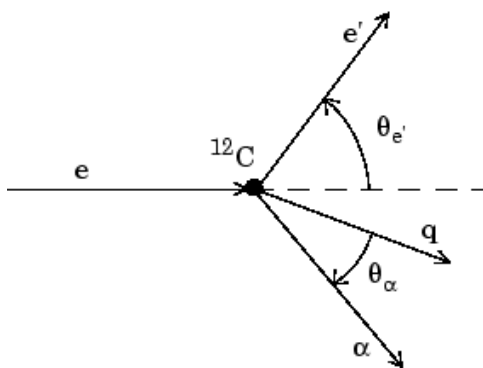
e is the incident electron.
^{12}C is the carbon target.
q is the momentum transferred to ^8Be.
e' is the scattered electron.
α is the emitted alpha particle.
$\theta_{e'}$ is the electron scattering angle.
θ_α is the alpha scattering angle.

The data is collected by placing solid state detectors at values of Θ_α ranging from $10°$ to $240°$ in $10°$ increments.

It is sometimes useful to describe a variable expressed as a function of angle in terms of Legendre polynomials

$$y(x) = \sum_{n=0}^{\infty} a_n P_n(x)$$

where $P_n(x)$ is a Legendre polynomial of degree n, x is $\cos(\Theta_\alpha)$, and a_n are the coefficients of the fit. For information about generating Legendre polynomials, see the `legendre` function.

For the alpha-emission data, you can directly associate the coefficients with the nuclear dynamics by invoking a theoretical model. Additionally, the theoretical model introduces constraints for the infinite sum shown above. In particular, by considering the angular momentum of the reaction, a fourth-degree Legendre polynomial using only even terms should describe the data effectively.

You can generate Legendre polynomials with Rodrigues' formula:

$$P_n(x) = \frac{1}{2^n n!} \left(\frac{d}{dx}\right)^n (x^2 - 1)^n$$

Legendre Polynomials Up to Fourth Degree

n	$P_n(x)$
0	1
1	x
2	$(1/2)(3x^2 - 1)$
3	$(1/2)(5x^3 - 3x)$
4	$(1/8)(35x^4 - 30x^2 + 3)$

This example shows how to fit the data using a fourth-degree Legendre polynomial with only even terms:

$$y_1(x) = a_0 + a_2\left(\frac{1}{2}\right)(3x^2 - 1) + a_4\left(\frac{1}{8}\right)(35x^4 - 30x^2 + 3)$$

1 Load the ^{12}C alpha-emission data by entering

```
load carbon12alpha
```

The workspace now contains two new variables:

- `angle` is a vector of angles (in radians) ranging from 10° to 240° in 10° increments.
- `counts` is a vector of raw alpha particle counts that correspond to the emission angles in `angle`.

2 Open the Curve Fitting app by entering:

```
cftool
```

3 In the Curve Fitting app, select `angle` and `counts` for **X data** and **Y data** to create a default polynomial fit to the two variables.

4 Change the fit type from `Polynomial` to `Linear Fitting` to create a default custom linear fit.
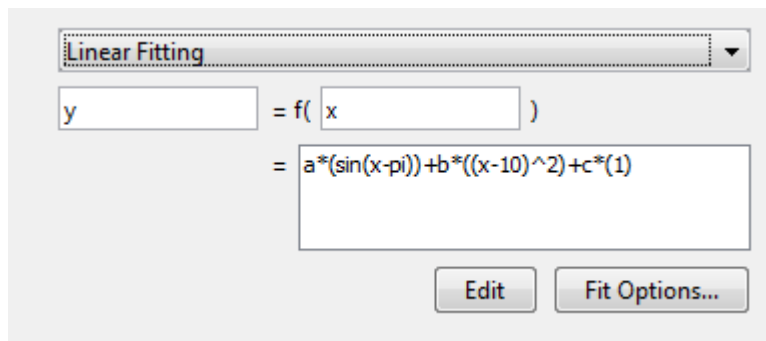
You use `Linear Fitting` instead of `Custom Equation` fit type, because the Legendre polynomials depend only on the predictor variable and constants. The equation you will specify for the model is $y_1(x)$ (that is, the equation given at the beginning of this procedure). Because `angle` is given in radians, the argument of the Legendre terms is given by $\cos(\Theta_\alpha)$.

5 Click **Edit** to change the equation terms in the Edit Custom Linear Terms dialog box.

   a Change the **Coefficients** names to `a2`, `a4`, and `a0`.

   b Change the **Terms** for `a2` to

```
(1/2)*(3*cos(x)^2-1)
```

The Curve Fitting app updates the fit as you edit the terms.

c Change the **Terms** for a4 to

```
(1/8)*(35*cos(x)^4-30*cos(x)^2+3)
```
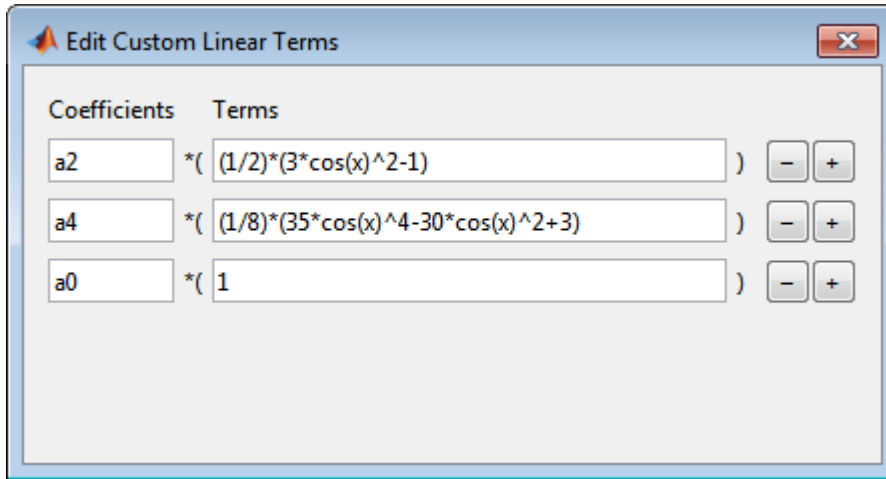

The fit appears in the Curve Fitting app.

- Rename the **Fit name** to Leg4Even.
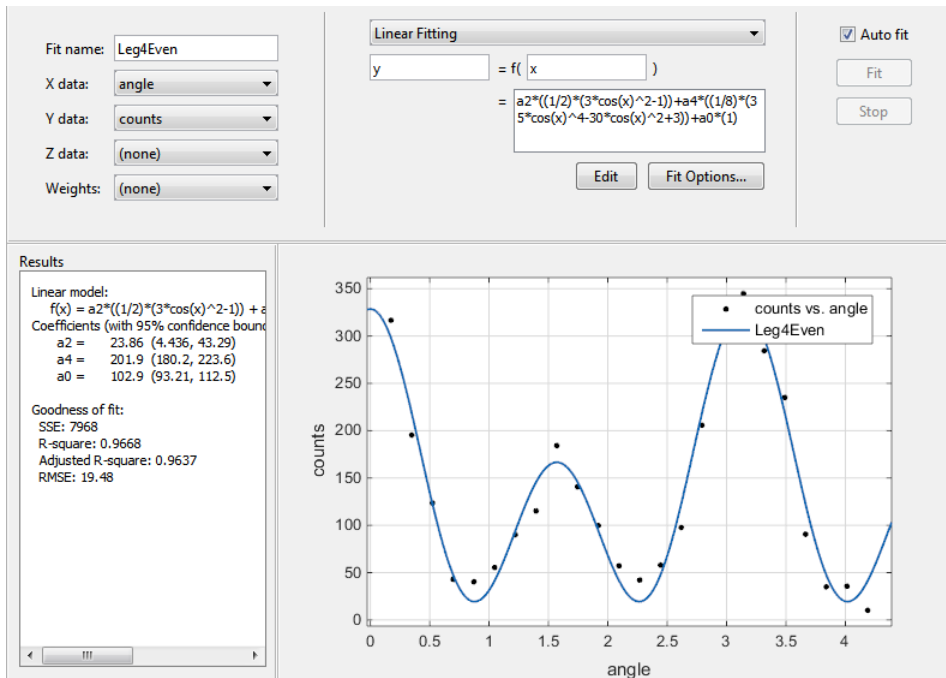
- Display the residuals by selecting **View > Residuals Plot**.
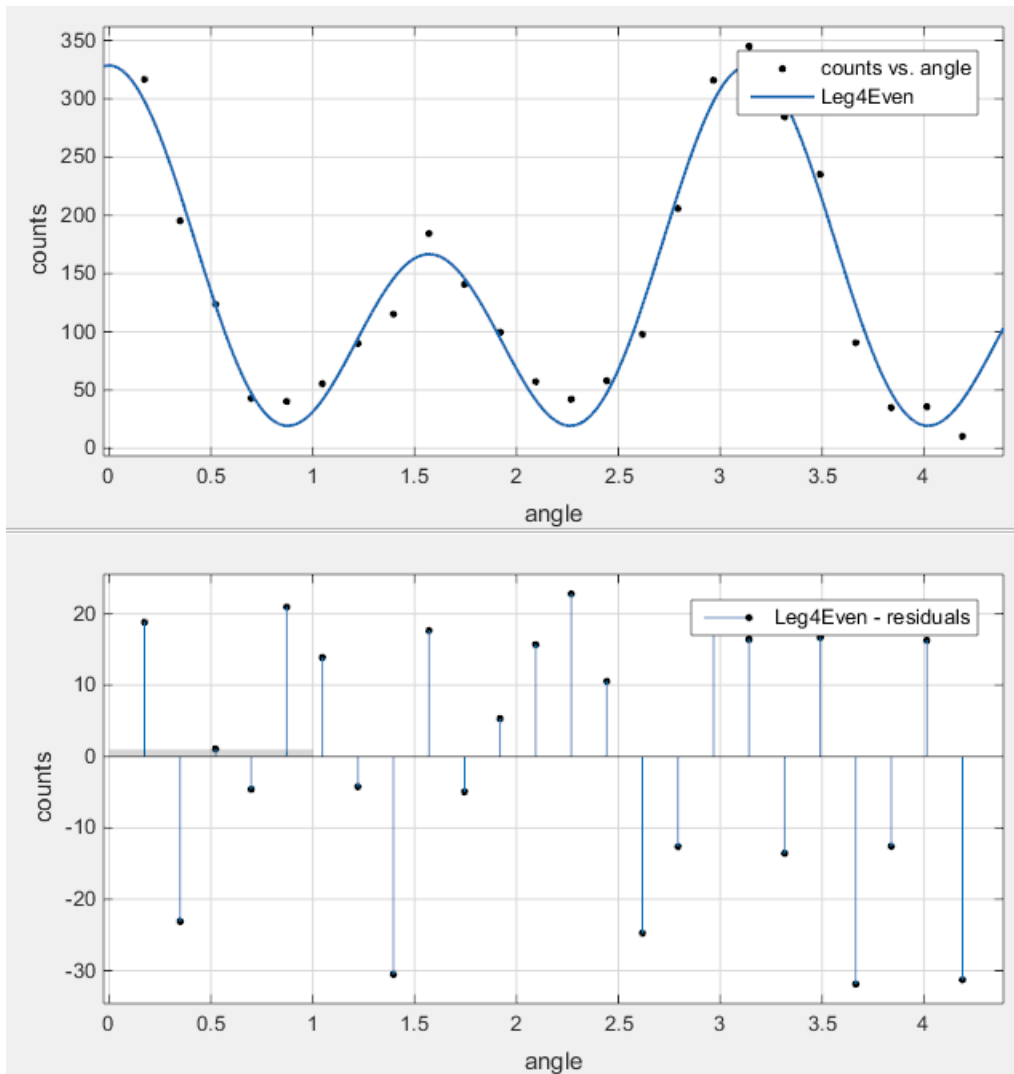
The fit appears to follow the trend of the data well, while the residuals appear to be randomly distributed and do not exhibit any systematic behavior.

- Examine the numerical fit results in the **Results** pane. Look at each coefficient value and its confidence bounds in parentheses. The 95% confidence bounds indicate that the coefficients associated with $a_0(x)$ and $a_4(x)$ are known fairly accurately, but that the $a_2(x)$ coefficient has a relatively large uncertainty.

Results

Linear model:
 f(x) = a2*((1/2)*(3*cos(x)^2-1)) + a4*((1/8)*(35*cos(x)^4-30*cos(x)^2+3)) + a0
Coefficients (with 95% confidence bounds):
  a2 =   23.86 (4.436, 43.29)
  a4 =   201.9 (180.2, 223.6)
  a0 =   102.9 (93.21, 112.5)

Goodness of fit:
 SSE: 7968
 R-square: 0.9668
 Adjusted R-square: 0.9637
 RMSE: 19.48

- Select **Fit > Duplicate Leg4Even** to make a copy of your previous Legendre polynomial fit to modify.

  The duplicated fit appears in a new tab.

  To confirm the theoretical argument that the alpha-emission data is best described by a fourth-degree Legendre polynomial with only even terms, next fit the data using both even and odd terms:

  $$y_2(x) = y_1(x) + a_1 x + a_3 \left(\frac{1}{2}\right)(5x^3 - 3x)$$

- Rename the new fit to `Leg4EvenOdd`.
- Click **Edit** to change the equation terms. The Edit Custom Linear Terms dialog box opens.

  Edit the terms as follows to fit the model given by $y_2(x)$:

  1 Click the + button to add a term twice, to add the odd Legendre terms.

  2 Change the new coefficient names to `a1` and `a3`.

  3 Change the **Terms** for `a1` to

     `cos(x)`

  4 Change the **Terms** for `a3` to

     `pro(1/2)*(5*cos(x)^3-3*cos(x))`
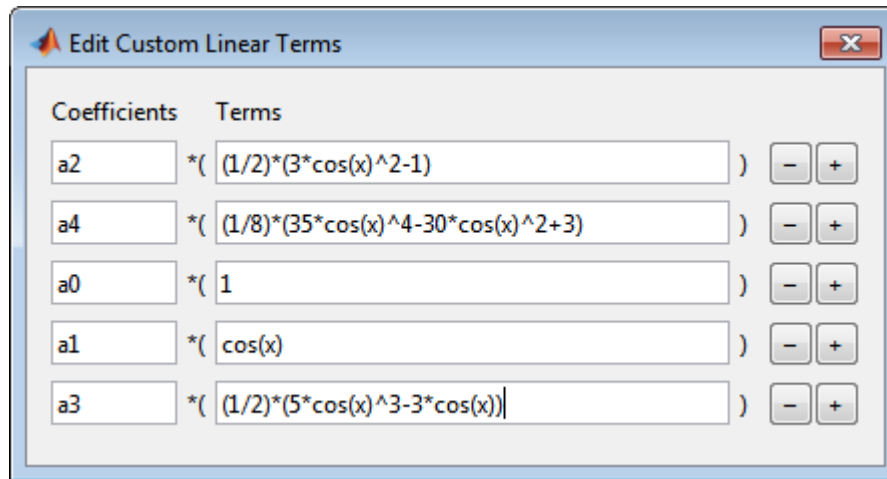
- Observe the new fit plotted in the Curve Fitting app, and examine the numerical results in the **Results** pane.

Results

Linear model:
   f(x) = a2*((1/2)*(3*cos(x)^2-1)) + a4*((1/8)*(35*cos(x)^4-30*cos(x)^2+3)
) + a0 + a1*cos(x) + a3*((1/2)*(5*cos(x)^3-3*cos(x)))
Coefficients (with 95% confidence bounds):
     a2 =   24.19 (3.291, 45.1)
     a4 =   201.5 (177.6, 225.5)
     a0 =   103.1 (92.71, 113.6)
     a1 =   1.837 (-12.89, 16.56)
     a3 =   -1.21 (-22.52, 20.1)

Goodness of fit:
  SSE: 7940
  R-square: 0.9669
  Adjusted R-square: 0.96
  RMSE: 20.44
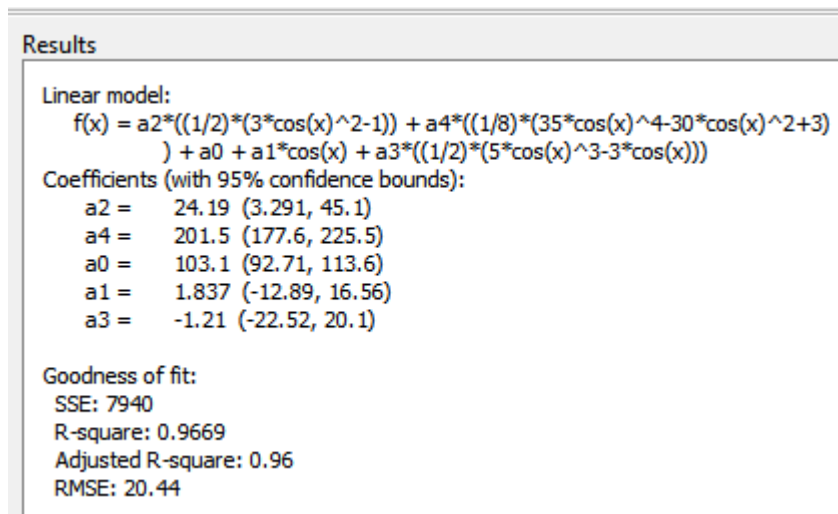
Note that the odd Legendre coefficients (`a1` and `a3`) are likely candidates for removal to simplify the fit, because their values are small and their confidence bounds contain zero. These results indicate that the odd Legendre terms do not contribute significantly to the fit, and the even Legendre terms are essentially unchanged from the previous fit. This confirms that the initial model choice in the `Leg4Even` fit is the best one.

- To compare the fits side by side, select **Left/Right** tile. You can display only the plots by hiding the fit settings and results panes using the Curve Fitting app **View** menu.
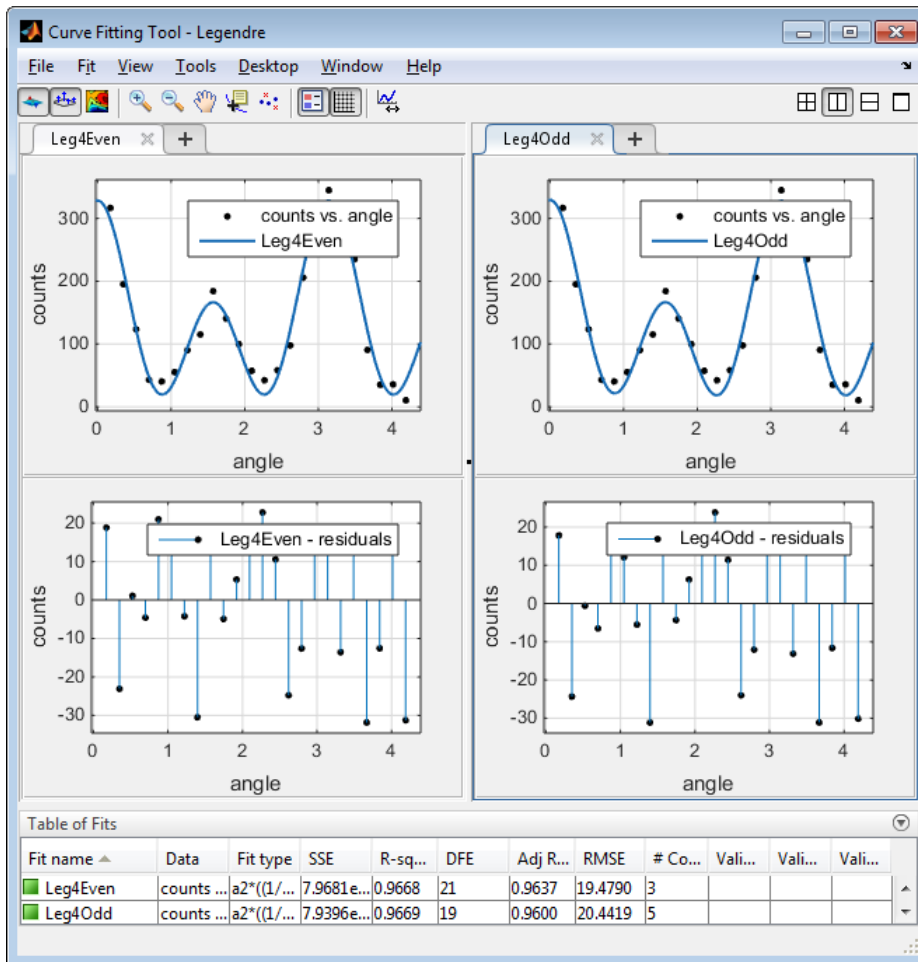
Fit Custom Linear Legendre Polynomials at the Command Line

Fit the same model at the command line that you created in Curve Fitting app.

1   To use a linear fitting algorithm, specify a cell array of model terms as an input to the `fittype` function. Use the same **Terms** you entered in Curve Fitting app for the `Leg4Even` fit, and do not specify any coefficients.

```
linearft = fittype({'(1/2)*(3*cos(x)^2-1)', ...
'(1/8)*(35*cos(x)^4-30*cos(x)^2+3)','1'})

linearft =

    Linear model:
    linearft(a,b,c,x) = a*((1/2)*(3*cos(x)^2-1))...
       + b*((1/8)*(35*cos(x)^4-30*cos(x)^2+3)) + c
```

2   Load the `angle` and `counts` variables in the workspace.

```
load carbon12alpha
```

3   Use the `fittype` as an input to the `fit` function, and specify the `angle` and `counts` variables in the workspace.

```
f = fit(angle, counts, linearft)
```

```
f =

     Linear model:
     f(x) = a*((1/2)*(3*cos(x)^2-1))...
     + b*((1/8)*(35*cos(x)^4-30*cos(x)^2+3)) + c
     Coefficients (with 95% confidence bounds):
       a =       23.86  (4.436, 43.29)
       b =       201.9  (180.2, 223.6)
       c =       102.9  (93.21, 112.5)
```

4 Plot the fit and data.

```
plot(f, angle, counts)
```

For more details on linear model terms, see the `fittype` function.

Custom Nonlinear Census Fitting

This example shows how to fit a custom equation to census data, specifying bounds, coefficients, and a problem-dependent parameter.

Load and plot the data in census.mat:

```
load census
plot(cdate,pop,'o')
hold on
```
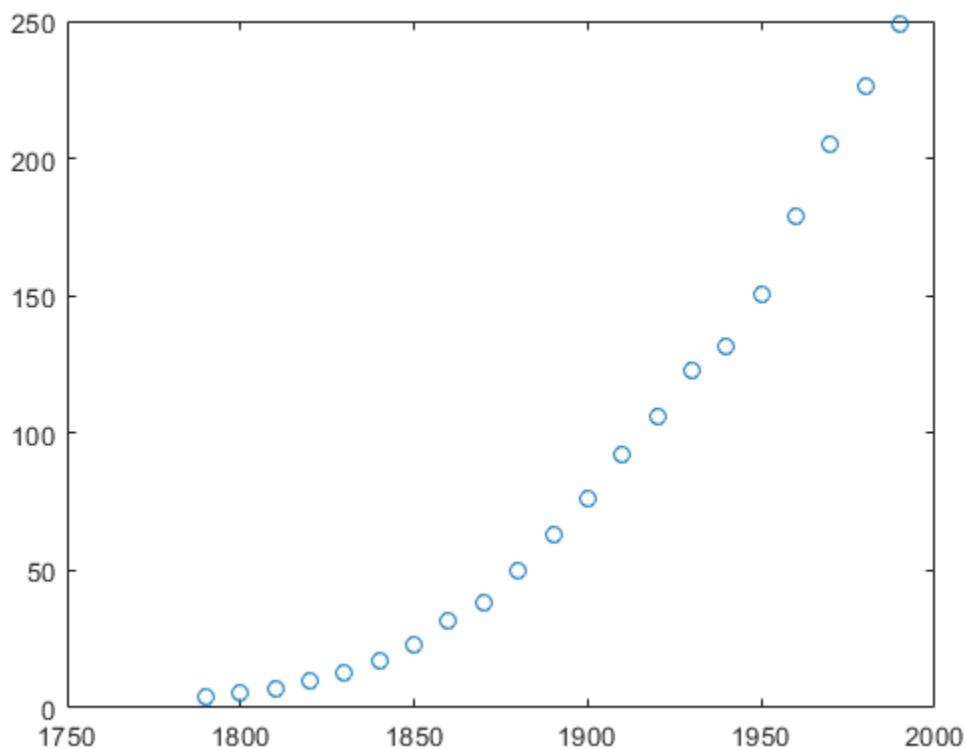

Create a fit options structure and a fittype object for the custom nonlinear model $y = a(x-b)n$, where a and b are coefficients and n is a problem-dependent parameter. See the fittype function page for more details on problem-dependent parameters.

```
s = fitoptions('Method','NonlinearLeastSquares',...
               'Lower',[0,0],...
               'Upper',[Inf,max(cdate)],...
               'Startpoint',[1 1]);
f = fittype('a*(x-b)^n','problem','n','options',s);
```

Fit the data using the fit options and a value of n = 2:

```
[c2,gof2] = fit(cdate,pop,f,'problem',2)

c2 =
     General model:
```

```
    c2(x) = a*(x-b)^n
    Coefficients (with 95% confidence bounds):
      a =     0.006092  (0.005743, 0.006441)
      b =          1789  (1784, 1793)
    Problem parameters:
      n =             2

gof2 = struct with fields:
          sse: 246.1543
      rsquare: 0.9980
          dfe: 19
    adjrsquare: 0.9979
         rmse: 3.5994
```

Fit the data using the fit options and a value of n = 3:

```
[c3,gof3] = fit(cdate,pop,f,'problem',3)

c3 =
    General model:
    c3(x) = a*(x-b)^n
    Coefficients (with 95% confidence bounds):
      a =   1.359e-05  (1.245e-05, 1.474e-05)
      b =          1725  (1718, 1731)
    Problem parameters:
      n =             3

gof3 = struct with fields:
          sse: 232.0058
      rsquare: 0.9981
          dfe: 19
    adjrsquare: 0.9980
         rmse: 3.4944
```

Plot the fit results and the data:

```
plot(c2,'m')
plot(c3,'c')
legend( 'fit with n=2', 'fit with n=3' )
```
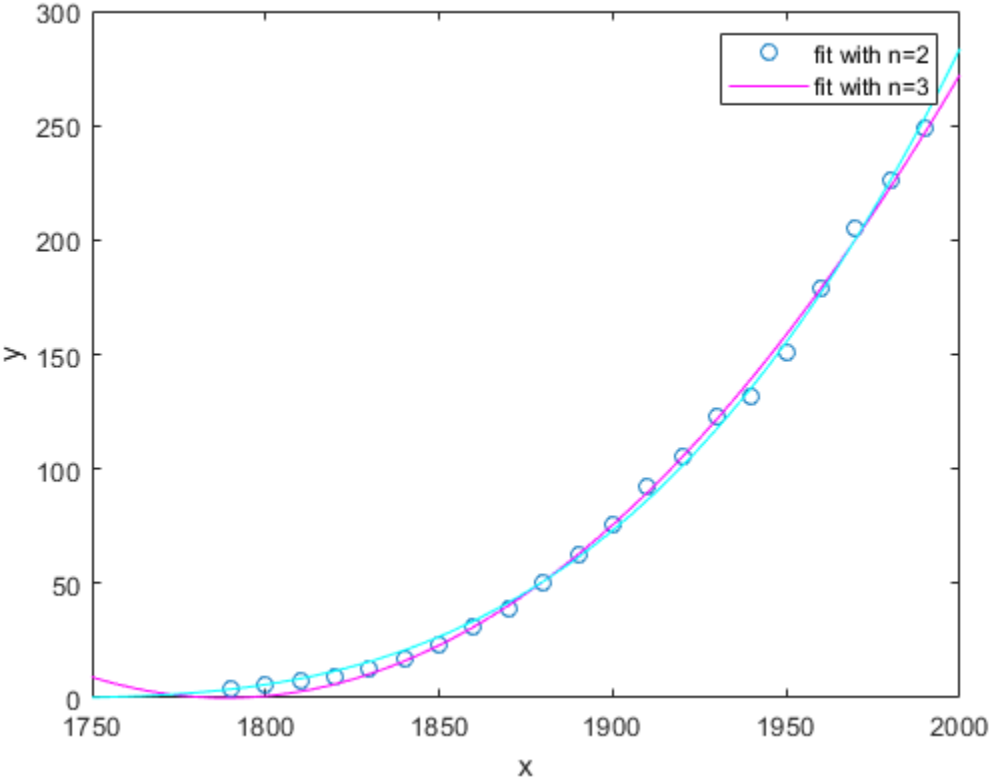
Custom Nonlinear ENSO Data Analysis

This example fits the ENSO data using several custom nonlinear equations. The ENSO data consists of monthly averaged atmospheric pressure differences between Easter Island and Darwin, Australia. This difference drives the trade winds in the southern hemisphere.

The ENSO data is clearly periodic, which suggests it can be described by a Fourier series:

$$y(x) = a_0 + \sum_{i=1}^{\infty} a_i \cos\left(2\pi \frac{x}{c_i}\right) + b_i \sin\left(2\pi \frac{x}{c_i}\right)$$

where a_i and b_i are the amplitudes, and c_i are the periods (cycles) of the data. The question to answer here is how many cycles exist?

As a first attempt, assume a single cycle and fit the data using one cosine term and one sine term.

$$y_1(x) = a_0 + a_1 \cos\left(2\pi \frac{x}{c_1}\right) + b_1 \sin\left(2\pi \frac{x}{c_1}\right)$$

If the fit does not describe the data well, add additional cosine and sine terms with unique period coefficients until a good fit is obtained.

The equation is nonlinear because an unknown coefficient c_1 is included as part of the trigonometric function arguments.

Load Data and Fit Library and Custom Fourier Models

1 Load the data and open the Curve Fitting app:

```
load enso
cftool
```

2 The toolbox includes the Fourier series as a nonlinear library equation. However, the library equation does not meet the needs of this example because its terms are defined as fixed multiples of the fundamental frequency w. Refer to "Fourier Series" on page 4-34 for more information. Create the built-in library Fourier fit to compare with your custom equations:

    a Select `month` for **X data** and `pressure` for **Y data**.

    b Select `Fourier` for the model type.

    c Enter `Fourier` for the **Fit name**.

    d Change the number of terms to 8.

    Observe the library model fit. In the next steps you will create custom equations to compare.
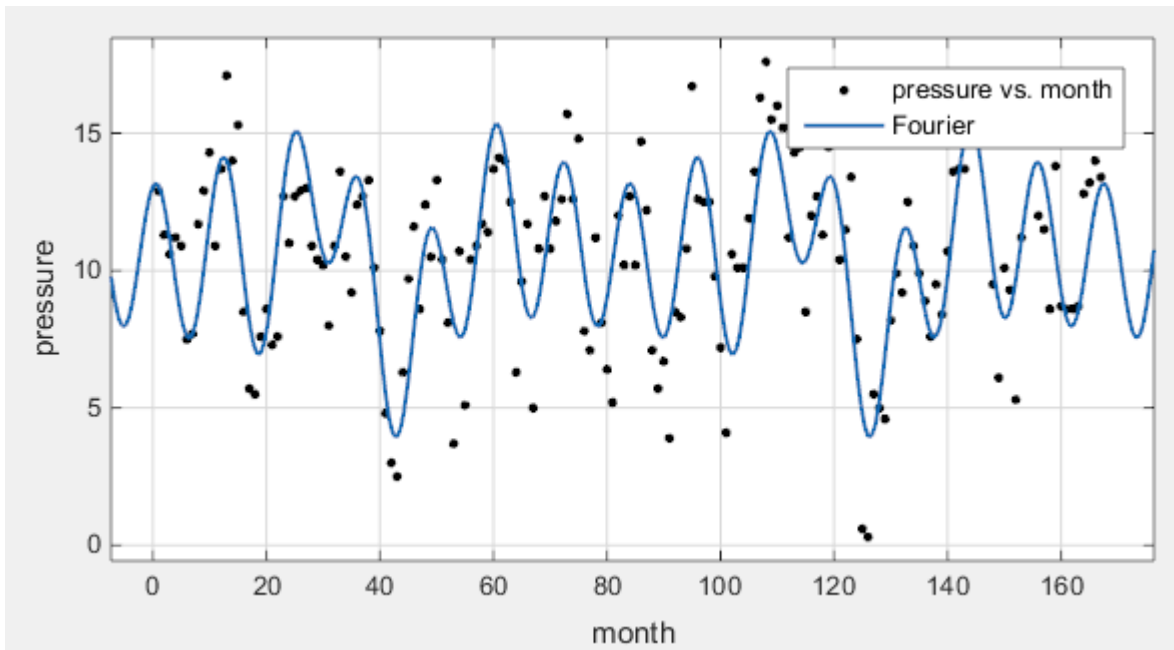
3   Duplicate your fit. Right-click your fit in the **Table of Fits** and select **Duplicate 'Fourier'**.

4   Name the new fit `Enso1Period`.

5   Change the fit type from `Fourier` to `Custom Equation`.

6   Replace the example text in the equation edit box with

```
a0+a1*cos(2*pi*x/c1)+b1*sin(2*pi*x/c1)
```


The toolbox applies the fit to the `enso` data.

The graphical and numerical results shown here indicate that the fit does not describe the data well. In particular, the fitted value for `c1` is unreasonably small. Your initial fit results might differ from these results because the starting points are randomly selected.
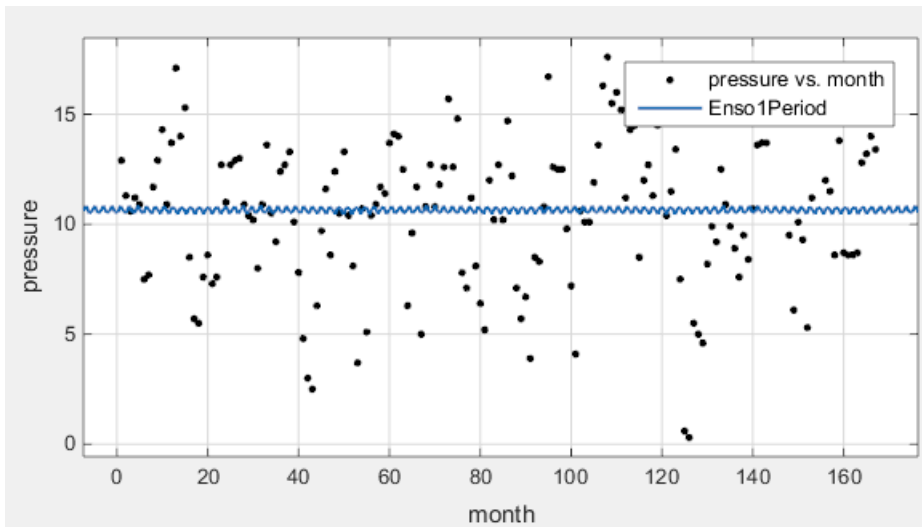
Results

General model:
   f(x) = a0+a1*cos(2*pi*x/c1)+b1*sin(2*pi*x/c1)
Coefficients (with 95% confidence bounds):
   a0 =   10.64 (10.11, 11.17)
   a1 =   -0.2215 (-1.111, 0.6678)
   b1 =   0.0915 (-1.302, 1.485)
   c1 =   0.6447 (0.6404, 0.6489)
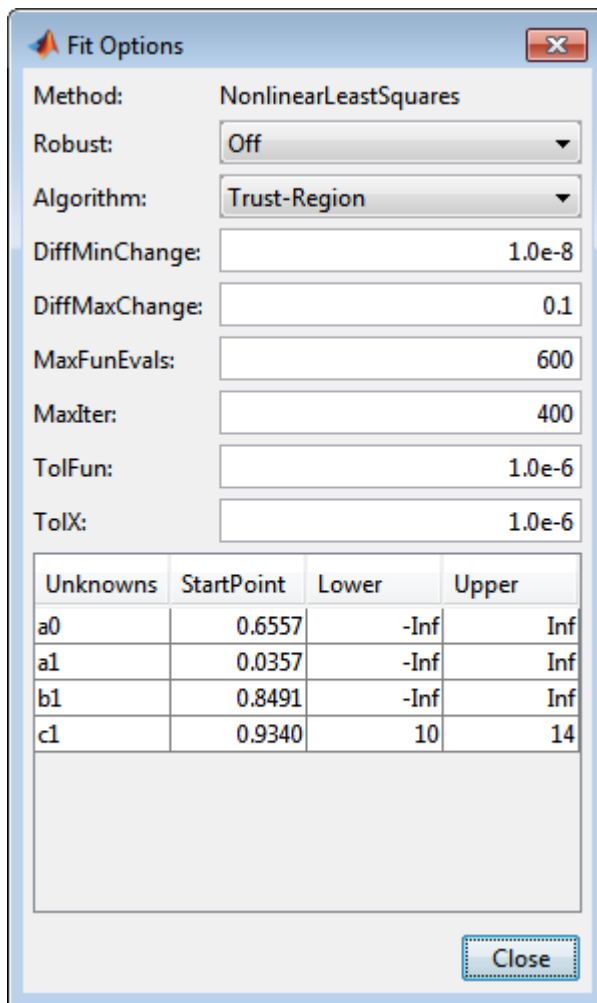
Goodness of fit:
 SSE: 1958
 R-square: 0.002451
 Adjusted R-square: -0.0158
 RMSE: 3.455

By default, the coefficients are unbounded and have random starting values from 0 to 1. The data include a periodic component with a period of about 12 months. However, with `c1` unconstrained and with a random starting point, this fit failed to find that cycle.

Use Fit Options to Constrain a Coefficient

1   To assist the fitting procedure, constrain `c1` to a value from 10 to 14. Click the **Fit Options** button to view and edit constraints for unknown coefficients.

2   In the Fit Options dialog box, observe that by default the coefficients are unbounded (bounds of -`Inf` and `Inf`).

3   Change the **Lower** and **Upper** bounds for `c1` to constrain the cycle from 10 to 14 months, as shown next.

Fit Options				
Method:	NonlinearLeastSquares			
Robust:	Off			
Algorithm:	Trust-Region			
DiffMinChange:	1.0e-8			
DiffMaxChange:	0.1			
MaxFunEvals:	600			
MaxIter:	400			
TolFun:	1.0e-6			
TolX:	1.0e-6			

Unknowns	StartPoint	Lower	Upper
a0	0.6557	-Inf	Inf
a1	0.0357	-Inf	Inf
b1	0.8491	-Inf	Inf
c1	0.9340	10	14

Close

4 Click **Close**. The Curve Fitting app refits.

5 Observe the new fit and the residuals plot. If necessary, select **View > Residuals Plot** or use the toolbar button.
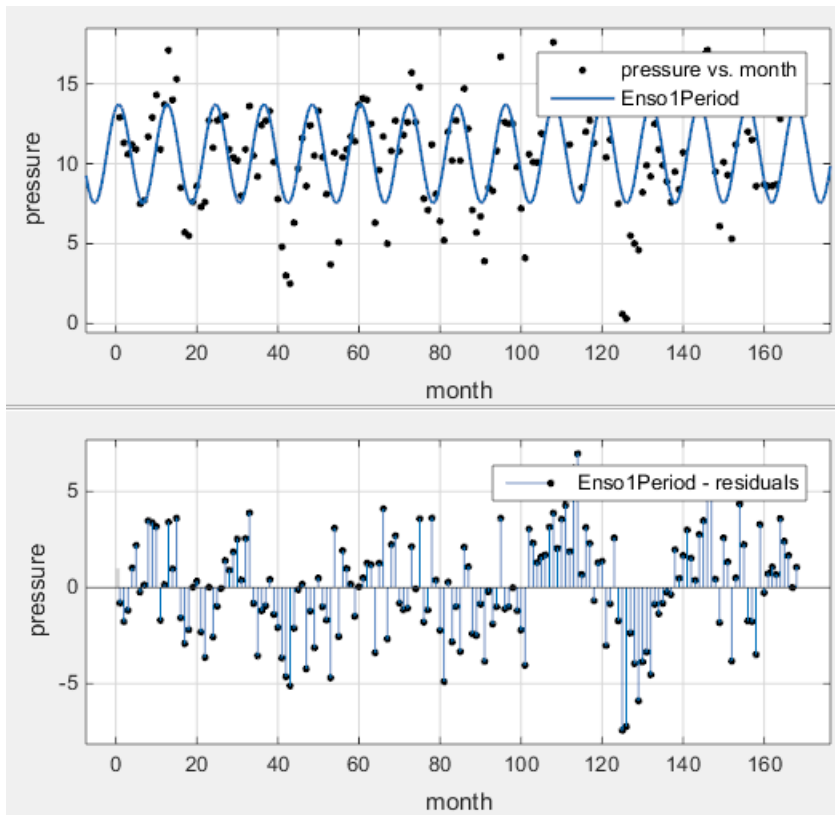
The fit appears to be reasonable for some data points but clearly does not describe the entire data set very well. As predicted, the numerical results in the **Results** pane (`c1=11.94`) indicate a cycle of approximately 12 months. However, the residuals show a systematic periodic distribution, indicating that at least one more cycle exists. There are additional cycles that you should include in the fit equation.

Create Second Custom Fit with Additional Terms and Constraints

To refine your fit, you need to add an additional sine and cosine term to $y_1(x)$ as follows:

$$y_2(x) = y_1(x) + a_2\cos\left(2\pi\frac{x}{c_2}\right) + b_2\sin\left(2\pi\frac{x}{c_2}\right)$$

and constrain the upper and lower bounds of c_2 to be roughly twice the bounds used for c_1.

1   Duplicate your fit by right-clicking it in the **Table of Fits** and selecting **Duplicate 'Enso1Period'**.

2   Name the new fit `Enso2Period`.

3   Add these terms to the end of the previous equation:

`+a2*cos(2*pi*x/c2)+b2*sin(2*pi*x/c2)`

4   Click **Fit Options**. When you edit the custom equation, the tool remembers your fit options. Observe the **Lower** and **Upper** bounds for `c1` still constrain the cycle from 10 to 14 months. Add more fit options:

    a   Change the **Lower** and **Upper** for $c2$ to be roughly twice the bounds used for $c1$ (20<c2<30).

    b   Change the **StartPoint** for $a0$ to 5.

As you change each setting, the Curve Fitting app refits. The fit and residuals are shown next.
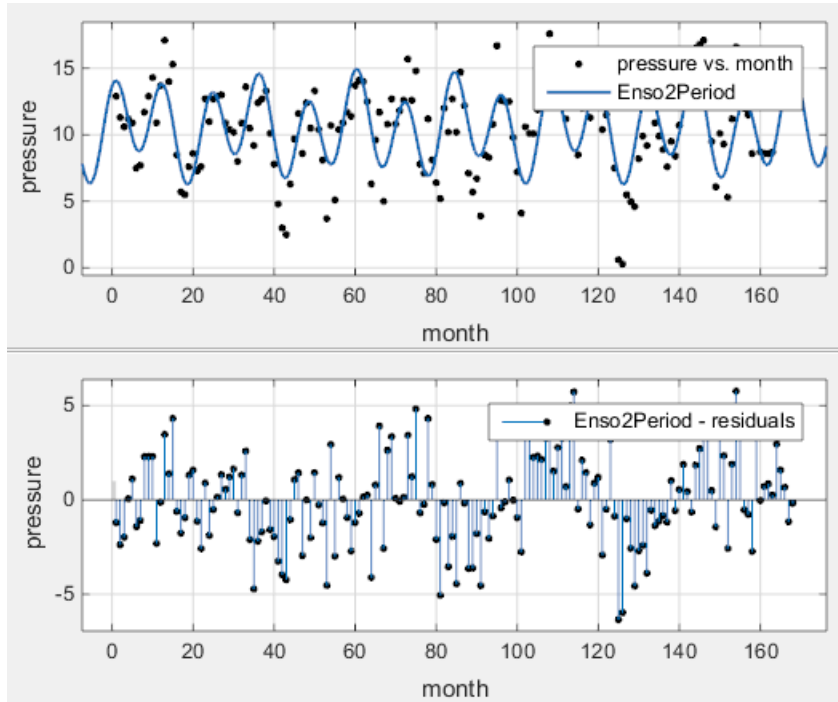
The fit appears reasonable for most data points. However, the residuals indicate that you should include another cycle to the fit equation.
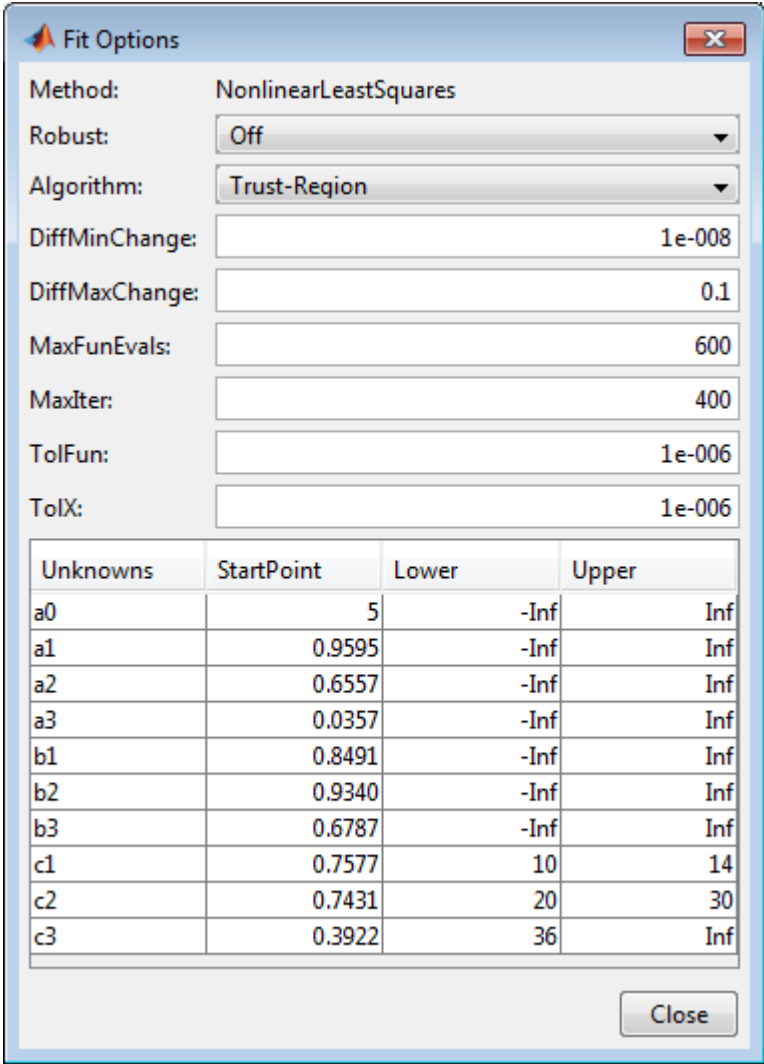
Create a Third Custom Fit with Additional Terms and Constraints

As a third attempt, add an additional sine and cosine term to $y_2(x)$

$$y_3(x) = y_2(x) + a_3\cos\left(2\pi\frac{x}{c_3}\right) + b_3\sin\left(2\pi\frac{x}{c_3}\right)$$

and constrain the lower bound of c_3 to be roughly triple the value of c_1.

1   Duplicate your fit by right-clicking it in the **Table of Fits** and selecting **Duplicate 'Enso2Period'**.

2   Name the new fit `Enso3Period`.

3   Add these terms to the end of the previous equation:

    `+a3*cos(2*pi*x/c3)+b3*sin(2*pi*x/c3)`

4   Click **Fit Options** Observe your previous fit options are still present.

5   **a**   Change the **Lower** bound for $c3$ to be 36, which is roughly triple the value of $c1$.

Fit Options

Method:	NonlinearLeastSquares
Robust:	Off ▾
Algorithm:	Trust-Region ▾
DiffMinChange:	1e-008
DiffMaxChange:	0.1
MaxFunEvals:	600
MaxIter:	400
TolFun:	1e-006
TolX:	1e-006

Unknowns	StartPoint	Lower	Upper
a0	5	-Inf	Inf
a1	0.9595	-Inf	Inf
a2	0.6557	-Inf	Inf
a3	0.0357	-Inf	Inf
b1	0.8491	-Inf	Inf
b2	0.9340	-Inf	Inf
b3	0.6787	-Inf	Inf
c1	0.7577	10	14
c2	0.7431	20	30
c3	0.3922	36	Inf

Close

b Close the dialog box. The Curve Fitting app refits. The fit and residuals appear next.
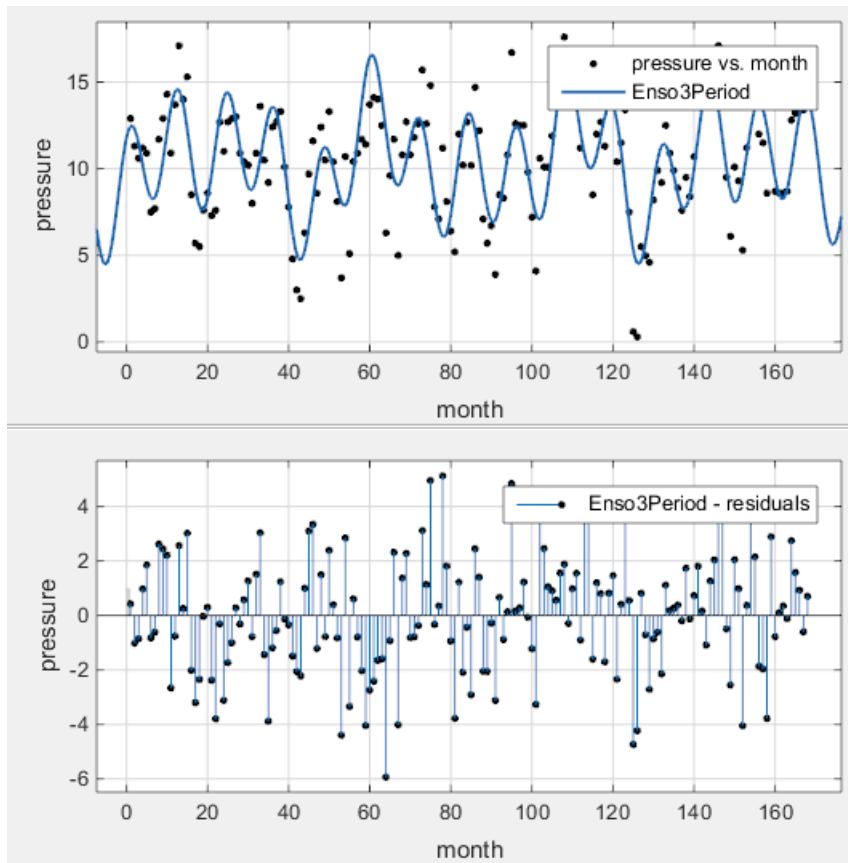
The fit is an improvement over the previous two fits, and appears to account for most of the cycles in the ENSO data set. The residuals appear random for most of the data, although a pattern is still visible indicating that additional cycles might be present, or you can improve the fitted amplitudes.

In conclusion, Fourier analysis of the data reveals three significant cycles. The annual cycle is the strongest, but cycles with periods of approximately 44 and 22 months are also present. These cycles correspond to El Nino and the Southern Oscillation (ENSO).

Gaussian Fitting with an Exponential Background

This example fits two poorly resolved Gaussian peaks on a decaying exponential background using a general (nonlinear) custom model.

Fit the data using this equation

$$y(x) = ae^{-bx} + a_1 e^{-\left(\frac{x - b_1}{c_1}\right)^2} + a_2 e^{-\left(\frac{x - b_2}{c_2}\right)^2}$$

where a_i are the peak amplitudes, b_i are the peak centroids, and c_i are related to the peak widths. Because unknown coefficients are part of the exponential function arguments, the equation is nonlinear.

1   Load the data and open the Curve Fitting app:

```
load gauss3
cftool
```

The workspace contains two new variables:

- `xpeak` is a vector of predictor values.
- `ypeak` is a vector of response values.

2   In the Curve Fitting app, select `xpeak` for **X data** and `ypeak` for **Y data**.

3   Enter `Gauss2exp1` for the **Fit name**.

4   Select `Custom Equation` for the model type.

5   Replace the example text in the equation edit box with these terms:

```
a*exp(-b*x)+a1*exp(-((x-b1)/c1)^2)+a2*exp(-((x-b2)/c2)^2)
```

The fit is poor (or incomplete) at this point because the starting points are randomly selected and no coefficients have bounds.

6   Specify reasonable coefficient starting points and constraints. Deducing the starting points is particularly easy for the current model because the Gaussian coefficients have a straightforward interpretation and the exponential background is well defined. Additionally, as the peak amplitudes and widths cannot be negative, constrain a_1, a_2, c_1, and c_2 to be greater than 0.

   a   Click **Fit Options**.

   b   Change the **Lower** bound for a_1, a_2, c_1, and c_2 to 0, as the peak amplitudes and widths cannot be negative.

   c   Enter start points as shown for the unknown coefficients.

Unknowns	Start Point
a	100
a1	100
a2	80
b	0.1
b1	110
b2	140

Unknowns	Start Point
c1	20
c2	20

Fit Options

Method:	NonlinearLeastSquares
Robust:	Off
Algorithm:	Trust-Region
DiffMinChange:	1.0e-8
DiffMaxChange:	0.1
MaxFunEvals:	600
MaxIter:	400
TolFun:	1.0e-6
TolX:	1.0e-6

Unknowns	StartPoint	Lower	Upper
a	100	-Inf	Inf
a1	100	0	Inf
a2	80	0	Inf
b	0.1000	-Inf	Inf
b1	110	-Inf	Inf
b2	140	-Inf	Inf
c1	20	0	Inf
c2	20	0	Inf
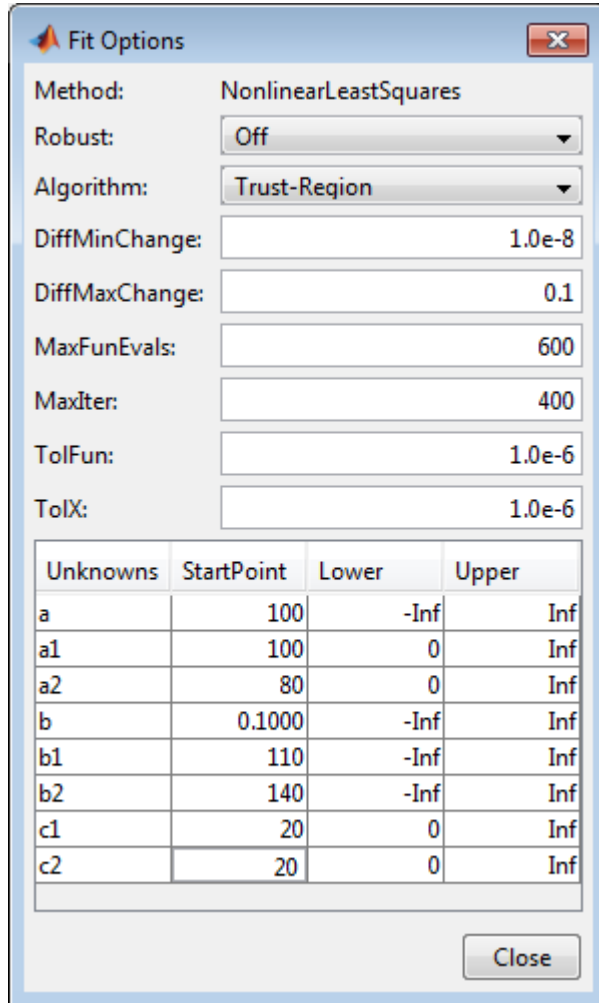
Close

As you change fit options, the Curve Fitting app refits. Press **Enter** or close the Fit Options dialog box to ensure your last change is applied to the fit.

Following are the fit and residuals.
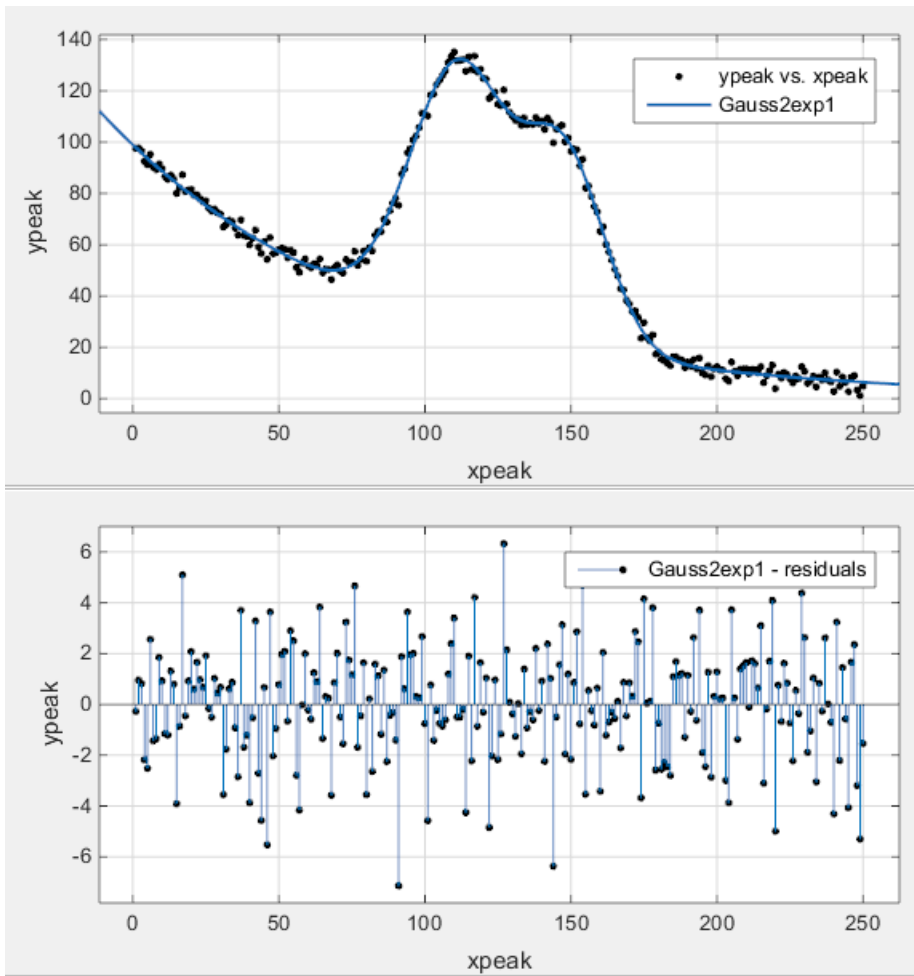
Surface Fitting to Biopharmaceutical Data

Curve Fitting Toolbox software provides some example data for an anesthesia drug interaction study. You can use Curve Fitting app to fit response surfaces to this data to analyze drug interaction effects. Response surface models provide a good method for understanding the pharmacodynamic interaction behavior of drug combinations.

This data is based on the results in this paper:

- Kern SE, Xie G, White JL, Egan TD. Opioid-hypnotic synergy: A response surface analysis of propofol-remifentanil pharmacodynamic interaction in volunteers. Anesthesiology 2004; 100: 1373–81.

Anesthesia is typically at least a two-drug process, consisting of an opioid and a sedative hypnotic. This example uses Propofol and Reminfentanil as drug class prototypes. Their interaction is measured by four different measures of the analgesic and sedative response to the drug combination. Algometry, Tetany, Sedation, and Laryingoscopy comprise the four measures of surrogate drug effects at various concentration combinations of Propofol and Reminfentanil.

To interactively create response surfaces for this drug combination:

1 Use the Current Folder browser to locate and view the folder `matlab\toolbox\curvefit \curvefit`.

2 Right-click the file `OpioidHypnoticSynergy.txt`, and select **Import Data**. The Import Wizard appears.

   a Leave the default **Column delimiters** set to **Tab** and **Column vectors** in the Import tab.

      Review the six variables selected for import: Propofol, Reminfentanil, Algometry, Tetany, Sedation, and Laryingoscopy.

   b On the Import tab, in the Import section, click **Import Selection** to import the dose-response data into the MATLAB workspace.

   Alternatively, you can import the data programmatically. Enter the following code to read the dose-response data from the file into the MATLAB workspace.

   ```
   data = importdata( 'OpioidHypnoticSynergy.txt' );
   Propofol      = data.data(:,1);
   Remifentanil  = data.data(:,2);
   Algometry     = data.data(:,3);
   Tetany        = data.data(:,4);
   Sedation      = data.data(:,5);
   Laryingoscopy = data.data(:,6);
   ```

3 To create response surfaces you must select the two drugs for the X and Y inputs, and one of the four effects for the Z output. After you load the variables into your workspace, you can either open the tool and select variables interactively, or specify the initial fit variables with the `cftool` command.

   Enter the following to open Curve Fitting app (if necessary) and create a new response surface for `Algometry`:

   ```
   cftool(Propofol, Remifentanil, Algometry)
   ```

Review the Curve Fitting app X, Y, and Z input and output controls. The tool displays the selected variables `Propofol`, `Remifentanil` and `Algometry`, with a surface fit. The default fit is an interpolating surface that passes through the data points.
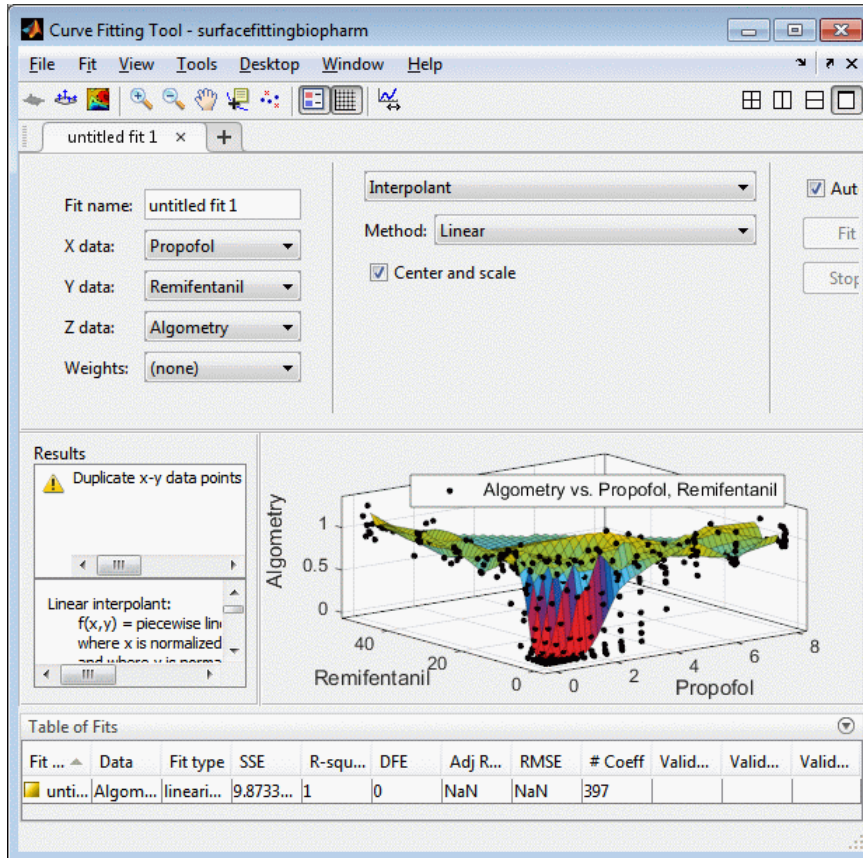
4   Create a copy of the current surface fit by either:

a   Selecting **Fit > Duplicate "*Current Fit Name*"**.

b   Right-clicking a fit in the **Table of Fits**, and selecting **Duplicate**.

5   Select the **Custom Equation** fit type from the drop-down list to define your own equation to fit the data.

6   Select and delete the example custom equation text in the edit box.

You can use the custom equation edit box to enter MATLAB code to define your model. The equation that defines the model must depend on the input variables x and y and a list of fixed parameters, estimable parameters, or both.

The model from the paper is:

$$
E = \frac{E_{\max} \cdot \left(\frac{C_A}{IC50_A} + \frac{C_B}{IC50_B} + \alpha \cdot \frac{C_A}{IC50_A} \cdot \frac{C_B}{IC50_B} \right)^n}{1 + \left(\frac{C_A}{IC50_A} + \frac{C_B}{IC50_B} + \alpha \cdot \frac{C_A}{IC50_A} \cdot \frac{C_B}{IC50_B} \right)^n}
$$

where C_A and C_B are the drug concentrations, and IC50A, IC50B, alpha, and n are the coefficients to be estimated.

You can define this in MATLAB code as

```
Effect = Emax*( CA/IC50A + CB/IC50B + alpha*( CA/IC50A )...
            .* ( CB/IC50B ) ).^n ./(( CA/IC50A + CB/IC50B + ...
            alpha*( CA/IC50A ) .* ( CB/IC50B ) ).^n  + 1);
```

Telling the tool which variables to fit and which parameters to estimate, requires rewriting the variable names CA and CB to x, and y. You must include x and y when you enter a custom equation in the edit box. Assume Emax = 1 because the effect output is normalized.

7 Enter the following text in the custom equation edit box.

```
( x/IC50A + y/IC50B + alpha*( x/IC50A ) .* ( y/IC50B ) ).^n
  ./(( x/IC50A + y/IC50B + alpha*( x/IC50A ) .*
  ( y/IC50B ) ).^n  + 1);
```

Curve Fitting app fits a surface to the data using the custom equation model.
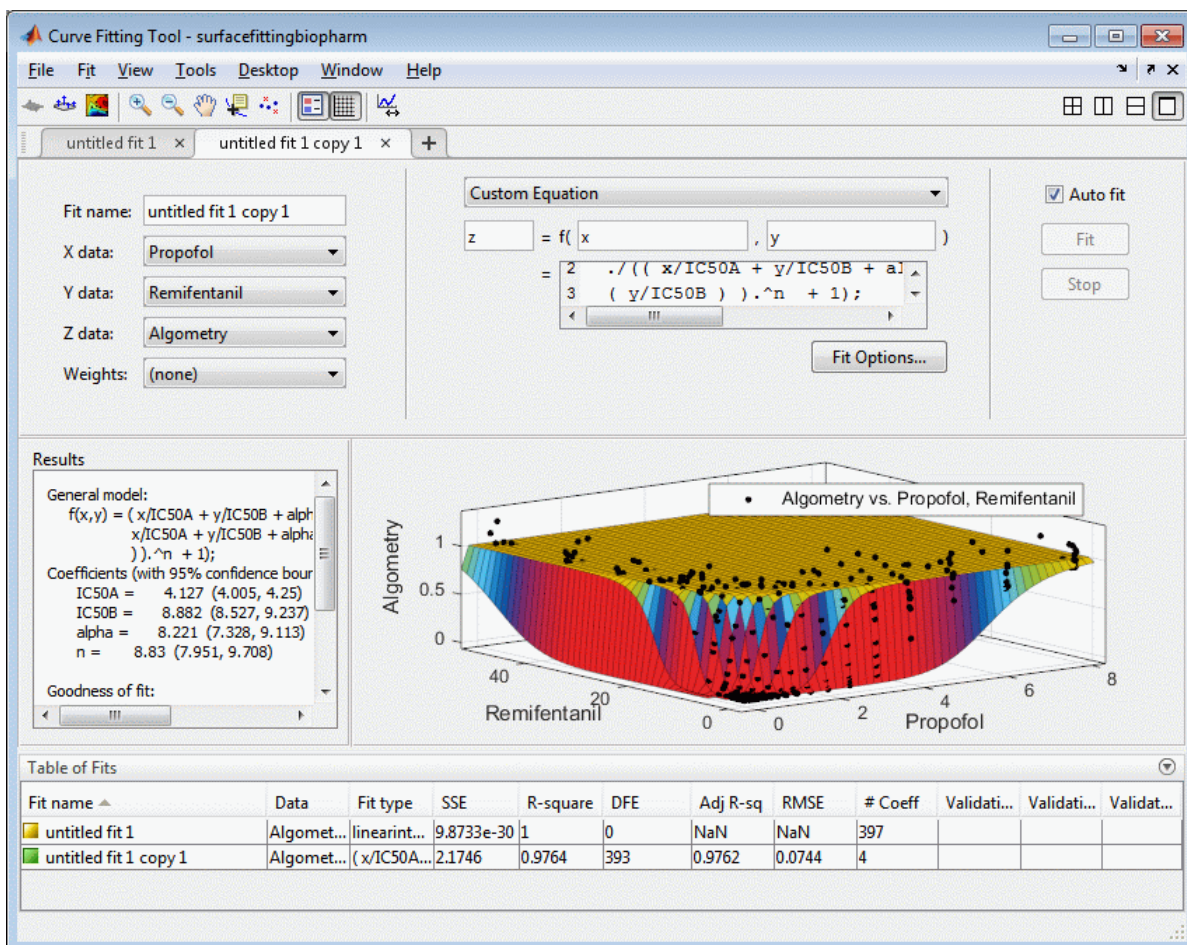
8 Set some of the fit options by clicking **Fit Options** under your custom equation.

In the Fit Options dialog box:

a Set **Robust** to Lar

b Set the **alpha StartPoint** to 1 and lower bound to –5.

Fit Options

Method:	NonlinearLeastSquares
Robust:	Lar
Algorithm:	Trust-Region
DiffMinChange:	1.0e-8
DiffMaxChange:	0.1
MaxFunEvals:	600
MaxIter:	400
TolFun:	1.0e-6
TolX:	1.0e-6

Unknowns	StartPoint	Lower	Upper
IC50A	0.4719	-Inf	Inf
IC50B	0.4225	-Inf	Inf
alpha	1	-5	Inf
n	0.8455	-Inf	Inf
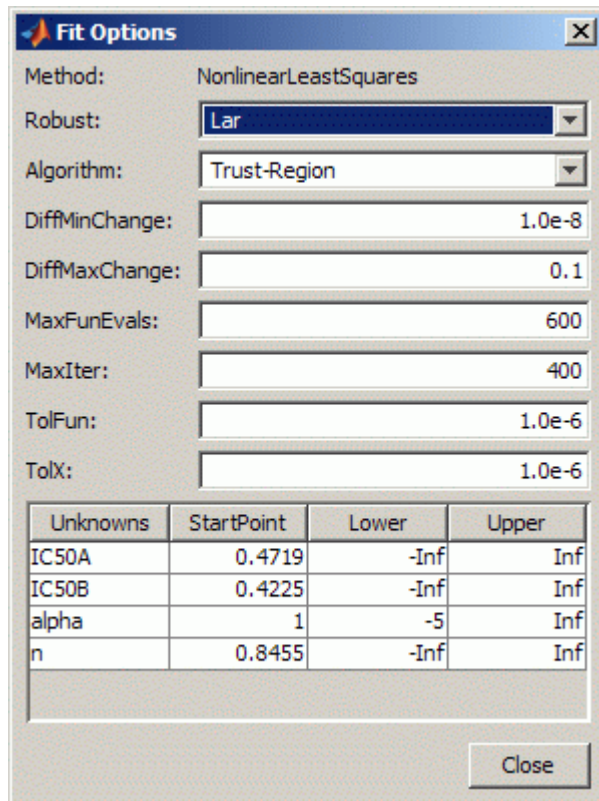
c Leave the other defaults, and click **Close**.

The tool refits with your new options.

9 Review the **Results** pane. View (and, optionally, copy) any of these results:

- The model equation
- The values of the estimated coefficients
- The goodness-of-fit statistics

10 Display the residuals plot to check the distribution of points relative to the surface by clicking the toolbar button or selecting **View > Residuals Plot**.
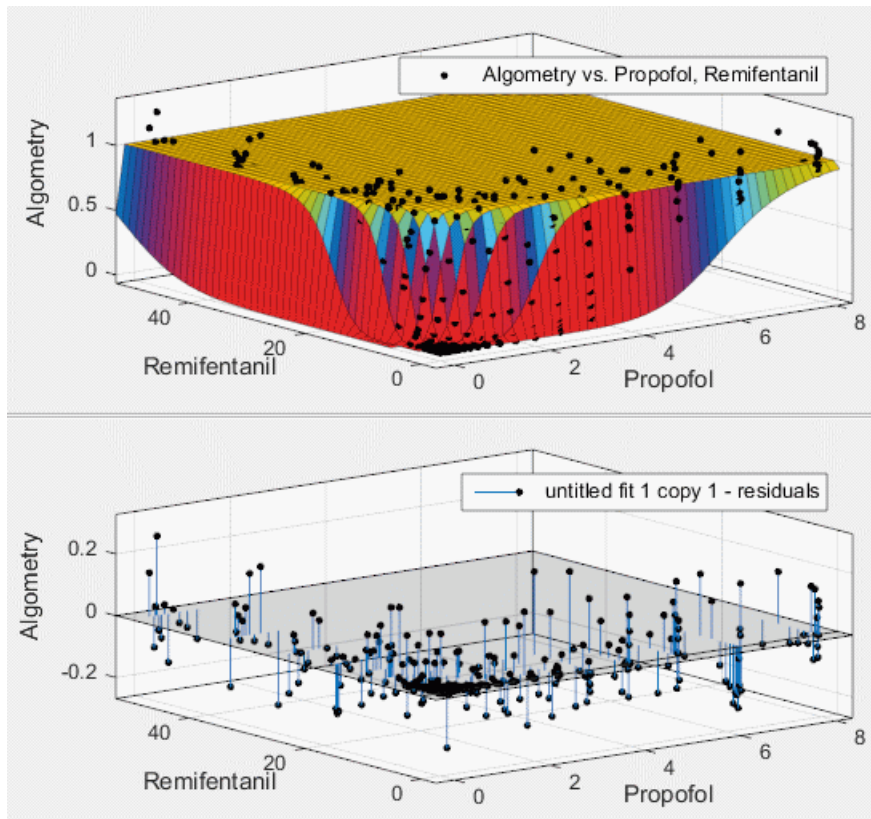
11 To generate code for all fits and plots in your Curve Fitting app session, select **File > Generate Code**.

The Curve Fitting app generates code from your session and displays the file in the MATLAB Editor. The file includes all fits and plots in your current session.

12 Save the file with the default name, `createFits.m`.

13 You can recreate your fits and plots by calling the file from the command line (with your original data or new data as input arguments). In this case, your original data still appears in the workspace.

Highlight the first line of the file (excluding the word `function`), and evaluate it by either right-clicking and selecting **Evaluate Selection in Command Window**, pressing **F9**, or copying and pasting the following to the command line:

```
[fitresult, gof] = createFits(Propofol,...
 Remifentanil, Algometry)
```

The function creates a figure window for each fit you had in your session. The custom fit figure shows both the surface and residuals plots that you created interactively in the Curve Fitting app.

14 Create a new fit to the Tetany response instead of Algometry by entering:

```
[fitresult, gof] = createFits(Propofol,...
 Remifentanil, Tetany)
```

You need to edit the file if you want the new response label on the plots. You can use the generated code as a starting point to change the surface fits and plots to fit your needs. For a list of methods you can use, see `sfit`.

To see how to programmatically fit surfaces to the same example problem, see "Surface Fitting With Custom Equations to Biopharmaceutical Data" on page 12-14.

6

Interpolation and Smoothing

Nonparametric Fitting

In some cases, you are not concerned about extracting or interpreting fitted parameters. Instead, you might simply want to draw a smooth curve through your data. Fitting of this type is called *nonparametric fitting*. The Curve Fitting Toolbox software supports these nonparametric fitting methods:

- "Interpolation Methods" on page 6-3 — Estimate values that lie between known data points.
- "Smoothing Splines" on page 6-9 — Create a smooth curve through the data. You adjust the level of smoothness by varying a parameter that changes the curve from a least-squares straight-line approximation to a cubic spline interpolant.
- "Lowess Smoothing" on page 6-16 — Create a smooth surface through the data using locally weighted linear regression to smooth data.

For details about interpolation, see "Interpolation".

You can also use smoothing techniques on response data. See "Filtering and Smoothing Data" on page 6-27.

To view all available model types, see "List of Library Models for Curve and Surface Fitting" on page 4-10.

See Also

Related Examples

- "Selecting an Interpolant Fit" on page 6-6
- "Smoothing Splines" on page 6-9
- "Lowess Smoothing" on page 6-16
- "Filtering and Smoothing Data" on page 6-27

Interpolation Methods

Interpolation is a process for estimating values that lie between known data points.

Interpolation involves the construction of a function f that matches given *data values*, y_i, at given *data sites*, x_i, in the sense that $f(x_i) = y_i$, all *i*.

The interpolant, *f*, is usually constructed as the unique function of the form

$$f(x) = \sum_j f_j(x) a_j$$

that matches the given data, with the functions f_j chosen "appropriately".

In spline interpolation, one chooses the f_j to be the n consecutive B-splines $B_j(x) = B(x|t_j,...,t_{j+k})$, $j = 1{:}n$, of order k for some knot sequence $t_1 \le t_2 \le ... \le t_{n+k}$.

About Interpolation Methods

Method	Description
Linear	Linear interpolation. This method fits a different linear polynomial between each pair of data points for curves, or between sets of three points for surfaces.
Nearest neighbor	Nearest neighbor interpolation. This method sets the value of an interpolated point to the value of the nearest data point. Therefore, this method does not generate any new data points.
Cubic spline	Cubic spline interpolation. This method fits a different cubic polynomial between each pair of data points for curves, or between sets of three points for surfaces.
Shape-preserving	Piecewise cubic Hermite interpolation (PCHIP). This method preserves monotonicity and the shape of the data. For curves only.
Biharmonic (v4)	MATLAB 4 `griddata` method. For surfaces only.
Thin-plate spline	Thin-plate spline interpolation. This method fits smooth surfaces that also extrapolate well. For surfaces only.

For surfaces, the Interpolant fit type uses the MATLAB `scatteredInterpolant` function for linear and nearest methods, and the MATLAB `griddata` function for cubic and biharmonic methods. The thin-plate spline method uses the `tpaps` function.

The type of interpolant to use depends on the characteristics of the data being fit, the required smoothness of the curve, speed considerations, post-fit analysis requirements, and so on. The linear and nearest neighbor methods are fast, but the resulting curves are not very smooth. The cubic spline and shape-preserving and v4 methods are slower, but the resulting curves are very smooth.

For example, the nuclear reaction data from the `carbon12alpha.mat` file is shown here with a nearest neighbor interpolant fit and a shape-preserving (PCHIP) interpolant fit. Clearly, the nearest

neighbor interpolant does not follow the data as well as the shape-preserving interpolant. The difference between these two fits can be important if you are interpolating. However, if you want to integrate the data to get a sense of the total strength of the reaction, then both fits provide nearly identical answers for reasonable integration bin widths.
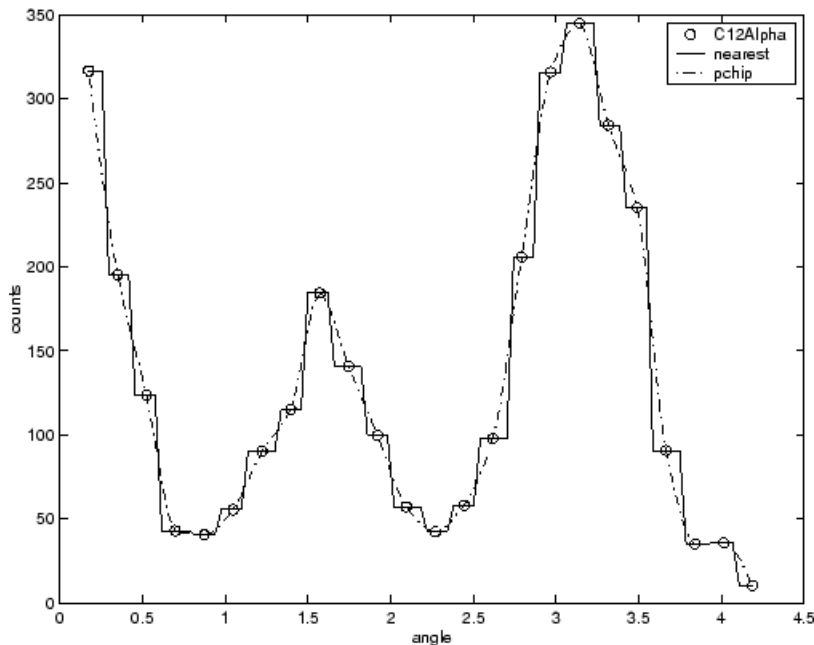
Note Goodness-of-fit statistics, prediction bounds, and weights are not defined for interpolants. Additionally, the fit residuals are always 0 (within computer precision) because interpolants pass through the data points.

Interpolants are defined as *piecewise polynomials* because the fitted curve is constructed from many "pieces" (except for `Biharmonic` for surfaces which is a radial basis function interpolant). For cubic spline and PCHIP interpolation, each piece is described by four coefficients, which the toolbox calculates using a cubic (third-degree) polynomial.

- Refer to the `spline` function for more information about cubic spline interpolation.
- Refer to the `pchip` function for more information about shape-preserving interpolation, and for a comparison of the two methods.
- Refer to the `scatteredInterpolant`, `griddata`, and `tpaps` functions for more information about surface interpolation.

It is possible to fit a single "global" polynomial interpolant to data, with a degree one less than the number of data points. However, such a fit can have wildly erratic behavior between data points. In contrast, the piecewise polynomials described here always produce a well-behaved fit, so they are more flexible than parametric polynomials and can be effectively used for a wider range of data sets.

See Also

Related Examples

- "Selecting an Interpolant Fit" on page 6-6

Selecting an Interpolant Fit

In this section...

Selecting an Interpolant Fit Interactively

In the Curve Fitting app, select `Interpolant` from the model type list.

The `Interpolant` fit category fits an interpolating curve or surface that passes through every data point. For surfaces, the Interpolant fit type uses the MATLAB `scatteredInterpolant` function for linear and nearest methods, the MATLAB `griddata` function for cubic and biharmonic methods, and the `tpaps` function for thin-plate spline interpolation.

The settings are shown here.
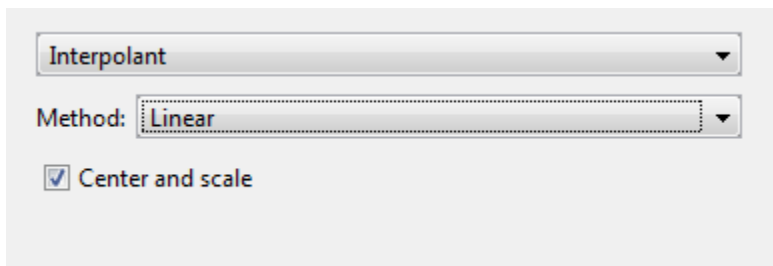
You can specify the **Method** setting: `Nearest neighbor`, `Linear`, `Cubic`, `Shape-preserving (PCHIP)` (for curves), `Biharmonic (v4)` (for surfaces) or `Thin-plate spline` (for surfaces). For details, see "About Interpolation Methods" on page 6-3.

Tip If you are fitting a surface and your input variables have different scales, turn the **Center and scale** option on and off to see the difference in the surface fit. Normalizing the inputs can strongly influence the results of the triangle-based (i.e., piecewise `Linear` and `Cubic` interpolation) and `Nearest neighbor` surface interpolation methods.

For surfaces, try thin-plate splines when you require both smooth surface interpolation and good extrapolation properties.

Fit Linear Interpolant Models Using the fit Function

This example shows how to use the `fit` function to fit linear interpolant models to data.

Interpolant Fitting Methods

Specify the interpolant model method when calling the fit function using one of the options outlined in "Interpolant Model Names" on page 4-13. None of the interpolant methods has any additional fit option parameters.

Fit a Linear Interpolant Model

Load data and fit a linear interpolant model using the `'linearinterp'` option.

```
load census
f = fit(cdate,pop,'linearinterp');
plot(f,cdate,pop);
```
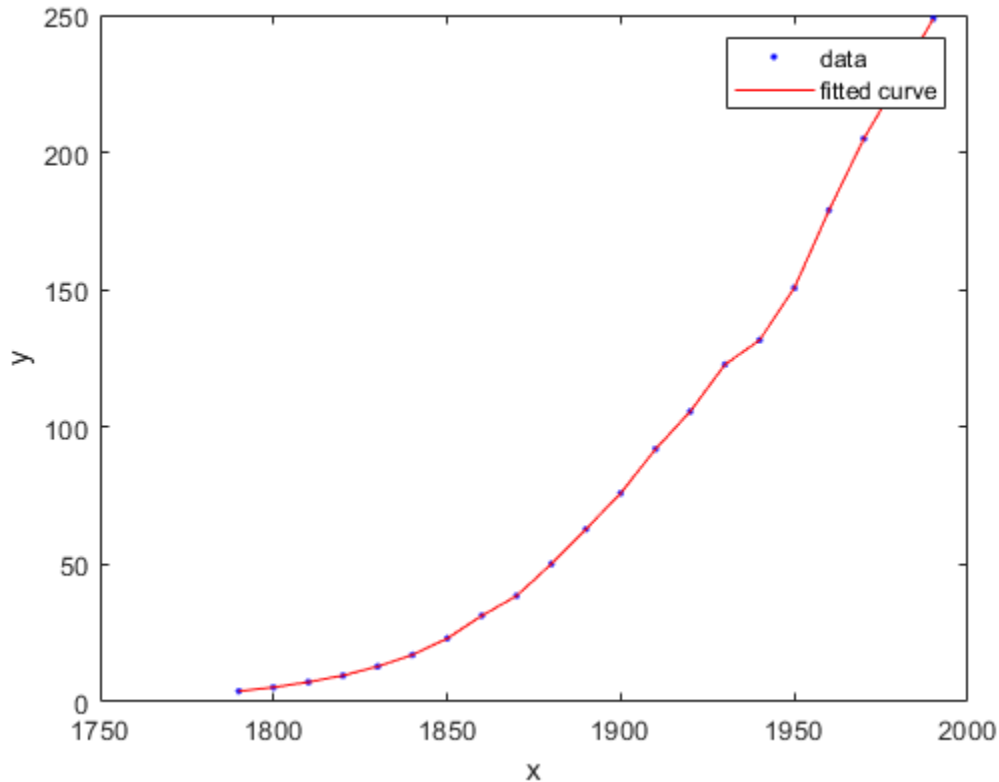

Compare Linear Interpolant Models

Load data and create both nearest neighbor and pchip interpolant fits using the `'nearestinterp'` and `'pchip'` options.

```
load carbon12alpha
f1 = fit(angle,counts,'nearestinterp');
f2 = fit(angle,counts,'pchip');
```

Compare the fitted curves `f1` and `f2` on a plot.

```
p1 = plot(f1,angle,counts);
xlim([min(angle),max(angle)])
hold on

p2 = plot(f2,'b');
hold off
legend([p1;p2],'Counts per Angle','Nearest Neighbor','pchip',...
    'Location','northwest')
```
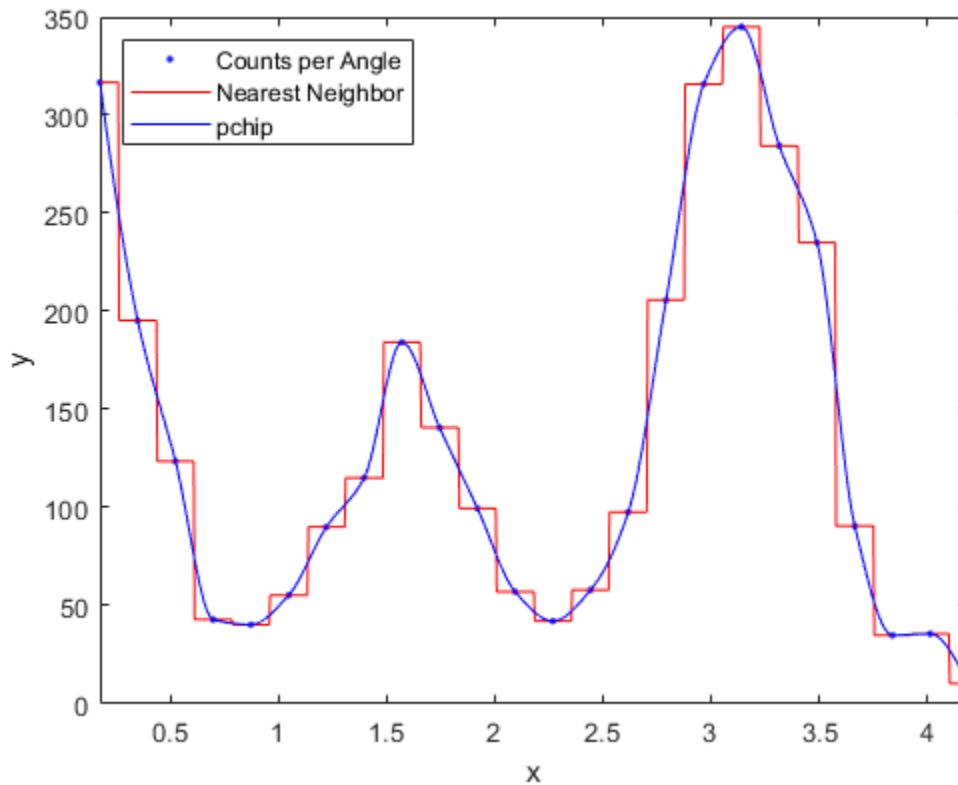
For an alternative to 'cubicinterp' or 'pchipinterp', you can use other spline functions that give you greater control over what you create. See "About Splines in Curve Fitting Toolbox" on page 8-2.

See Also

`fit`

Related Examples

- "Interpolation Methods" on page 6-3

Smoothing Splines

About Smoothing Splines

If your data is noisy, you might want to fit it using a smoothing spline. Alternatively, you can use one of the smoothing methods described in "Filtering and Smoothing Data" on page 6-27.

The smoothing spline s is constructed for the specified *smoothing parameter p* and the specified weights w_i. The smoothing spline minimizes

$$p \sum_i w_i (y_i - s(x_i))^2 + (1 - p) \int \left(\frac{d^2 s}{dx^2} \right)^2 dx$$

If the weights are not specified, they are assumed to be 1 for all data points.

p is defined between 0 and 1. $p = 0$ produces a least-squares straight-line fit to the data, while $p = 1$ produces a cubic spline interpolant. If you do not specify the smoothing parameter, it is automatically selected in the "interesting range." The interesting range of p is often near $1/(1+h^3/6)$ where h is the average spacing of the data points, and it is typically much smaller than the allowed range of the parameter. Because smoothing splines have an associated smoothing parameter, you might consider these fits to be parametric in that sense. However, smoothing splines are also piecewise polynomials like cubic spline or shape-preserving interpolants and are considered a nonparametric fit type in this guide.

Note The smoothing spline algorithm is based on the `csaps` function.

The nuclear reaction data from the file *carbon12alpha.mat* is shown here with three smoothing spline fits. The default smoothing parameter ($p = 0.99$) produces the smoothest curve. The cubic spline curve ($p = 1$) goes through all the data points, but is not quite as smooth. The third curve ($p = 0.95$) misses the data by a wide margin and illustrates how small the "interesting range" of p can be.
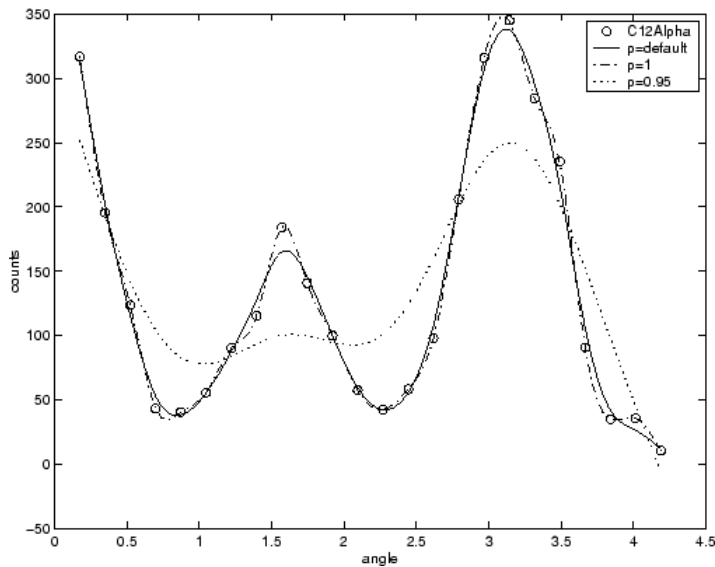
Selecting a Smoothing Spline Fit Interactively

In the Curve Fitting app, select `Smoothing Spline` from the model type list.
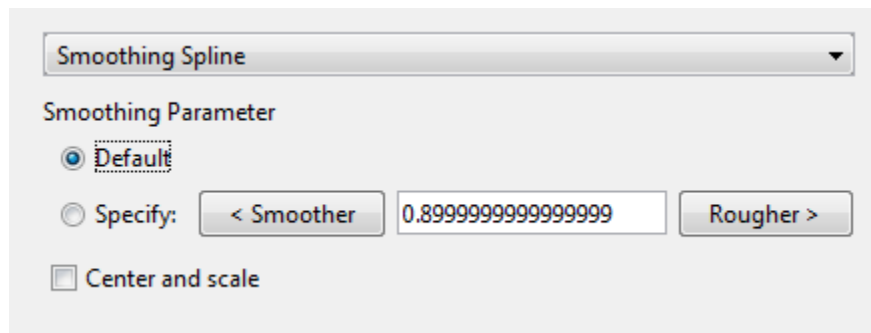
You can specify the following options:

• To make a smoother fit further from the data, click the **< Smoother** button repeatedly until the plot shows the smoothness you want.

• To make a rougher fit closer to the data, click the **Rougher >** button until you are satisfied with the plot.

• Alternatively, specify any value from 0 to 1 for the smoothing parameter. 0 produces a linear polynomial fit (a least-squares straight-line fit to the data), while 1 produces a piecewise cubic polynomial fit that passes through all the data points (a cubic spline interpolant).

• Click **Default** to return to the initial value. The toolbox attempts to select a default value appropriate for your data. See "About Smoothing Splines" on page 6-9.

For example:

1   Load the data in "About Smoothing Splines" on page 6-9 by entering:

```
load carbon12alpha
```

2 In the Curve Fitting app, select `angle` for **X data** and `counts` for **Y data**.

3 Select the `Smoothing Spline` fit type.

4 Try smoothing parameter values `1`, `0.95`, and the default value (0.99).

Fit Smoothing Spline Models Using the fit Function

This example shows how to use the `fit` function to fit a smoothing spline model to data.

Fit a Smoothing Spline Model

Load data and fit a smoothing spline model by specifying `'smoothingspline'` when calling the fit function.

```
load enso
f = fit(month,pressure,'smoothingspline');
plot(f,month,pressure)
```
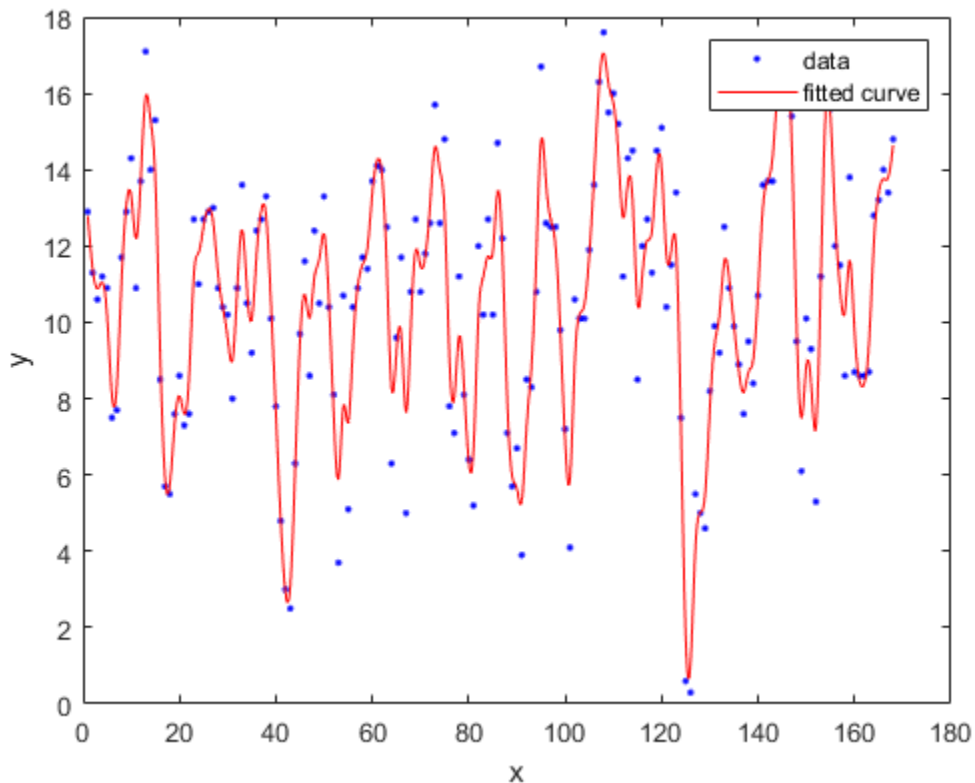

View Calculated Smoothing Parameter

Create the model again and use the third output argument to view the calculated smoothing parameter. The smoothing parameter is the `p` value in the `out` structure. The default value depends on the data set.

```
[f,gof,out] = fit(month,pressure,'smoothingspline');
```

```
out.p
```

```
ans = 0.9000
```

Specify Smoothing Parameter using `'SmoothingParam'`

Specify the smoothing parameter for a new fit with the `'SmoothingParam'` option. Its value must be between 0 and 1.

```
f = fit(month,pressure,'smoothingspline','SmoothingParam',0.07);
plot(f,month,pressure)
```
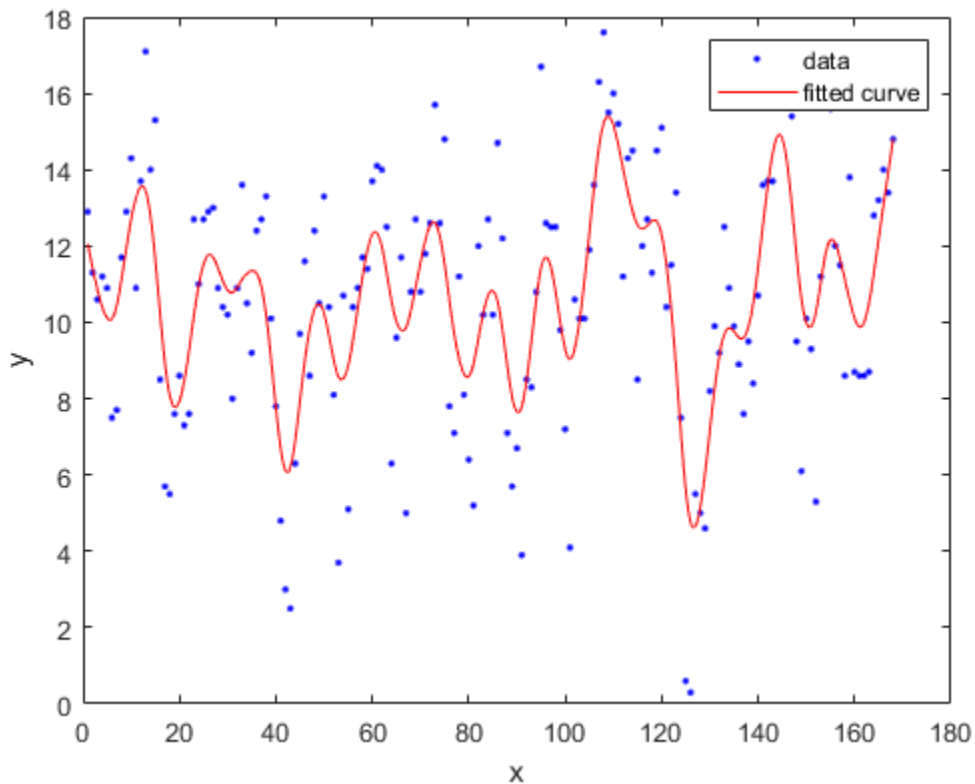

Specify Smoothing Parameter using `fitoptions`

Alternatively, use `fitoptions` to specify a smoothing parameter before fitting.

```
options = fitoptions('Method','Smooth','SmoothingParam',0.07);
[f,gof,out] = fit(month,pressure,'smooth',options);
out.p
```

```
ans = 0.0700
```

For an alternative to `'smoothingspline'`, you can use the csaps cubic smoothing spline function or other spline functions that allow greater control over what you can create. See "About Splines in Curve Fitting Toolbox" on page 8-2.

Example: Nonparametric Fitting with Cubic and Smoothing Splines

This example fits some data using a cubic spline interpolant and several smoothing splines.

1   Create the variables in your workspace:

```
x = (4*pi)*[0 1 rand(1,25)];
y = sin(x) + .2*(rand(size(x))-.5);
```

2   Open the Curve Fitting app by entering:

```
cftool
```

3   Select x and y from the **X data** and **Y data** lists.

The Curve Fitting app fits and plots the data.

4   Fit the data with a cubic spline interpolant by selecting **Interpolant** fit type and the **Method**
Cubic.
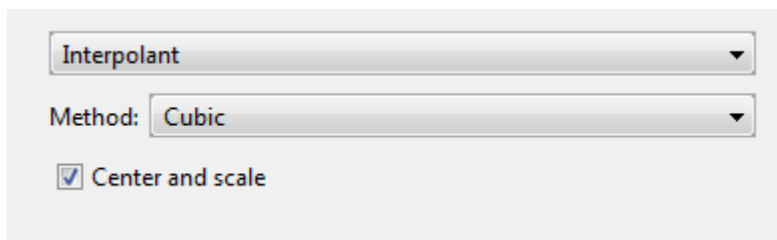
The Curve Fitting app fits and plots the cubic spline interpolant.

5   Enter the **Fit name** cubicsp.

6   View the **Results** pane. Goodness-of-fit statistics such as RMSE are not defined (shown as NaN)
for interpolants.
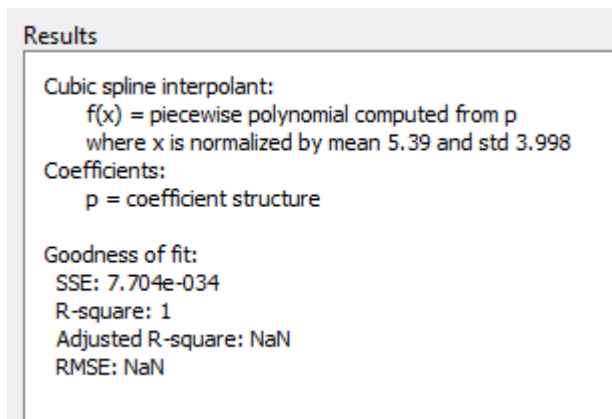
A cubic spline interpolation is defined as a piecewise polynomial that results in a structure of
coefficients (p). The number of "pieces" in the structure is one less than the number of fitted data
points, and the number of coefficients for each piece is four because the polynomial degree is
three. You can examine the coefficient structure p if you export your fit to the workspace (e.g.,
enter *fitname*.p). For information on the structure of coefficients, see "Constructing and
Working with ppform Splines" on page 10-10.

7   Create another fit to compare. Right-click your fit in the **Table of Fits** and select **Duplicate
'cubicsp'**.

8   Fit the data with a smoothing spline by selecting **Smoothing Spline**.
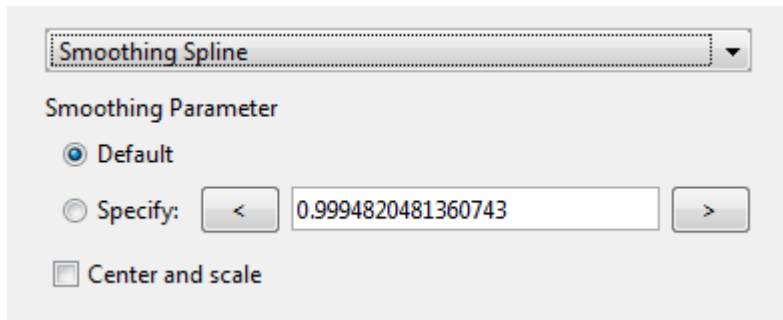
The level of smoothness is given by the **Smoothing Parameter**. The default smoothing parameter value depends on the data set, and is automatically calculated by the toolbox.

For this data set, the default smoothing parameter is close to 1, indicating that the smoothing spline is nearly cubic and comes very close to passing through each data point.

9   Name the default smoothing parameter fit `Smooth1`. If you do not like the level of smoothing produced by the default smoothing parameter, you can specify any value from 0 to 1. 0 produces a linear polynomial fit, while 1 produces a piecewise cubic polynomial fit that passes through all the data points.

The numerical results for the smoothing spline fit are shown here.
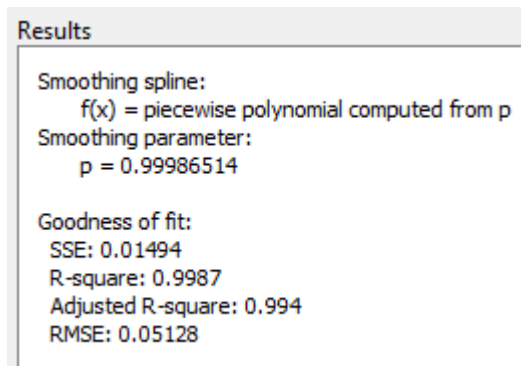
10   For comparison purposes, create another smoothing spline fit. Right-click your fit in the **Table of Fits** and select **Duplicate 'smooth1'**. Change the smoothing parameter to `0.5` and name the fit `Smooth2`.

11   Compare the plots for your three fits. Explore the fit behavior beyond the data limits by increasing the default abscissa scale. You change the axes limits with **Tools > Axes Limit Control** menu item.

Note Your results depend on random start points and may vary from those described.

Note that the default smoothing parameter produces a curve that is smoother than the interpolant, but is a good fit to the data. In this case, decreasing the smoothing parameter from the default value produces a curve that is smoother still, but is not a good fit to the data. As the smoothing parameter increases beyond the default value, the associated curve approaches the cubic spline interpolant. The cubic spline and default smoothing spline are similar for interior points, but diverge at the end points.

See Also

`fit`

Related Examples

- "Nonparametric Fitting" on page 6-2
- "Filtering and Smoothing Data" on page 6-27

Lowess Smoothing

In this section...
"About Lowess Smoothing" on page 6-16
"Selecting a Lowess Fit Interactively" on page 6-16
"Fit Lowess Models Using the fit Function" on page 6-17

About Lowess Smoothing

Use Lowess models to fit smooth surfaces to your data. The names "lowess" and "loess" are derived from the term "locally weighted scatter plot smooth," as both methods use locally weighted linear regression to smooth data. The process is *weighted* because the toolbox defines a regression weight function for the data points contained within the span. In addition to the regression weight function, the **Robust** option is a weight function that can make the process resistant to outliers.

For more information on these two types of smoothing fit, see "Local Regression Smoothing" on page 6-29.

Selecting a Lowess Fit Interactively

In the Curve Fitting app, select `Lowess` from the model type list.

You can use the `Lowess` model type to fit smooth surfaces to your data with either `lowess` or `loess` methods. The `Lowess` fits use locally weighted linear regression to smooth data.
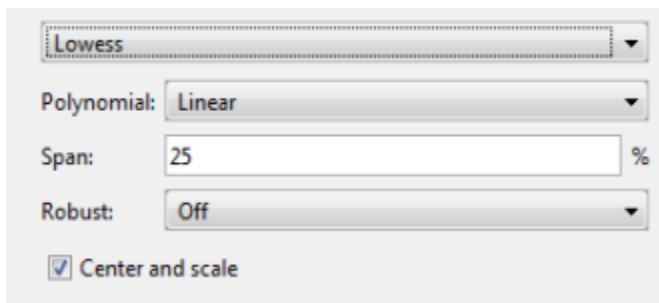
You can specify the following options:

- Select `Linear` or `Quadratic` from the list to specify the type of **Polynomial** model to use in the regression. In Curve Fitting Toolbox, `lowess` fitting uses a linear polynomial, while `loess` fitting uses a quadratic polynomial.

- Use **Span** to specify the span as a percentage of the total number of data points in the data set. The toolbox uses neighboring data points defined within the span to determine each smoothed value. This role of neighboring points is the reason why the smoothing process is called "local."

  Tip Increase the span to make the surface smoother. Reduce the span to make the surface follow the data more closely.

- The **Robust** linear least-squares fitting method you want to use (`Off`, `LAR`, or `Bisquare`). The local regression uses the **Robust** option. Using the **Robust** weight function can make the process resistant to outliers. For details, see `Robust` on the `fitoptions` reference page.

For an interactive example using Lowess, see "Surface Fitting to Franke Data" on page 2-26.

Fit Lowess Models Using the fit Function

This example shows how to use the `fit` function to fit a Lowess model to data.

Load some data and fit a Lowess model by specifying `'lowess'` when calling the fit function.

```
load franke
f = fit([x y],z,'lowess')

    Locally weighted smoothing linear regression:
      f(x,y) = lowess (linear) smoothing regression computed from p
    Coefficients:
      p = coefficient structure
```

```
plot(f,[x y],z)
```
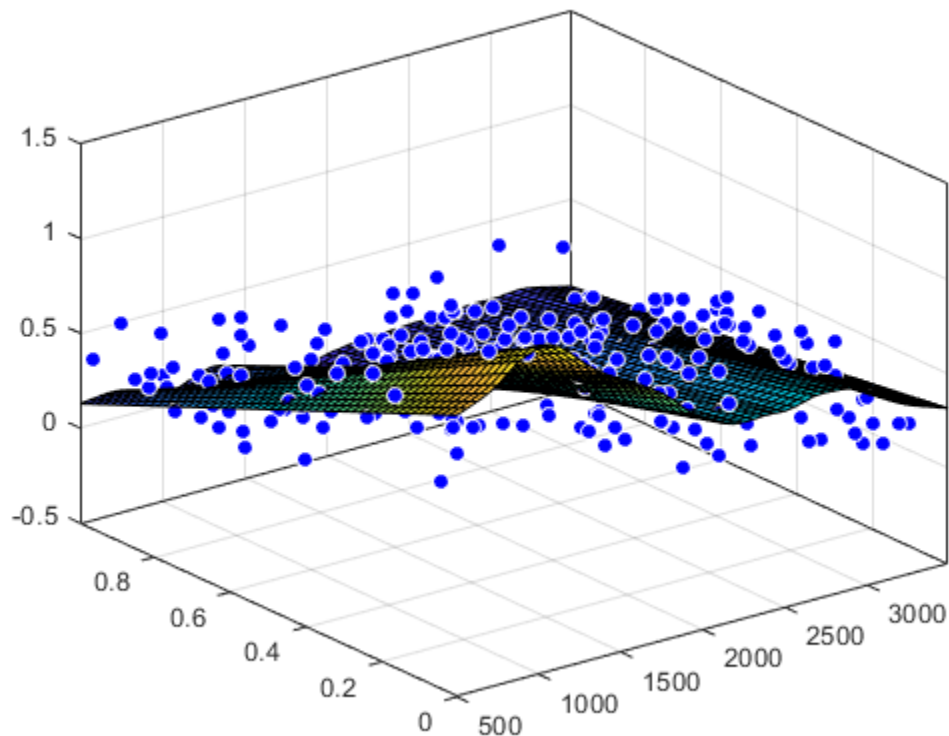

For a command-line example using Lowess, see "Fit Smooth Surfaces To Investigate Fuel Efficiency" on page 6-19.

See Also
fit

Related Examples

- "Nonparametric Fitting" on page 6-2
- "Fit Smooth Surfaces To Investigate Fuel Efficiency" on page 6-19
- "Filtering and Smoothing Data" on page 6-27
- "Surface Fitting to Franke Data" on page 2-26

Fit Smooth Surfaces To Investigate Fuel Efficiency

This example shows how to use Curve Fitting Toolbox™ to fit a response surface to some automotive data to investigate fuel efficiency.

The toolbox provides sample data generated from a GTPOWER predictive combustion engine model. The model emulates a naturally aspirated spark-ignition, 2-liter, inline 4-cylinder engine. You can fit smooth lowess surfaces to this data to find minimum fuel consumption.

The data set includes the required variables to model response surfaces:

- Speed is in revolutions per minute (rpm) units.
- Load is the normalized cylinder air mass (the ratio of cylinder aircharge to maximum naturally aspirated cylinder aircharge at standard temperature and pressure).
- BSFC is the brake-specific fuel consumption in g/KWh. That is, the energy flow in, divided by mechanical power out (fuel efficiency).

The aim is to model a response surface to find the minimum BSFC as a function of speed and load. You can use this surface as a table, included as part of a hybrid vehicle optimization algorithm combining the use of a motor and your engine. To operate the engine as fuel efficiently as possible, the table must operate the engine near the bottom of the BSFC bowl.

Load and Preprocess Data

Load the data from the XLS spreadsheet. Use the `'basic'` command option for non-Windows® platforms.

Create a variable `n` that has all the numeric data in one array.

```
n = xlsread( 'Engine_Data_SI_NA_2L_I4.xls', 'SI NA 2L I4', '', 'basic' );
```

Extract from the variable `n` the columns of interest.

```
SPEED = n(:,2);
LOAD_CMD = n(:,3);
LOAD = n(:,8);
BSFC = n(:,22);
```

Process the data before fitting, to pick out the minimum BSFC values from each sweep. The data points are organized in sweeps on speed/load.

Get a list of the speed/load sites:

```
SL = unique( [SPEED, LOAD_CMD], 'rows' );
nRuns = size( SL, 1 );
```

For each speed/load site, find the data at the site and extract the actual measured load and the minimum BSFC.

```
minBSFC  = zeros( nRuns, 1 );
Load     = zeros( nRuns, 1 );
Speed    = zeros( nRuns, 1 );
for i = 1:nRuns
    idx = SPEED == SL(i,1) & LOAD_CMD == SL(i,2);
```

```
    minBSFC(i) = min(  BSFC(idx) );
    Load(i)    = mean( LOAD(idx) );
    Speed(i)   = mean( SPEED(idx) );
end
```

Fit a Surface

Fit a surface of fuel efficiency to the preprocessed data.

```
f1 = fit( [Speed, Load], minBSFC, 'Lowess', 'Normalize', 'on' )
```

```
Locally weighted smoothing linear regression:
  f1(x,y) = lowess (linear) smoothing regression computed from p
  where x is normalized by mean 3407 and std 1214
  and where y is normalized by mean 0.5173 and std 0.1766
Coefficients:
  p = coefficient structure
```

Plot Fit

```
plot( f1, [Speed, Load], minBSFC );
xlabel( 'Speed [RPM]' );
ylabel( 'Load [%]' );
zlabel( 'Minimum BSFC [g/Kwh]' );
```
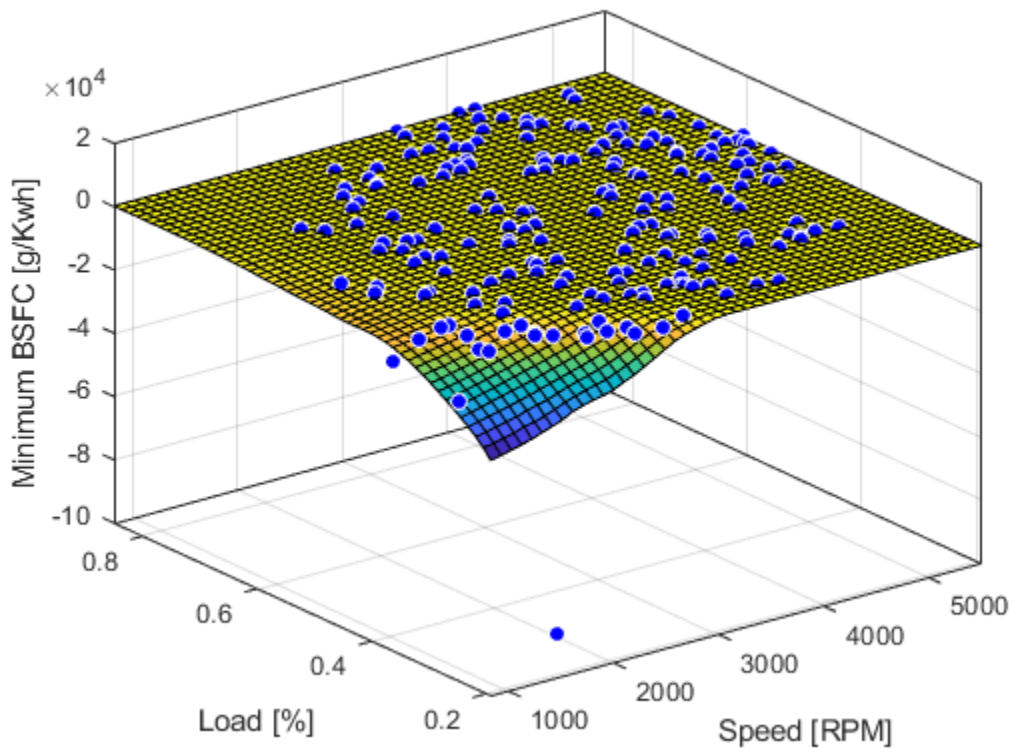
Remove Problem Points

Review the resulting plot.

There are points where BSFC is negative because this data is generated by an engine simulation.

Remove those problem data points by keeping points in the range [0, Inf].

```
out = excludedata( Speed, minBSFC, 'Range', [0, Inf] );
f2 = fit( [Speed, Load], minBSFC, 'Lowess', ...
    'Normalize', 'on', 'Exclude', out )
```

```
    Locally weighted smoothing linear regression:
      f2(x,y) = lowess (linear) smoothing regression computed from p
      where x is normalized by mean 3443 and std 1187
      and where y is normalized by mean 0.521 and std 0.175
    Coefficients:
      p = coefficient structure
```

Plot the new fit. Note that the excluded points are plotted as red crosses.

```
plot( f2, [Speed, Load], minBSFC, 'Exclude', out );
xlabel( 'Speed [RPM]' );
ylabel( 'Load [%]' );
zlabel( 'Minimum BSFC [g/Kwh]' );
```
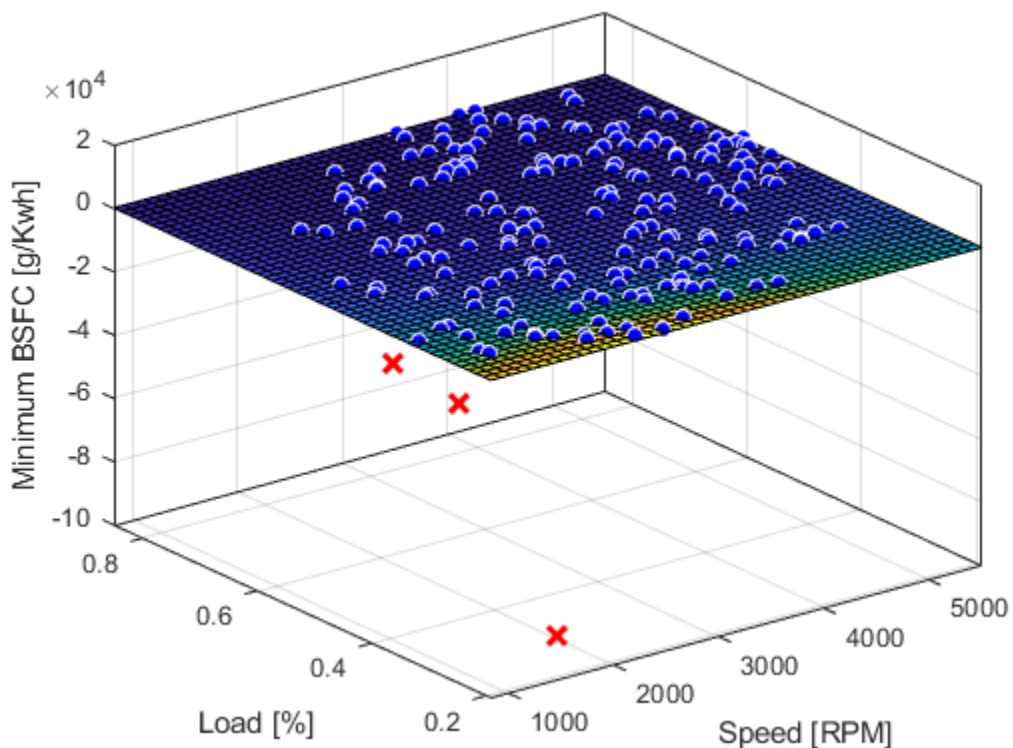
Zoom In

Zoom in on the part of the z-axis of interest.

```
zlim([0, max( minBSFC )])
```
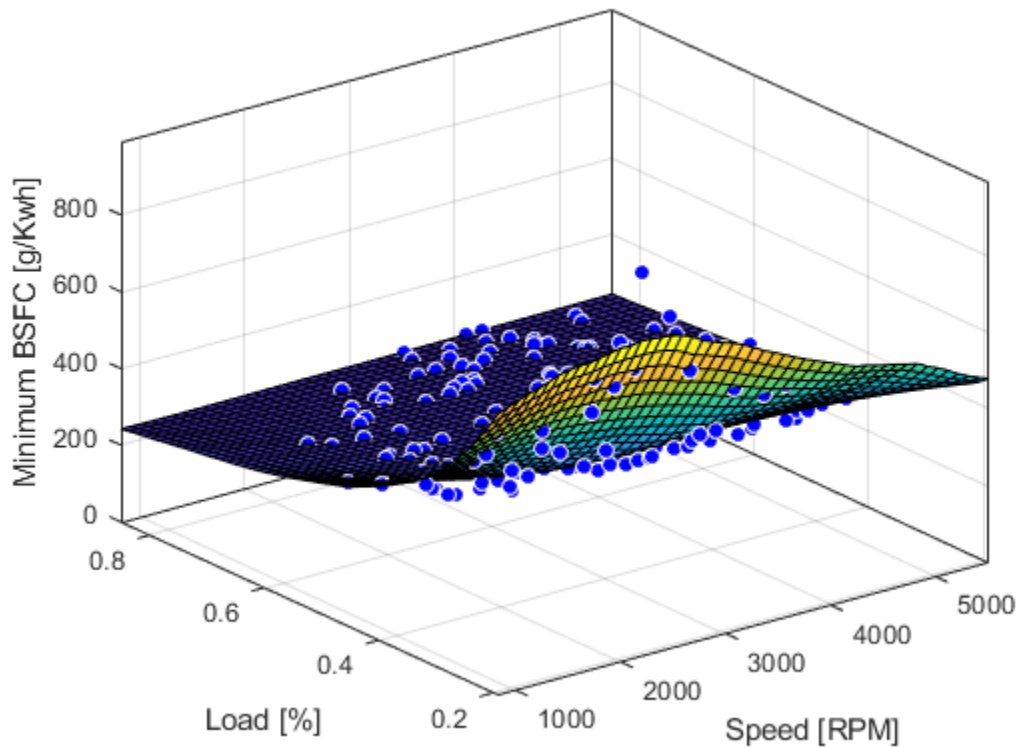

You want to operate the engine efficiently, so create a contour plot to see the region where the BSFC is low. Use the plot function, and specify the name/value parameter pair `'style','Contour'`.

```
plot( f2, [Speed, Load], minBSFC, 'Exclude', out, 'Style', 'Contour' );
xlabel( 'Speed [RPM]' );
ylabel( 'Load [%]' );
colorbar
```
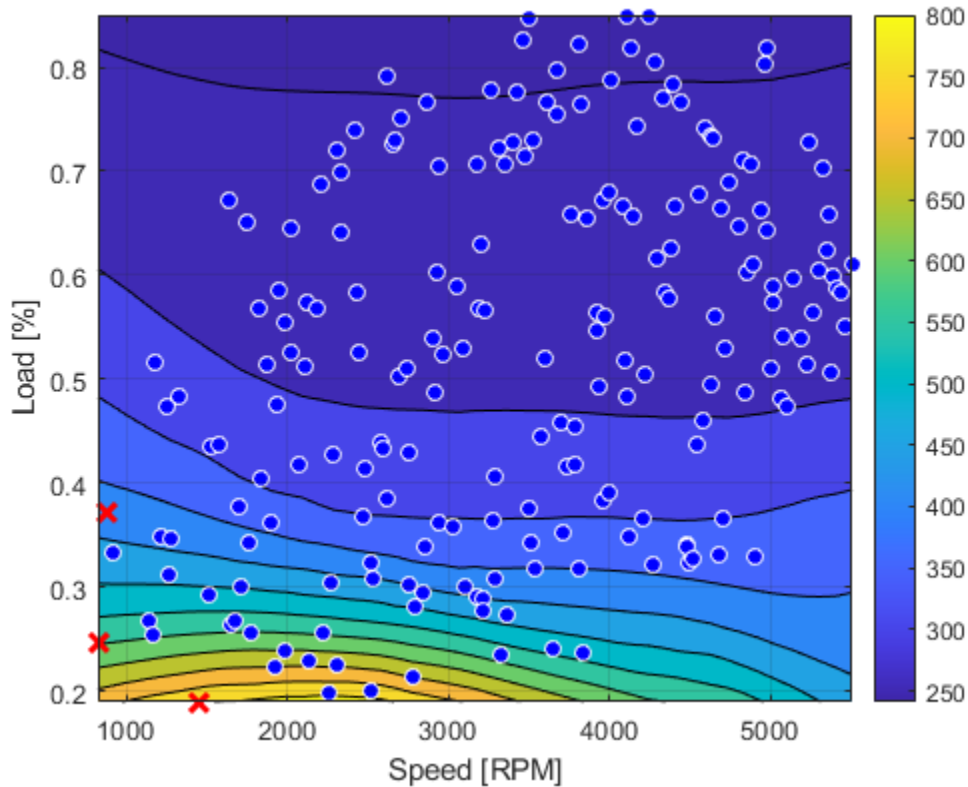
Create a Table from the Surface

Generate a table by evaluating the model f2 over a grid of points.

Create variables for the table breakpoints.

```
speedbreakpoints = linspace( 1000, 5500, 17 );
loadbreakpoints =  linspace( 0.2, 0.8, 13 );
```

To generate values for the table, evaluate the model over a grid of points.

```
[tSpeed, tLoad] = meshgrid( speedbreakpoints, loadbreakpoints );
tBSFC = f2( tSpeed, tLoad );
```

Examine the rows and columns of the table at the command line.

```
tBSFC(1:2:end,1:2:end)
```

```
ans =

  Columns 1 through 7

   722.3280   766.7608   779.4296   757.4574   694.5378   624.4095   576.5235
   503.9880   499.9201   481.7240   458.2803   427.7338   422.1099   412.1624
   394.7579   364.3421   336.1811   330.1550   329.1635   328.1810   329.1144
   333.7740   307.7736   295.1777   291.2068   290.3637   290.0173   287.8672
   295.9729   282.7567   273.8287   270.8869   269.8485   271.0547   270.5502
```

```
273.7512   264.5167   259.7631   257.9215   256.9350   258.3228   258.6638
251.5652   247.6746   247.2747   247.4699   247.3570   248.2433   248.8139

Columns 8 through 9

532.1533   466.9610
396.3209   398.0199
335.3871   346.3882
286.3077   291.0075
269.6837   272.2054
258.0298   260.5269
249.0083   250.4165
```

Plot the Table Against the Original Model

The grid on the model surface shows the table breakpoints.

```
h = plot( f2 );
h.EdgeColor = 'none';
hold on
mesh( tSpeed, tLoad, tBSFC, ...
    'LineStyle', '-', 'LineWidth', 2, 'EdgeColor', 'k', ...
    'FaceColor', 'none', 'FaceAlpha', 1 );
hold off
xlabel( 'Speed [RPM]' );
ylabel( 'Load [%]' );
zlabel( 'Minimum BSFC [g/Kwh]' );
```
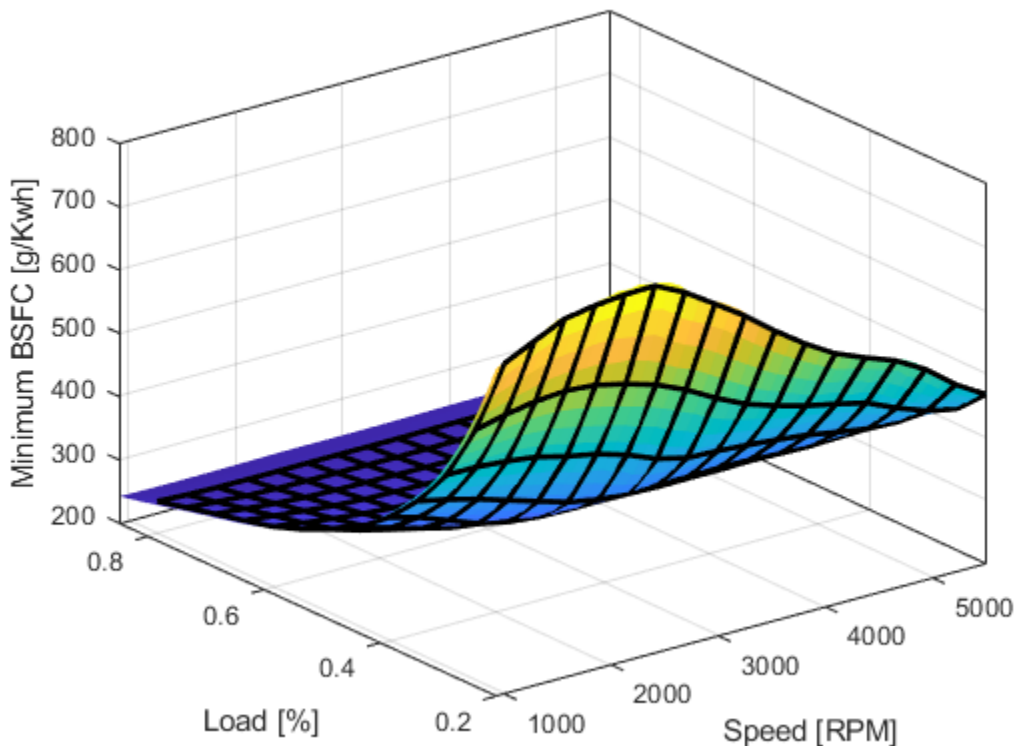
Check the Table Accuracy

View the difference between the model and the table by plotting the difference between them on a finer grid. Then, use this difference in prediction accuracy between the table and the model to determine the most efficient table size for your accuracy requirements.

The following code evaluates the model over a finer grid and plots the difference between the model and the table.

```
[tfSpeed, tfLoad] = meshgrid( ...
    linspace( 1000, 5500, 8*17+1 ), ...
    linspace( 0.2, 0.8, 8*13+1 ) );
tfBSFC_model = f2( tfSpeed, tfLoad );
tfBSFC_table = interp2( tSpeed, tLoad, tBSFC, tfSpeed, tfLoad, 'linear' );
tfDiff = tfBSFC_model - tfBSFC_table;

surf( tfSpeed, tfLoad, tfDiff, 'LineStyle', 'none'  );
hold on
mesh( tSpeed, tLoad, zeros( size( tBSFC ) ), ...
    'LineStyle', '-', 'LineWidth', 2, 'EdgeColor', 'k', ...
    'FaceColor', 'none', 'FaceAlpha', 1 );
hold off
axis tight
xlabel( 'Speed [RPM]' );
ylabel( 'Load [%]' );
zlabel( 'Difference between model and table [g/Kwh]' );
title( sprintf( 'Max difference: %g', max( abs( tfDiff(:) ) ) ) );
```
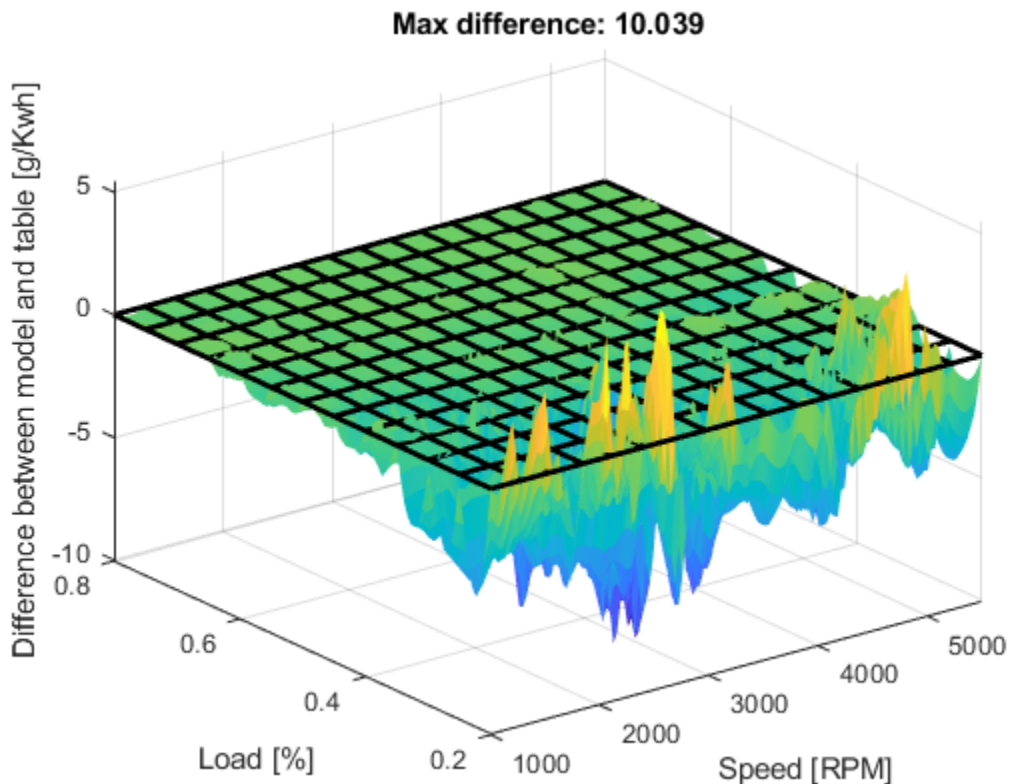
Create a Table Array Including Breakpoint Values

After creating a table by evaluating a model fit over a grid of points, it can be useful to export your table data from MATLAB. Before exporting, create a table array that includes the breakpoint values in the first row and column. The following command reshapes your data to this table format:

- X (`speedbreakpoints`) is a (1 x M) vector
- Y (`loadbreakpoints`) is an (N x 1) vector
- Z (`tBSFC`) is an (M x N) matrix

```
table = [
    {'Load\Speed'}, num2cell(speedbreakpoints(:).' )
    num2cell(loadbreakpoints (:) ), num2cell( tBSFC )
    ];
```

Export Table to Spreadsheet File

You can use the `xlswrite` function to export your table data to a new Excel Spreadsheet. Execute the following command to create a spreadsheet file.

```
xlswrite( 'tabledata.xlsx', table )
```

Create a Lookup Table Block

If you have Simulink™ software, you can create a Look Up Table block as follows. Execute the following code to try it out.

1. Create a model with a 2-D Lookup Table block.

```
simulink
new_system( 'my_model' )
open_system( 'my_model' )
add_block( 'Simulink/Lookup Tables/2-D Lookup Table', 'my_model/surfaceblock' )
```

2. Populate the Lookup Table with speed breakpoints, load breakpoints, and a lookup table.

```
set_param( 'my_model/surfaceblock',...
    'BreakpointsForDimension1', 'loadbreakpoints',...
    'BreakpointsForDimension2', 'speedbreakpoints',...
    'Table', 'tBSFC' );
```

3. Examine the populated Lookup Table block.

Filtering and Smoothing Data

In this section...

About Data Smoothing and Filtering

You can use the `smooth` function to smooth response data. You can use optional methods for moving average, Savitzky-Golay filters, and local regression with and without weights and robustness (`lowess`, `loess`, `rlowess` and `rloess`).

Moving Average Filtering

A moving average filter smooths data by replacing each data point with the average of the neighboring data points defined within the span. This process is equivalent to lowpass filtering with the response of the smoothing given by the difference equation

$$y_s(i) = \frac{1}{2N+1}(y(i+N) + y(i+N-1) + ... + y(i-N))$$

where $y_s(i)$ is the smoothed value for the `ith` data point, N is the number of neighboring data points on either side of $y_s(i)$, and $2N+1$ is the span.

The moving average smoothing method used by Curve Fitting Toolbox follows these rules:

- The span must be odd.
- The data point to be smoothed must be at the center of the span.
- The span is adjusted for data points that cannot accommodate the specified number of neighbors on either side.
- The end points are not smoothed because a span cannot be defined.

Note that you can use `filter` function to implement difference equations such as the one shown above. However, because of the way that the end points are treated, the toolbox moving average result will differ from the result returned by `filter`. Refer to Difference Equations and Filtering for more information.

For example, suppose you smooth data using a moving average filter with a span of 5. Using the rules described above, the first four elements of y_s are given by

```
ys(1) = y(1)
ys(2) = (y(1)+y(2)+y(3))/3
ys(3) = (y(1)+y(2)+y(3)+y(4)+y(5))/5
ys(4) = (y(2)+y(3)+y(4)+y(5)+y(6))/5
```

Note that $y_s(1)$, $y_s(2)$, ... ,$y_s(end)$ refer to the order of the data after sorting, and not necessarily the original order.

The smoothed values and spans for the first four data points of a generated data set are shown below.

Moving Average Smoothing
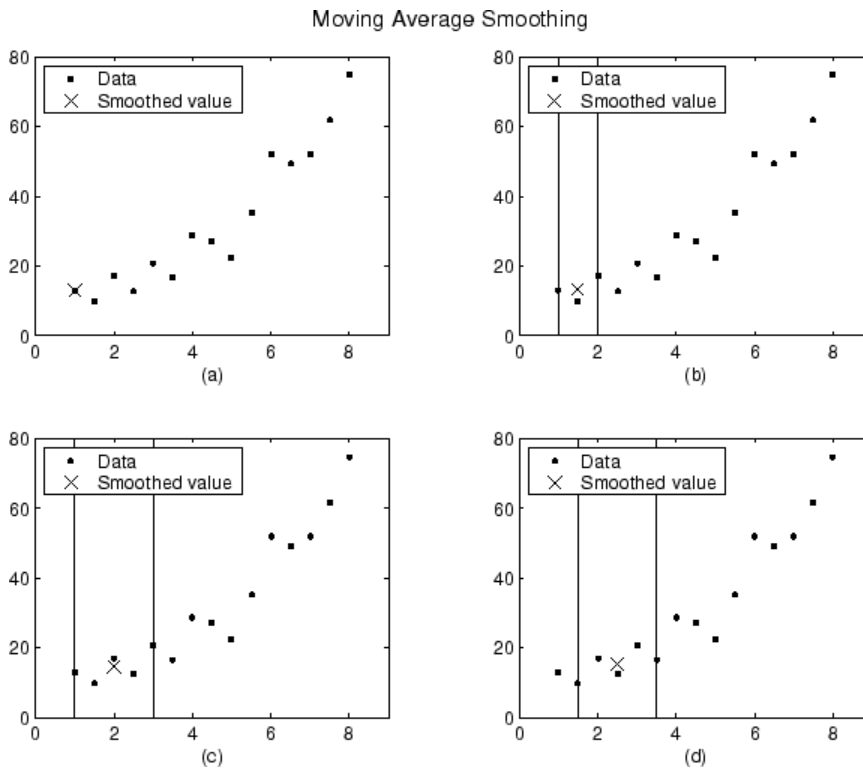
Plot (a) indicates that the first data point is not smoothed because a span cannot be constructed. Plot (b) indicates that the second data point is smoothed using a span of three. Plots (c) and (d) indicate that a span of five is used to calculate the smoothed value.

Savitzky-Golay Filtering

Savitzky-Golay filtering can be thought of as a generalized moving average. You derive the filter coefficients by performing an unweighted linear least-squares fit using a polynomial of a given degree. For this reason, a Savitzky-Golay filter is also called a digital smoothing polynomial filter or a least-squares smoothing filter. Note that a higher degree polynomial makes it possible to achieve a high level of smoothing without attenuation of data features.

The Savitzky-Golay filtering method is often used with frequency data or with spectroscopic (peak) data. For frequency data, the method is effective at preserving the high-frequency components of the signal. For spectroscopic data, the method is effective at preserving higher moments of the peak such as the line width. By comparison, the moving average filter tends to filter out a significant portion of the signal's high-frequency content, and it can only preserve the lower moments of a peak such as the centroid. However, Savitzky-Golay filtering can be less successful than a moving average filter at rejecting noise.

The Savitzky-Golay smoothing method used by Curve Fitting Toolbox software follows these rules:

- The span must be odd.

- The polynomial degree must be less than the span.

- The data points are not required to have uniform spacing.

    Normally, Savitzky-Golay filtering requires uniform spacing of the predictor data. However, the Curve Fitting Toolbox algorithm supports nonuniform spacing. Therefore, you are not required to perform an additional filtering step to create data with uniform spacing.

The plot shown below displays generated Gaussian data and several attempts at smoothing using the Savitzky-Golay method. The data is very noisy and the peak widths vary from broad to narrow. The span is equal to 5% of the number of data points.
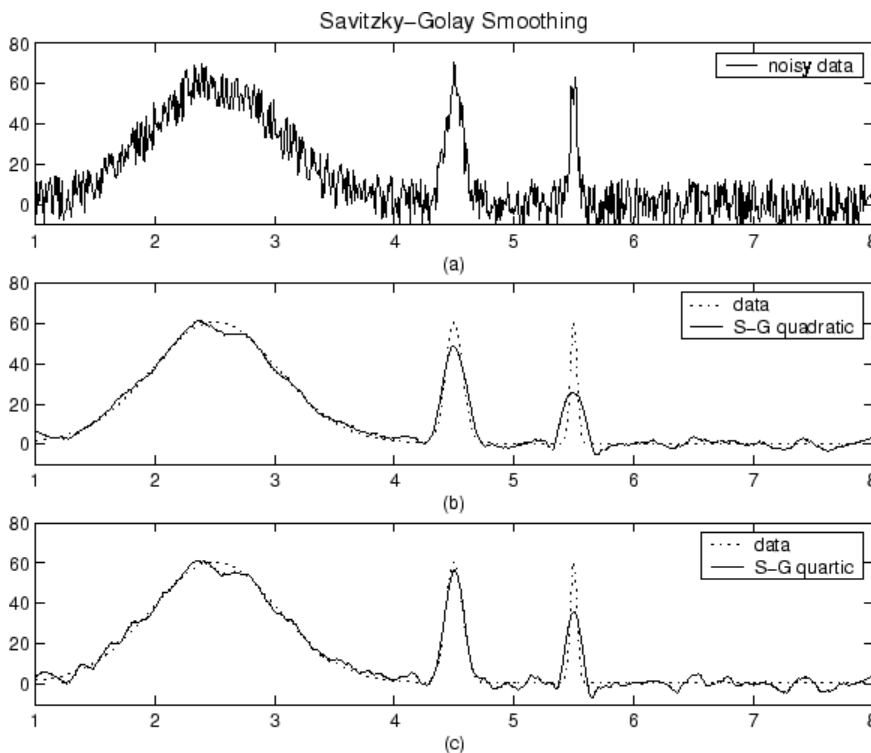
Plot (a) shows the noisy data. To more easily compare the smoothed results, plots (b) and (c) show the data without the added noise.

Plot (b) shows the result of smoothing with a quadratic polynomial. Notice that the method performs poorly for the narrow peaks. Plot (c) shows the result of smoothing with a quartic polynomial. In general, higher degree polynomials can more accurately capture the heights and widths of narrow peaks, but can do poorly at smoothing wider peaks.

Local Regression Smoothing

- "Lowess and Loess" on page 6-30
- "The Local Regression Method" on page 6-30
- "Robust Local Regression" on page 6-32

Lowess and Loess

The names "lowess" and "loess" are derived from the term "locally weighted scatter plot smooth," as both methods use locally weighted linear regression to smooth data.

The smoothing process is considered local because, like the moving average method, each smoothed value is determined by neighboring data points defined within the span. The process is weighted because a regression weight function is defined for the data points contained within the span. In addition to the regression weight function, you can use a robust weight function, which makes the process resistant to outliers. Finally, the methods are differentiated by the model used in the regression: lowess uses a linear polynomial, while loess uses a quadratic polynomial.

The local regression smoothing methods used by Curve Fitting Toolbox software follow these rules:

- The span can be even or odd.
- You can specify the span as a percentage of the total number of data points in the data set. For example, a span of 0.1 uses 10% of the data points.

The Local Regression Method

The local regression smoothing process follows these steps for each data point:

1. Compute the *regression weights* for each data point in the span. The weights are given by the tricube function shown below.

$$w_i = \left(1 - \left|\frac{x - x_i}{d(x)}\right|^3\right)^3$$

   x is the predictor value associated with the response value to be smoothed, x_i are the nearest neighbors of x as defined by the span, and $d(x)$ is the distance along the abscissa from x to the most distant predictor value within the span. The weights have these characteristics:

   - The data point to be smoothed has the largest weight and the most influence on the fit.
   - Data points outside the span have zero weight and no influence on the fit.

2. A weighted linear least-squares regression is performed. For lowess, the regression uses a first degree polynomial. For loess, the regression uses a second degree polynomial.

3. The smoothed value is given by the weighted regression at the predictor value of interest.

If the smooth calculation involves the same number of neighboring data points on either side of the smoothed data point, the weight function is symmetric. However, if the number of neighboring points is not symmetric about the smoothed data point, then the weight function is not symmetric. Note that unlike the moving average smoothing process, the span never changes. For example, when you smooth the data point with the smallest predictor value, the shape of the weight function is truncated by one half, the leftmost data point in the span has the largest weight, and all the neighboring points are to the right of the smoothed value.

The weight function for an end point and for an interior point is shown below for a span of 31 data points.

Local Regression Weight Function

The weight function for the leftmost data point

The weight function for an interior data point
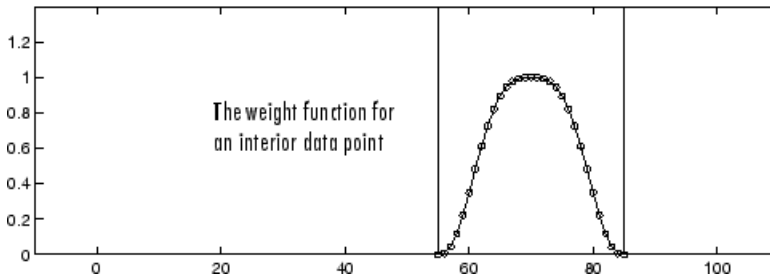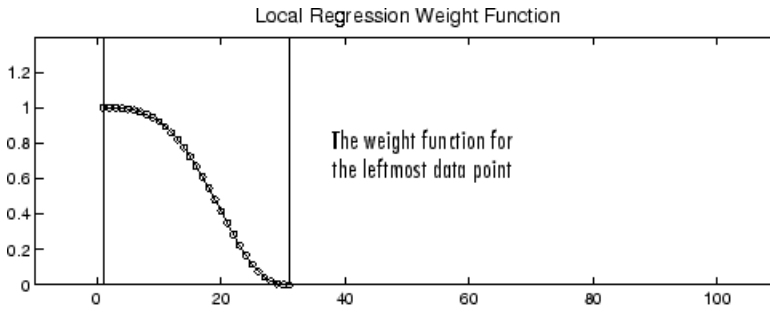
Using the lowess method with a span of five, the smoothed values and associated regressions for the first four data points of a generated data set are shown below.

Lowess Smoothing

(a)

(b)

(c)

(d)
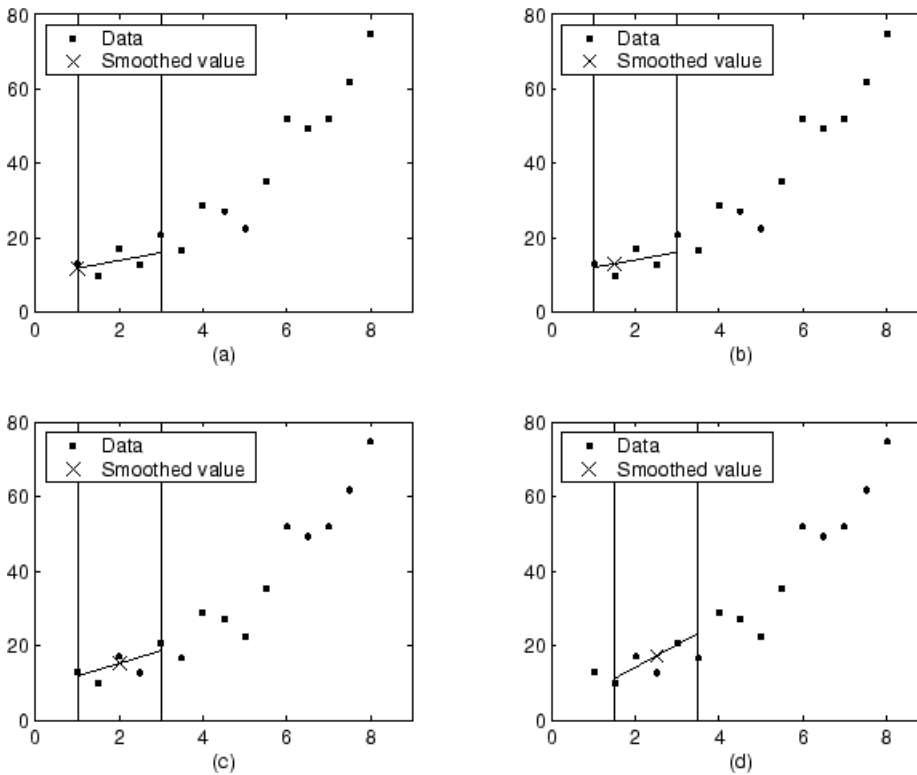
Notice that the span does not change as the smoothing process progresses from data point to data point. However, depending on the number of nearest neighbors, the regression weight function might

not be symmetric about the data point to be smoothed. In particular, plots (a) and (b) use an asymmetric weight function, while plots (c) and (d) use a symmetric weight function.

For the loess method, the graphs would look the same except the smoothed value would be generated by a second-degree polynomial.

Robust Local Regression

If your data contains outliers, the smoothed values can become distorted, and not reflect the behavior of the bulk of the neighboring data points. To overcome this problem, you can smooth the data using a robust procedure that is not influenced by a small fraction of outliers. For a description of outliers, refer to "Residual Analysis" on page 7-48.

Curve Fitting Toolbox software provides a robust version for both the lowess and loess smoothing methods. These robust methods include an additional calculation of robust weights, which is resistant to outliers. The robust smoothing procedure follows these steps:

1 Calculate the residuals from the smoothing procedure described in the previous section.

2 Compute the *robust weights* for each data point in the span. The weights are given by the bisquare function,

$$w_i = \begin{cases} \left(1 - (r_i/6MAD)^2\right)^2, & |ri| < 6MAD, \\ 0, & |ri| \geq 6MAD, \end{cases}$$

where r_i is the residual of the ith data point produced by the regression smoothing procedure, and MAD is the median absolute deviation of the residuals,

$$MAD = \text{median}(|r|).$$

The median absolute deviation is a measure of how spread out the residuals are. If r_i is small compared to $6MAD$, then the robust weight is close to 1. If r_i is greater than $6MAD$, the robust weight is 0 and the associated data point is excluded from the smooth calculation.

3 Smooth the data again using the robust weights. The final smoothed value is calculated using both the local regression weight and the robust weight.

4 Repeat the previous two steps for a total of five iterations.

The smoothing results of the lowess procedure are compared below to the results of the robust lowess procedure for a generated data set that contains a single outlier. The span for both procedures is 11 data points.

Robust Lowess Smoothing
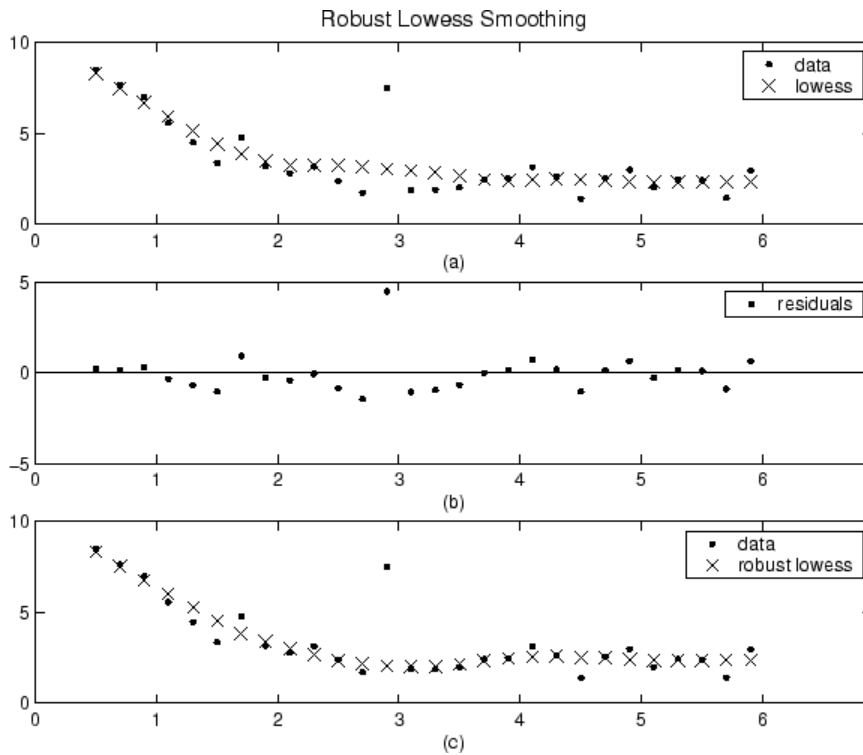
Plot (`a`) shows that the outlier influences the smoothed value for several nearest neighbors. Plot (`b`) suggests that the residual of the outlier is greater than six median absolute deviations. Therefore, the robust weight is zero for this data point. Plot (`c`) shows that the smoothed values neighboring the outlier reflect the bulk of the data.

Example: Smoothing Data

Load the data in `count.dat`:

```
load count.dat
```

The 24-by-3 array `count` contains traffic counts at three intersections for each hour of the day.

First, use a moving average filter with a 5-hour span to smooth all of the data at once (by linear index):

```
c = smooth(count(:));
C1 = reshape(c,24,3);
```

Plot the original data and the smoothed data:

```
subplot(3,1,1)
plot(count,':');
hold on
plot(C1,'-');
title('Smooth C1 (All Data)')
```

Second, use the same filter to smooth each column of the data separately:

```
C2 = zeros(24,3);
for I = 1:3,
```

```
    C2(:,I) = smooth(count(:,I));
end
```

Again, plot the original data and the smoothed data:

```
subplot(3,1,2)
plot(count,':');
hold on
plot(C2,'-');
title('Smooth C2 (Each Column)')
```

Plot the difference between the two smoothed data sets:

```
subplot(3,1,3)
plot(C2 - C1,'o-')
title('Difference C2 - C1')
```
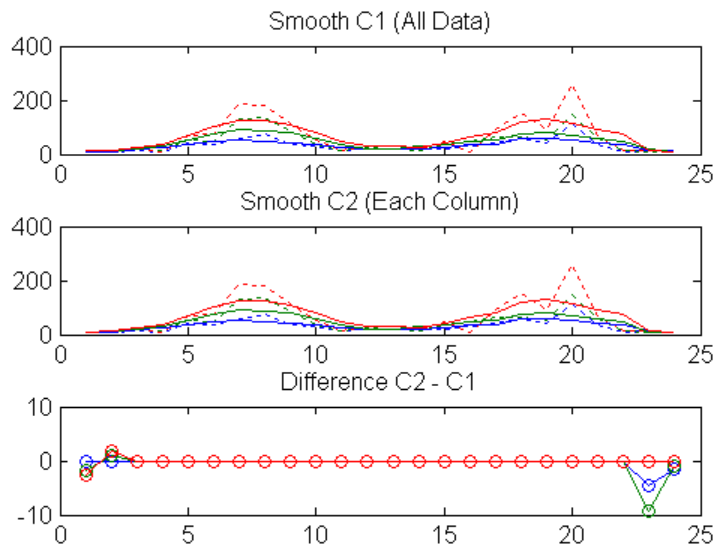

Note the additional end effects from the 3-column smooth.

Example: Smoothing Data Using Loess and Robust Loess

Create noisy data with outliers:

```
x = 15*rand(150,1);
y = sin(x) + 0.5*(rand(size(x))-0.5);
y(ceil(length(x)*rand(2,1))) = 3;
```

Smooth the data using the `loess` and `rloess` methods with a span of 10%:

```
yy1 = smooth(x,y,0.1,'loess');
yy2 = smooth(x,y,0.1,'rloess');
```

Plot original data and the smoothed data.

```
[xx,ind] = sort(x);
subplot(2,1,1)
plot(xx,y(ind),'b.',xx,yy1(ind),'r-')
set(gca,'YLim',[-1.5 3.5])
```

```
legend('Original Data','Smoothed Data Using ''loess''',...
        'Location','NW')
subplot(2,1,2)
plot(xx,y(ind),'b.',xx,yy2(ind),'r-')
set(gca,'YLim',[-1.5 3.5])
legend('Original Data','Smoothed Data Using ''rloess''',...
        'Location','NW')
```
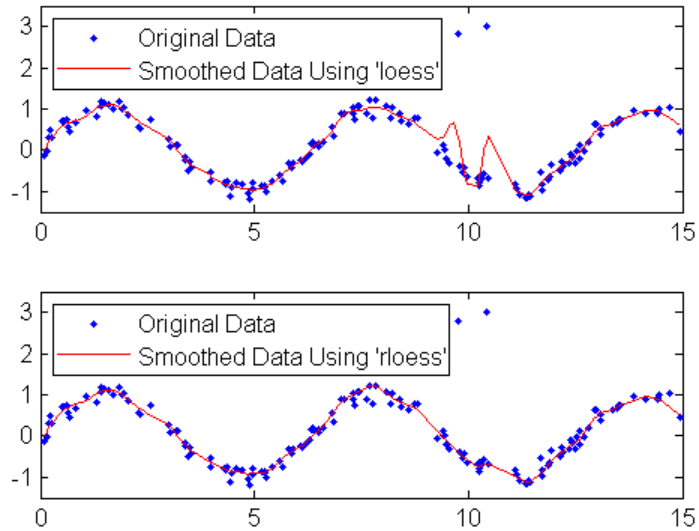

Note that the outliers have less influence on the robust method.

See Also

`smooth`

Related Examples

- "Nonparametric Fitting" on page 6-2
- "Smoothing Splines" on page 6-9
- "Lowess Smoothing" on page 6-16

7

Fit Postprocessing

Explore and Customize Plots

In this section...
"Displaying Fit and Residual Plots" on page 7-2
"Viewing Surface Plots and Contour Plots" on page 7-3
"Using Zoom, Pan, Data Cursor, and Outlier Exclusion" on page 7-4
"Customizing the Fit Display" on page 7-5
"Print to MATLAB Figures" on page 7-6

Displaying Fit and Residual Plots

Within each fit figure, you can display up to three plots simultaneously to examine the fit. Use the toolbar or **View** menu to select the type of plot to display:

- **Main Plot** shows the curve or surface fit.
- **Residuals Plot** shows the errors between your fit and your data
- **Contour Plot** shows a contour map of a surface fit (not available for curve fits).

The next example shows a main plot with a curve fit and prediction bounds, and the residuals plot.
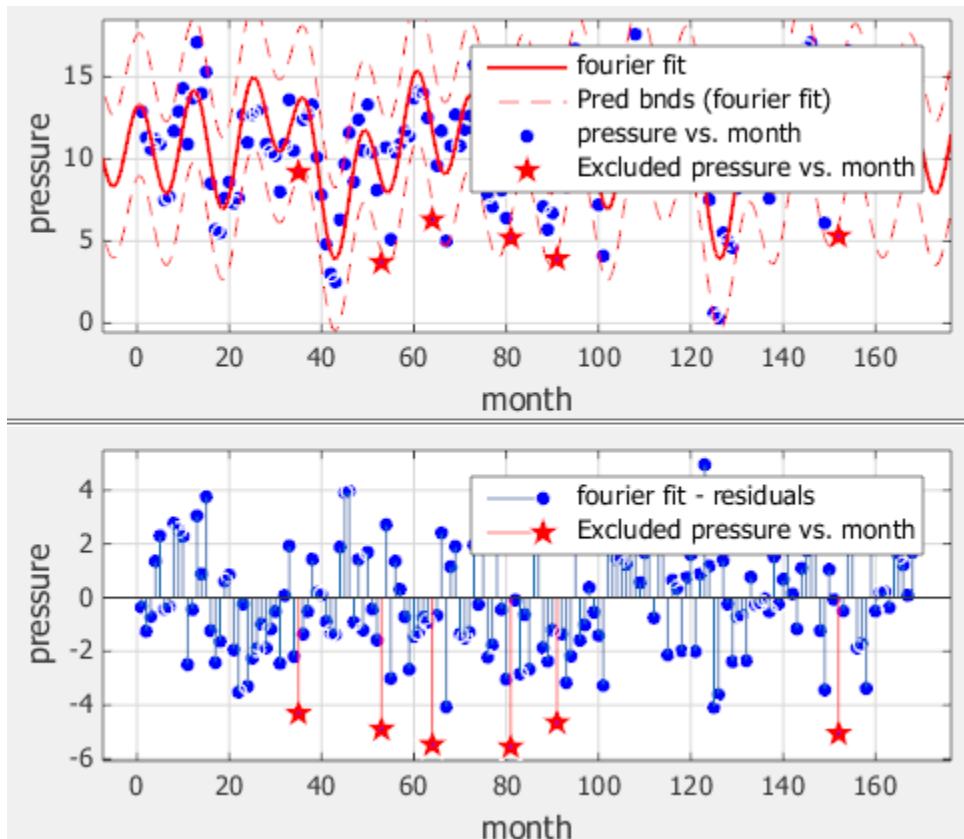
When you select **Tools > Prediction Bounds**, two additional curves (or surfaces) are plotted to show the prediction bounds on both sides of your model fit.

Choose which bounds to display: `None`, `90`%, `95`%, `99`%, or `Custom`. The custom option opens a dialog box where you can enter the required confidence level.

See also "Customizing the Fit Display" on page 7-5.

Residuals Plot

On the residuals plot, you can view the errors between your fit and your data, and you can remove outliers. See "Remove Outliers" on page 7-8. This example shows a residuals plot with some excluded outliers.
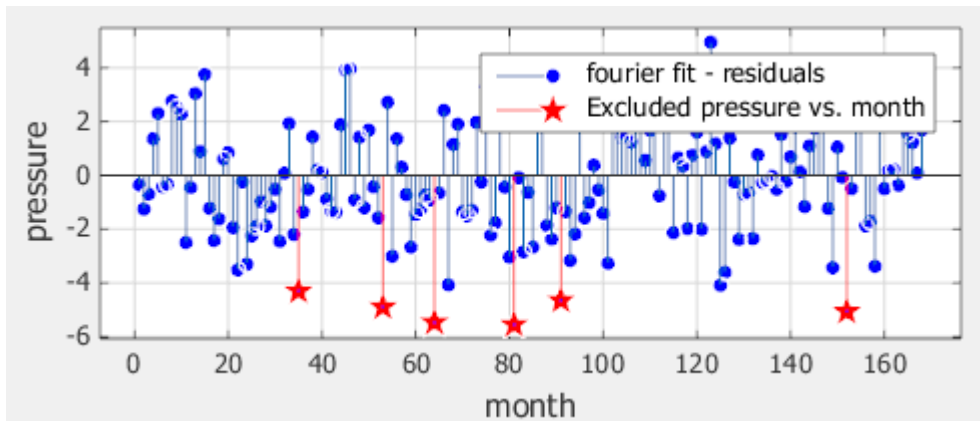
Viewing Surface Plots and Contour Plots

If you fit a surface, then the main plot shows your surface fit. Click-and-drag rotation or **Rotate 3D** is the default mouse mode for surface plots in the Curve Fitting app. Rotate mode in the Curve Fitting app is the same as **Rotate 3D** in MATLAB figures. You can change the mouse mode for manipulating plots just as for curve plots. See "Using Zoom, Pan, Data Cursor, and Outlier Exclusion" on page 7-4.

Tip To return to rotate mode, turn off any other mouse mode.

If you turn on a mouse mode for zoom, pan, data cursor or exclude outliers, turn the mode off again to return to rotate mode. For example, click the Zoom in toolbar button a second time to clear it and return to rotate mode.

If you have a surface fit, use the contour plot to examine a contour map of your surface. Contour plots are not available for curve fits. On a surface fit, a contour plot makes it easier to see points that have the same height. An example follows.
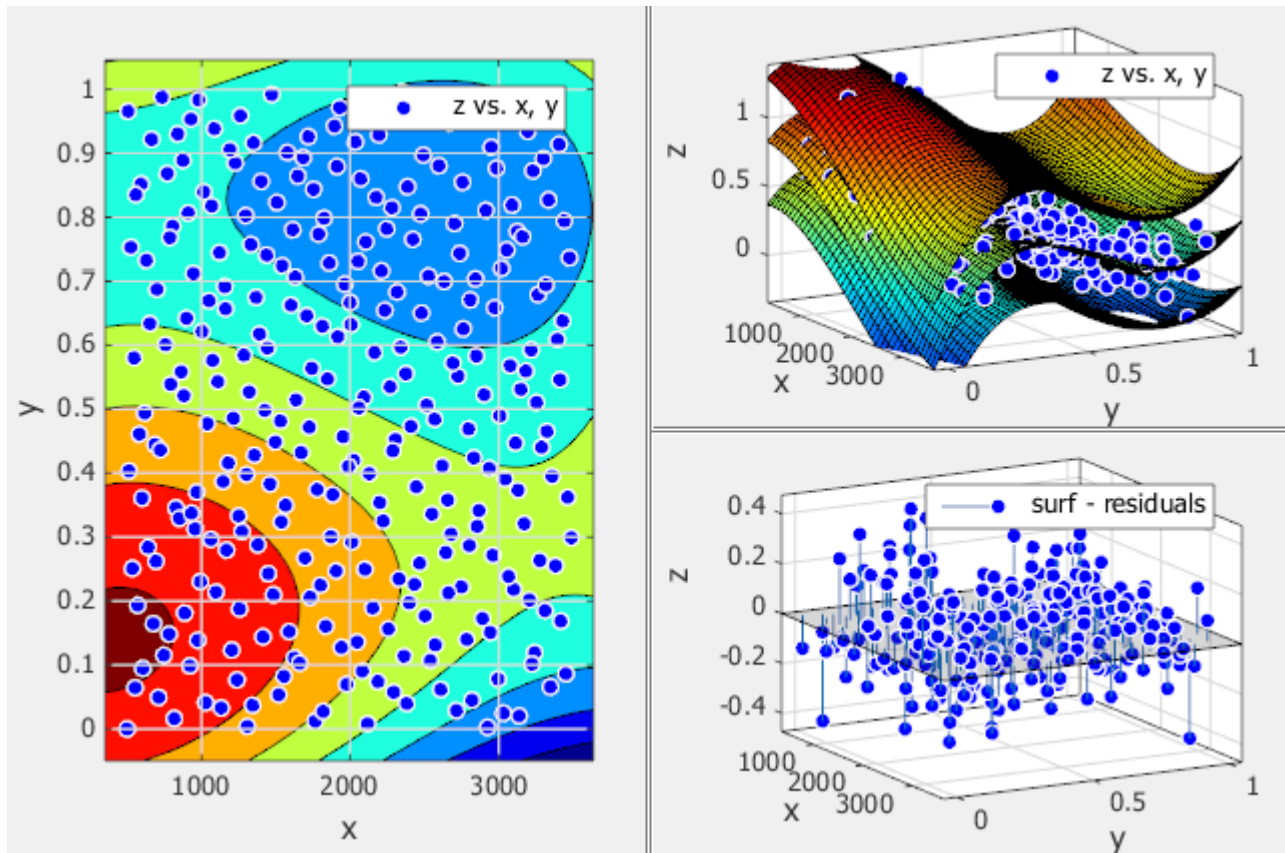
For polynomial and custom fits, you also can use the **Tools** menu to display prediction bounds. When you display prediction bounds, two additional surfaces are plotted to show the prediction bounds on both sides of your model fit. The previous example shows prediction bounds. You can see three surfaces on the plot. The top and bottom surfaces show the prediction bounds at the specified confidence level on either side of your model fit surface.

You can also switch your surface plot to a 2-D plot if desired. Your plot cursor must be in rotate mode. Clear any other mouse mode if necessary. Then, right-click the plot to select **X-Y**, **X-Z**, or **Y-Z** view, or to select **Rotate Options**. All these context menu options are standard MATLAB 3-D plot tools. For more information, see "Interactively Explore Plotted Data".

Using Zoom, Pan, Data Cursor, and Outlier Exclusion

You can change mouse mode for manipulating plots. Use the toolbar or **Tools** menu to switch to **Zoom**, **Pan**, **Data Cursor**, or **Exclude Outliers** modes.

The Curve Fitting app remembers your selected mouse mode in each fit figure within a session.

Use the toolbar or **Tools** menu to toggle mouse mode in your plots:

- **Zoom In**, **Zoom Out**, **Pan**, and **Data Cursor** are standard MATLAB plot tools.

  On surfaces, turn all these modes off to return to **Rotate 3D** mode. For surface plots, rotation is the default mouse mode in the Curve Fitting app. See "Viewing Surface Plots and Contour Plots" on page 7-3.

- — **Data Cursor** selects data cursor mode, where you can click points to display input and output values.

- — **Exclude Outliers** selects outlier mode, where you can click points to remove or include in your fit. Exclude outliers is a mouse mode for graphically excluding data from your fit. See "Remove Outliers" on page 7-8.

Customizing the Fit Display

To customize your plot display, use the toolbar, **Tools** menu, or the **View** menu. See also "Create Multiple Fits in Curve Fitting App" on page 2-11.

Tools Menu and Toolbar

- — **Legend** toggles display of the legend on all plots in the currently selected fit tab.

- — **Grid** toggles display of the grid on all plots in the currently selected fit tab.

- **Tools** > **Prediction Bounds** lets you choose which bounds to display: None, 90%, 95%, 99%, or Custom. The custom option opens a dialog box where you can enter the required confidence level.

- — **Axes Limits** opens a dialog box where you can specify upper and lower bounds for the X- and Y-axes of plots. Click **Reset** to return to the default axes limits.

View Menu and Toolbar

Use the View controls to customize the display to show or hide the plots, fit settings, results and table of fits.

- Available in the **View** menu and the toolbar:

  - — **Main Plot** toggles the display of the main fit plot in the currently selected fit figure. This item is disabled if only the main plot is displayed.

  - — **Residuals Plot** toggles the display of the residuals plot in the currently selected fit tab. This item is disabled if only the residuals plot is displayed.

  - — **Contour Plot** toggles the display of the contour plot in the currently selected fit tab. This item is disabled if only the contour plot is displayed.

- **View** > **Fit Settings** toggles the display of the fit controls pane in the currently selected fit tab (**Fit name**, inputs, fit type, and so on).

- **View** > **Fit Results** toggles the display of the **Results** pane in the currently selected fit tab. When you display the **Results** pane, you can see model terms and coefficients, goodness-of-fit statistics, and information messages about the fit.

- **View** > **Table of Fits** toggles the display of the **Table of Fits** pane in the Curve Fitting app.

Tip For more space to view and compare plots, as shown next, use the **View** menu to hide or show the **Fit Settings**, **Fit Results**, or **Table of Fits** panes.
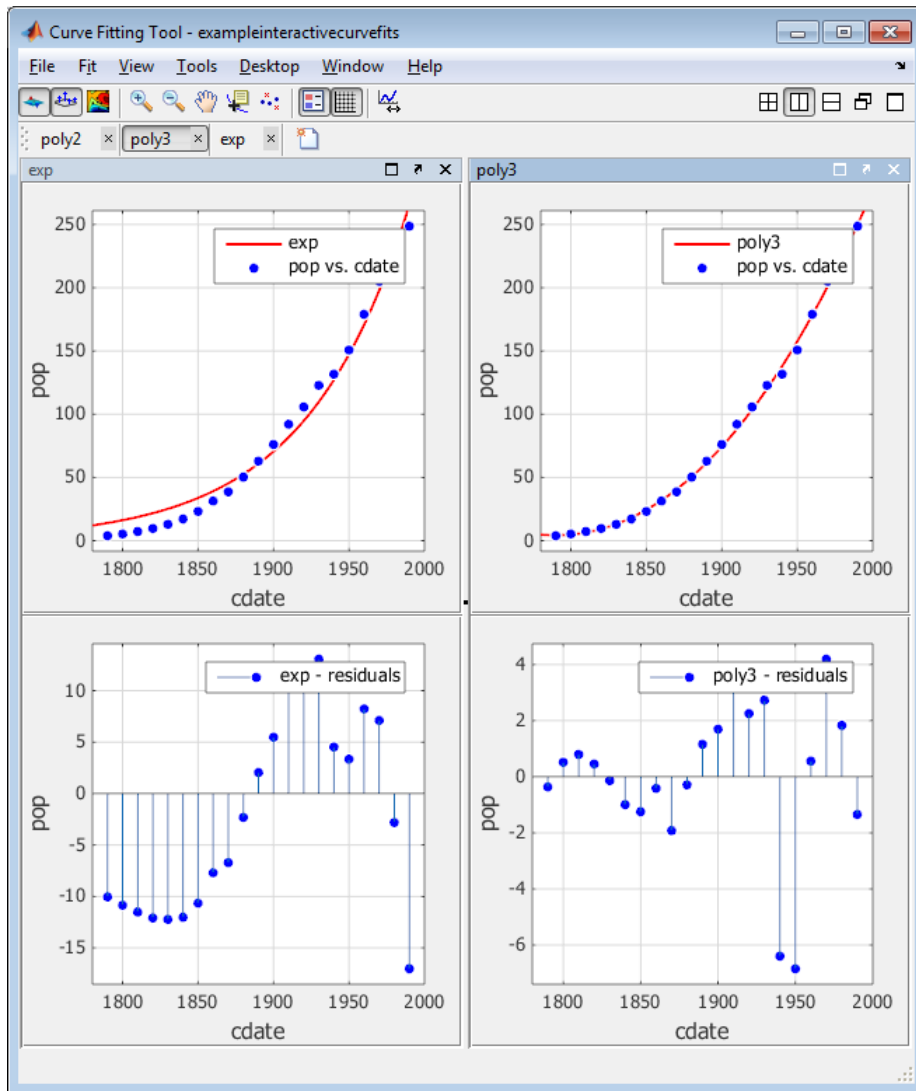
See also "Displaying Multiple Fits Simultaneously" on page 2-12.

Print to MATLAB Figures

In the Curve Fitting app, select **File > Print to Figure** to produce MATLAB figures from the results of curve fitting. **Print to Figure** creates a figure containing all plots for the current fit. You can then use the interactive plotting tools to edit the figures showing fitting results for presentation purposes, and export these in different formats.

See Also

Related Examples

- "Remove Outliers" on page 7-8
- "Compare Fits in Curve Fitting App" on page 2-16

- "Create Multiple Fits in Curve Fitting App" on page 2-11
- "Generate Code and Export Fits to the Workspace" on page 7-13
- "Compare Fits Programmatically" on page 7-32

Remove Outliers

In this section...

Remove Outliers Interactively

To remove outliers in the Curve Fitting app, follow these steps:

1 Select **Tools > Exclude Outliers** or click the toolbar button .

When you move the mouse cursor to the plot, it changes to a cross-hair to show you are in outlier selection mode.

2 Click a point that you want to exclude in the main plot or residuals plot. Alternatively, click and drag to define a rectangle and remove all enclosed points.

A removed plot point becomes a red cross in the plots. If you have **Auto-fit** selected, the Curve Fitting app refits the surface without the point. Otherwise, you can click **Fit** to refit.

3 Repeat for all points you want to exclude.

When removing outliers from surface fits, it can be helpful to display a 2-D residuals plot for examining and removing outliers. With your plot cursor in rotation mode, right-click the plot to select **X-Y**, **X-Z**, or **Y-Z** view.

To replace individual excluded points in the fit, click an excluded point again in **Exclude Outliers** mode. To replace all excluded points in the fit, right-click and select **Clear all exclusions**.

In surface plots, to return to rotation mode, click the **Exclude outliers** toolbar button again to turn off outlier selection mode.

Exclude Data Ranges

To exclude sections of data by range in the Curve Fitting app, follow these steps:

1 Select **Tools > Exclude By Rule**.

2 Specify data to exclude. Enter numbers in any of the boxes to define beginning or ending intervals to exclude in the X, Y, or Z data.

3 Press **Enter** to apply the exclusion rule.

Curve Fitting app displays shaded pink areas on the plots to show excluded ranges. Excluded points become red.

Remove Outliers Programmatically

This example shows how to remove outliers when curve fitting programmatically, using the 'Exclude' name/value pair argument with the fit or fitoptions functions. You can plot excluded data by supplying an Exclude or outliers argument with the plot function.

Exclude Data Using a Simple Rule

For a simple example, load data and fit a Gaussian, excluding some data with an expression, then plot the fit, data and the excluded points.

```
[x, y] = titanium;
f1 = fit(x',y','gauss2', 'Exclude', x<800);
plot(f1,x,y,x<800)
```
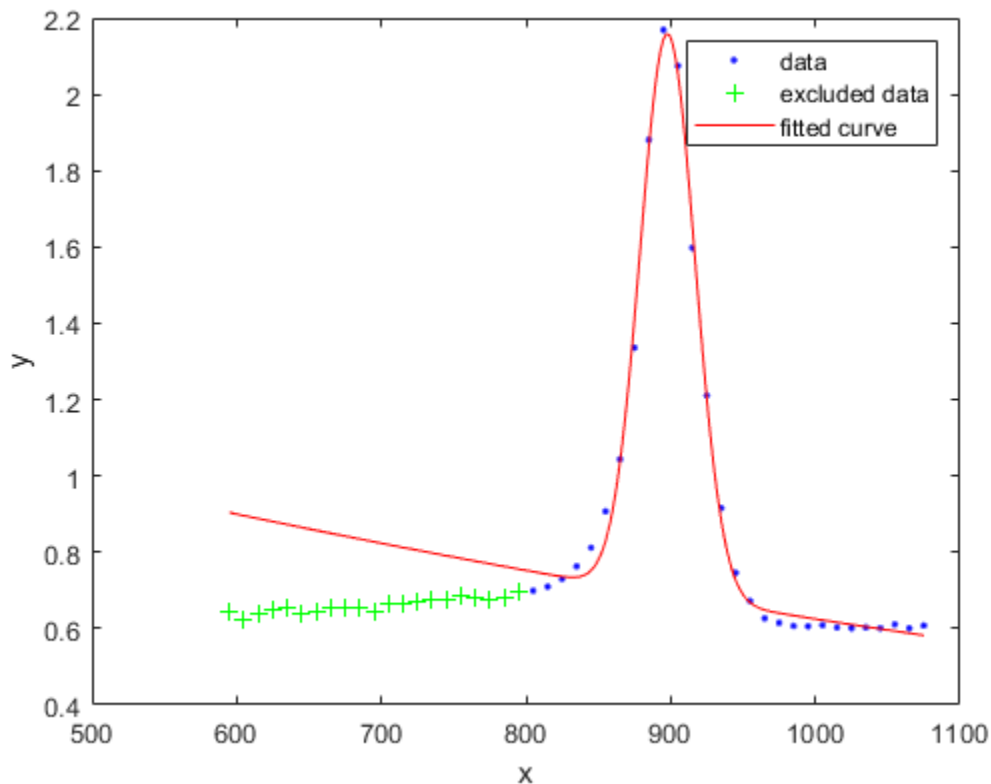

Exclude Data by Distance from the Model

It can be useful to exclude outliers by distance from the model, using standard deviations. The following example shows how to identify outliers using distance greater than 1.5 standard deviations from the model, and compares with a robust fit which gives lower weight to outliers.

Create a baseline sinusoidal signal:

```
xdata = (0:0.1:2*pi)';
y0 = sin(xdata);
```

Add noise to the signal with non-constant variance:

```
% Response-dependent Gaussian noise
gnoise = y0.*randn(size(y0));

% Salt-and-pepper noise
spnoise = zeros(size(y0));
p = randperm(length(y0));
sppoints = p(1:round(length(p)/5));
spnoise(sppoints) = 5*sign(y0(sppoints));

ydata = y0 + gnoise + spnoise;
```

Fit the noisy data with a baseline sinusoidal model:

```
f = fittype('a*sin(b*x)');
fit1 = fit(xdata,ydata,f,'StartPoint',[1 1]);
```

Identify "outliers" as points at a distance greater than 1.5 standard deviations from the baseline model, and refit the data with the outliers excluded:

```
fdata = feval(fit1,xdata);
I = abs(fdata - ydata) > 1.5*std(ydata);
outliers = excludedata(xdata,ydata,'indices',I);

fit2 = fit(xdata,ydata,f,'StartPoint',[1 1],...
           'Exclude',outliers);
```

Compare the effect of excluding the outliers with the effect of giving them lower bisquare weight in a robust fit:

```
fit3 = fit(xdata,ydata,f,'StartPoint',[1 1],'Robust','on');
```

Plot the data, the outliers, and the results of the fits:

```
plot(fit1,'r-',xdata,ydata,'k.',outliers,'m*')
hold on
plot(fit2,'c--')
plot(fit3,'b:')
xlim([0 2*pi])
```

See Also

excludedata | fit

Related Examples

- "Explore and Customize Plots" on page 7-2

Select Validation Data

To specify validation data for the currently selected fit, follow these steps:

1 Select **Fit > Specify Validation Data**. The Specify Validation Data dialog box opens.

2 Select variables for **X data** and **Y data** (and **Z data** for surfaces).

When you select two or three variables, depending on whether your fit data is for a curve or a surface, the tool calculates validation statistics (SSE and RMSE) and displays them in the **Results** pane and the **Table of Fits**. For definitions of these statistics, see "Using the Statistics in the Table of Fits" on page 2-13. Your validation data points display on the main plot and residual plot along with the original data.

3 Close the dialog box.

Generate Code and Export Fits to the Workspace

In this section...
"Generating Code from the Curve Fitting App" on page 7-13
"Exporting a Fit to the Workspace" on page 7-14

Generating Code from the Curve Fitting App

You can generate and use MATLAB code from an interactive session in the Curve Fitting app. In this way, you can transform your interactive analysis into reusable functions for batch processing of multiple data sets. You can use the generated file without modification, or you can edit and customize the file as needed.

To generate code for all fits and plots in your Curve Fitting app session follow these steps:

1 Select **File > Generate Code**.

The Curve Fitting app generates code from your session and displays the file in the MATLAB Editor. The file includes all fits and plots in your current session. The file captures the following information:

- Names of fits and their variables
- Fit settings and options
- Plots
- Curve and surface fitting objects and methods used to create the fits:

  - A cell-array of `cfit` or `sfit` objects representing the fits
  - A structure array with goodness-of fit information.

2 Save the file.

To recreate your fits and plots, call the file from the command line with your original data as input arguments. You also can call the file with new data.

For example, enter:

```
[fitresult, gof] = myFileName(a, b, c)
```

where `a`, `b`, and `c` are your variable names, and *myFileName* is the file name.

Calling the file from the command line does *not* recreate your Curve Fitting app and session. When you call the file, you get the same plots you had in your Curve Fitting app session in standard MATLAB figure windows. There is one window for each fit. For example, if your fit in the Curve Fitting app session displayed main, residual and contour plots, all three plots appear in a single figure window.

Curve Fitting Functions

The curve and surface fit objects (`cfit` and `sfit`) store the results from a fitting operation, making it easy to plot and analyze fits at the command line.

To learn about available functions for working with fits, see "Curve and Surface Fitting" on page 3-2.

Exporting a Fit to the Workspace

To export a fit to the MATLAB workspace, follow these steps:

1 Select a fit and save it to the MATLAB workspace using one of these methods:

- Right-click the fit listed in the Table of Fits and select **Save *myfitname* to Workspace**
- Select a fit figure in the Curve Fitting app and select **Fit > Save to Workspace**.

The Save Fit to MATLAB Workspace dialog box opens.
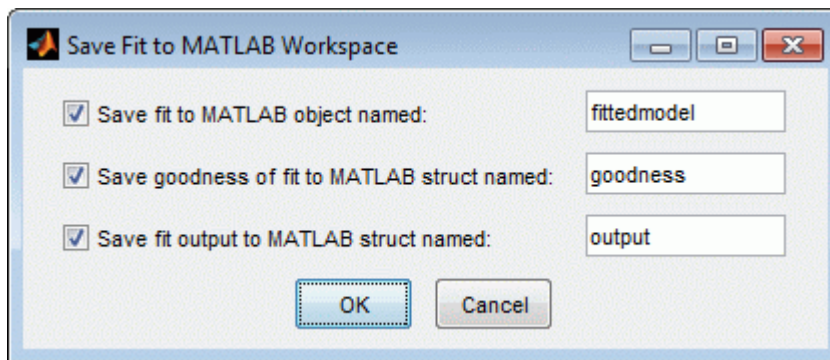
2 Edit the names as appropriate. If you previously exported fits, the toolbox automatically adds a numbered suffix to the default names so there is no danger of overwriting them.

3 Choose which options you want to export by selecting the check boxes. Check box options are as follows:

- **Save fit to MATLAB object named *fittedmodel*** — This option creates a `cfit` or `sfit` object, that encapsulates the result of fitting a curve or surface to data. You can examine the fit coefficients at the command line, for example:

```
fittedmodel
   Linear model Poly22:
      fittedmodel1(x,y) = p00 + p10*x + p01*y + p20*x^2...
                          + p11*x*y + p02*y^2
   Coefficients (with 95% confidence bounds):
     p00 =        302.1  (247.3, 356.8)
     p10 =        -1395  (-1751, -1039)
     p01 =      0.03525  (0.01899, 0.05151)
     p20 =         1696  (1099, 2293)
     p11 =      -0.1119  (-0.1624, -0.06134)
     p02 =    2.36e-006  (-8.72e-007, 5.593e-006)
```

  You also can treat the `cfit` or `sfit` object as a function to make predictions or evaluate the fit at values of X (or X and Y). See the `cfit` and `sfit` reference page.

- **Save goodness of fit to MATLAB struct named *goodness*** — This option creates a structure array that contains statistical information about the fit, for example:

```
goodness =
          sse: 0.0234
      rsquare: 0.9369
          dfe: 128
    adjrsquare: 0.9345
         rmse: 0.0135
```

- **Save fit output to MATLAB struct named *output*** — This option creates a structure array that contains information such as numbers of observations and parameters, residuals, and so on. For example:

```
output =
        numobs: 134
      numparam: 6
     residuals: [134x1 double]
      Jacobian: [134x6 double]
      exitflag: 1
     algorithm: 'QR factorization and solve'
    iterations: 1
```

> **Note** **Goodness of fit** and **Output** arrays are outputs of the `fit` function. See the `fit` reference page.

4 Click **OK** to save the fit options to the workspace.

After you save your fit to the workspace, you can use fit postprocessing functions. For an example, see "Analyzing Your Best Fit in the Workspace" on page 2-22. For more information and a list of functions, see "Fit Postprocessing".

See Also

Related Examples

- "Evaluate a Curve Fit" on page 7-16
- "Evaluate a Surface Fit" on page 7-25

Evaluate a Curve Fit

This example shows how to work with a curve fit.

Load Data and Fit a Polynomial Curve

```
load census
curvefit = fit(cdate,pop,'poly3','normalize','on')

curvefit =
    Linear model Poly3:
    curvefit(x) = p1*x^3 + p2*x^2 + p3*x + p4
      where x is normalized by mean 1890 and std 62.05
    Coefficients (with 95% confidence bounds):
      p1 =       0.921  (-0.9743, 2.816)
      p2 =       25.18  (23.57, 26.79)
      p3 =       73.86  (70.33, 77.39)
      p4 =       61.74  (59.69, 63.8)
```

The output displays the fitted model equation, the fitted coefficients, and the confidence bounds for the fitted coefficients.

Plot the Fit, Data, Residuals, and Prediction Bounds

```
plot(curvefit,cdate,pop)
```
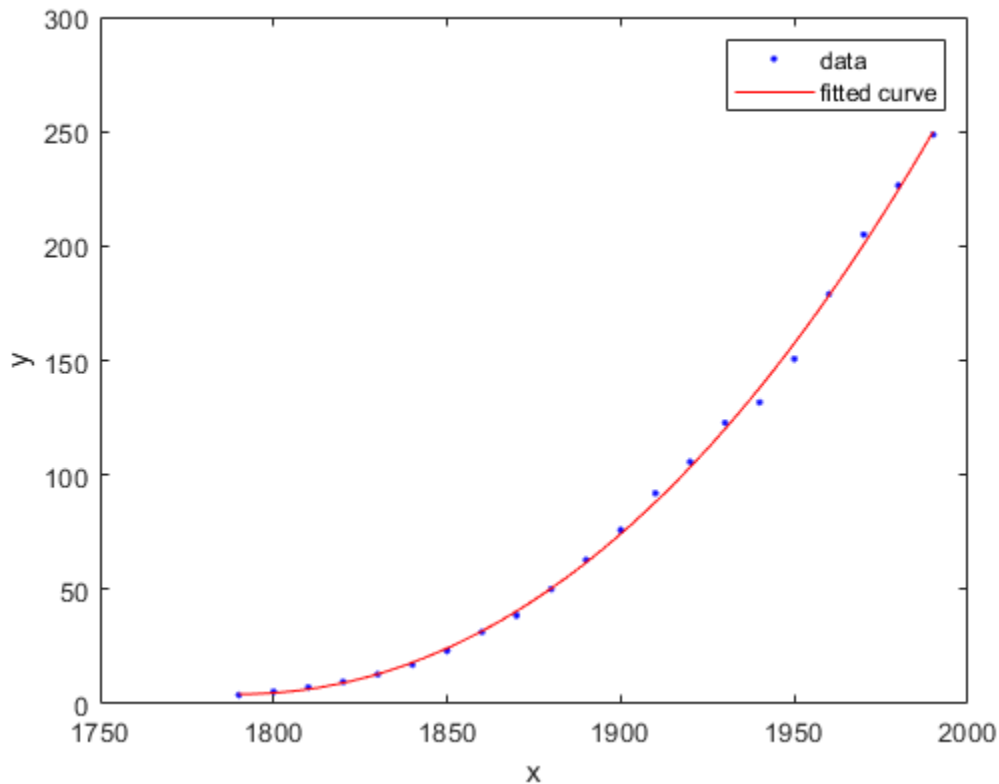

Plot the residuals fit.

```
plot(curvefit,cdate,pop,'Residuals')
```
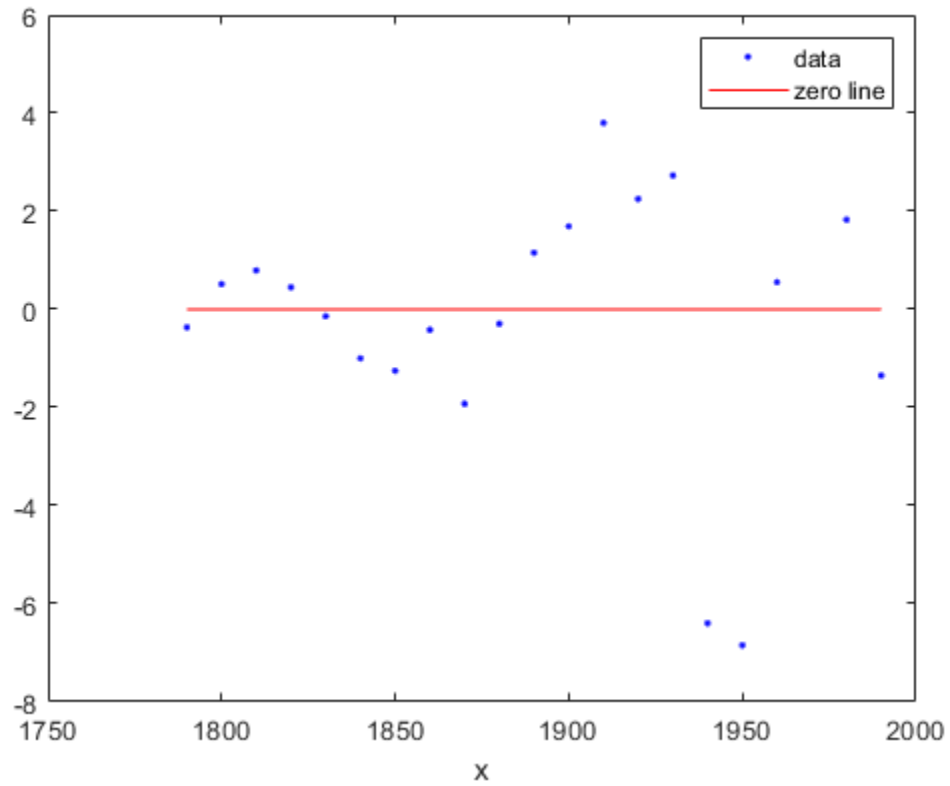

Plot the prediction bounds on the fit.

```
plot(curvefit,cdate,pop,'predfunc')
```
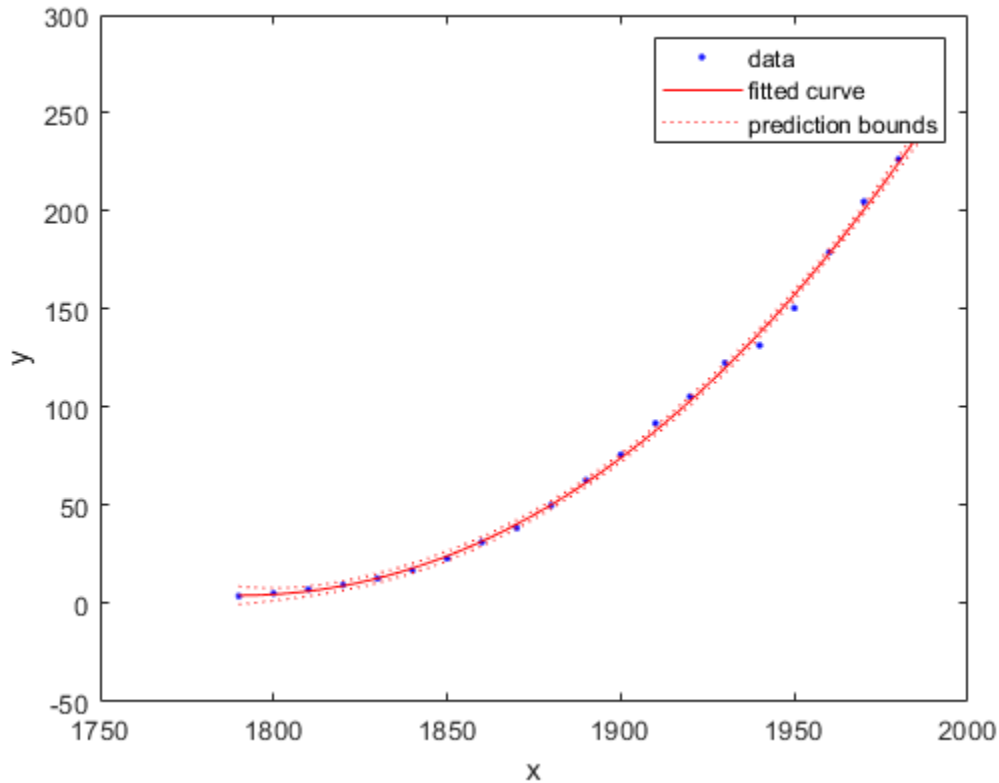
Evaluate the Fit at a Specified Point

Evaluate the fit at a specific point by specifying a value for x , using this form: y = fittedmodel(x).

```
curvefit(1991)
```

```
ans = 252.6690
```

Evaluate the Fit Values at Many Points

Evaluate the model at a vector of values to extrapolate to the year 2050.

```
xi = (2000:10:2050).';
curvefit(xi)
```

```
ans = 6×1
```

```
  276.9632
  305.4420
  335.5066
  367.1802
  400.4859
  435.4468
```

Get prediction bounds on those values.

```
ci = predint(curvefit,xi)
```

```
ci = 6×2

  267.8589   286.0674
  294.3070   316.5770
  321.5924   349.4208
  349.7275   384.6329
  378.7255   422.2462
  408.5919   462.3017
```

Plot the fit and prediction intervals across the extrapolated fit range. By default, the fit is plotted over the range of the data. To see values extrapolated from the fit, set the upper x-limit of the axes to 2050 before plotting the fit. To plot prediction intervals, use `predobs` or `predfun` as the plot type.

```
plot(cdate,pop,'o')
xlim([1900,2050])
hold on
plot(curvefit,'predobs')
hold off
```
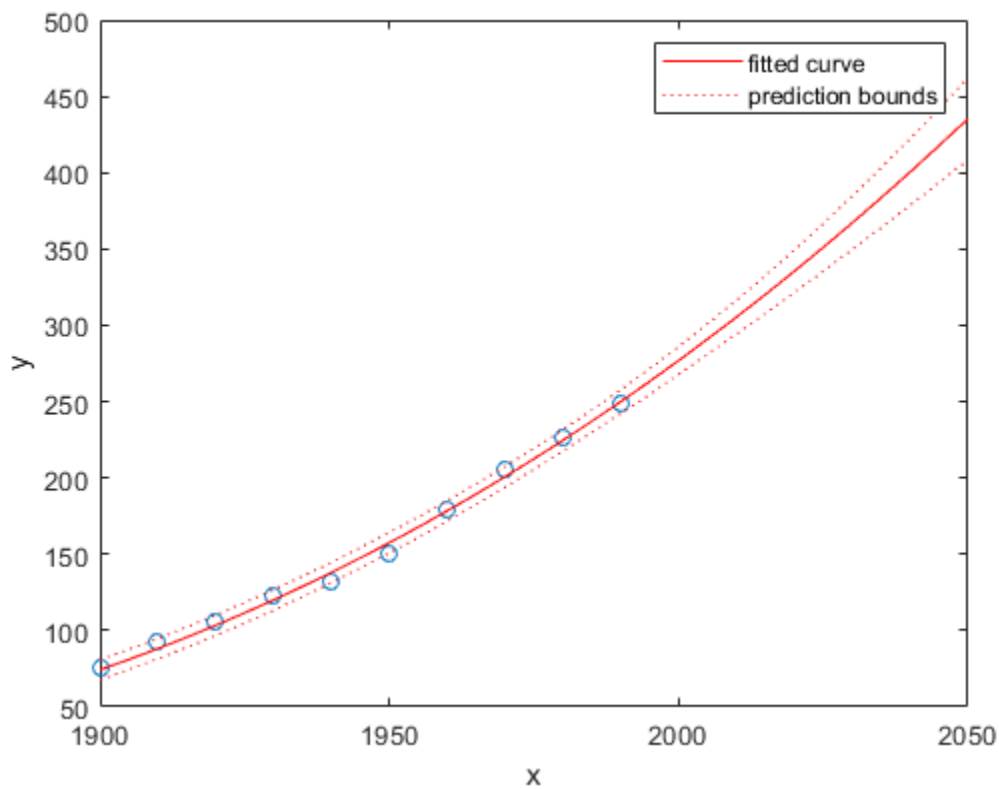

Get the Model Equation

Enter the fit name to display the model equation, the fitted coefficients, and the confidence bounds for the fitted coefficients.

```
curvefit
```

```
curvefit =
    Linear model Poly3:
    curvefit(x) = p1*x^3 + p2*x^2 + p3*x + p4
      where x is normalized by mean 1890 and std 62.05
    Coefficients (with 95% confidence bounds):
      p1 =        0.921  (-0.9743, 2.816)
      p2 =        25.18  (23.57, 26.79)
      p3 =        73.86  (70.33, 77.39)
      p4 =        61.74  (59.69, 63.8)
```

To get only the model equation, use `formula`.

`formula(curvefit)`

```
ans =
'p1*x^3 + p2*x^2 + p3*x + p4'
```

Get Coefficient Names and Values

Specify a coefficient by name.

`p1 = curvefit.p1`

`p1 = 0.9210`

`p2 = curvefit.p2`

`p2 = 25.1834`

Get all the coefficient names. Look at the fit equation (for example, `f(x) = p1*x^3+...`) to see the model terms for each coefficient.

`coeffnames(curvefit)`

```
ans = 4x1 cell
    {'p1'}
    {'p2'}
    {'p3'}
    {'p4'}
```

Get all the coefficient values.

`coeffvalues(curvefit)`

```
ans = 1×4

    0.9210   25.1834   73.8598   61.7444
```

Get Confidence Bounds on the Coefficients

Use confidence bounds on coefficients to help you evaluate and compare fits. The confidence bounds on the coefficients determine their accuracy. Bounds that are far apart indicate uncertainty. If the bounds cross zero for linear coefficients, this means you cannot be sure that these coefficients differ from zero. If some model terms have coefficients of zero, then they are not helping with the fit.

`confint(curvefit)`

```
ans = 2×4

   -0.9743    23.5736    70.3308    59.6907
    2.8163    26.7931    77.3888    63.7981
```

Examine Goodness-of-Fit Statistics

To get goodness-of-fit statistics at the command line, you can either:

- Open Curve Fitting app and select **Fit > Save to Workspace** to export your fit and goodness of fit to the workspace.

- Specify the `gof` output argument using the `fit` function.

Recreate the fit specifying the `gof` and output arguments to get goodness-of-fit statistics and fitting algorithm information.

```
[curvefit,gof,output] = fit(cdate,pop,'poly3','normalize','on')
```

```
curvefit =
     Linear model Poly3:
     curvefit(x) = p1*x^3 + p2*x^2 + p3*x + p4
       where x is normalized by mean 1890 and std 62.05
     Coefficients (with 95% confidence bounds):
       p1 =        0.921  (-0.9743, 2.816)
       p2 =        25.18  (23.57, 26.79)
       p3 =        73.86  (70.33, 77.39)
       p4 =        61.74  (59.69, 63.8)
```

```
gof = struct with fields:
           sse: 149.7687
       rsquare: 0.9988
           dfe: 17
     adjrsquare: 0.9986
          rmse: 2.9682
```

```
output = struct with fields:
        numobs: 21
      numparam: 4
     residuals: [21x1 double]
      Jacobian: [21x4 double]
      exitflag: 1
     algorithm: 'QR factorization and solve'
    iterations: 1
```

Plot a histogram of the residuals to look for a roughly normal distribution.

```
histogram(output.residuals,10)
```
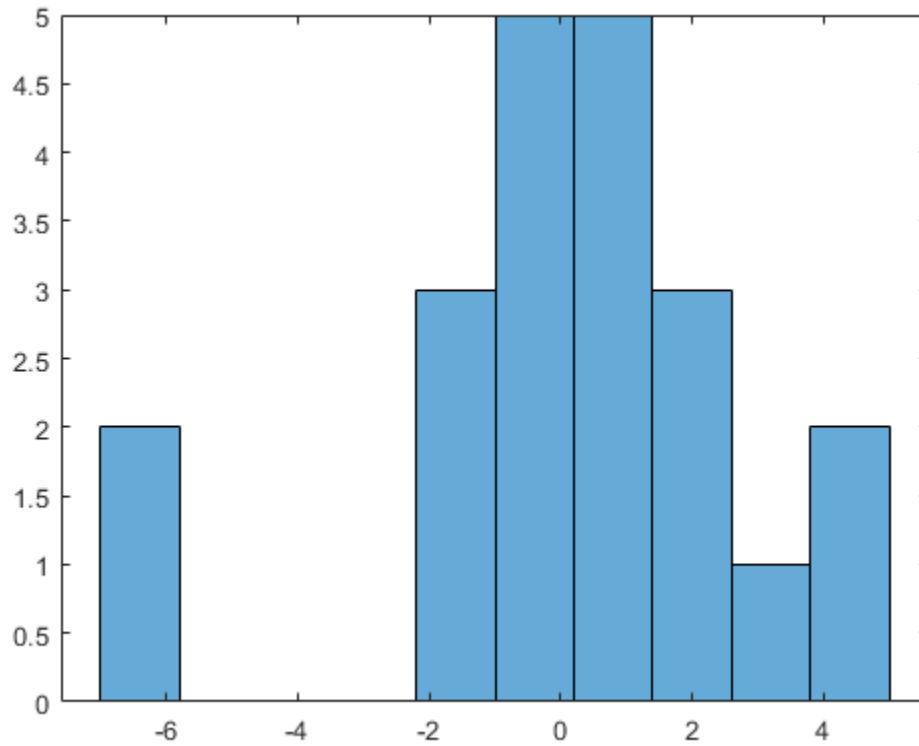
Plot the Fit, Data, and Residuals

```
plot(curvefit,cdate,pop,'fit','residuals')
legend Location SouthWest
subplot(2,1,1)
legend Location NorthWest
```
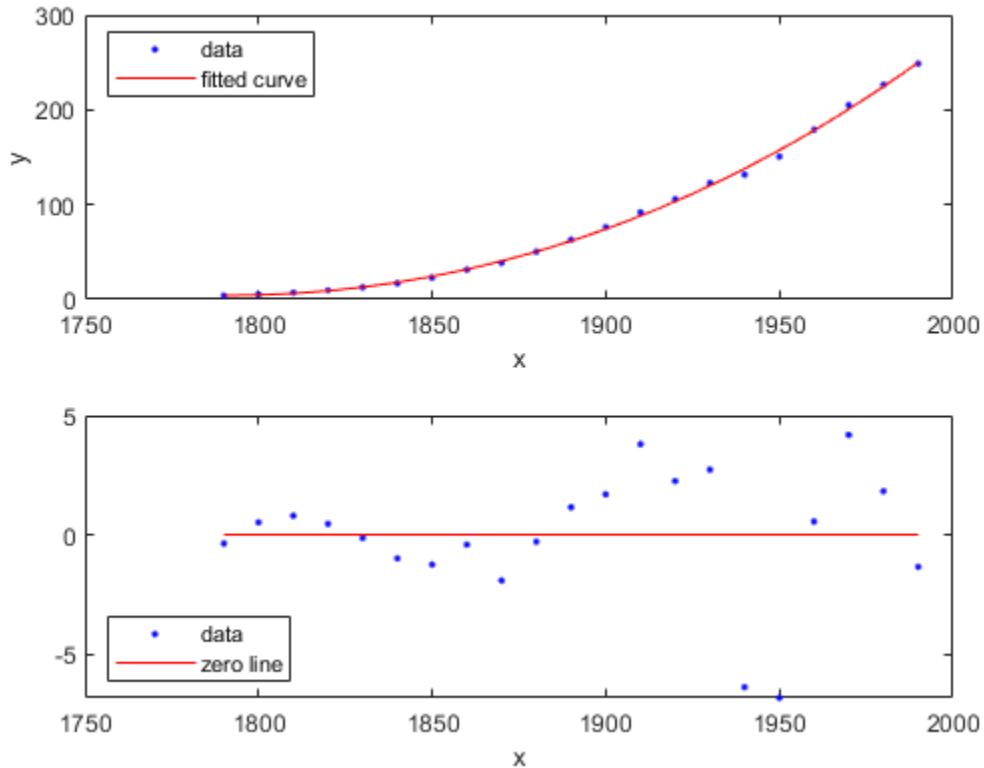
Find Methods

List every method that you can use with the fit.

```
methods(curvefit)
```

```
Methods for class cfit:
```

```
argnames      confint       formula       numcoeffs     setoptions
category      dependnames   indepnames    plot          type
cfit          differentiate integrate     predint
coeffnames    feval         islinear      probnames
coeffvalues   fitoptions    numargs       probvalues
```

Use the `help` command to find out how to use a fit method.

```
help cfit/differentiate
```

```
 DIFFERENTIATE  Differentiate a fit result object.
    DERIV1 = DIFFERENTIATE(FITOBJ,X) differentiates the model FITOBJ at the
    points specified by X and returns the result in DERIV1. FITOBJ is a Fit
    object generated by the FIT or CFIT function. X is a vector. DERIV1 is
    a vector with the same size as X. Mathematically speaking, DERIV1 =
    D(FITOBJ)/D(X).

    [DERIV1,DERIV2] = DIFFERENTIATE(FITOBJ, X) computes the first and
    second derivatives, DERIV1 and DERIV2 respectively, of the model
    FITOBJ.
```

See also CFIT/INTEGRATE, FIT, CFIT.

Evaluate a Surface Fit

This example shows how to work with a surface fit.

Load Data and Fit a Polynomial Surface

```
load franke;
surffit = fit([x,y],z,'poly23','normalize','on')

    Linear model Poly23:
    surffit(x,y) = p00 + p10*x + p01*y + p20*x^2 + p11*x*y + p02*y^2 + p21*x^2*y
                   + p12*x*y^2 + p03*y^3
      where x is normalized by mean 1982 and std 868.6
      and where y is normalized by mean 0.4972 and std 0.2897
    Coefficients (with 95% confidence bounds):
      p00 =      0.4253  (0.3928, 0.4578)
      p10 =      -0.106  (-0.1322, -0.07974)
      p01 =     -0.4299  (-0.4775, -0.3822)
      p20 =     0.02104  (0.001457, 0.04062)
      p11 =     0.07153  (0.05409, 0.08898)
      p02 =    -0.03084  (-0.05039, -0.01129)
      p21 =     0.02091  (0.001372, 0.04044)
      p12 =     -0.0321  (-0.05164, -0.01255)
      p03 =      0.1216  (0.09929, 0.1439)
```

The output displays the fitted model equation, the fitted coefficients, and the confidence bounds for the fitted coefficients.

Plot the Fit, Data, Residuals, and Prediction Bounds

```
plot(surffit,[x,y],z)
```
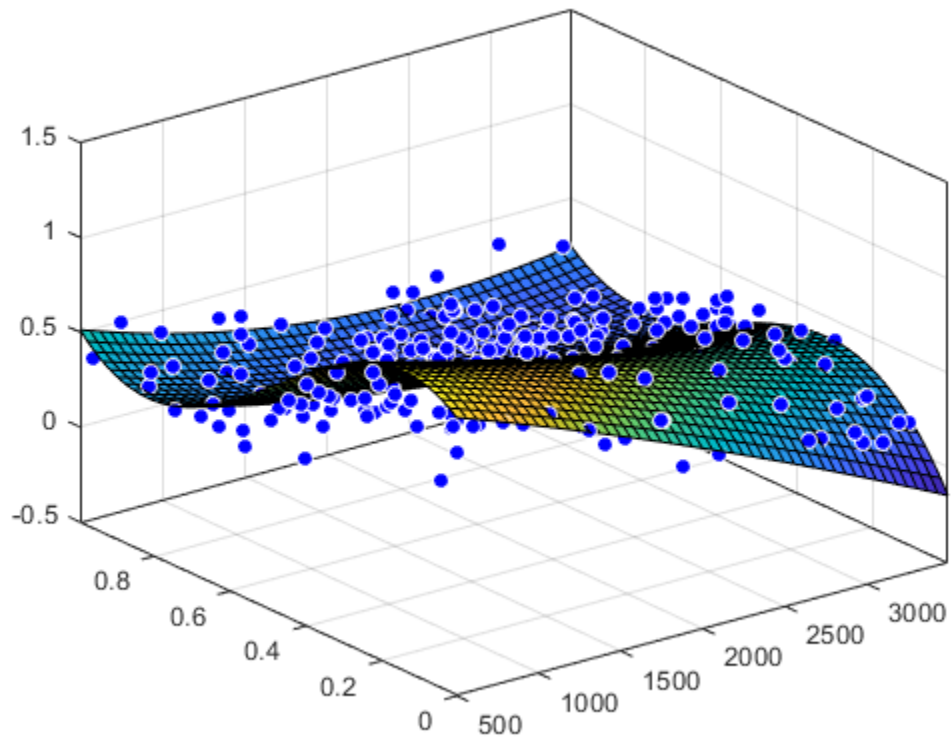
Plot the residuals fit.

```
plot(surffit,[x,y],z,'Style','Residuals')
```
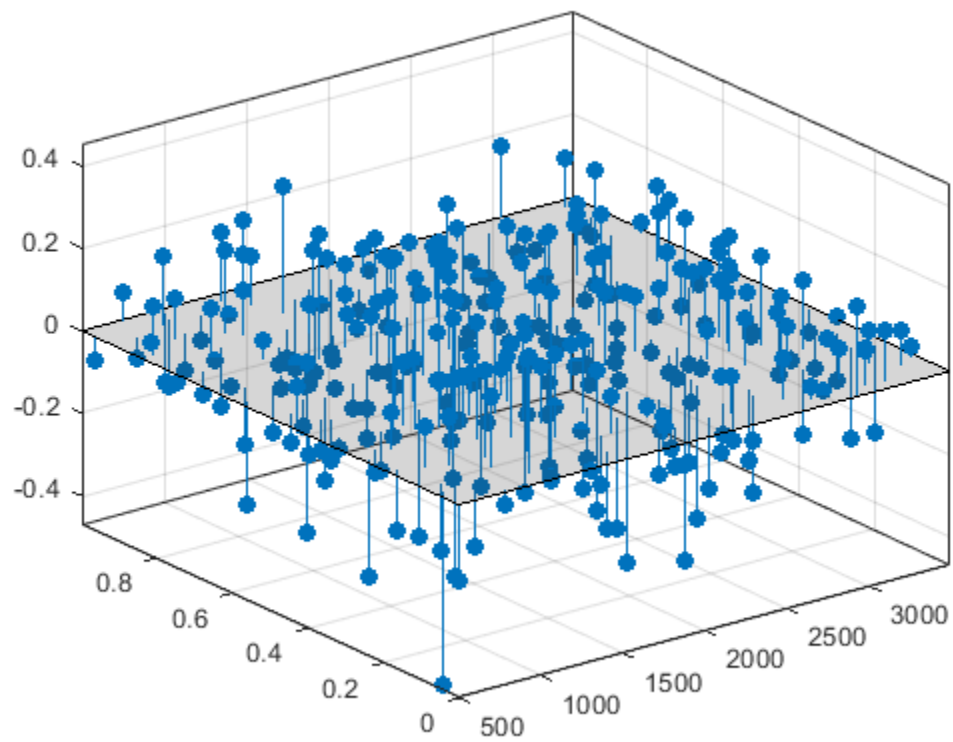
Plot prediction bounds on the fit.

```
plot(surffit,[x,y],z,'Style','predfunc')
```
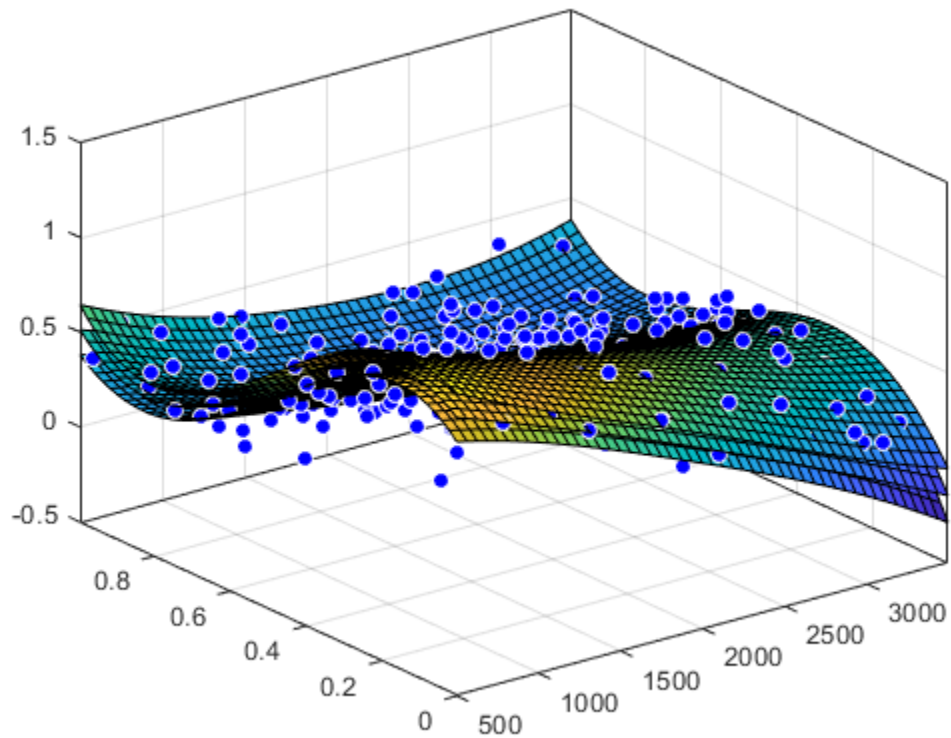
Evaluate the Fit at a Specified Point

Evaluate the fit at a specific point by specifying a value for x and y , using this form: z = fittedmodel(x,y).

```
surffit(1000,0.5)
```

```
ans = 0.5673
```

Evaluate the Fit Values at Many Points

```
xi = [500;1000;1200];
yi = [0.7;0.6;0.5];
surffit(xi,yi)
```

```
ans = 3×1

    0.3771
    0.4064
    0.5331
```

Get prediction bounds on those values.

```
[ci, zi] = predint(surffit,[xi,yi])
```

```
ci = 3×2
```

```
    0.0713    0.6829
    0.1058    0.7069
    0.2333    0.8330


zi = 3×1

    0.3771
    0.4064
    0.5331
```

Get the Model Equation

Enter the fit name to display the model equation, fitted coefficients, and confidence bounds for the fitted coefficients.

```
surffit
```

```
    Linear model Poly23:
    surffit(x,y) = p00 + p10*x + p01*y + p20*x^2 + p11*x*y + p02*y^2 + p21*x^2*y
                   + p12*x*y^2 + p03*y^3
      where x is normalized by mean 1982 and std 868.6
      and where y is normalized by mean 0.4972 and std 0.2897
    Coefficients (with 95% confidence bounds):
      p00 =       0.4253  (0.3928, 0.4578)
      p10 =       -0.106  (-0.1322, -0.07974)
      p01 =      -0.4299  (-0.4775, -0.3822)
      p20 =      0.02104  (0.001457, 0.04062)
      p11 =      0.07153  (0.05409, 0.08898)
      p02 =     -0.03084  (-0.05039, -0.01129)
      p21 =      0.02091  (0.001372, 0.04044)
      p12 =      -0.0321  (-0.05164, -0.01255)
      p03 =       0.1216  (0.09929, 0.1439)
```

To get only the model equation, use `formula`.

```
formula(surffit)
```

```
ans =
'p00 + p10*x + p01*y + p20*x^2 + p11*x*y + p02*y^2 + p21*x^2*y + p12*x*y^2 + p03*y^3'
```

Get Coefficient Names and Values

Specify a coefficient by name.

```
p00 = surffit.p00
```

```
p00 = 0.4253
```

```
p03 = surffit.p03
```

```
p03 = 0.1216
```

Get all the coefficient names. Look at the fit equation (for example, `f(x,y) = p00 + p10*x...`) to see the model terms for each coefficient.

```
coeffnames(surffit)
```

```
ans = 9x1 cell
    {'p00'}
```

```
{'p10'}
{'p01'}
{'p20'}
{'p11'}
{'p02'}
{'p21'}
{'p12'}
{'p03'}
```

Get all the coefficient values.

```
coeffvalues(surffit)
```

ans = *1×9*

```
    0.4253   -0.1060   -0.4299    0.0210    0.0715   -0.0308    0.0209   -0.0321    0.1216
```

Get Confidence Bounds on the Coefficients

Use confidence bounds on coefficients to help you evaluate and compare fits. The confidence bounds on the coefficients determine their accuracy. Bounds that are far apart indicate uncertainty. If the bounds cross zero for linear coefficients, this means you cannot be sure that these coefficients differ from zero. If some model terms have coefficients of zero, then they are not helping with the fit.

```
confint(surffit)
```

ans = *2×9*

```
    0.3928   -0.1322   -0.4775    0.0015    0.0541   -0.0504    0.0014   -0.0516    0.0993
    0.4578   -0.0797   -0.3822    0.0406    0.0890   -0.0113    0.0404   -0.0126    0.1439
```

Find Methods

List every method that you can use with the fit.

```
methods(surffit)
```

Methods for class sfit:

```
argnames        dependnames     indepnames      predint         sfit
category        differentiate   islinear        probnames       type
coeffnames      feval           numargs         probvalues
coeffvalues     fitoptions      numcoeffs       quad2d
confint         formula         plot            setoptions
```

Use the `help` command to find out how to use a fit method.

```
help sfit/quad2d
```

```
 QUAD2D  Numerically integrate a surface fit object.
    Q = QUAD2D(FO, A, B, C, D) approximates the integral of the surface fit
    object FO over the planar region A <= x <= B and C(x) <= y <= D(x). C and D
    may each be a scalar, a function handle or a curve fit (CFIT) object.

    [Q,ERRBND] = QUAD2D(...) also returns an approximate upper bound on the
    absolute error, ERRBND.
```

[Q,ERRBND] = QUAD2D(FUN,A,B,C,D,PARAM1,VAL1,PARAM2,VAL2,...) performs
the integration with specified values of optional parameters.

See QUAD2D for details of the upper bound and the optional parameters.

See also: QUAD2D, FIT, SFIT, CFIT.

Compare Fits Programmatically

This example shows how to fit and compare polynomials up to sixth degree using Curve Fitting Toolbox, fitting some census data. It also shows how to fit a single-term exponential equation and compare this to the polynomial models.

The steps show how to:

- Load data and create fits using different library models.
- Search for the best fit by comparing graphical fit results, and by comparing numerical fit results including the fitted coefficients and goodness of fit statistics.

Load and Plot the Data

The data for this example is the file census.mat.

```
load census
```

The workspace contains two new variables:

- cdate is a column vector containing the years 1790 to 1990 in 10-year increments.

- pop is a column vector with the U.S. population figures that correspond to the years in cdate .

```
whos cdate pop
plot(cdate,pop,'o')
```

```
  Name         Size            Bytes  Class     Attributes

  cdate        21x1              168  double
  pop          21x1              168  double
```
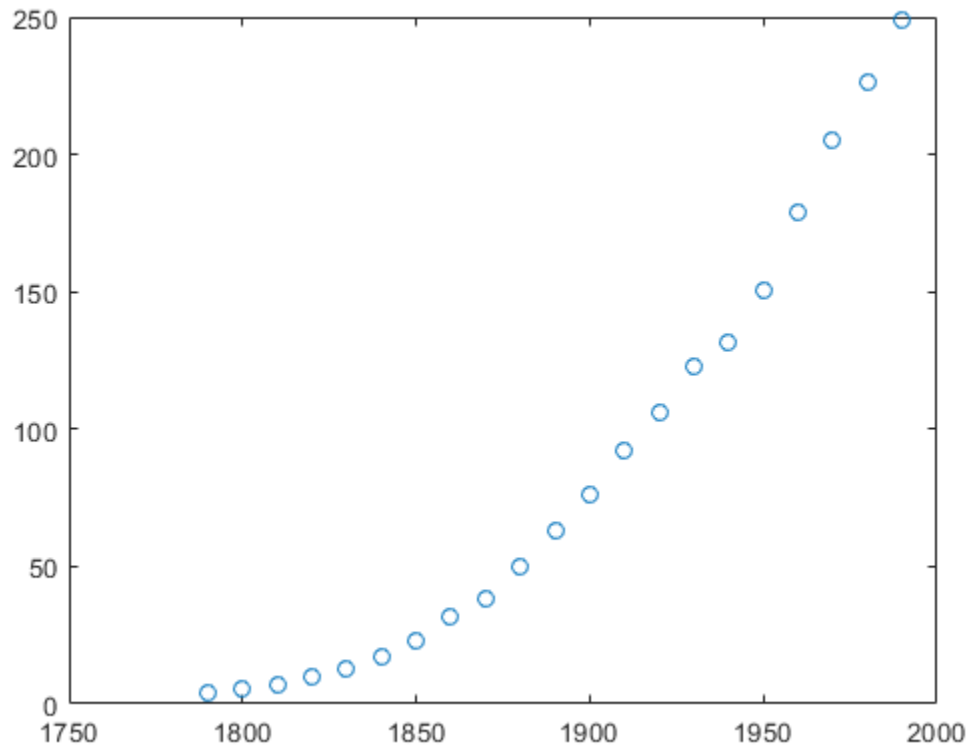
Create and Plot a Quadratic

Use the fit function to fit a polynomial to data. You specify a quadratic, or second-degree polynomial, with the string 'poly2'. The first output from fit is the polynomial, and the second output, gof, contains the goodness of fit statistics you will examine in a later step.

```
[population2,gof] = fit(cdate,pop,'poly2');
```

To plot the fit, use the plot method.

```
plot(population2,cdate,pop);
% Move the legend to the top left corner.
legend('Location','NorthWest');
```

Create and Plot a Selection of Polynomials

To fit polynomials of different degrees, change the fittype string, e.g., for a cubic or third-degree polynomial use 'poly3'. The scale of the input, cdate, is quite large, so you can obtain better results by centering and scaling the data. To do this, use the 'Normalize' option.

```
population3 = fit(cdate,pop,'poly3','Normalize','on');
population4 = fit(cdate,pop,'poly4','Normalize','on');
population5 = fit(cdate,pop,'poly5','Normalize','on');
population6 = fit(cdate,pop,'poly6','Normalize','on');
```

A simple model for population growth tells us that an exponential equation should fit this census data well. To fit a single term exponential model, use 'exp1' as the fittype.

```
populationExp = fit(cdate,pop,'exp1');
```

Plot all the fits at once, and add a meaningful legend in the top left corner of the plot.

```
hold on
plot(population3,'b');
plot(population4,'g');
plot(population5,'m');
plot(population6,'b--');
plot(populationExp,'r--');
hold off
legend('cdate v pop','poly2','poly3','poly4','poly5','poly6','exp1',...
    'Location','NorthWest');
```

Plot the Residuals to Evaluate the Fit

To plot residuals, specify 'residuals' as the plot type in the plot method.

```
plot(population2,cdate,pop,'residuals');
```

The fits and residuals for the polynomial equations are all similar, making it difficult to choose the best one. If the residuals display a systematic pattern, it is a clear sign that the model fits the data poorly.

```
plot(populationExp,cdate,pop,'residuals');
```

The fit and residuals for the single-term exponential equation indicate it is a poor fit overall. Therefore, it is a poor choice and you can remove the exponential fit from the candidates for best fit.

Examine Fits Beyond the Data Range

Examine the behavior of the fits up to the year 2050. The goal of fitting the census data is to extrapolate the best fit to predict future population values. By default, the fit is plotted over the range of the data. To plot a fit over a different range, set the x-limits of the axes before plotting the fit. For example, to see values extrapolated from the fit, set the upper x-limit to 2050.

```
plot(cdate,pop,'o');
xlim([1900, 2050]);
hold on
plot(population6);
hold off
```

Examine the plot. The behavior of the sixth-degree polynomial fit beyond the data range makes it a poor choice for extrapolation and you can reject this fit.

Plot Prediction Intervals

To plot prediction intervals, use 'predobs' or 'predfun' as the plot type. For example, to see the prediction bounds for the fifth-degree polynomial for a new observation up to year 2050:

```
plot(cdate,pop,'o');
xlim([1900, 2050])
hold on
plot(population5,'predobs');
hold off
```

Plot prediction intervals for the cubic polynomial up to year 2050.

```
plot(cdate,pop,'o');
xlim([1900, 2050])
hold on
plot(population3,'predobs')
hold off
```

Examine Goodness-of-Fit Statistics

The struct gof shows the goodness-of-fit statistics for the 'poly2' fit. When you created the 'poly2' fit with the fit function in an earlier step, you specified the gof output argument.

```
gof
```

```
gof =

  struct with fields:

          sse: 159.0293
      rsquare: 0.9987
          dfe: 18
    adjrsquare: 0.9986
         rmse: 2.9724
```

Examine the sum of squares due to error (SSE) and the adjusted R-square statistics to help determine the best fit. The SSE statistic is the least-squares error of the fit, with a value closer to zero indicating a better fit. The adjusted R-square statistic is generally the best indicator of the fit quality when you add additional coefficients to your model.

The large SSE for 'exp1' indicates it is a poor fit, which you already determined by examining the fit and residuals. The lowest SSE value is associated with 'poly6'. However, the behavior of this fit

beyond the data range makes it a poor choice for extrapolation, so you already rejected this fit by examining the plots with new axis limits.

The next best SSE value is associated with the fifth-degree polynomial fit, 'poly5', suggesting it might be the best fit. However, the SSE and adjusted R-square values for the remaining polynomial fits are all very close to each other. Which one should you choose?

Compare the Coefficients and Confidence Bounds to Determine the Best Fit

Resolve the best fit issue by examining the coefficients and confidence bounds for the remaining fits: the fifth-degree polynomial and the quadratic.

Examine population2 and population5 by displaying the models, the fitted coefficients, and the confidence bounds for the fitted coefficients:

```
population2
```

```
population5
```

```
population2 =

     Linear model Poly2:
     population2(x) = p1*x^2 + p2*x + p3
     Coefficients (with 95% confidence bounds):
       p1 =    0.006541  (0.006124, 0.006958)
       p2 =      -23.51  (-25.09, -21.93)
       p3 =   2.113e+04  (1.964e+04, 2.262e+04)

population5 =

     Linear model Poly5:
     population5(x) = p1*x^5 + p2*x^4 + p3*x^3 + p4*x^2 + p5*x + p6
       where x is normalized by mean 1890 and std 62.05
     Coefficients (with 95% confidence bounds):
       p1 =      0.5877  (-2.305, 3.48)
       p2 =      0.7047  (-1.684, 3.094)
       p3 =     -0.9193  (-10.19, 8.356)
       p4 =       23.47  (17.42, 29.52)
       p5 =       74.97  (68.37, 81.57)
       p6 =       62.23  (59.51, 64.95)
```

You can also get the confidence intervals by using confint.

```
ci = confint(population5)
```

```
ci =

   -2.3046   -1.6841  -10.1943   17.4213   68.3655   59.5102
    3.4801    3.0936    8.3558   29.5199   81.5696   64.9469
```

The confidence bounds on the coefficients determine their accuracy. Check the fit equations (e.g. f(x)=p1*x+p2*x...) to see the model terms for each coefficient. Note that p2 refers to the p2*x term in 'poly2' and the p2*x^4 term in 'poly5'. Do not compare normalized coefficients directly with non-normalized coefficients.

The bounds cross zero on the p1, p2, and p3 coefficients for the fifth-degree polynomial. This means you cannot be sure that these coefficients differ from zero. If the higher order model terms may have coefficients of zero, they are not helping with the fit, which suggests that this model over fits the census data.

The fitted coefficients associated with the constant, linear, and quadratic terms are nearly identical for each normalized polynomial equation. However, as the polynomial degree increases, the coefficient bounds associated with the higher degree terms cross zero, which suggests over fitting.

However, the small confidence bounds do not cross zero on p1, p2, and p3 for the quadratic fit, indicating that the fitted coefficients are known fairly accurately.

Therefore, after examining both the graphical and numerical fit results, you should select the quadratic population2 as the best fit to extrapolate the census data.

Evaluate the Best Fit at New Query Points

Now you have selected the best fit, population2, for extrapolating this census data, evaluate the fit for some new query points.

```
cdateFuture = (2000:10:2020).';
popFuture = population2(cdateFuture)


popFuture =

  274.6221
  301.8240
  330.3341
```

To compute 95% confidence bounds on the prediction for the population in the future, use the predint method:

```
ci = predint(population2,cdateFuture,0.95,'observation')


ci =

  266.9185   282.3257
  293.5673   310.0807
  321.3979   339.2702
```

Plot the predicted future population, with confidence intervals, against the fit and data.

```
plot(cdate,pop,'o');
xlim([1900, 2040])
hold on
plot(population2)
h = errorbar(cdateFuture,popFuture,popFuture-ci(:,1),ci(:,2)-popFuture,'.');
hold off
legend('cdate v pop','poly2','prediction','Location','NorthWest')
```
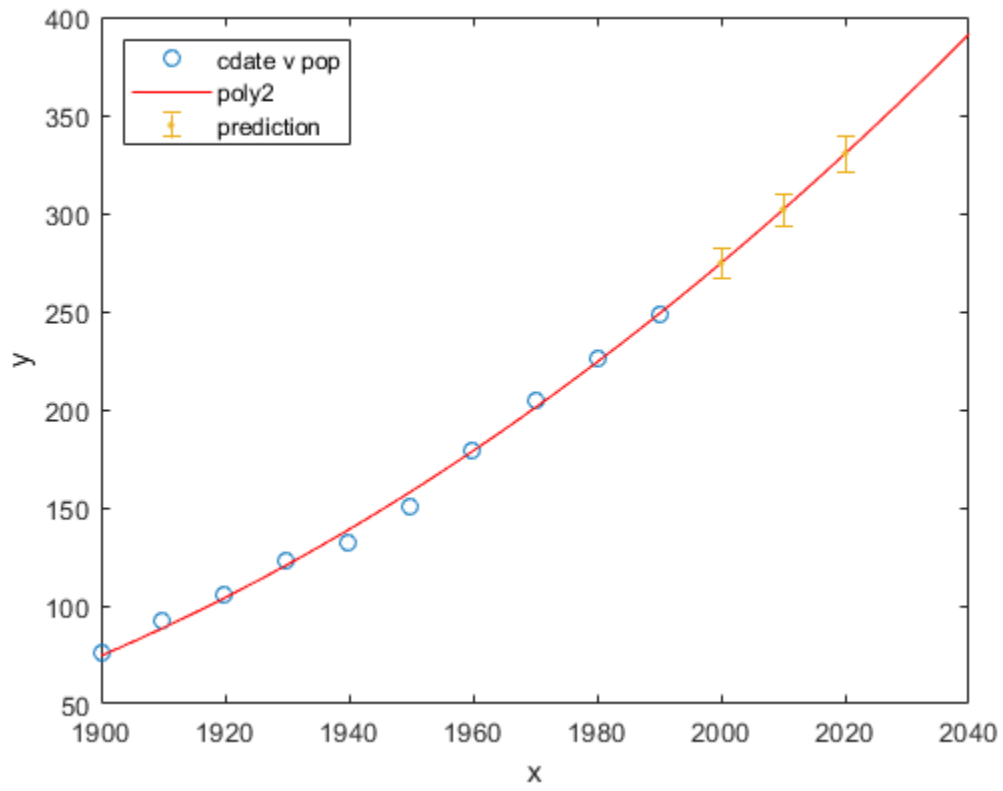
Evaluating Goodness of Fit

In this section...
"How to Evaluate Goodness of Fit" on page 7-44
"Goodness-of-Fit Statistics" on page 7-45

How to Evaluate Goodness of Fit

After fitting data with one or more models, you should evaluate the goodness of fit. A visual examination of the fitted curve displayed in Curve Fitting app should be your first step. Beyond that, the toolbox provides these methods to assess goodness of fit for both linear and nonlinear parametric fits:

- "Goodness-of-Fit Statistics" on page 7-45
- "Residual Analysis" on page 7-48
- "Confidence and Prediction Bounds" on page 7-52

As is common in statistical literature, the term *goodness of fit* is used here in several senses: A "good fit" might be a model

- that your data could reasonably have come from, given the assumptions of least-squares fitting
- in which the model coefficients can be estimated with little uncertainty
- that explains a high proportion of the variability in your data, and is able to predict new observations with high certainty

A particular application might dictate still other aspects of model fitting that are important to achieving a good fit, such as a simple model that is easy to interpret. The methods described here can help you determine goodness of fit in all these senses.

These methods group into two types: graphical and numerical. Plotting residuals and prediction bounds are graphical methods that aid visual interpretation, while computing goodness-of-fit statistics and coefficient confidence bounds yield numerical measures that aid statistical reasoning.

Generally speaking, graphical measures are more beneficial than numerical measures because they allow you to view the entire data set at once, and they can easily display a wide range of relationships between the model and the data. The numerical measures are more narrowly focused on a particular aspect of the data and often try to compress that information into a single number. In practice, depending on your data and analysis requirements, you might need to use both types to determine the best fit.

Note that it is possible that none of your fits can be considered suitable for your data, based on these methods. In this case, it might be that you need to select a different model. It is also possible that all the goodness-of-fit measures indicate that a particular fit is suitable. However, if your goal is to extract fitted coefficients that have physical meaning, but your model does not reflect the physics of the data, the resulting coefficients are useless. In this case, understanding what your data represents and how it was measured is just as important as evaluating the goodness of fit.

Goodness-of-Fit Statistics

After using graphical methods to evaluate the goodness of fit, you should examine the goodness-of-fit statistics. Curve Fitting Toolbox software supports these goodness-of-fit statistics for parametric models:

- The sum of squares due to error (SSE)
- R-square
- Adjusted R-square
- Root mean squared error (RMSE)

For the current fit, these statistics are displayed in the **Results** pane in the Curve Fitting app. For all fits in the current curve-fitting session, you can compare the goodness-of-fit statistics in the **Table of fits**.

To get goodness-of-fit statistics at the command line, either:

- In Curve Fitting app, select **Fit > Save to Workspace** to export your fit and goodness of fit to the workspace.
- Specify the `gof` output argument with the `fit` function.

Sum of Squares Due to Error

This statistic measures the total deviation of the response values from the fit to the response values. It is also called the summed square of residuals and is usually labeled as *SSE*.

$$SSE = \sum_{i=1}^{n} w_i(y_i - \widehat{y}_i)^2$$

A value closer to 0 indicates that the model has a smaller random error component, and that the fit will be more useful for prediction.

R-Square

This statistic measures how successful the fit is in explaining the variation of the data. Put another way, R-square is the square of the correlation between the response values and the predicted response values. It is also called the square of the multiple correlation coefficient and the coefficient of multiple determination.

R-square is defined as the ratio of the sum of squares of the regression (*SSR*) and the total sum of squares (*SST*). *SSR* is defined as

$$SSR = \sum_{i=1}^{n} w_i(\widehat{y}_i - \bar{y})^2$$

SST is also called the sum of squares about the mean, and is defined as

$$SST = \sum_{i=1}^{n} w_i(y_i - \bar{y})^2$$

where *SST = SSR + SSE*. Given these definitions, R-square is expressed as

$$\text{R-square} = \frac{SSR}{SST} = 1 - \frac{SSE}{SST}$$

R-square can take on any value between 0 and 1, with a value closer to 1 indicating that a greater proportion of variance is accounted for by the model. For example, an R-square value of 0.8234 means that the fit explains 82.34% of the total variation in the data about the average.

If you increase the number of fitted coefficients in your model, R-square will increase although the fit may not improve in a practical sense. To avoid this situation, you should use the degrees of freedom adjusted R-square statistic described below.

Note that it is possible to get a negative R-square for equations that do not contain a constant term. Because R-square is defined as the proportion of variance explained by the fit, if the fit is actually worse than just fitting a horizontal line then R-square is negative. In this case, R-square cannot be interpreted as the square of a correlation. Such situations indicate that a constant term should be added to the model.

Degrees of Freedom Adjusted R-Square

This statistic uses the R-square statistic defined above, and adjusts it based on the residual degrees of freedom. The residual degrees of freedom is defined as the number of response values n minus the number of fitted coefficients m estimated from the response values.

$v = n - m$

v indicates the number of independent pieces of information involving the n data points that are required to calculate the sum of squares. Note that if parameters are bounded and one or more of the estimates are at their bounds, then those estimates are regarded as fixed. The degrees of freedom is increased by the number of such parameters.

The adjusted R-square statistic is generally the best indicator of the fit quality when you compare two models that are *nested* — that is, a series of models each of which adds additional coefficients to the previous model.

$$\text{adjusted R-square} = 1 - \frac{SSE(n-1)}{SST(v)}$$

The adjusted R-square statistic can take on any value less than or equal to 1, with a value closer to 1 indicating a better fit. Negative values can occur when the model contains terms that do not help to predict the response.

Root Mean Squared Error

This statistic is also known as the fit standard error and the standard error of the regression. It is an estimate of the standard deviation of the random component in the data, and is defined as

$$RMSE = s = \sqrt{MSE}$$

where MSE is the mean square error or the residual mean square

$$MSE = \frac{SSE}{v}$$

Just as with SSE, an MSE value closer to 0 indicates a fit that is more useful for prediction.

See Also

fit

Related Examples

- "Generate Code and Export Fits to the Workspace" on page 7-13
- "Evaluate a Curve Fit" on page 7-16
- "Evaluate a Surface Fit" on page 7-25

Residual Analysis

Plotting and Analysing Residuals

The residuals from a fitted model are defined as the differences between the response data and the fit to the response data at each predictor value.

residual = data – fit

You display the residuals in Curve Fitting app by selecting the toolbar button or menu item **View > Residuals Plot**.

Mathematically, the residual for a specific predictor value is the difference between the response value y and the predicted response value $\hat{y}$.

$r = y - \hat{y}$

Assuming the model you fit to the data is correct, the residuals approximate the random errors. Therefore, if the residuals appear to behave randomly, it suggests that the model fits the data well. However, if the residuals display a systematic pattern, it is a clear sign that the model fits the data poorly. Always bear in mind that many results of model fitting, such as confidence bounds, will be invalid should the model be grossly inappropriate for the data.

A graphical display of the residuals for a first degree polynomial fit is shown below. The top plot shows that the residuals are calculated as the vertical distance from the data point to the fitted curve. The bottom plot displays the residuals relative to the fit, which is the zero line.
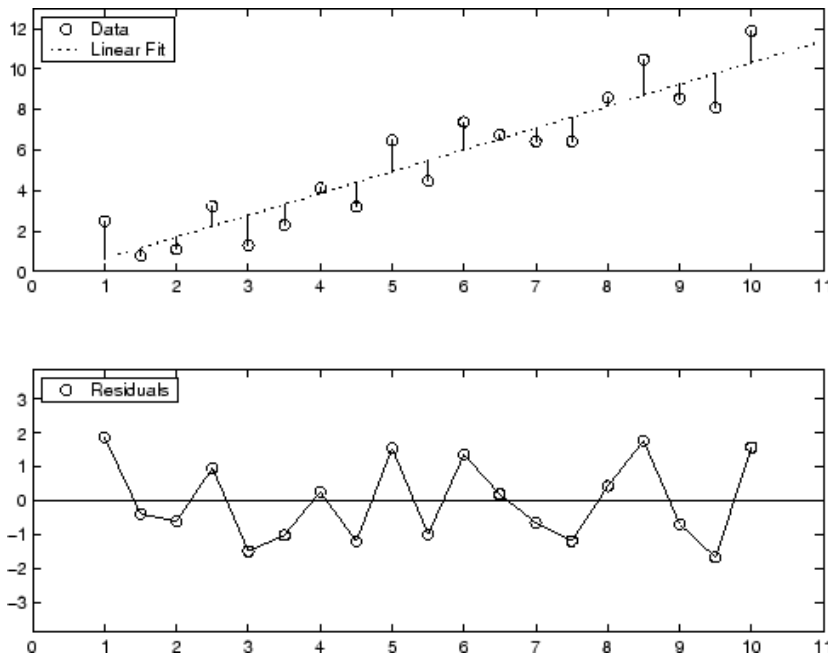
The residuals appear randomly scattered around zero indicating that the model describes the data well.

A graphical display of the residuals for a second-degree polynomial fit is shown below. The model includes only the quadratic term, and does not include a linear or constant term.
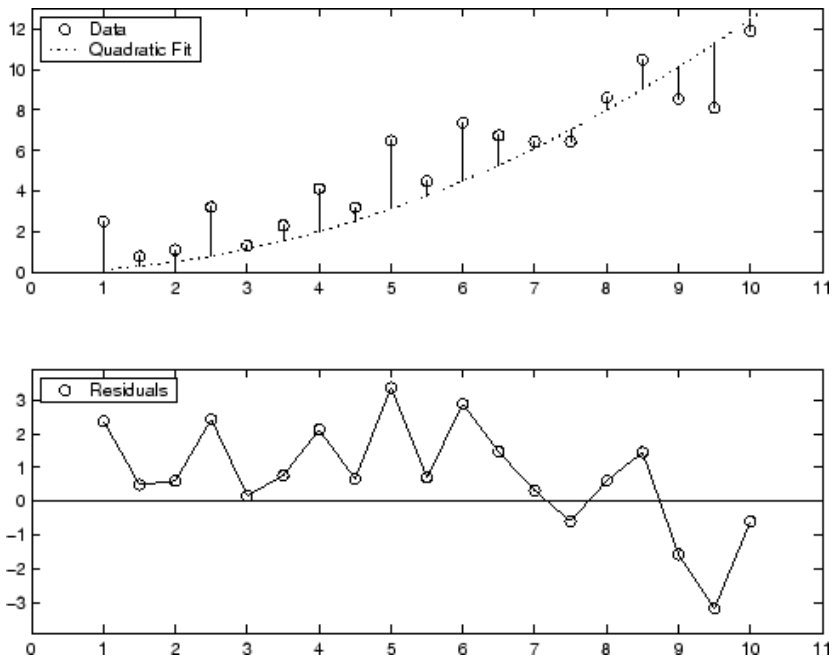
The residuals are systematically positive for much of the data range indicating that this model is a poor fit for the data.

Example: Residual Analysis

This example fits several polynomial models to generated data and evaluates how well those models fit the data and how precisely they can predict. The data is generated from a cubic curve, and there is a large gap in the range of the x variable where no data exist.

```
x = [1:0.1:3 9:0.1:10]';
c = [2.5 -0.5 1.3 -0.1];
y = c(1) + c(2)*x + c(3)*x.^2 + c(4)*x.^3 + (rand(size(x))-0.5);
```

Fit the data in the Curve Fitting app using a cubic polynomial and a fifth degree polynomial. The data, fits, and residuals are shown below. Display the residuals in the Curve Fitting app by selecting **View > Residuals Plot**.
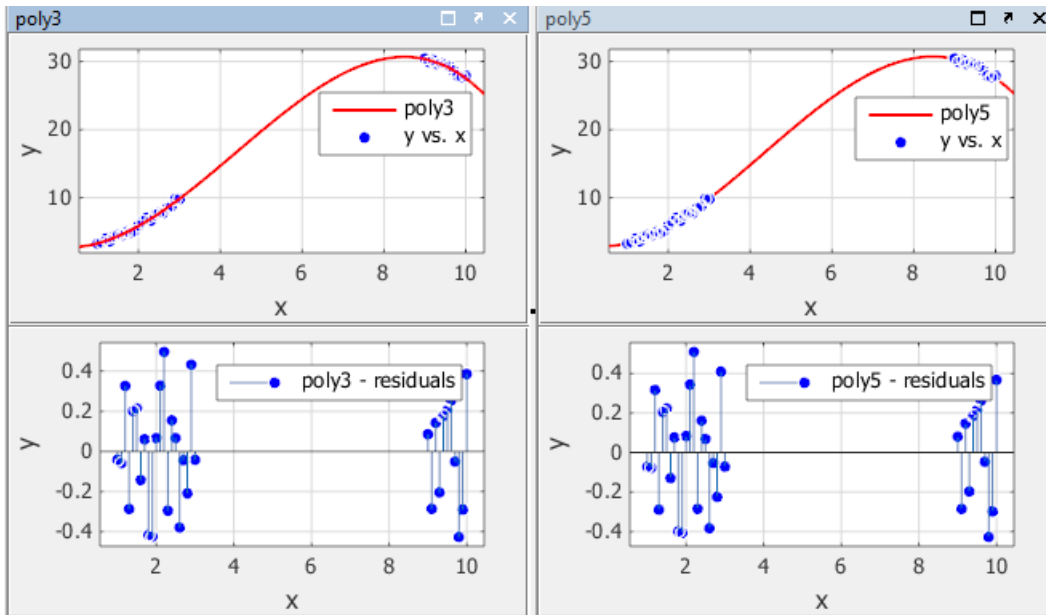
Both models appear to fit the data well, and the residuals appear to be randomly distributed around zero. Therefore, a graphical evaluation of the fits does not reveal any obvious differences between the two equations.

Look at the numerical fit results in the **Results** pane and compare the confidence bounds for the coefficients.

The results show that the cubic fit coefficients are accurately known (bounds are small), while the quintic fit coefficients are not accurately known. As expected, the fit results for `poly3` are reasonable because the generated data follows a cubic curve. The 95% confidence bounds on the fitted coefficients indicate that they are acceptably precise. However, the 95% confidence bounds for `poly5` indicate that the fitted coefficients are not known precisely.

The goodness-of-fit statistics are shown in the **Table of Fits**. By default, the adjusted R-square and RMSE statistics are displayed in the table. The statistics do not reveal a substantial difference between the two equations. To choose statistics to display or hide, right-click the column headers.

The 95% nonsimultaneous prediction bounds for new observations are shown below. To display prediction bounds in the Curve Fitting app, select **Tools > Prediction Bounds > 95%**.
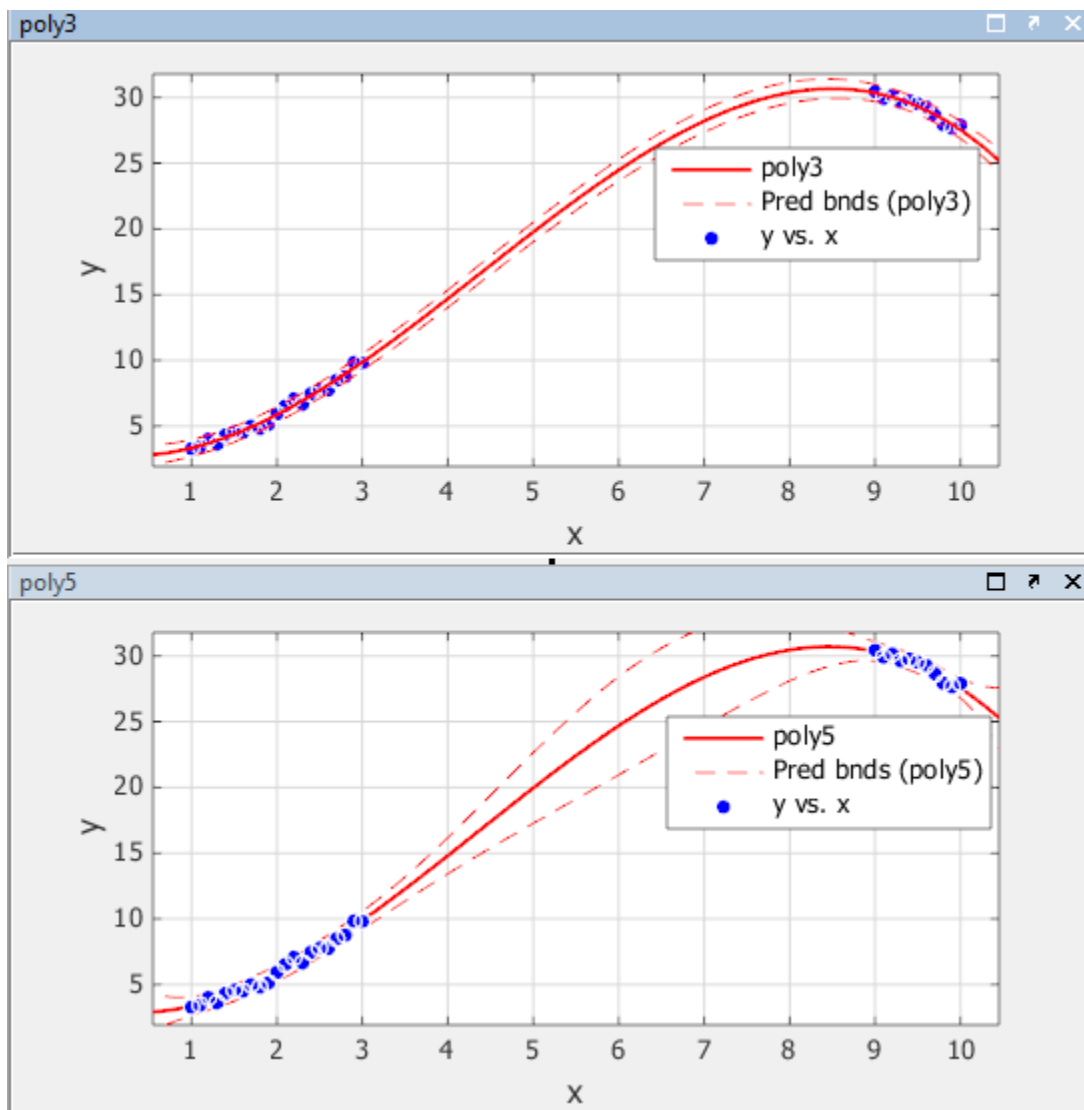
The prediction bounds for `poly3` indicate that new observations can be predicted with a small uncertainty throughout the entire data range. This is not the case for `poly5`. It has wider prediction bounds in the area where no data exist, apparently because the data does not contain enough information to estimate the higher degree polynomial terms accurately. In other words, a fifth-degree polynomial overfits the data.

The 95% prediction bounds for the fitted function using `poly5` are shown below. As you can see, the uncertainty in predicting the function is large in the center of the data. Therefore, you would conclude that more data must be collected before you can make precise predictions using a fifth-degree polynomial.

In conclusion, you should examine all available goodness-of-fit measures before deciding on the fit that is best for your purposes. A graphical examination of the fit and residuals should always be your initial approach. However, some fit characteristics are revealed only through numerical fit results, statistics, and prediction bounds.

Confidence and Prediction Bounds

In this section...
"About Confidence and Prediction Bounds" on page 7-52
"Confidence Bounds on Coefficients" on page 7-52
"Prediction Bounds on Fits" on page 7-53
"Compute Prediction Intervals" on page 7-55

About Confidence and Prediction Bounds

Curve Fitting Toolbox software lets you calculate confidence bounds for the fitted coefficients, and prediction bounds for new observations or for the fitted function. Additionally, for prediction bounds, you can calculate simultaneous bounds, which take into account all predictor values, or you can calculate nonsimultaneous bounds, which take into account only individual predictor values. The coefficient confidence bounds are presented numerically, while the prediction bounds are displayed graphically and are also available numerically.

The available confidence and prediction bounds are summarized below.

Types of Confidence and Prediction Bounds

Interval Type	Description
Fitted coefficients	Confidence bounds for the fitted coefficients
New observation	Prediction bounds for a new observation (response value)
New function	Prediction bounds for a new function value

Note Prediction bounds are also often described as confidence bounds because you are calculating a confidence interval for a predicted response.

Confidence and prediction bounds define the lower and upper values of the associated interval, and define the width of the interval. The width of the interval indicates how uncertain you are about the fitted coefficients, the predicted observation, or the predicted fit. For example, a very wide interval for the fitted coefficients can indicate that you should use more data when fitting before you can say anything very definite about the coefficients.

The bounds are defined with a level of certainty that you specify. The level of certainty is often 95%, but it can be any value such as 90%, 99%, 99.9%, and so on. For example, you might want to take a 5% chance of being incorrect about predicting a new observation. Therefore, you would calculate a 95% prediction interval. This interval indicates that you have a 95% chance that the new observation is actually contained within the lower and upper prediction bounds.

Confidence Bounds on Coefficients

The confidence bounds for fitted coefficients are given by

$$C = b \pm t\sqrt{S}$$

where b are the coefficients produced by the fit, t depends on the confidence level, and is computed using the inverse of Student's t cumulative distribution function, and S is a vector of the diagonal elements from the estimated covariance matrix of the coefficient estimates, $(X^TX)^{-1}s^2$. In a linear fit, X is the design matrix, while for a nonlinear fit X is the Jacobian of the fitted values with respect to the coefficients. X^T is the transpose of X, and s^2 is the mean squared error.

The confidence bounds are displayed in the **Results** pane in the Curve Fitting app using the following format.

```
p1 = 1.275  (1.113, 1.437)
```

The fitted value for the coefficient `p1` is 1.275, the lower bound is 1.113, the upper bound is 1.437, and the interval width is 0.324. By default, the confidence level for the bounds is 95%.

You can calculate confidence intervals at the command line with the `confint` function.

Prediction Bounds on Fits

As mentioned previously, you can calculate prediction bounds for the fitted curve. The prediction is based on an existing fit to the data. Additionally, the bounds can be simultaneous and measure the confidence for all predictor values, or they can be nonsimultaneous and measure the confidence only for a single predetermined predictor value. If you are predicting a new observation, nonsimultaneous bounds measure the confidence that the new observation lies within the interval given a single predictor value. Simultaneous bounds measure the confidence that a new observation lies within the interval regardless of the predictor value.

Bound Type	Observation	Functional
Simultaneous	$y \pm f\sqrt{s^2 + xSx^T}$	$y \pm f\sqrt{xSx^T}$
Nonsimultaneous	$y \pm t\sqrt{s^2 + xSx^T}$	$y \pm t\sqrt{xSx^T}$

Where:

- s^2 is the mean squared error
- t depends on the confidence level, and is computed using the inverse of Student's t cumulative distribution function
- f depends on the confidence level, and is computed using the inverse of the F cumulative distribution function.
- S is the covariance matrix of the coefficient estimates, $(X^TX)^{-1}s^2$.
- x is a row vector of the design matrix or Jacobian evaluated at a specified predictor value.

You can graphically display prediction bounds using Curve Fitting app. With Curve Fitting app, you can display nonsimultaneous prediction bounds for new observations with **Tools > Prediction Bounds**. By default, the confidence level for the bounds is 95%. You can change this level to any value with **Tools > Prediction Bounds > Custom**.

You can display numerical prediction bounds of any type at the command line with the `predint` function.

To understand the quantities associated with each type of prediction interval, recall that the data, fit, and residuals are related through the formula

data = fit + residuals

where the fit and residuals terms are estimates of terms in the formula

data = model + random error

Suppose you plan to take a new observation at the predictor value x_{n+1}. Call the new observation $y_{n+1}(x_{n+1})$ and the associated error ε_{n+1}. Then

$$y_{n+1}(x_{n+1}) = f(x_{n+1}) + \varepsilon_{n+1}$$

where $f(x_{n+1})$ is the true but unknown function you want to estimate at x_{n+1}. The likely values for the new observation or for the estimated function are provided by the nonsimultaneous prediction bounds.

If instead you want the likely value of the new observation to be associated with any predictor value, the previous equation becomes

$$y_{n+1}(x) = f(x) + \varepsilon$$

The likely values for this new observation or for the estimated function are provided by the simultaneous prediction bounds.

The types of prediction bounds are summarized below.

Types of Prediction Bounds

Type of Bound	Simultaneous or Non-simultaneous	Associated Equation
Observation	Non-simultaneous	$y_{n+1}(x_{n+1})$
	Simultaneous	$y_{n+1}(x)$, for all x
Function	Non-simultaneous	$f(x_{n+1})$
	Simultaneous	$f(x)$, for all x

The nonsimultaneous and simultaneous prediction bounds for a new observation and the fitted function are shown below. Each graph contains three curves: the fit, the lower confidence bounds, and the upper confidence bounds. The fit is a single-term exponential to generated data and the bounds reflect a 95% confidence level. Note that the intervals associated with a new observation are wider than the fitted function intervals because of the additional uncertainty in predicting a new response value (the curve plus random errors).

Compute Prediction Intervals

Compute and plot observation and functional prediction intervals for a fit to noisy data.

Generate noisy data with an exponential trend.

```
x = (0:0.2:5)';
y = 2*exp(-0.2*x) + 0.5*randn(size(x));
```

Fit a curve to the data using a single-term exponential.

```
fitresult = fit(x,y,'exp1');
```

Compute 95% observation and functional prediction intervals, both simultaneous and nonsimultaneous. Non-simultaneous bounds are for individual elements of x; simultaneous bounds are for all elements of x.

```
p11 = predint(fitresult,x,0.95,'observation','off');
p12 = predint(fitresult,x,0.95,'observation','on');
p21 = predint(fitresult,x,0.95,'functional','off');
p22 = predint(fitresult,x,0.95,'functional','on');
```

Plot the data, fit, and prediction intervals. Observation bounds are wider than functional bounds because they measure the uncertainty of predicting the fitted curve plus the random variation in the new observation.

```
subplot(2,2,1)
plot(fitresult,x,y), hold on, plot(x,p11,'m--'), xlim([0 5]), ylim([-1 5])
title('Nonsimultaneous Observation Bounds','FontSize',9)
legend off

subplot(2,2,2)
plot(fitresult,x,y), hold on, plot(x,p12,'m--'), xlim([0 5]), ylim([-1 5])
title('Simultaneous Observation Bounds','FontSize',9)
legend off

subplot(2,2,3)
plot(fitresult,x,y), hold on, plot(x,p21,'m--'), xlim([0 5]), ylim([-1 5])
title('Nonsimultaneous Functional Bounds','FontSize',9)
legend off

subplot(2,2,4)
plot(fitresult,x,y), hold on, plot(x,p22,'m--'), xlim([0 5]), ylim([-1 5])
title('Simultaneous Functional Bounds','FontSize',9)
legend({'Data','Fitted curve', 'Prediction intervals'},...
        'FontSize',8,'Location','northeast')
```

Differentiating and Integrating a Fit

This example shows how to find the first and second derivatives of a fit, and the integral of the fit, at the predictor values.

Create a baseline sinusoidal signal:

```
xdata = (0:.1:2*pi)';
y0 = sin(xdata);
```

Add noise to the signal:

```
noise = 2*y0.*randn(size(y0)); % Response-dependent noise
ydata = y0 + noise;
```

Fit the noisy data with a custom sinusoidal model:

```
f = fittype('a*sin(b*x)');
fit1 = fit(xdata,ydata,f,'StartPoint',[1 1]);
```

Find the derivatives of the fit at the predictors:

```
[d1,d2] = differentiate(fit1,xdata);
```

Plot the data, the fit, and the derivatives:

```
subplot(3,1,1)
plot(fit1,xdata,ydata) % cfit plot method
subplot(3,1,2)
plot(xdata,d1,'m') % double plot method
grid on
legend('1st derivative')
subplot(3,1,3)
plot(xdata,d2,'c') % double plot method
grid on
legend('2nd derivative')
```
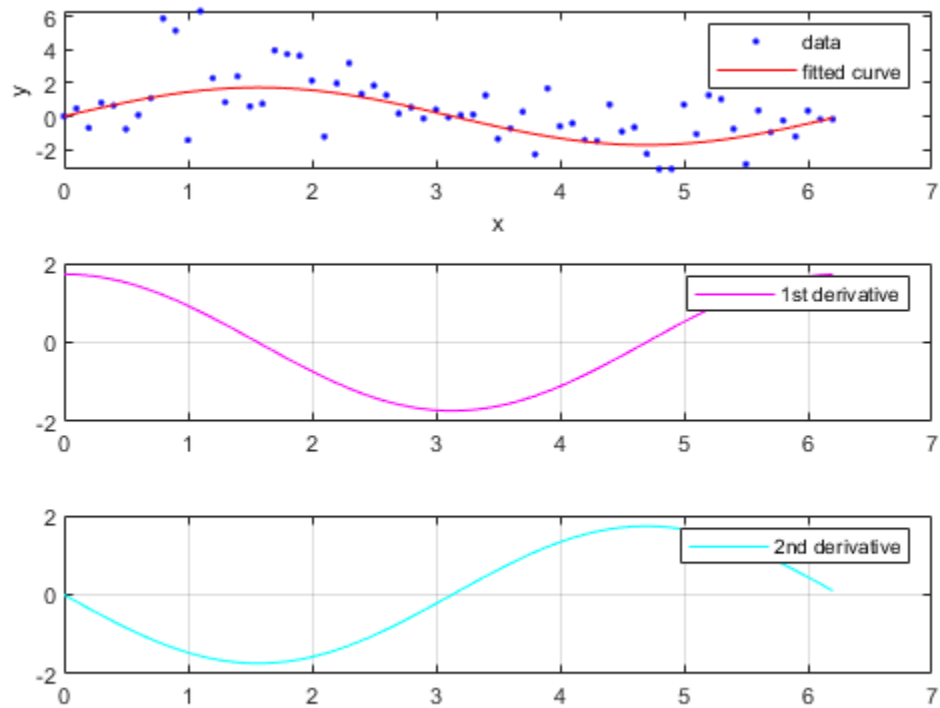
Note that derivatives can also be computed and plotted directly with the cfit plot method, as follows. The plot method, however, does not return data on the derivatives.

```
plot(fit1,xdata,ydata,{'fit','deriv1','deriv2'})
```
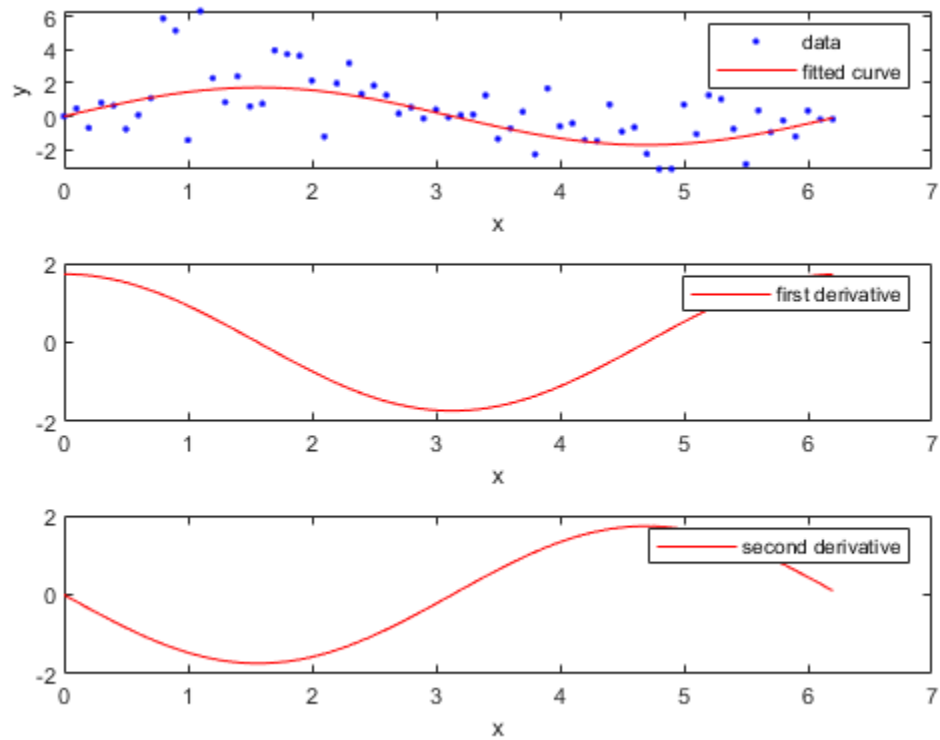
Find the integral of the fit at the predictors:

```
int = integrate(fit1,xdata,0);
```

Plot the data, the fit, and the integral:

```
subplot(2,1,1)
plot(fit1,xdata,ydata) % cfit plot method
subplot(2,1,2)
plot(xdata,int,'m') % double plot method
grid on
legend('integral')
```
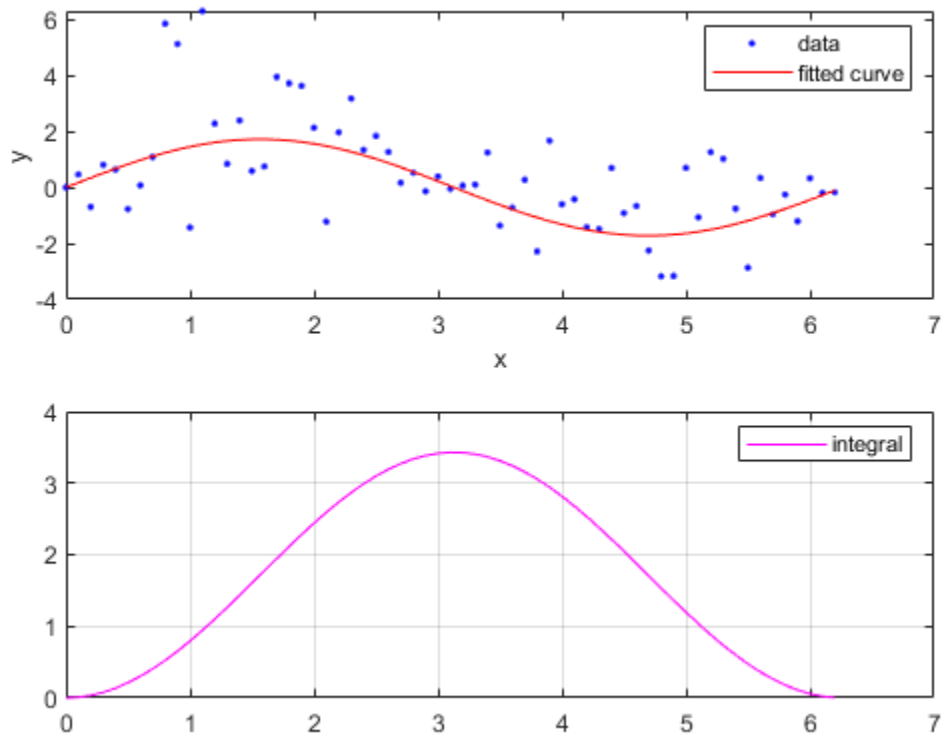
Note that integrals can also be computed and plotted directly with the cfit plot method, as follows. The plot method, however, does not return data on the integral.

```
plot(fit1,xdata,ydata,{'fit','integral'})
```
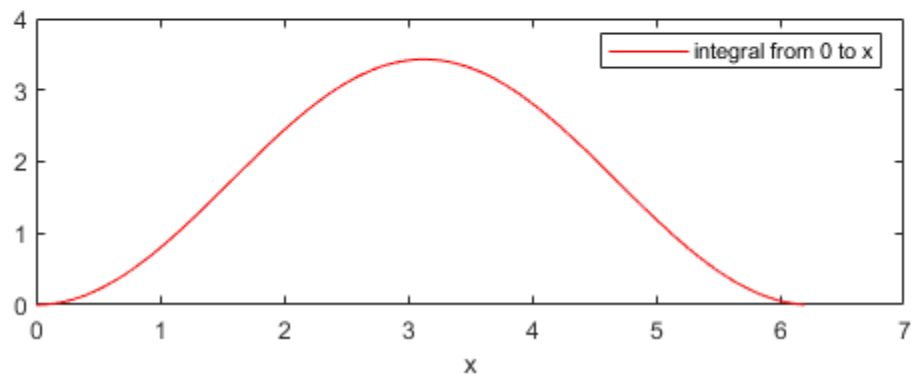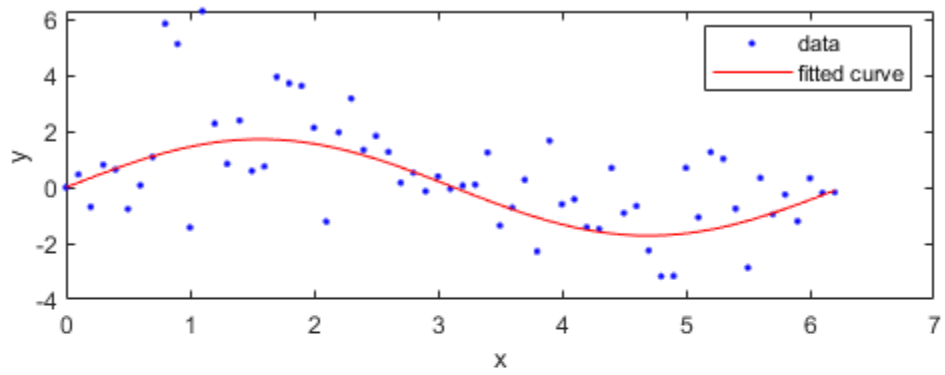
Spline Fitting

About Splines

- "Introducing Spline Fitting" on page 8-2
- "Curve Fitting Toolbox Splines and MATLAB Splines" on page 8-4

Introducing Spline Fitting

About Splines in Curve Fitting Toolbox

You can work with splines in Curve Fitting Toolbox in several ways.

Using the Curve Fitting app or the `fit` function you can:

- Fit cubic spline interpolants to curves or surfaces
- Fit smoothing splines and shape-preserving cubic spline interpolants to curves (but not surfaces)
- Fit thin-plate splines to surfaces (but not curves)

The toolbox also contains specific splines functions to allow greater control over what you can create. For example, you can use the `csapi` function for cubic spline interpolation. Why would you use `csapi` instead of the `fit` function `'cubicinterp'` option? You might require greater flexibility to work with splines for the following reasons:

- You want to combine the results with other splines, e.g., by addition.
- You want vector-valued splines. You can use `csapi` with scalars, vectors, matrices, and ND-arrays. The `fit` function only allows scalar-valued splines.
- You want other types of splines such as ppform, B-form, tensor-product, rational, and stform thin-plate splines.
- You want to create splines without data.
- You want to specify breaks, optimize knot placement, and use specialized functions for spline manipulation such as differentiation and integration.

If you require specialized spline functions, see the following sections for an overview of splines, and interactive and programmatic spline fitting.

Spline Overview

The Curve Fitting Toolbox spline functions are a collection of tools for creating, viewing, and analyzing spline approximations of data. Splines are smooth piecewise polynomials that can be used to represent functions over large intervals, where it would be impractical to use a single approximating polynomial.

The spline functionality includes a graphical user interface (GUI) that provides easy access to functions for creating, visualizing, and manipulating splines. The toolbox also contains functions that enable you to evaluate, plot, combine, differentiate, and integrate splines. Because all toolbox functions are implemented in the open MATLAB language, you can inspect the algorithms, modify the source code, and create your own custom functions.

Key spline features:

- GUIs that let you create, view, and manipulate splines and manage and compare spline approximations
- Functions for advanced spline operations, including differentiation, integration, break/knot manipulation, and optimal knot placement
- Support for piecewise polynomial form (ppform) and basis form (B-form) splines
- Support for tensor-product splines and rational splines (including NURBS)

Interactive Spline Fitting

You can access all spline functions from the `splinetool` GUI. You can use the GUI to:

- Vary spline parameters and tolerances
- View and modify data, breaks, knots, and weights
- View the error of the spline, or the spline's first or second derivative
- Observe the toolbox commands that generated your spline
- Create and import data, including built-in instructive data sets, and save splines to the workspace

See `splinetool`.

Programmatic Spline Fitting

To programmatically fit splines, see "Spline Construction" for descriptions of types of splines and numerous examples.

Curve Fitting Toolbox Splines and MATLAB Splines

In this section...

Curve Fitting Toolbox Splines

Curve Fitting Toolbox spline functions contain versions of the essential MATLAB programs of the B-spline package (extended to handle also *vector*-valued splines) as described in *A Practical Guide to Splines,* (Applied Math. Sciences Vol. 27, Springer Verlag, New York (1978), xxiv + 392p; revised edition (2001), xviii+346p), hereafter referred to as *PGS*. The toolbox makes it easy to create and work with piecewise-polynomial functions.

The typical use envisioned for this toolbox involves the construction and subsequent use of a piecewise-polynomial approximation. This construction would involve data fitting, but there is a wide range of possible data that could be fit. In the simplest situation, one is given points (t_i,y_i) and is looking for a piecewise-polynomial function f that satisfies $f(t_i) = y_i$, all i, more or less. An exact fit would involve *interpolation*, an approximate fit might involve *least-squares approximation* or the *smoothing spline*. But the function to be approximated may also be described in more implicit ways, for example as the solution of a differential or integral equation. In such a case, the data would be of the form (Af)(t_i), with A some differential or integral operator. On the other hand, one might want to construct a spline *curve* whose exact location is less important than is its overall shape. Finally, in all of this, one might be looking for functions of more than one variable, such as *tensor product splines*.

Care has been taken to make this work as painless and intuitive as possible. In particular, the user need not worry about just how splines are constructed or stored for later use, nor need the casual user worry about such items as "breaks" or "knots" or "coefficients". It is enough to know that each function constructed is just another variable that is freely usable as input (where appropriate) to many of the commands, including all commands beginning with `fn`, which stands for `function`. At times, it may be also useful to know that, internal to the toolbox, splines are stored in different forms, with the command `fn2fm` available to convert between forms.

At present, the toolbox supports two major forms for the representation of piecewise-polynomial functions, because each has been found to be superior to the other in certain common situations. The B-form is particularly useful during the construction of a spline, while the ppform is more efficient when the piecewise-polynomial function is to be evaluated extensively. These two forms are almost exactly the B-representation and the pp representation used in *A Practical Guide to Splines*.

But, over the years, the Curve Fitting Toolbox spline functions have gone beyond the programs in *A Practical Guide to Splines*. The toolbox now supports the `scattered translates' form, or stform, in order to handle the construction and use of bivariate thin-plate splines, and also two ways to represent rational splines, the rBform and the rpform, in order to handle NURBS.

Splines can be very effective for data fitting because the linear systems to be solved for this are banded, hence the work needed for their solution, done properly, grows only linearly with the number of data points. In particular, the MATLAB sparse matrix facilities are used in the Curve Fitting Toolbox spline functions when that is more efficient than the toolbox's own equation solver, `slvblk`, which relies on the fact that some of the linear systems here are even almost block diagonal.

All polynomial spline construction commands are equipped to produce bivariate (or even multivariate) piecewise-polynomial functions as tensor products of the univariate functions used here, and the various `fn...` commands also work for these multivariate functions.

There are various examples, all accessible through the Help browser. You are strongly urged to have a look at some of them, or at the GUI `splinetool`, to help you work with splines.

Splines

Consider the set

$$S := \Pi_{\xi,k}^{\mu}$$

of all (scalar-valued) piecewise-polynomials of order k with breaks $\xi_1 < ... < \xi_{l+1}$ that, for $i = 2...l$, may have a jump across ξ_i in its μ_ith derivative but have no jump there in any lower order derivative. This set is a linear space, in the sense that any scalar multiple of a function in S is again in S, as is the sum of any two functions in S.

Accordingly, S contains a basis (in fact, infinitely many bases), that is, a sequence $f_1,...,f_n$ so that every f in S can be written *uniquely* in the form

$$f(x) = \sum_{j=1}^{n} f_j(x)a_j,$$

for suitable coefficients a_j. The number n appearing here is the *dimension* of the linear space S. The coefficients a_j are often referred to as the *coordinates* of f with respect to this basis.

In particular, according to the Curry-Schoenberg Theorem, our space S has a basis consisting of B-splines, namely the sequence of all B-splines of the form $B(\cdot \mid t_j, ..., t_{j+k}), j = 1...n$, with the knot sequence t obtained from the break sequence ξ and the sequence μ by the following conditions:

- Have both ξ_1 and ξ_{l+1} occur in t exactly k times
- For each $i = 2:l$, have ξ_i occur in t exactly $k - \mu_i$ times
- Make sure the sequence is nondecreasing and only contains elements from ξ

Note the correspondence between the multiplicity of a knot and the smoothness of the spline across that knot. In particular, at a simple knot, that is a knot that appears exactly once in the knot sequence, only the $(k - 1)$st derivative may be discontinuous.

MATLAB Splines

The MATLAB technical computing environment provides spline approximation via the command `spline`. If called in the form `cs = spline(x,y)`, it returns the ppform of the cubic spline with break sequence x that takes the value `y(i)` at `x(i)`, all i, and satisfies the not-a-knot end condition. In other words, the command `cs = spline(x,y)` gives the same result as the command `cs =`

`csapi(x,y)` available in the Curve Fitting Toolbox spline functions. But only the latter also works when x,y describe multivariate gridded data. In MATLAB, cubic spline interpolation to multivariate gridded data is provided by the command `interpn(x1,...,xd,v,y1,...,yd,'spline')` which returns values of the interpolating tensor product cubic spline at the grid specified by y1,...,yd.

Further, any of the Curve Fitting Toolbox spline `fn...` commands can be applied to the output of the MATLAB `spline(x,y)` command, with simple versions of the Curve Fitting Toolbox spline commands `fnval`, `ppmak`, `fnbrk` available directly in MATLAB, as the commands `ppval`, `mkpp`, `unmkpp`, respectively.

Expected Background

The Curve Fitting Toolbox spline functions started out as an extension of the MATLAB environment of interest to experts in spline approximation, to aid them in the construction and testing of new methods of spline approximation. Such people will have mastered the material in *A Practical Guide to Splines*.

However, the basic commands for constructing and using spline approximations are set up to be usable with no more knowledge than it takes to understand what it means to, say, construct an interpolant or a least squares approximant to some data, or what it means to differentiate or integrate a function.

With that in mind, there are sections, like "Cubic Spline Interpolation" on page 9-2, that are meant even for the novice, while sections devoted to a detailed example, like the one on constructing a Chebyshev spline or on constructing and using tensor products, are meant for users interested in developing their own spline commands.

Vector Data Type Support

The Curve Fitting Toolbox spline functions can handle *vector*-valued splines, i.e., splines whose values lie in R^d. Since MATLAB started out with just one variable type, that of a matrix, there is even now some uncertainty about how to deal with vectors, i.e., lists of numbers. MATLAB sometimes stores such a list in a matrix with just one row, and other times in a matrix with just one column. In the first instance, such a *1-row matrix* is called a row-vector; in the second instance, such a *1-column matrix* is called a column-vector. Either way, these are merely different ways for *storing* vectors, not different *kinds* of vectors.

In this toolbox, *vectors*, i.e., lists of numbers, may also end up stored in a 1-row matrix or in a 1-column matrix, but with the following agreements.

A point in R^d, i.e., a d-vector, is always stored as a column vector. In particular, if you want to supply an n-list of d-vectors to one of the commands, you are expected to provide that list as the n columns of a matrix of size `[d,n]`.

While other lists of numbers (e.g., a knot sequence or a break sequence) may be stored internally as row vectors, you may supply such lists as you please, as a row vector or a column vector.

Spline Function Naming Conventions

Most of the spline commands in this toolbox have names that follow one of the following patterns:

`cs...` commands construct cubic splines (in ppform)

`sp...` commands construct splines in B-form

`fn...` commands operate on spline functions

`..2...` commands convert something

`..api` commands construct an approximation by interpolation

`..aps` commands construct an approximation by smoothing

`..ap2` commands construct a least-squares approximation

`...knt` commands construct (part of) a particular knot sequence

`...dem` commands are examples.

Arguments for Curve Fitting Toolbox Spline Functions

For ease of use, most Curve Fitting Toolbox spline functions have default arguments. In the reference entry under Syntax, we usually first list the function with all *necessary* input arguments and then with all *possible* input arguments. When there is more than one optional argument, then, sometimes, but not always, their exact order is immaterial. When their order does matter, you have to specify every optional argument preceding the one(s) you are interested in. In this situation, you can specify the default value for an optional argument by using `[]` (the empty matrix) as the input for it. The description in the reference page tells you the default value for each optional input argument.

As in MATLAB, only the output arguments explicitly specified are returned to the user.

Acknowledgments

MathWorks® would like to acknowledge the contributions of **Carl de Boor** to the Curve Fitting Toolbox spline functions. Professor de Boor authored the Spline Toolbox™ from its first release until Version 3.3.4 (2008).

Professor de Boor received the John von Neumann Prize in 1996 and the National Medal of Science in 2003. He is a member of both the American Academy of Arts and Sciences and the National Academy of Sciences. He is the author of *A Practical Guide to Splines* (Springer, 2001).

Some of the spline function naming conventions are the result of a discussion with Jörg Peters, then a graduate student in Computer Sciences at the University of Wisconsin-Madison.

Simple Spline Examples

Cubic Spline Interpolation

In this section...

Cubic Spline Interpolant of Smooth Data

Suppose you want to interpolate some smooth data, e.g., to

```
rng(6), x = (4*pi)*[0 1 rand(1,15)]; y = sin(x);
```

You can use the cubic spline interpolant obtained by

```
cs = csapi(x,y);
```

and plot the spline, along with the data, with the following code:

```
fnplt(cs);
hold on
plot(x,y,'o')
legend('cubic spline','data')
hold off
```

This produces a figure like the following.
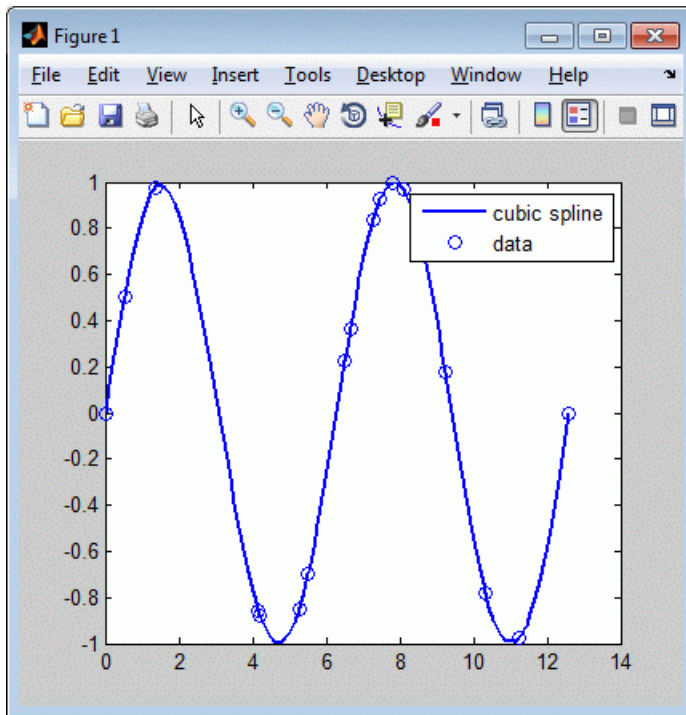
Cubic Spline Interpolant of Smooth Data

This is, more precisely, the cubic spline interpolant with the not-a-knot end conditions, meaning that it is the unique piecewise cubic polynomial with two continuous derivatives with breaks at all *interior* data sites except for the leftmost and the rightmost one. It is the same interpolant as produced by the MATLAB `spline` command, `spline(x,y)`.

Periodic Data

The sine function is 2π-periodic. To check how well your interpolant does on that score, compute, e.g., the difference in the value of its first derivative at the two endpoints,

```
diff(fnval(fnder(cs),[0 4*pi]))
ans = -.0100
```

which is not so good. If you prefer to get an interpolant whose first and second derivatives at the two endpoints, `0` and `4*pi`, match, use instead the command `csape` which permits specification of many different kinds of end conditions, including periodic end conditions. So, use instead

```
pcs = csape(x,y,'periodic');
```

for which you get

```
diff(fnval(fnder(pcs),[0 4*pi]))
```

Output is `ans = 0` as the difference of end slopes. Even the difference in end second derivatives is small:

```
diff(fnval(fnder(pcs,2),[0 4*pi]))
```

Output is `ans = -4.6074e-015`.

Other End Conditions

Other end conditions can be handled as well. For example,

```
cs = csape(x,[3,y,-4],[1 2]);
```

provides the cubic spline interpolant with breaks at the $x(i)$ and with its slope at the leftmost data site equal to 3, and its second derivative at the rightmost data site equal to -4.

General Spline Interpolation

If you want to interpolate at sites other than the breaks and/or by splines other than cubic splines with simple knots, then you use the `spapi` command. In its simplest form, you would say `sp = spapi(k,x,y)`; in which the first argument, `k`, specifies the *order* of the interpolating spline; this is the number of coefficients in each polynomial piece, i.e., 1 more than the nominal degree of its polynomial pieces. For example, the next figure shows a linear, a quadratic, and a quartic spline interpolant to your data, as obtained by the statements

```
sp2 = spapi(2,x,y); fnplt(sp2,2), hold on
sp3 = spapi(3,x,y); fnplt(sp3,2,'k--'),
sp5 = spapi(5,x,y); fnplt(sp5,2,'r-.'), plot(x,y,'o')
legend('linear','quadratic','quartic','data'), hold off
```
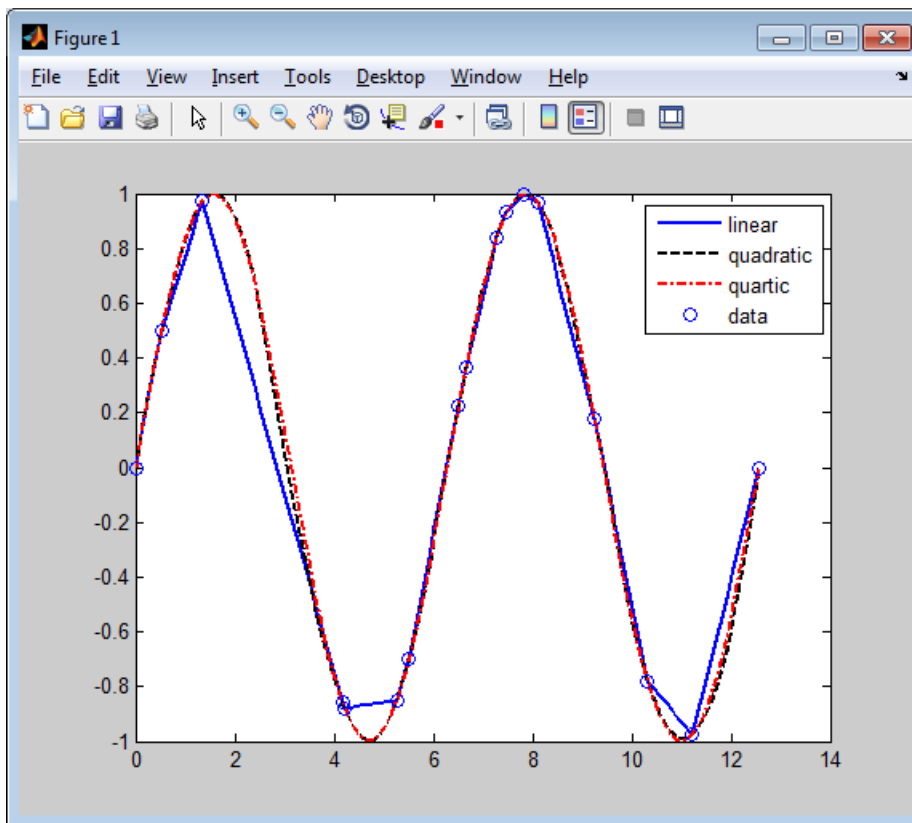

Spline Interpolants of Various Orders of Smooth Data

Even the cubic spline interpolant obtained from `spapi` is different from the one provided by `csapi` and `spline`. To emphasize their difference, compute and plot their second derivatives, as follows:

```
fnplt(fnder(spapi(4,x,y),2)), hold on,
fnplt(fnder(csapi(x,y),2),2,'k--'),plot(x,zeros(size(x)),'o')
legend('from spapi','from csapi','data sites'), hold off
```

This gives the following graph:
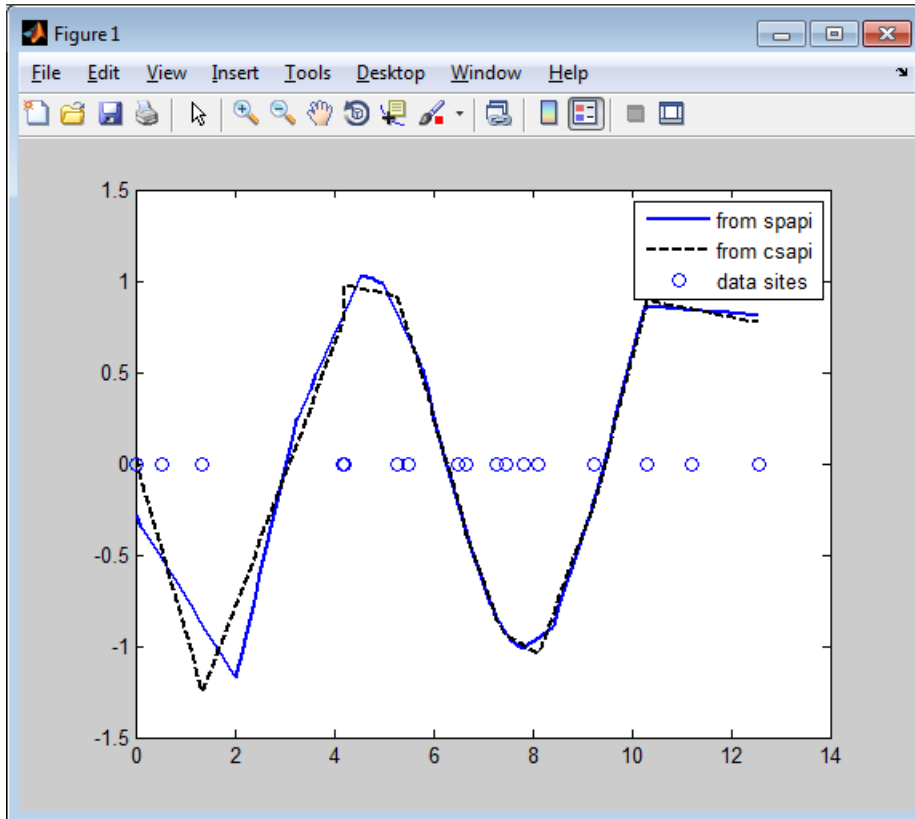
Second Derivative of Two Cubic Spline Interpolants of the Same Smooth Data

Since the second derivative of a cubic spline is a broken line, with vertices at the breaks of the spline, you can see clearly that csapi places breaks at the data sites, while spapi does not.

Knot Choices

It is, in fact, possible to specify explicitly just where the spline interpolant should have its breaks, using the command sp = spapi(knots,x,y); in which the sequence knots supplies, in a certain way, the breaks to be used. For example, recalling that you had chosen y to be sin(x), the command

```
ch = spapi(augknt(x,4,2),[x x],[y cos(x)]);
```

provides a cubic Hermite interpolant to the sine function, namely the piecewise cubic function, with breaks at all the x(i)'s, that matches the sine function in value *and* slope at all the x(i)'s. This makes the interpolant continuous with continuous first derivative but, in general, it has jumps across the breaks in its second derivative. Just how does this command know which part of the data value array [y cos(x)] supplies the values and which the slopes? Notice that the data site array here is given as [x x], i.e., each data site appears twice. Also notice that y(i) is associated with the first occurrence of x(i), and cos(x(i)) is associated with the second occurrence of x(i). The data

value associated with the first appearance of a data site is taken to be a function value; the data value associated with the second appearance is taken to be a slope. If there were a third appearance of that data site, the corresponding data value would be taken as the second derivative value to be matched at that site. See "Constructing and Working with B-form Splines" on page 10-17 for a discussion of the command `augknt` used here to generate the appropriate "knot sequence".

Smoothing

What if the data are noisy? For example, suppose that the given values are

```
noisy = y + .3*(rand(size(x))-.5);
```

Then you might prefer to approximate instead. For example, you might try the cubic smoothing spline, obtained by the command

```
scs = csaps(x,noisy);
```

and plotted by

```
fnplt(scs,2), hold on, plot(x,noisy,'o'),
legend('smoothing spline','noisy data'), hold off
```

This produces a figure like this:
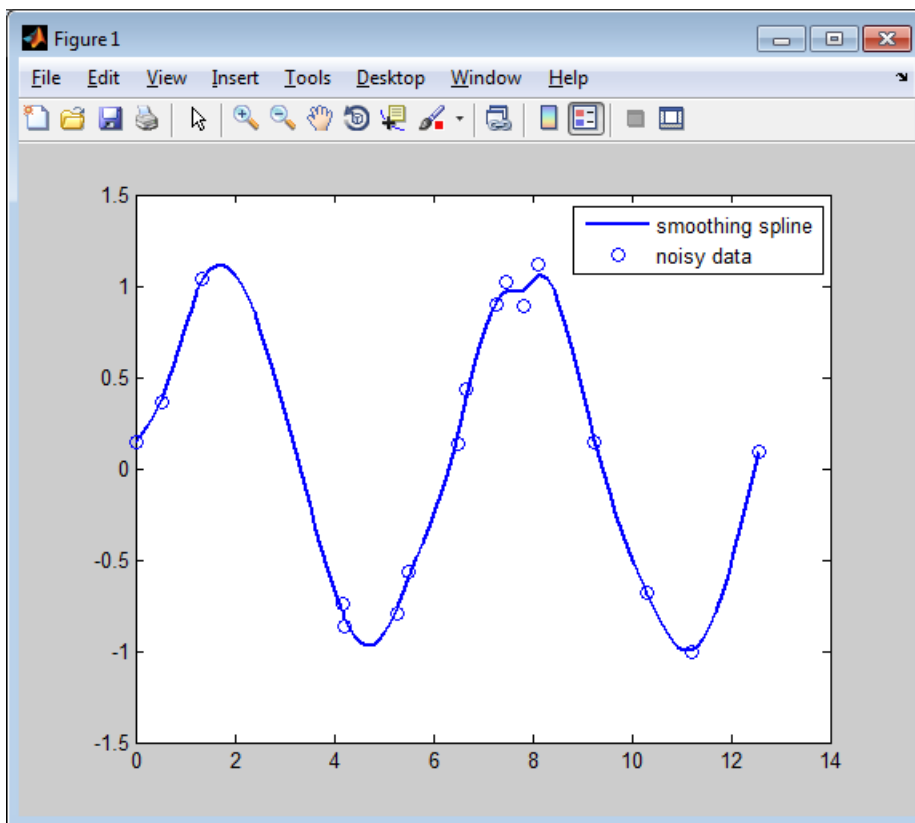
Cubic Smoothing Spline of Noisy Data

If you don't like the level of smoothing done by `csaps(x,y)`, you can change it by specifying the smoothing parameter, `p`, as an optional third argument. Choose this number anywhere between 0 and

1. As `p` changes from 0 to 1, the smoothing spline changes, correspondingly, from one extreme, the least squares straight-line approximation to the data, to the other extreme, the "natural" cubic spline interpolant to the data. Since `csaps` returns the smoothing parameter actually used as an optional second output, you could now experiment, as follows:

```
[scs,p] = csaps(x,noisy); fnplt(scs,2), hold on
fnplt(csaps(x,noisy,p/2),2,'k--'),
fnplt(csaps(x,noisy,(1+p)/2),2,'r:'), plot(x,noisy,'o')
legend('smoothing spline','more smoothed','less smoothed',...
'noisy data'), hold off
```

This produces the following picture.
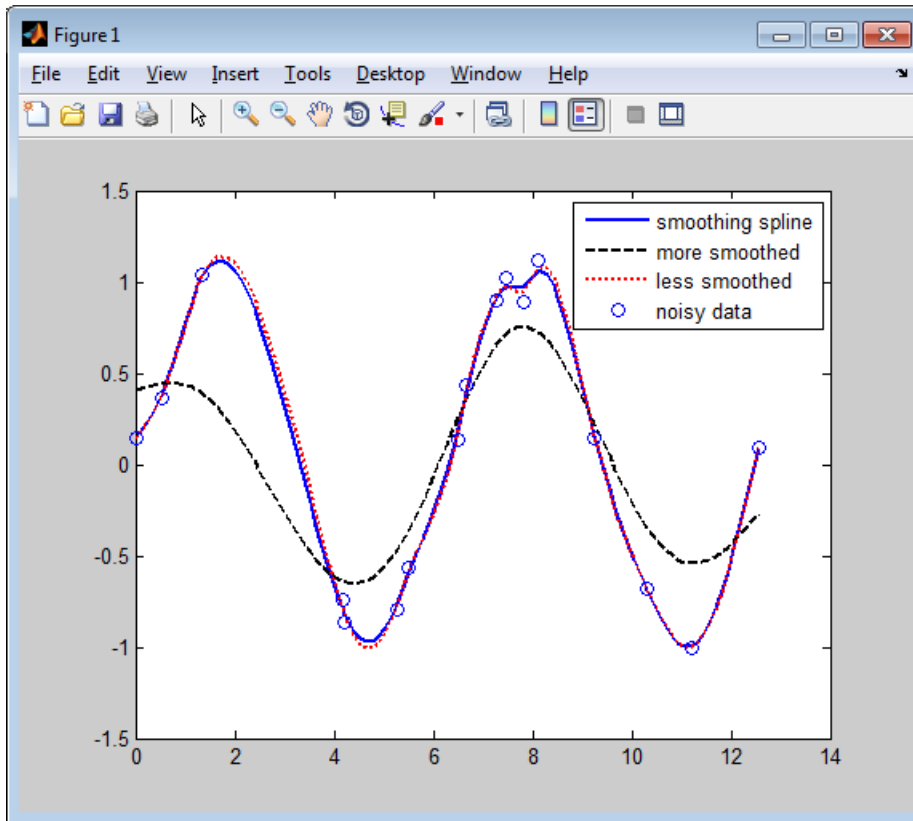
Noisy Data More or Less Smoothed

At times, you might prefer simply to get the smoothest cubic spline `sp` that is within a specified tolerance `tol` of the given data in the sense that `norm(noisy - fnval(sp,x))^2 <= tol`. You create this spline with the command `sp = spaps(x,noisy,tol)` for your defined tolerance `tol`.

Least Squares

If you prefer a least squares approximant, you can obtain it by the statement `sp = spap2(knots,k,x,y);` in which both the knot sequence `knots` and the order `k` of the spline must be provided.

The popular choice for the order is 4, and that gives you a cubic spline. If you have no clear idea of how to choose the knots, simply specify the number of polynomial pieces you want used. For example,

```
sp = spap2(3,4,x,y);
```

gives a cubic spline consisting of three polynomial pieces. If the resulting error is uneven, you might try for a better knot distribution by using `newknt` as follows:

```
sp = spap2(newknt(sp),4,x,y);
```

Vector-Valued Functions

The toolbox supports *vector-valued* splines. For example, if you want a spline *curve* through given planar points $(x(i), y(i))$, $i = 1, ..., n$, then the following code defines some data and then creates and plots such a spline curve, using chord-length parameterization and cubic spline interpolation with the not-a-knot end condition.

```
x=[19 43 62 88 114 120 130 129 113 76 135 182 232 298 ...
 348 386 420 456 471 485 463 444 414 348 275 192 106 ...
 30 48 83 107 110 109 92 66 45 23 22 30 40 55 55 52 34 20 16];
y=[306 272 240 215 218 237 275 310 368 424 425 427 428 ...
 397 353 302 259 200 148 105 77 47 28 17 10 12 23 41 43 ...
 77 96 133 155 164 157 148 142 162 181 187 192 202 217 245 266 303];

xy = [x;y]; df = diff(xy,1,2);
t = cumsum([0, sqrt([1 1]*(df.*df))]);
cv = csapi(t,xy);
fnplt(cv), hold on, plot(x,y,'o'), hold off
```
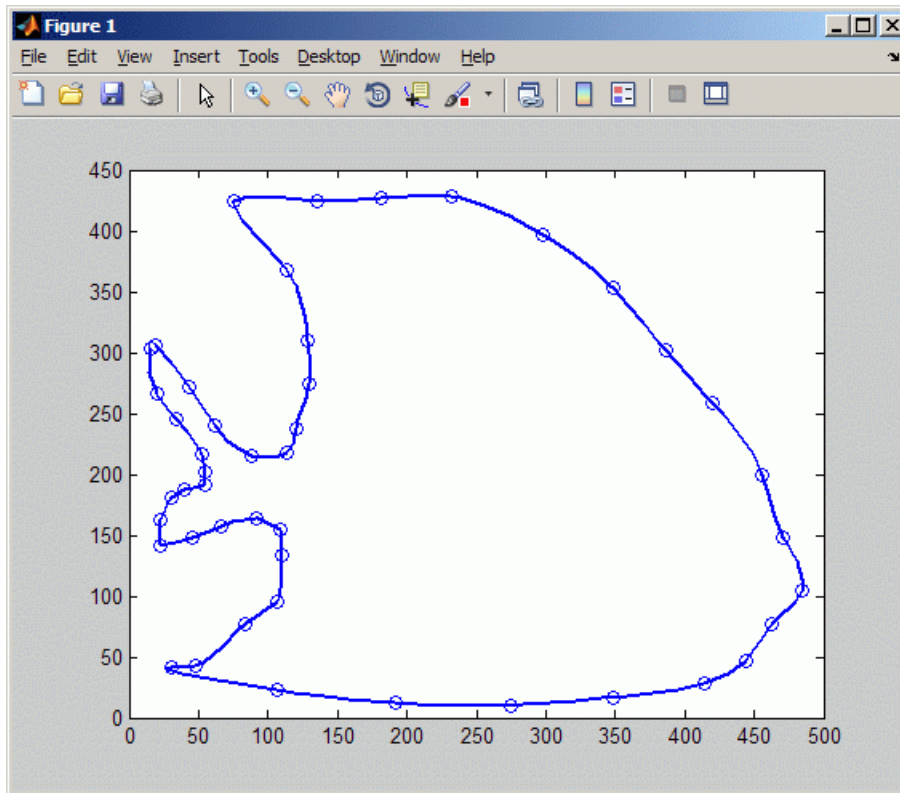

If you then wanted to know the area enclosed by this curve, you would want to evaluate the integral $\int y(t)dx(t) = \int y(t)Dx(t)dt$, with $(x(t),y(t))$ the point on the curve corresponding to the parameter value t. For the spline curve in `cv` just constructed, this can be done exactly in one (somewhat complicated) command:

```
area = diff(fnval(fnint( ...
        fncmb(fncmb(cv,[0 1]),'*',fnder(fncmb(cv,[1 0]))) ...
                        ),fnbrk(cv,'interval')));
```

To explain, y=fncmb(cv,[0 1]) picks out the second component of the curve in cv, Dx=fnder(fncmb(cv,[1 0])) provides the derivative of the first component, and yDx=fncmb(y,'*',Dx) constructs their pointwise product. Then IyDx=fnint(yDx) constructs the indefinite integral of yDx and, finally, diff(fnval(IyDx,fnbrk(cv,'interval'))) evaluates that indefinite integral at the endpoints of the basic interval and then takes the difference of the second from the first value, thus getting the definite integral of yDx over its basic interval. Depending on whether the enclosed area is to the right or to the left as the curve point travels with increasing parameter, the resulting number is either positive or negative.

Further, all the values Y (if any) for which the point (X,Y) lies on the spline curve in cv just constructed can be obtained by the following (somewhat complicated) command:

```
X=250; %Define a value of X
Y = fnval(fncmb(cv,[0 1]), ...
        mean(fnzeros(fncmb(fncmb(cv,[1 0]),'-',X))))
```

To explain: x = fncmb(cv,[1 0]) picks out the first component of the curve in cv; xmX = fncmb(x,'-',X) translates that component by X; t = mean(fnzeros(xmX)) provides all the parameter values for which xmX is zero, i.e., for which the first component of the curve equals X; y = fncmb(cv,[0,1]) picks out the second component of the curve in cv; and, finally, Y = fnval(y,t) evaluates that second component at those parameter sites at which the first component of the curve in cv equals X.

As another example of the use of vector-valued functions, suppose that you have solved the equations of motion of a particle in some specified force field in the plane, obtaining, at discrete times $t_j = t(j), j = 1:n$, the position $(x(t_j), y(t_j))$ as well as the velocity $(\dot{x}(t_j), \dot{y}(t_j))$ stored in the 4-vector $z(:,j)$, as you would if, in the standard way, you had solved the equivalent first-order system numerically. Then the following statement, which uses cubic Hermite interpolation, will produce a plot of the particle path: fnplt(spapi(augknt(t,4,2),t,reshape(z,2,2*n))).

Fitting Values at N-D Grid with Tensor-Product Splines

Vector-valued splines are also used in the approximation to *gridded data*, in any number of variables, using *tensor-product* splines. The same spline-construction commands are used, only the form of the input differs. For example, if x is an m-vector, y is an n-vector, and z is an array of size [m,n], then cs = csapi({x,y},z); describes a bicubic spline *f* satisfying $f(x(i),y(j))=z(i,j)$ for $i=1$:m, $j=1$:n. Such a multivariate spline can be vector-valued. For example,

```
x = 0:4; y=-2:2; s2 = 1/sqrt(2);
z(3,:,:) = [0 1 s2 0 -s2 -1 0].'*[1 1 1 1 1];
z(2,:,:) = [1 0 s2 1 s2 0 -1].'*[0 1 0 -1 0];
z(1,:,:) = [1 0 s2 1 s2 0 -1].'*[1 0 -1 0 1];
sph = csape({x,y},z,{'clamped','periodic'});
fnplt(sph), axis equal, axis off
```

gives a perfectly acceptable sphere. Its projection onto the (x, z)-plane is plotted by

```
fnplt(fncmb(sph,[1 0 0; 0 0 1])), axis equal, axis off
```

Both plots are shown below.
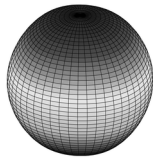
A Sphere Made by a 3-D-Valued Bivariate Tensor Product Spline
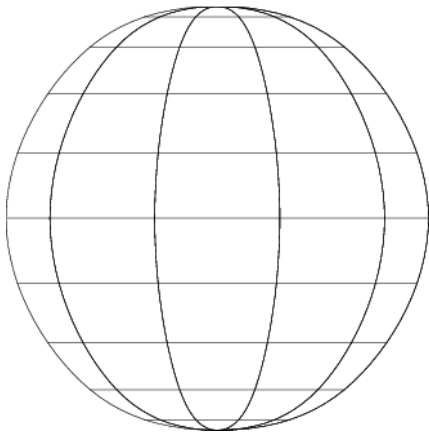
Planar Projection of Spline Sphere

9-11

Fitting Values at Scattered 2-D Sites with Thin-Plate Smoothing Splines

Tensor-product splines are good for gridded (bivariate and even multivariate) data. For work with scattered bivariate data, the toolbox provides the thin-plate smoothing spline. Suppose you have given data values y(j) at scattered data sites x(:,j), j=1:N, in the plane. To give a specific example,

```
n = 65; t = linspace(0,2*pi,n+1);
x = [cos(t);sin(t)]; x(:,end) = [0;0];
```

provides 65 sites, namely 64 points equally spaced on the unit circle, plus the center of that circle. Here are corresponding data values, namely noisy values of the very nice function
$g(x) = (x(1) + 1/2)^2 + (x(2) + 1/2)^2$.

```
y = (x(1,:)+.5).^2 + (x(2,:)+.5).^2;
noisy = y + (rand(size(y))-.5)/3;
```

Then you can compute a reasonable approximation to these data by

```
st = tpaps(x,noisy);
```

and plot the resulting approximation along with the noisy data by

```
fnplt(st); hold on
plot3(x(1,:),x(2,:),noisy,'wo','markerfacecolor','k')
hold off
```

and so produce the following picture:
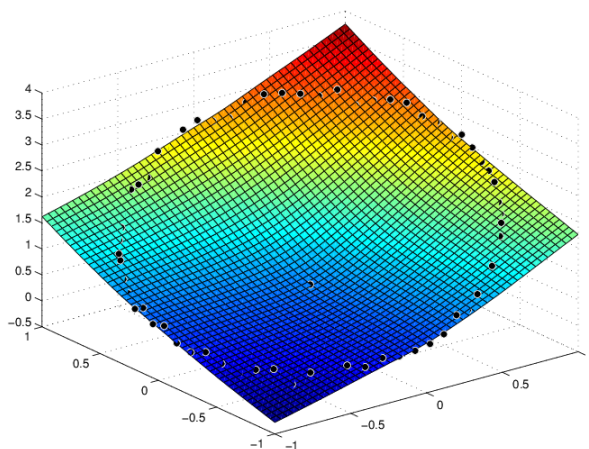
Thin-Plate Smoothing Spline Approximation to Noisy Data

Postprocessing Splines

You can use the following commands with any example spline, such as the `cs`, `ch` and `sp` examples constructed in the section "Cubic Spline Interpolation" on page 9-2.

First construct a spline, for example:

```
sp = spmak(1:6,0:2)
```

To display a plot of the spline:

```
fnplt(sp)
```

To get the value at `a`, use the syntax `fnval(f,a)`, for example:

```
fnval(sp,4)
```

To construct the spline's second derivative:

```
DDf = fnder(fnder(sp))
```

An alternative way to construct the second derivative:

```
DDf = fnder(sp,2);
```

To obtain the spline's definite integral over an interval [`a`..`b`], in this example from 2 to 5:

```
diff(fnval(fnint(sp),[2;5]))
```

To compute the difference between two splines, use the form `fncmb(sp1,'-',sp2)`, for example:

```
fncmb(sp,'-',DDf);
```

Types of Splines

Types of Splines: ppform and B-form

In this section...
"Polynomials vs. Splines" on page 10-2
"ppform" on page 10-2
"B-form" on page 10-2
"Knot Multiplicity" on page 10-3

Polynomials vs. Splines

Polynomials are the approximating functions of choice when a smooth function is to be approximated locally. For example, the truncated Taylor series

$$\sum_{i=0}^{n} (x-a)^i D^i f(a)/i!$$

provides a satisfactory approximation for $f(x)$ if f is sufficiently smooth and x is sufficiently close to a. But if a function is to be approximated on a larger interval, the degree, n, of the approximating polynomial may have to be chosen unacceptably large. The alternative is to subdivide the interval $[a..b]$ of approximation into sufficiently small intervals $[\xi_j..\xi_{j+1}]$, with $a = \xi_1 < \cdots < \xi_{l+1} = b$, so that, on each such interval, a polynomial p_j of relatively low degree can provide a good approximation to f. This can even be done in such a way that the polynomial pieces blend smoothly, i.e., so that the resulting patched or composite function $s(x)$ that equals $p_j(x)$ for $x \epsilon [\xi_j \, \xi_{j+1}]$, all j, has several continuous derivatives. Any such smooth piecewise polynomial function is called a *spline*. I.J. Schoenberg coined this term because a twice continuously differentiable cubic spline with sufficiently small first derivative approximates the shape of a draftsman's spline.

There are two commonly used ways to represent a polynomial spline, the ppform and the B-form. In this toolbox, a spline in ppform is often referred to as a piecewise polynomial, while a piecewise polynomial in B-form is often referred to as a spline. This reflects the fact that piecewise polynomials and (polynomial) splines are just two different views of the same thing.

ppform

The *ppform* of a polynomial spline of *order k* provides a description in terms of its *breaks* $\xi_1..\xi_{l+1}$ and the *local polynomial coefficients* c_{ji} of its l pieces.

$$p_j(x) = \sum_{i=1}^{k} \left(x - \xi_j\right)^{k-i} c_{ji}, \; j = 1:l$$

For example, a cubic spline is of order 4, corresponding to the fact that it requires four coefficients to specify a cubic polynomial. The ppform is convenient for the evaluation and other *uses* of a spline.

B-form

The *B-form* has become the standard way to represent a spline during its *construction*, because the B-form makes it easy to build in smoothness requirements across breaks and leads to banded linear systems. The B-form describes a spline as a weighted sum

$$\sum_{j=1}^{n} B_{j,k} a_j$$

of B-splines of the required order k, with their number, n, at least as big as k–1 plus the number of polynomial pieces that make up the spline. Here, $B_{j,k} = B\ (\cdot|t_j, \ldots, t_{j+k})$ is the jth B-spline of *order k* for the *knot sequence* $t_1 \le t_2 \le \cdots \le t_{n+k}$. In particular, $B_{j,k}$ is piecewise-polynomial of degree $< k$, with breaks $t_j, \ldots, t_{j+k}$, is nonnegative, is zero outside the interval $[t_j, ..t_{j+k}]$, and is so normalized that

$$\sum_{j=1}^{n} B_{j,k}(x) = 1 \ \ on \ \ [t_k..t_{n+1}]$$

Knot Multiplicity

The multiplicity of the knots governs the smoothness, in the following way: If the number τ occurs exactly r times in the sequence $t_j, \ldots t_{j+k}$, then $B_{j,k}$ and its first *k-r-1* derivatives are continuous across the break τ, while the *(k-r)*th derivative has a jump at τ. You can experiment with all these properties of the B-spline in a very visual and interactive way using the command `bspligui`.

See Also

Related Examples

- "Constructing and Working with ppform Splines" on page 10-10
- "Constructing and Working with B-form Splines" on page 10-17

B-Splines and Smoothing Splines

In this section...
"B-Spline Properties" on page 10-4
"Variational Approach and Smoothing Splines" on page 10-5

In this toolbox, the definition of a B-spline with knots t_j, ..., t_{j+k} is given by

$$B_{j,k}(x) = B(x | t_j, ..., t_{j+k}) = (t_{j+k} - t_j)[t_j, ..., t_{j+k}](x - \cdot)_+^{k-1}.$$

This is only one of several reasonable normalizations of the B-spline. It is chosen so that

$$\sum_{j=1}^{n} B_{j,k}(x) = 1, \quad t_k \le x \le t_{n+1}.$$

But, instead of trying to understand the above formula for the B-spline, look at the reference pages for the GUI `bspligui` for some of the basic properties of the B-spline, and use that GUI to gain some firsthand experience with this intriguing function. Its most important property for the purposes of this toolbox is also the reason the letter B is in its name:

Every space of (univariate) piecewise-polynomials of a given order has a Basis consisting of B-*splines (hence the "B" in B-spline).*

B-Spline Properties

Because $B_{j,k}$ is nonzero only on the interval $(t_j..t_{j+k})$, the linear system for the B-spline coefficients of the spline to be determined, by interpolation or least squares approximation, or even as the approximate solution of some differential equation, is *banded*, making the solving of that linear system particularly easy. For example, to construct a spline s of order k with knot sequence $t_1 \le t_2 \le \cdots \le t_{n+k}$ so that $s(x_i)=y_i$ for $i=1, ..., n$, use the linear system

$$\sum_{j=1}^{n} B_{j,k}(x_i)a_j = y_i \ i = 1:n$$

for the unknown B-spline coefficients a_j in which each equation has at most k nonzero entries.

Also, many theoretical facts concerning splines are most easily stated and/or proved in terms of B-splines. For example, it is possible to match arbitrary data at sites $x_1 < \cdots < x_n$ uniquely by a spline of order k with knot sequence $(t_1, ..., t_{n+k})$ if and only if $B_{j,k}(x_j) \ne 0$ for all j (Schoenberg-Whitney Conditions). Computations with B-splines are facilitated by stable *recurrence relations*

$$B_{j,k}(x) = \frac{x - t_j}{t_{j+k-1} - t_j}B_{j,k-1}(x) + \frac{t_{j+k} - x}{t_{j+k} - t_{j+1}}B_{j+1,k-1}(x)$$

which are also of help in the conversion from B-form to ppform. The dual functional

$$a_j(s): = \sum_{i < k}(-D)^{k-i-1}\Psi_j(\tau)D^i s(\tau)$$

provides a useful expression for the jth B-spline coefficient of the spline s in terms of its value and derivatives at an arbitrary site τ between t_j and t_{j+k}, and with $\psi_j(t):=(t_{j+1}-t)\cdots(t_{j+k-1}-t)/(k-1)!$ It can be

used to show that $a_j(s)$ is closely related to s on the interval $[t_j..t_{j+k}]$, and seems the most efficient means for converting from ppform to B-form.

Variational Approach and Smoothing Splines

The above *constructive* approach is not the only avenue to splines. In the *variational* approach, a spline is obtained as a *best interpolant*, e.g., as the function with smallest *m*th derivative among all those matching prescribed function values at certain sites. As it turns out, among the many such splines available, only those that are piecewise-polynomials or, perhaps, piecewise-exponentials have found much use. Of particular practical interest is the *smoothing spline* $s = s_p$ which, for given data (x_i, y_i) with $x \epsilon [a..b]$, all i, and given corresponding positive weights w_i, and for given *smoothing parameter p*, minimizes

$$p \sum_i w_i |y_i - f(x_i)|^2 + (1 - p) \int_a^b |D^m f(t)|^2 dt$$

over all functions f with m derivatives. It turns out that the smoothing spline s is a spline of order *2m* with a break at every data site. The smoothing parameter, *p*, is chosen artfully to strike the right balance between wanting the *error measure*

$$E(s) = \sum_i w_i |y_i - s(x_i)|^2$$

small and wanting the *roughness measure*

$$F(D^m s) = \int_a^b |D^m s(t)|^2 dt$$

small. The hope is that s contains as much of the information, and as little of the supposed noise, in the data as possible. One approach to this (used in `spaps`) is to make $F(D^m f)$ as small as possible subject to the condition that $E(f)$ be no bigger than a prescribed tolerance. For computational reasons, `spaps` uses the (equivalent) smoothing parameter $\rho = p/(1-p)$, i.e., minimizes $\rho E(f) + F(D^m f)$. Also, it is useful at times to use the more flexible roughness measure

$$F(D^m s) = \int_a^b \lambda(t) |D^m s(t)|^2 dt$$

with λ a suitable positive weight function.

See Also

Related Examples
* "Constructing and Working with B-form Splines" on page 10-17

Multivariate and Rational Splines

In this section...
"Multivariate Splines" on page 10-6
"Rational Splines" on page 10-7

Multivariate Splines

Multivariate splines can be obtained from univariate splines by the tensor product construct. For example, a trivariate spline in B-form is given by

$$f(x, y, z) = \sum_{u=1}^{U} \sum_{v=1}^{V} \sum_{w=1}^{W} B_{u,k}(x) B_{v,l}(y) B_{w,m}(z) a_{u,v,w}$$

with $B_{u,k}, B_{v,l}, B_{w,m}$ univariate B-splines. Correspondingly, this spline is of order k in x, of order l in y, and of order m in z. Similarly, the ppform of a tensor-product spline is specified by break sequences in each of the variables and, for each hyper-rectangle thereby specified, a coefficient array. Further, as in the univariate case, the coefficients may be vectors, typically 2-vectors or 3-vectors, making it possible to represent, e.g., certain surfaces in $\Re^3$.

A very different bivariate spline is the *thin-plate spline*. This is a function of the form

$$f(x) = \sum_{j=1}^{n-3} \Psi(x - c_j) a_j + x(1) a_{n-2} + x(2) a_{n-1} + a_n$$

with $\psi(x) = |x|^2 \log|x|^2$ the thin-plate spline basis function, and $|x|$ denoting the Euclidean length of the vector x. Here, for convenience, denote the independent variable by x, but x is now a *vector* whose two components, $x(1)$ and $x(2)$, play the role of the two independent variables earlier denoted x and y. Correspondingly, the sites c_j are points in $\Re^2$.

Thin-plate splines arise as bivariate *smoothing splines*, meaning a thin-plate spline minimizes

$$p \sum_{i=1}^{n-3} |y_i - fc_i2| + (1 - p) \int \left(|D_1 D_1 f|^2 + 2|D_1 D_2 f|^2 + |D_2 D_2 f|^2 \right)$$

over all sufficiently smooth functions f. Here, the y_i are data values given at the data sites c_i, p is the smoothing parameter, and $D_j f$ denotes the partial derivative of f with respect to $x(j)$. The integral is taken over the entire $\Re^2$. The upper summation limit, n–3, reflects the fact that 3 degrees of freedom of the thin-plate spline are associated with its polynomial part.

Thin-plate splines are functions in stform, meaning that, up to certain polynomial terms, they are a weighted sum of arbitrary or scattered translates $\Psi(\cdot\text{ -c})$ of one fixed function, Ψ. This so-called basis function for the thin-plate spline is special in that it is radially symmetric, meaning that $\Psi(x)$ only depends on the Euclidean length, $|x|$, of x. For that reason, thin-plate splines are also known as RBFs or radial basis functions. See "Constructing and Working with stform Splines" on page 10-28 for more information.

Rational Splines

A *rational spline* is any function of the form $r(x) = s(x)/w(x)$, with both s and w splines and, in particular, w a scalar-valued spline, while s often is vector-valued.

Rational splines are attractive because it is possible to describe various basic geometric shapes, like conic sections, exactly as the range of a rational spline. For example, a circle can so be described by a quadratic rational spline with just two pieces.

In this toolbox, there is the additional requirement that both s and w be of the same form and even of the same order, and with the same knot or break sequence. This makes it possible to store the rational spline r as the ordinary spline R whose value at x is the vector $[s(x);w(x)]$. Depending on whether the two splines are in B-form or ppform, such a representation is called here the rBform or the rpform of such a rational spline.

It is easy to obtain r from R. For example, if `v` is the value of R at x, then `v(1:end-1)/v(end)` is the value of r at x. As another example, consider getting derivatives of r from those of R. Because $s = wr$, Leibniz' rule tells us that

$$D^m s = \sum_{j=0}^{m} \binom{m}{j} D^j w D^{m-j} r$$

where $D^m s$ the mth derivative of s.

Hence, if `v(:,j)` contains $D^{j-1}R(x)$, $j = 1...\text{m} + 1$, then

$$\left(\left((v(1:\text{end} - 1, m + 1) - \sum_{j=1}^{m} \binom{m}{j} v(\text{end}, j + 1) v(1:\text{end} - 1, j + 1) \right) / v(\text{end}, 1) \right)$$

provides the value of $D^m R(x)$.

See Also

Related Examples

- "Constructing and Working with Rational Splines" on page 10-25

The ppform

In this section...

Introduction to ppform

A univariate *piecewise polynomial f* is specified by its *break sequence* breaks and the *coefficient array* coefs of the local power form (see equation in "Definition of ppform" on page 10-8) of its polynomial pieces; see "Multivariate Tensor Product Splines" on page 10-21 for a discussion of multivariate piecewise-polynomials. The coefficients may be (column-)vectors, matrices, even ND-arrays. For simplicity, the present discussion deals only with the case when the coefficients are scalars.

The break sequence is assumed to be strictly increasing,

```
breaks(1)
< breaks(2) < ... < breaks(l+1)
```

with l the number of polynomial pieces that make up *f*.

While these polynomials may be of varying degrees, they are all recorded as polynomials of the same *order* k, i.e., the coefficient array coefs is of size [l,k], with coefs(j,:) containing the k coefficients in the local power form for the jth polynomial piece, from the highest to the lowest power; see equation in "Definition of ppform" on page 10-8.

Definition of ppform

The items breaks, coefs, l, and k, make up the *ppform* of *f*, along with the dimension d of its coefficients; usually d equals 1. The *basic interval* of this form is the interval [breaks(1) .. breaks(l+1)]. It is the default interval over which a function in ppform is plotted by the plot command fnplt.

In these terms, the precise description of the piecewise-polynomial *f* is

$$f(t) = polyval(coefs(j,:), t - breaks(j)) \tag{10-1}$$

for breaks(*j*)≤*t*<breaks(*j*+1).

Here, polyval(a,x) is the MATLAB function; it returns the number

$$\sum_{j=1}^{k} a(j)x^{k-j} = a(1)x^{k-1} + a(2)x^{k-2} + ... + a(k)x^0$$

This defines *f(t)* only for *t* in the half-open interval [breaks(1)..breaks(l+1)]. For any other *t*, *f(t)* is defined by

$$f(t) = polyval(coefs(j,:), t - breaks(j)) \quad j = \begin{matrix} 1, t < breaks(1) \\ l, t \geq breaks(l+1) \end{matrix}$$

i.e., by extending the first, respectively last, polynomial piece. In this way, a function in ppform has possible jumps, in its value and/or its derivatives, only across the interior breaks, `breaks(2:l)`. The end breaks, `breaks([1,l+1])`, mainly serve to define the basic interval of the ppform.

See Also

Related Examples
- "Constructing and Working with ppform Splines" on page 10-10

Constructing and Working with ppform Splines

In this section...
"Constructing a ppform" on page 10-10
"Working With ppform Splines" on page 10-10
"Example ppform" on page 10-11

Constructing a ppform

A piecewise-polynomial is usually constructed by some command, through a process of interpolation or approximation, or conversion from some other form e.g., from the B-form, and is output as a variable. But it is also possible to make one up from scratch, using the statement

```
pp
= ppmak(breaks,coefs)
```

For example, if you enter `pp=ppmak(-5:-1,-22:-11)`, or, more explicitly,

```
breaks = -5:-1;
coefs = -22:-11; pp = ppmak(breaks,coefs);
```

you specify the uniform break sequence `-5:-1` and the coefficient sequence `-22:-11`. Because this break sequence has 5 entries, hence 4 break intervals, while the coefficient sequence has 12 entries, you have, in effect, specified a piecewise-polynomial of order 3 (= 12/4). The command

```
fnbrk(pp)
```

prints out all the constituent parts of this piecewise-polynomial, as follows:

```
breaks(1:l+1)
    -5 -4 -3 -2 -1
coefficients(d*l,k)
     -22 -21 -20
     -19 -18 -17
     -16 -15 -14
     -13 -12 -11
   pieces number l
     4
order k
   3
dimension d of target
     1
```

Further, `fnbrk` can be used to supply each of these parts separately. But the point of Curve Fitting Toolbox spline functionality is that you usually need not concern yourself with these details. You simply use `pp` as an argument to commands that evaluate, differentiate, integrate, convert, or plot the piecewise-polynomial whose description is contained in `pp`.

Working With ppform Splines

Here are some functions for operations you can perform on a piecewise-polynomial.

`v = fnval(pp,x)`	Evaluates
`dpp = fnder(pp)`	Differentiates
`dirpp = fndir(pp,dir)`	Differentiates in the direction `dir`
`ipp = fnint(pp)`	Integrates
`fnmin(pp,[a,b])`	Finds the minimum value in given interval
`fnzeros(pp,[a,b])`	Finds the zeros in the given interval
`pj = fnbrk(pp,j)`	Pulls out the `j`th polynomial piece
`pc = fnbrk(pp,[a b])`	Restricts/extends to the interval [a..b]
`po = fnxtr(pp,order)`	Extends outside its basic interval by polynomial of specified order
`fnplt(pp,[a,b])`	Plots on given interval
`sp = fn2fm(pp,'B-')`	Converts to B-form
`pr = fnrfn(pp,morebreaks)`	Inserts additional breaks

Inserting additional breaks comes in handy when you want to add two piecewise-polynomials with different breaks, as is done in the command `fncmb`.

Example ppform

Execute the following commands to create and plot the particular piecewise-polynomial (ppform) described in the "Constructing a ppform" on page 10-10 section.

1 Create the piecewise-polynomial with break sequence `-5:-1` and coefficient sequence `-22:-11`:

```
pp=ppmak(-5:-1,-22:-11)
```

2 Create the basic plot:

```
x = linspace(-5.5,-.5,101);
plot(x, fnval(pp,x),'x')
```

3 Add the break lines to the plot:

```
breaks=fnbrk(pp,'b'); yy=axis; hold on
for j=1:fnbrk(pp,'l')+1
   plot(breaks([j j]),yy(3:4))
end
```

4 Superimpose the plot of the polynomial that supplies the third polynomial piece:

```
plot(x,fnval(fnbrk(pp,3),x),'linew',1.3)
set(gca,'ylim',[-60 -10]), hold off
```
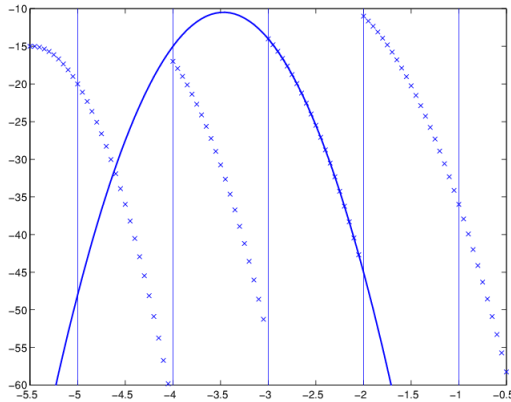
A Piecewise-Polynomial Function, Its Breaks, and the Polynomial Giving Its Third Piece

The figure above is the final picture. It shows the piecewise-polynomial as a sequence of points and, solidly on top of it, the polynomial from which its third polynomial piece is taken. It is quite noticeable that the value of a piecewise-polynomial at a break is its limit from the *right*, and that the value of the piecewise-polynomial outside its basic interval is obtained by extending its leftmost, respectively its rightmost, polynomial piece.

While the ppform of a piecewise-polynomial is efficient for evaluation, the *construction* of a piecewise-polynomial from some data is usually more efficiently handled by determining first its *B-form*, i.e., its representation as a linear combination of B-splines.

See Also

More About

*   "Types of Splines: ppform and B-form" on page 10-2
*   "The ppform" on page 10-8

The B-form

Introduction to B-form

A univariate spline f is specified by its nondecreasing knot sequence `t` and by its B-spline coefficient sequence `a`. See "Multivariate Tensor Product Splines" on page 10-21 for a discussion of multivariate splines. The coefficients may be (column-)vectors, matrices, even ND-arrays. When the coefficients are 2-vectors or 3-vectors, f is a curve in R^2 or R^3 and the coefficients are called the *control points* for the curve.

Roughly speaking, such a spline is a piecewise-polynomial of a certain order and with breaks $t(i)$. But knots are different from breaks in that they may be repeated, i.e., `t` need not be *strictly* increasing. The resulting knot *multiplicities* govern the smoothness of the spline across the knots, as detailed below.

With `[d,n] = size(a)`, and `n+k = length(t)`, the spline is of *order* k. This means that its polynomial pieces have degree $<$ k. For example, a *cubic* spline is a spline of *order 4* because it takes four coefficients to specify a cubic polynomial.

Definition of B-form

These four items, t, a, n, and k, make up the B-form of the spline f.

This means, explicitly, that

$$f = \sum_{i=1}^{n} B_{i,k} a(:,i)$$

with $B_{i,k}=B(\cdot|t(i:i+k))$ the ith B-spline of order k for the given knot sequence `t`, i.e., the B-spline with knots $t(i),...,t(i+k)$. The basic interval of this B-form is the interval $[t(1)..t(n+k)]$. It is the default interval over which a spline in B-form is plotted by the command `fnplt`. Note that a spline in B-form is zero outside its basic interval while, after conversion to ppform via `fn2fm`, this is usually not the case because, outside its basic interval, a piecewise-polynomial is defined by extension of its first or last polynomial piece. In particular, a function in B-form may have jumps in value and/or one of its derivative not only across its interior knots, i.e., across $t(i)$ with $t(1)<t(i)<t(n+k)$, but also across its end knots, $t(1)$ and $t(n+k)$.

B-form and B-Splines

The building blocks for the B-form of a spline are the B-splines. "A B-Spline of Order 4, and the Four Cubic Polynomials from Which It Is Made" on page 10-14 shows a picture of such a B-spline, the one

with the knot sequence [0 1.5 2.3 4 5], hence of order 4, together with the polynomials whose pieces make up the B-spline. The information for that picture could be generated by the command

```
bspline([0 1.5 2.3 4 5])
```
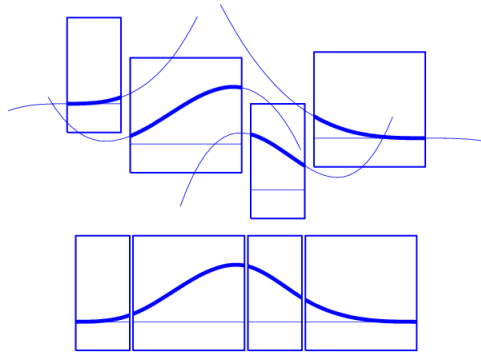

A B-Spline of Order 4, and the Four Cubic Polynomials from Which It Is Made

To summarize: The B-spline with knots $t(i) \leq \cdots \leq t(i+k)$ is positive on the interval $(t(i)..t(i+k))$and is zero outside that interval. It is piecewise-polynomial of order k with breaks at the sites $t(i),...,t(i+k)$. These knots may coincide, and the precise *multiplicity* governs the smoothness with which the two polynomial pieces join there.

Definition of B-Splines

The shorthand

$$f \in S_{k,t}$$

is one of several ways to indicate that *f* is a spline of order k with knot sequence t, i.e., *a linear combination of the B-splines* of order k for the knot sequence t.

A word of caution: The term *B-spline* has been expropriated by the Computer-Aided Geometric Design (CAGD) community to mean what is called here a *spline in B-form*, with the unhappy result that, in any discussion between mathematicians/approximation theorists and people in CAGD, one now always has to check in what sense the term is being used.

B-Spline Knot Multiplicity

The rule is

knot multiplicity + condition multiplicity = order
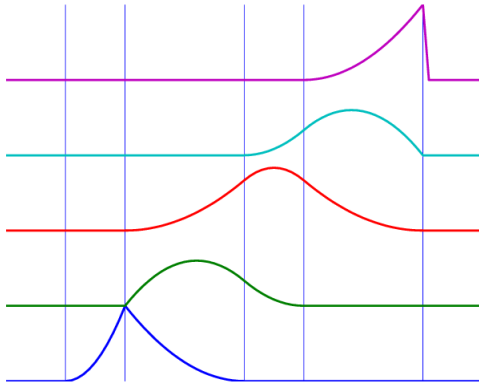
All Third-Order B-Splines for a Certain Knot Sequence with Various Knot Multiplicities

For example, for a B-spline of order 3, a simple knot would mean two smoothness conditions, i.e., continuity of function and first derivative, while a double knot would only leave one smoothness condition, i.e., just continuity, and a triple knot would leave no smoothness condition, i.e., even the function would be discontinuous.

"All Third-Order B-Splines for a Certain Knot Sequence with Various Knot Multiplicities" on page 10-15 shows a picture of all the third-order B-splines for a certain mystery knot sequence `t`. The breaks are indicated by vertical lines. For each break, try to determine its multiplicity in the knot sequence (it is 1,2,1,1,3), as well as its multiplicity as a knot in each of the B-splines. For example, the second break has multiplicity 2 but appears only with multiplicity 1 in the third B-spline and not at all, i.e., with multiplicity 0, in the last two B-splines. Note that only one of the B-splines shown has all its knots simple. It is the only one having three different nontrivial polynomial pieces. Note also that you can tell the knot-sequence multiplicity of a knot by the number of B-splines whose nonzero part begins or ends there. The picture is generated by the following MATLAB statements, which use the command `spcol` from this toolbox to generate the function values of all these B-splines at a fine net `x`.

```
t=[0,1,1,3,4,6,6,6]; x=linspace(-1,7,81);
c=spcol(t,3,x);[l,m]=size(c);
c=c+ones(l,1)*[0:m-1];
axis([-1 7 0 m]); hold on
for tt=t, plot([tt tt],[0 m],'-'), end
plot(x,c,'linew',2), hold off, axis off
```

Further illustrated examples are provided by the example "Construct and Work with the B-form". You can also use the GUI `bspligui` to study the dependence of a B-spline on its knots experimentally.

Choice of Knots for B-form

The rule "knot multiplicity + condition multiplicity = order" has the following consequence for the process of choosing a knot sequence for the B-form of a spline approximant. Suppose the spline s is to be of order k, with basic interval $[a..b]$, and with interior breaks $\xi_2 < \cdots < \xi_l$. Suppose, further, that, at ξ_i, the spline is to satisfy μ_i smoothness conditions, i.e.,

$$jump_{\xi_i} D^j s := D^j s(\xi_{i+}) - D^j s(\xi_{i-}) = 0, \ 0 \le j < \mu_i, \ i = 2, ..., l$$

Then, the appropriate knot sequence t should contain the break ξ_i exactly $k - \mu_i$ times, $i=2,...,l$. In addition, it should contain the two endpoints, a and b, of the basic interval exactly k times. This last requirement can be relaxed, but has become standard. With this choice, there is exactly one way to write each spline s with the properties described as a weighted sum of the B-splines of order k with knots a segment of the knot sequence t. This is the reason for the B in *B-spline*: B-splines are, in Schoenberg's terminology, *basic* splines.

For example, if you want to generate the B-form of a cubic spline on the interval [1 .. 3], with interior breaks 1.5, 1.8, 2.6, and with two continuous derivatives, then the following would be the appropriate knot sequence:

```
t = [1, 1, 1, 1, 1.5, 1.8, 2.6, 3, 3, 3, 3];
```

This is supplied by `augknt([1, 1.5, 1.8, 2.6, 3], 4)`. If you wanted, instead, to allow for a corner at 1.8, i.e., a possible jump in the first derivative there, you would triple the knot 1.8, i.e., use

```
t = [1, 1, 1, 1, 1.5, 1.8, 1.8, 1.8, 2.6, 3, 3, 3, 3];
```

and this is provided by the statement

```
t = augknt([1, 1.5, 1.8, 2.6, 3], 4, [1, 3, 1] );
```

See Also

Related Examples

*   "Constructing and Working with B-form Splines" on page 10-17

Constructing and Working with B-form Splines

In this section...
"Construction of B-form" on page 10-17
"Working With B-form Splines" on page 10-17
"Example: B-form Spline Approximation to a Circle" on page 10-18

Construction of B-form

Usually, a spline is constructed from some information, like function values and/or derivative values, or as the approximate solution of some ordinary differential equation. But it is also possible to make up a spline from scratch, by providing its knot sequence and its coefficient sequence to the command `spmak`.

For example, if you enter

```
sp = spmak(1:10,3:8);
```

you supply the uniform knot sequence `1:10` and the coefficient sequence `3:8`. Because there are 10 knots and 6 coefficients, the order must be 4(= 10 – 6), i.e., you get a cubic spline. The command

```
fnbrk(sp)
```

prints out the constituent parts of the B-form of this cubic spline, as follows:

```
knots(1:n+k)
   1 2 3 4 5 6 7 8 9 10
coefficients(d,n)
   3 4 5 6 7 8
number n of coefficients
   6
order k
   4
dimension d of target
   1
```

Further, `fnbrk` can be used to supply each of these parts separately.

But the point of the Curve Fitting Toolbox spline functionality is that there shouldn't be any need for you to look up these details. You simply use `sp` as an argument to commands that evaluate, differentiate, integrate, convert, or plot the spline whose description is contained in `sp`.

Working With B-form Splines

The following commands are available for spline work. There is `spmak` and `fnbrk` to make up a spline and take it apart again. Use `fn2fm` to convert from B-form to ppform. You can evaluate, differentiate, integrate, minimize, find zeros of, plot, refine, or selectively extrapolate a spline with the aid of `fnval`, `fnder`, `fndir`, `fnint`, `fnmin`, `fnzeros`, `fnplt`, `fnrfn`, and `fnxtr`.

There are five commands for generating knot sequences:

- `augknt` for providing boundary knots and also controlling the multiplicity of interior knots

- `brk2knt` for supplying a knot sequence with specified multiplicities
- `aptknt` for providing a knot sequence for a spline space of given order that is suitable for interpolation at given data sites
- `optknt` for providing an *optimal* knot sequence for interpolation at given sites
- `newknt` for a knot sequence perhaps more suitable for the function to be approximated

In addition, there is:

- `aveknt` to supply certain knot averages (the Greville sites) as recommended sites for interpolation
- `chbpnt` to supply such sites
- `knt2brk` and `knt2mlt` for extracting the breaks and/or their multiplicities from a given knot sequence

To display a spline *curve* with given two-dimensional coefficient sequence and a uniform knot sequence, use `spcrv`.

You can also write your own spline construction commands, in which case you will need to know the following. The construction of a spline satisfying some interpolation or approximation conditions usually requires a *collocation matrix*, i.e., the matrix that, in each row, contains the sequence of numbers $D^r B_{j,k}(\tau)$, i.e., the *r*th derivative at τ of the *j*th B-spline, for all *j*, for some *r* and some site τ. Such a matrix is provided by `spcol`. An optional argument allows for this matrix to be supplied by `spcol` in a space-saving spline-almost-block-diagonal-form or as a MATLAB sparse matrix. It can be fed to `slvblk`, a command for solving linear systems with an almost-block-diagonal coefficient matrix. If you are interested in seeing how `spcol` and `slvblk` are used in this toolbox, have a look at the commands `spapi`, `spap2`, and `spaps`.

In addition, there are routines for constructing *cubic* splines. `csapi` and `csape` provide the cubic spline interpolant at knots to given data, using the not-a-knot and various other end conditions, respectively. A parametric cubic spline curve through given points is provided by `cscvn`. The cubic *smoothing* spline is constructed in `csaps`.

Example: B-form Spline Approximation to a Circle

As another simple example,

```
points = .95*[0 -1 0 1;1 0 -1 0];
sp = spmak(-4:8,[points points]);
```

provides a planar, quartic, spline curve whose middle part is a pretty good approximation to a circle, as the plot on the next page shows. It is generated by a subsequent

```
plot(points(1,:),points(2,:),'x'), hold on
fnplt(sp,[0,4]), axis equal square, hold off
```

Insertion of additional control points $(\pm 0.95, \pm 0.95)/\sqrt{1.9}$ would make a visually perfect circle.

Here are more details. The spline curve generated has the form $\Sigma^8_{j=1} B_{j,5} a(:, j)$, with `-4:8` the uniform knot sequence, and with its control points $a(:,j)$ the sequence $(0,\alpha),(-\alpha,0),(0,-\alpha),(\alpha,0),(0,\alpha),(-\alpha,0),(0,-\alpha),(\alpha,0)$ with $\alpha=0.95$. Only the curve part between the parameter values 0 and 4 is actually plotted.

To get a feeling for how close to circular this part of the curve actually is, compute its unsigned curvature. The curvature $\kappa(t)$ at the curve point $\gamma(t) = (x(t), y(t))$ of a space curve γ can be computed from the formula

$$\kappa = \frac{|x'y'' - y'x''|}{(x'^2 + y'^2)^{3/2}}$$

in which x', x″, y', and y″ are the first and second derivatives of the curve with respect to the parameter used (*t*). Treat the planar curve as a space curve in the (*x,y*)-plane, hence obtain the maximum and minimum of its curvature at 21 points as follows:

```
t = linspace(0,4,21);zt = zeros(size(t));
dsp = fnder(sp); dspt = fnval(dsp,t); ddspt = fnval(fnder(dsp),t);
kappa = abs(dspt(1,:).*ddspt(2,:)-dspt(2,:).*ddspt(1,:))./...
   (sum(dspt.^2)).^(3/2);
[min(kappa),max(kappa)]

ans =
     1.6747     1.8611
```

So, while the curvature is not quite constant, it is close to 1/radius of the circle, as you see from the next calculation:

```
1/norm(fnval(sp,0))

ans =
     1.7864
```
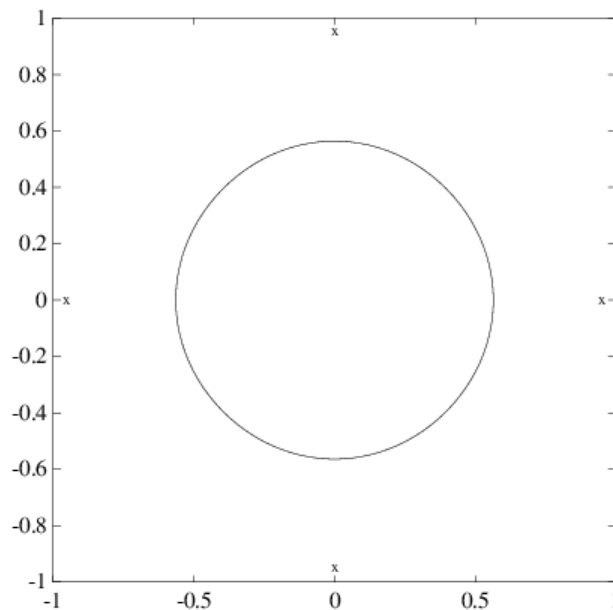

Spline Approximation to a Circle; Control Points Are Marked x

See Also

More About

- "Types of Splines: ppform and B-form" on page 10-2
- "The B-form" on page 10-13

Multivariate Tensor Product Splines

In this section...
"Introduction to Multivariate Tensor Product Splines" on page 10-21
"B-form of Tensor Product Splines" on page 10-21
"Construction With Gridded Data" on page 10-21
"ppform of Tensor Product Splines" on page 10-22
"Example: The Mobius Band" on page 10-22

Introduction to Multivariate Tensor Product Splines

The toolbox provides (polynomial) spline functions in any number of variables, as tensor products of univariate splines. These multivariate splines come in both standard forms, the B-form and the ppform, and their construction and use parallels entirely that of the univariate splines discussed in previous sections, "Constructing and Working with ppform Splines" on page 10-10 and "Constructing and Working with B-form Splines" on page 10-17. The same commands are used for their construction and use.

For simplicity, the following discussion deals just with bivariate splines.

B-form of Tensor Product Splines

The tensor-product idea is very simple. If f is a function of x, and g is a function of y, then their tensor-product $p(x,y) := f(x)g(y)$ is a function of x and y, i.e., a bivariate function. More generally, with $s = (s_1,...,s_{m+h})$ and $t = (t_1,...,t_{n+k})$ knot sequences and a_{ij} ($i = 1,...,m; j = 1,...n$) a corresponding coefficient array, you obtain a bivariate spline as

$$f(x, y) = \sum_{i=1}^{m} \sum_{j=1}^{n} B(x \,|\, s_i, ..., s_{i+h}) B\big(y \,\big|\, t_j, ..., t_{j+k}\big) a_{ij}$$

The B-form of this spline comprises the cell array $\{s,t\}$ of its knot sequences, the coefficient array a, the numbers vector $[m,n]$, and the orders vector $[h,k]$. The command

```
sp = spmak({s,t},a);
```

constructs this form. Further, `fnplt`, `fnval`, `fnder`, `fndir`, `fnrfn`, and `fn2fm` can be used to plot, evaluate, differentiate and integrate, refine, and convert this form.

Construction With Gridded Data

You are most likely to construct such a form by looking for an interpolant or approximant to gridded data. For example, if you know the values z(i,j)=g(x(i),y(j)),i=1:m, j=1:n, of some function g at all the points in a rectangular grid, then, assuming that the strictly increasing sequence x satisfies the Schoenberg-Whitney conditions with respect to the above knot sequence s, and that the strictly increasing sequence y satisfies the Schoenberg-Whitney conditions with respect to the above knot sequence t, the command

```
sp=spapi({s,t},[h,k],{x,y},z);
```

constructs the unique bivariate spline of the above form that matches the given values. The command `fnplt(sp)` gives you a quick plot of this interpolant. The command `pp = fn2fm(sp,'pp')` gives you the ppform of this spline, which is probably what you want when you want to evaluate the spline at a fine grid(`(xx(i),yy(j))` for `i=1:M, j=1:N`), by the command:

```
values = fnval(pp,{xx,yy});
```

ppform of Tensor Product Splines

The ppform of such a bivariate spline comprises, analogously, a cell array of break sequences, a multidimensional coefficient array, a vector of number pieces, and a vector of polynomial orders. Fortunately, the toolbox is set up in such a way that there is usually no reason for you to concern yourself with these details of either form. You use interpolation, approximation, or smoothing to construct splines, and then use the `fn...` commands to make use of them.

Example: The Mobius Band

Here is an example of a surface constructed as a 3-D-valued bivariate spline. The surface is the famous Möbius band, obtainable by taking a longish strip of paper and gluing its narrow ends together, but with a twist. The figure is obtained by the following commands:

```
x = 0:1; y = 0:4; h = 1/4; o2 = 1/sqrt(2); s = 2; ss = 4;
v(3,:,:) = h*[0, -1, -o2, 0, o2, 1, 0;0, 1, o2, 0, -o2, -1, 0];
v(2,:,:) = [ss, 0, s-h*o2, 0, -s-h*o2, 0, ss;...
            ss, 0, s+h*o2, 0,-s+h*o2, 0, ss];
v(1,:,:) = s*[0, 1, 0, -1+h, 0, 1, 0; 0, 1, 0, -1-h, 0, 1, 0];
cs = csape({x,y},v,{'variational','clamped'});
fnplt(cs), axis([-2 2 -2.5 2.5 -.5 .5]), shading interp
axis off, hold on
values = squeeze(fnval(cs,{1,linspace(y(1),y(end),51)}));
plot3(values(1,:), values(2,:), values(3,:),'k','linew',2)
view(-149,28), hold off
```
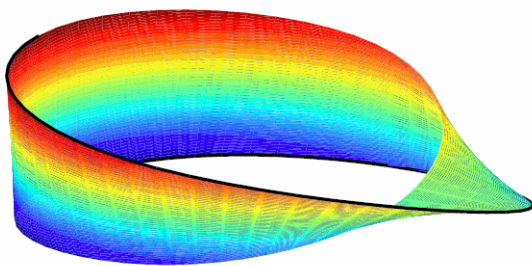

A Möbius Band Made by Vector-Valued Bivariate Spline Interpolation

See Also

More About

* "Types of Splines: ppform and B-form" on page 10-2

NURBS and Other Rational Splines

In this section...
"Introduction to Rational Splines" on page 10-23
"rsform: rpform, rBform" on page 10-23

Introduction to Rational Splines

A rational spline is, by definition, any function that is the ratio of two splines:

$$r(x) = s(x)/w(x)$$

This requires w to be scalar-valued, but s is often chosen to be vector-valued. Further, it is desirable that $w(x)$ be not zero for any x of interest.

Rational splines are popular because, in contrast to ordinary splines, they can be used to describe certain basic design shapes, like conic sections, exactly.

rsform: rpform, rBform

The two splines, s and w, in the rational spline $r(x)=s(x)/w(x)$ need not be related to one another. They could even be of different forms. But, in the context of this toolbox, it is convenient to restrict them to be of the same form, and even of the same order and with the same breaks or knots. For, under that assumption, you can represent such a rational spline by the (vector-valued) spline function

$$R(x) = [s(x); w(x)]$$

whose values are vectors with one more entry than the values of the rational spline r, and call this the *rsform* of the rational spline, or, more precisely, the *rpform* or *rBform*, depending on whether s and w are in ppform or in B-form. Internally, the only thing that distinguishes these rational forms from their corresponding ordinary spline forms, rpform and B-form, is their form part, i.e., the string obtained via `fnbrk(r,'form')`. This is enough to alert the `fn...` commands to act appropriately on a function in one of the rsforms.

For example, as is done in `fnval`, it is very easy to obtain $r(x)$ from $R(x)$. If `v` is the value of R at x, then `v(1:end-1)/v(end)` is the value of r at x. If, in addition, `dv` is $DR(x)$, then `(dv(1:end-1)-dv(end)*v(1:end-1))/v(end)` is $Dr(x)$. More generally, by Leibniz's formula,

$$D^j s = D^j (wr) = \sum_{i=0}^{j} \binom{j}{i} D^i w D^{j-i} r$$

Therefore,

$$D^j r = \left(D^j s - \sum_{i=1}^{j} \binom{j}{i} D^i w D^{j-i} r \right) / w$$

This shows that you can compute the derivatives of r inductively, using the derivatives of s and w (i.e., the derivatives of R) along with the derivatives of r of order less than j to compute the jth derivative of r. This inductive scheme is used in `fntlr` to provide the first so many derivatives of a rational spline. There is a corresponding formula for partial and directional derivatives for multivariate rational splines.

10-23

See Also

Related Examples

- "Constructing and Working with Rational Splines" on page 10-25

Constructing and Working with Rational Splines

In this section...

Rational Spline Example: Circle

For example,

```
circle = rsmak('circle');
```

provides a rational spline whose values on its basic interval trace out the unit circle, i.e., the circle of radius 1 with center at the origin, as the command

```
fnplt(circle), axis square
```

readily shows; the resulting output is the circle in the figure "A Circle and an Ellipse, Both Given by a Rational Spline" on page 10-25.

It is easy to manipulate this circle to obtain related shapes. For example, the next commands stretch the circle into an ellipse, rotate the ellipse 45 degrees, and translate it by (1,1), and then plot it on top of the circle.

```
ellipse = fncmb(circle,[2 0;0 1]);
s45 = 1/sqrt(2);
rtellipse = fncmb(fncmb(ellipse, [s45 -s45;s45 s45]), [1;1] );
hold on, fnplt(rtellipse), hold off
```

As a further example, the "circle" just constructed is put together from four pieces. To highlight the first such piece, use the following commands:

```
quarter = fnbrk(fn2fm(circle,'rp'),1);
hold on, fnplt(quarter,3), hold off
```

In the first command, `fn2fm` is used to change forms, from one based on the B-form to one based on the ppform, and then `fnbrk` is used to extract the first piece, and this piece is then plotted on top of the circle in "A Circle and an Ellipse, Both Given by a Rational Spline" on page 10-25, with linewidth 3 to make it stand out.
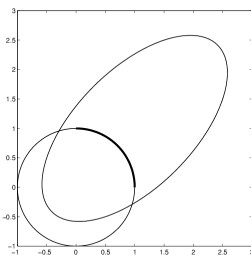
A Circle and an Ellipse, Both Given by a Rational Spline

Rational Spline Example: Sphere

As a surface example, the command `rsmak('southcap')` provides a 3-vector valued rational bicubic polynomial whose values on the unit square [-1 .. 1]^2 fill out a piece of the unit sphere. Adjoin to it five suitable rotates of it and you get the unit sphere exactly. For illustration, the following commands generate two-thirds of that sphere, as shown in "Part of a Sphere Formed by Four Rotates of a Quartic Rational" on page 10-26.

```
southcap = rsmak('southcap'); fnplt(southcap)
xpcap = fncmb(southcap,[0 0 -1;0 1 0;1 0 0]);
ypcap = fncmb(xpcap,[0 -1 0; 1 0 0; 0 0 1]);
northcap = fncmb(southcap,-1);
hold on, fnplt(xpcap), fnplt(ypcap), fnplt(northcap)
axis equal, shading interp, view(-115,10), axis off, hold off
```


Part of a Sphere Formed by Four Rotates of a Quartic Rational

Functions for Working With Rational Splines

Having chosen to represent the rational spline $r = s/w$ in this way by the ordinary spline $R=[s;w]$ makes it is easy to apply to a rational spline all the `fn...` commands in the Curve Fitting Toolbox spline functions, with the following exceptions. The integral of a rational spline need not be a rational spline, hence there is no way to extend `fnint` to rational splines. The derivative of a rational spline *is* again a rational spline but one of roughly twice the order. For that reason, `fnder` and `fndir` will not touch rational splines. Instead, there is the command `fntlr` for computing the value at a given `x` of all derivatives up to a given order of a given function. If that function is rational, the needed calculation is based on the considerations given in the preceding paragraph.

The command `r = rsmak(shape)` provides rational splines in rBform that describe exactly certain standard geometric shapes , like `'circle'`, `'arc'`, `'cylinder'`, `'sphere'`, `'cone'`, `'torus'`. The command `fncmb(r,trans)` can be used to apply standard transformations to the resulting shape. For example, if `trans` is a column-vector of the right length, the shape would be translated by that vector while, if `trans` is a suitable matrix like a rotation, the shape would be transformed by that matrix.

The command `r = rscvn(p)` constructs the quadratic rBform of a tangent-continuous curve made up of circular arcs and passing through the given sequence, `p`, of points in the plane.

A special rational spline form, called a NURBS, has become a standard tool in CAGD. A NURBS is, by definition, any rational spline for which both *s* and *w* are in the same B-form, with each coefficient for *s* containing explicitly the corresponding coefficient for *w* as a factor:

$$s = \sum_i B_i v(i) a(:,i), \ w = \sum_i B_i v(i)$$

The normalized coefficients $a(:,i)$ for the numerator spline are more readily used as control points than the unnormalized coefficients $v(i)a(:,i)$ used in the rBform. Nevertheless, this toolbox provides no special NURBS form, but only the more general rational spline, but in both B-form (called `rBform` internally) and in ppform (called `rpform` internally).

The rational spline `circle` used earlier is put together in `rsmak` by code like the following.

```
x = [1 1 0 -1 -1 -1  0  1 1]; y = [0 1 1  1  0 -1 -1 -1 0];
s45 = 1/sqrt(2); w =[1 s45 1 s45 1 s45 1 s45 1];
circle = rsmak(augknt(0:4,3,2), [w.*x;w.*y;w]);
```

Note the appearance of the denominator spline as the last component. Also note how the coefficients of the denominator spline appear here explicitly as factors of the corresponding coefficients of the numerator spline. The normalized coefficient sequence `[x;y]` is very simple; it consists of the vertices and midpoints, in proper order, of the "unit square". The resulting control polygon is tangent to the circle at the places where the four quadratic pieces that form the circle abut.

For a thorough discussion of NURBS, see [G. Farin, *NURBS*, 2nd ed., AKPeters Ltd, 1999] or [Les Piegl and Wayne Tiller, *The NURBS Book*, 2nd ed., Springer-Verlag, 1997].

See Also

More About

- "Multivariate and Rational Splines" on page 10-6

Constructing and Working with stform Splines

In this section...
"Introduction to the stform" on page 10-28
"Construction and Properties of the stform" on page 10-28
"Working with the stform" on page 10-29

Introduction to the stform

A multivariate function form quite different from the tensor-product construct is the scattered translates form, or stform for short. As the name suggests, it uses arbitrary or scattered translates $\psi(\cdot - c_j)$ of one fixed function ψ, in addition to some polynomial terms. Explicitly, such a form describes a function

$$f(x) = \sum_{j=1}^{n-k} \psi(x - c_j)a_j + p(x)$$

in terms of the *basis function* ψ, a sequence (c_j) of sites called *centers* and a corresponding sequence (a_j) of n coefficients, with the final k coefficients, $a_{n-k+1},...,a_n$, involved in the *polynomial part*, p.

When the basis function is radially symmetric, meaning that $\psi(x)$ depends only on the Euclidean length $|x|$ of its argument, x, then ψ is called a *radial basis function*, and, correspondingly, f is then often called an RBF.

At present, the toolbox works with just one kind of stform, namely a bivariate thin-plate spline and its first partial derivatives. For the thin-plate spline, the basis function is $\psi(x) = \varphi(|x|^2)$, with $\varphi(t) = t\log t$, i.e., a radial basis function. Its polynomial part is a linear polynomial, i.e., $p(x)=x(1)a_{n-2}+x(2)a_{n-1}+a_n$. The first partial derivative with respect to its first argument uses, correspondingly, the basis function $\psi(x)=\varphi(|x|^2)$, with $\varphi(t) = (D_1 t)\cdot(\log t+1)$ and $D_1 t = D_1 t(x) = 2x(1)$, and $p(x) = a_n$.

Construction and Properties of the stform

A function in stform can be put together from its center sequence `centers` and its coefficient sequence `coefs` by the command

```
f = stmak(centers, coefs, type);
```

with the string `type` one of `'tp00'`, `'tp10'`, `'tp01'`, to indicate, respectively, a thin-plate spline, a first partial of a thin-plate spline with respect to the first argument, and a first partial of a thin-plate spline with respect to the second argument. There is one other choice, `'tp'`; it denotes a thin-plate spline without any polynomial part and is likely to be used only during the construction of a thin-plate spline, as in `tpaps`.

A function f in stform depends linearly on its coefficients, meaning that

$$f(x) = \sum_{j=1}^{n} \psi_j(x)a_j$$

with ψ_j either a translate of the basis function Ψ or else some polynomial. Suppose you wanted to determine these coefficients a_j so that the function f matches prescribed values at prescribed sites x_i.

Then you would need the collocation matrix $(\psi_j(x_i))$. You can obtain this matrix by the command `stcol(centers,x,type)`. In fact, because the stform has a_j as the *j*th *column*, `coefs(:,j)`, of its coefficient array, it is worth noting that `stcol` can also supply the *transpose* of the collocation matrix. Thus, the command

```
values = coefs*stcol(centers,x,type,'tr');
```

would provide the values at the entries of `x` of the st function specified by `centers` and `type`.

The stform is attractive because, in contrast to piecewise polynomial forms, its complexity is the same in any number of variables. It is quite simple, yet, because of the complete freedom in the choice of centers, very flexible and adaptable.

On the negative side, the most attractive choices for a radial basis function share with the thin-plate spline that the evaluation at any site involves all coefficients. For example, plotting a scalar-valued thin-plate spline via `fnplt` involves evaluation at a 51-by-51 grid of sites, a nontrivial task when there are 1000 coefficients or more. The situation is worse when you want to determine these 1000 coefficients so as to obtain the stform of a function that matches function values at 1000 data sites, as this calls for solving a full linear system of order 1000, a task requiring O(10^9) flops if done by a direct method. Just the construction of the collocation matrix for this linear system (by `stcol`) takes O(10^6) flops.

The command `tpaps`, which constructs thin-plate spline interpolants and approximants, uses iterative methods when there are more than 728 data points, but convergence of such iteration may be slow.

Working with the stform

After you have constructed an approximating or interpolating thin-plate spline `st` with the aid of `tpaps` (or directly via `stmak`), you can use the following commands:

- `fnbrk` to obtain its parts or change its basic interval,
- `fnval` to evaluate it
- `fnplt` to plot it
- `fnder` to construct its two first partial derivatives, but no higher order derivatives as they become infinite at the centers.

  This is just one indication that the stform is quite different in nature from the other forms in this toolbox, hence other `fn...` commands by and large don't work with stforms. For example, it makes no sense to use `fnjmp`, and `fnmin` or `fnzeros` only work for univariate functions. It also makes no sense to use `fnint` on a function in stform because such functions cannot be integrated in closed form.

- The command `Ast = fncmb(st,A)` can be used on `st`, provided A is something that can be applied to the values of the function described by `st`. For example, A might be `'sin'`, in which case `Ast` is the stform of the function whose coefficients are the sine of the coefficients of `st`. In effect, `Ast` describes the function obtained by composing A with `st`. But, because of the singularities in the higher-order derivatives of a thin-plate spline, there seems little point to make `fndir` or `fntlr` applicable to such a `st`.

Advanced Spline Examples

Least-Squares Approximation by Natural Cubic Splines

The construction of a least-squares approximant usually requires that one have in hand a basis for the space from which the data are to be approximated. As the example of the space of "natural" cubic splines illustrates, the explicit construction of a basis is not always straightforward.

This section makes clear that an explicit basis is not actually needed; it is sufficient to have available some means of interpolating in some fashion from the space of approximants. For this, the fact that the Curve Fitting Toolbox spline functions support work with vector-valued functions is essential.

This section discusses these aspects of least-squares approximation by "natural" cubic splines.

- "Problem" on page 11-2
- "General Resolution" on page 11-2
- "Need for a Basis Map" on page 11-2
- "A Basis Map for "Natural" Cubic Splines" on page 11-3
- "The One-line Solution" on page 11-3
- "The Need for Proper Extrapolation" on page 11-3
- "The Correct One-Line Solution" on page 11-4
- "Least-Squares Approximation by Cubic Splines" on page 11-5

Problem

You want to construct the least-squares approximation to given data (x,y) from the space *S* of "natural" cubic splines with given breaks `b(1) < ...< b(l+1)`.

General Resolution

If you know a basis, (f1,f2,...,fm), for the linear space *S* of all "natural" cubic splines with break sequence b, then you have learned to find the least-squares approximation in the form `c(1)f1+ c(2)f2+ ... + c(m)fm`, with the vector c the least-squares solution to the linear system `A*c = y`, whose coefficient matrix is given by

```
A(i,j) = fj(x(i)),    i=1:length(x),   j=1:m .
```

In other words, `c = A\y`.

Need for a Basis Map

The general solution seems to require that you know a basis. However, in order to construct the coefficient sequence c, you only need to know the matrix A. For this, it is sufficient to have at hand a *basis map*, namely a function F say, so that `F(c)` returns the spline given by the particular weighted sum `c(1)f1+c(2)f2+... +c(m)fm`. For, with that, you can obtain, for `j=1:m`, the j-th column of A as `fnval(F(ej),x)`, with `ej` the j-th column of `eye(m)`, the identity matrix of order m.

Better yet, the Curve Fitting Toolbox spline functions can handle *vector-valued* functions, so you should be able to construct the basis map F to handle vector-valued coefficients `c(i)` as well. However, by agreement, in this toolbox, a vector-valued coefficient is a *column* vector, hence the sequence c is necessarily a row vector of column vectors, i.e., a *matrix*. With that, `F(eye(m))` is the

vector-valued spline whose `i`-th component is the basis element `fi, i=1:m`. Hence, assuming the vector `x` of data sites to be a row vector, `fnval(F(eye(m)),x)` is the matrix whose `(i,j)`-entry is the value of `fi` at `x(j)`, i.e., the *transpose* of the matrix `A` you are seeking. On the other hand, as just pointed out, your basis map `F` expects the coefficient sequence `c` to be a row vector, i.e., the transpose of the vector `A\y`. Hence, assuming, correspondingly, the vector `y` of data values to be a row vector, you can obtain the least-squares approximation from *S* to data `(x,y)` as

```
F(y/fnval(F(eye(m)),x))
```

To be sure, if you wanted to be prepared for `x` and `y` to be arbitrary vectors (of the same length), you would use instead

```
F(y(:).'/fnval(F(eye(m)),x(:).'))
```

A Basis Map for "Natural" Cubic Splines

What exactly is required of a basis map `F` for the linear space *S* of "natural" cubic splines with break sequence `b(1) < ... < b(l+1)`? Assuming the dimension of this linear space is `m`, the map `F` should set up a linear one-to-one correspondence between `m`-vectors and elements of *S*. But that is exactly what `csape(b, . ,'var')` does.

To be explicit, consider the following function `F`:

```
function s = F(c)
s = csape(b,c,'var');
```

For given vector `c` (of the same length as `b`), it provides the *unique* "natural" cubic spline with break sequence b that takes the value `c(i)` at `b(i)`, `i=1:l+1`. The uniqueness is key. It ensures that the correspondence between the vector `c` and the resulting spline `F(c)` is one-to-one. In particular, `m` equals `length(b)`. More than that, because the value *f(t)* of a function *f* at a point *t* depends linearly on *f*, this uniqueness ensures that `F(c)` depends linearly on `c` (because `c` equals `fnval(F(c),b)` and the inverse of an invertible linear map is again a linear map).

The One-line Solution

Putting it all together, you arrive at the following code

```
csape(b,y(:).'/fnval(csape(b,eye(length(b)),'var'),x(:).'),...
'var')
```

for the least-squares approximation by "natural" cubic splines with break sequence `b`.

The Need for Proper Extrapolation

Let's try it on some data, the census data, say, which is provided in MATLAB by the command

```
load census
```

and which supplies the years, `1790:10:1990`, as `cdate` and the values as `pop`. Use the break sequence `1810:40:1970`.

```
b = 1810:40:1970;
s = csape(b, ...
pop(:)'/fnval(csape(b,eye(length(b)),'var'),cdate(:)'),'var');
```

```
fnplt(s, [1750,2050],2.2);
hold on
plot(cdate,pop,'or');
hold off
```

Have a look at "Least-Squares Approximation by "Natural" Cubic Splines With Three Interior Breaks" on page 11-4 which shows, in thick blue, the resulting approximation, along with the given data.

This looks like a good approximation, -- except that it doesn't look like a "natural" cubic spline. A "natural" cubic spline, to recall, must be linear to the left of its first break and to the right of its last break, and this approximation satisfies neither condition. This is due to the following facts.

The "natural" cubic spline interpolant to given data is provided by `csape` in ppform, with the interval spanned by the data sites its basic interval. On the other hand, evaluation of a ppform outside its basic interval is done, in MATLAB `ppval` or Curve Fitting Toolbox spline function `fnval`, by using the relevant polynomial end piece of the ppform, i.e., by full-order extrapolation. In case of a "natural" cubic spline, you want instead second-order extrapolation. This means that you want, to the left of the first break, the straight line that agrees with the cubic spline in value and slope at the first break. Such an extrapolation is provided by `fnxtr`. Because the "natural" cubic spline has zero second derivative at its first break, such an extrapolation is even third-order, i.e., it satisfies three matching conditions. In the same way, beyond the last break of the cubic spline, you want the straight line that agrees with the spline in value and slope at the last break, and this, too, is supplied by `fnxtr`.
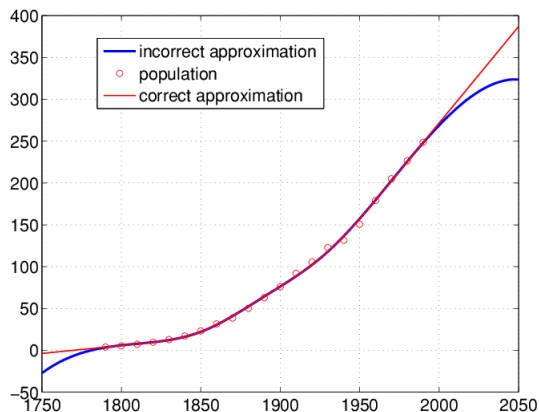
Least-Squares Approximation by "Natural" Cubic Splines With Three Interior Breaks

The Correct One-Line Solution

The following one-line code provides the correct least-squares approximation to data (x,y) by "natural" cubic splines with break sequence b:

```
fnxtr(csape(b,y(:).'/ ...
    fnval(fnxtr(csape(b,eye(length(b)),'var')),x(:).'),'var'))
```

But it is, admittedly, a rather long line.

The following code uses this correct formula and plots, in a thinner, red line, the resulting approximation on top of the earlier plots, as shown in "Least-Squares Approximation by "Natural" Cubic Splines With Three Interior Breaks" on page 11-4.

```
 ss = fnxtr(csape(b,pop(:)'/ ...
   fnval(fnxtr(csape(b,eye(length(b)),'var')),cdate(:)'),'var'));
hold on, fnplt(ss,[1750,2050],1.2,'r'),grid, hold off
legend('incorrect approximation','population', ...
'correct approximation')
```

Least-Squares Approximation by Cubic Splines

The one-line solution works perfectly if you want to approximate by the space *S* of all cubic splines with the given break sequence b. You don't even have to use the Curve Fitting Toolbox spline functions for this because you can rely on the MATLAB `spline`. You know that, with c a sequence containing two more entries than does b, `spline(b,c)` provides the unique cubic spline with break sequence b that takes the value `c(i+1)` at `b(i)`, all i, and takes the slope `c(1)` at `b(1)`, and the slope `c(end)` at `b(end)`. Hence, `spline(b,.)` is a basis map for S.

More than that, you know that `spline(b,c,xi)` provides the value(s) at `xi` of this interpolating spline. Finally, you know that `spline` can handle vector-valued data. Therefore, the following one-line code constructs the least-squares approximation by cubic splines with break sequence b to data (x,y):

```
spline(b,y(:)'/spline(b,eye(length(b)),x(:)'))
```

Solving A Nonlinear ODE

This section discusses these aspects of a nonlinear ODE problem:

- "Problem" on page 11-6
- "Approximation Space" on page 11-6
- "Discretization" on page 11-6
- "Numerical Problem" on page 11-7
- "Linearization" on page 11-7
- "Linear System to Be Solved" on page 11-7
- "Iteration" on page 11-8

You can run this example: "Solving a Nonlinear ODE with a Boundary Layer by Collocation".

Problem

Consider the nonlinear singularly perturbed problem:

$$\varepsilon D^2 g(x) + (g(x))^2 = 1 \ \ on \ \ [0..1]$$

$$Dg(0) = g(1) = 0$$

Approximation Space

Seek an approximate solution by collocation from C^1 piecewise cubics with a suitable break sequence; for instance,

```
breaks = (0:4)/4;
```

Because cubics are of order 4, you have

```
k = 4;
```

Obtain the corresponding knot sequence as

```
knots = augknt(breaks,k,2);
```

This gives a quadruple knot at both 0 and 1, which is consistent with the fact that you have cubics, i.e., have order 4.

This implies that you have

```
n = length(knots)-k;
n = 10;
```

i.e., 10 degrees of freedom.

Discretization

You collocate at two sites per polynomial piece, i.e., at eight sites altogether. This, together with the two side conditions, gives us 10 conditions, which matches the 10 degrees of freedom.

Choose the two Gaussian sites for each interval. For the *standard* interval [–0.5,0.5] of length 1, these are the two sites

```
gauss = .5773502692*[-1/2; 1/2];
```

From this, you obtain the whole collection of collocation sites by

```
ninterv = length(breaks)-1;
temp = ((breaks(2:ninterv+1)+breaks(1:ninterv))/2);
temp = temp([1 1],:) + gauss*diff(breaks);
colsites = temp(:).';
```

Numerical Problem

With this, the numerical problem you want to solve is to find $y \in S_{4,\,knots}$ that satisfies the nonlinear system

$$Dy(0) = 0$$

$$(^y + \varepsilon D^2 y(x) = 1 \text{ for } x \in \text{ colsites}$$

$$y(1) = 0$$

Linearization

If y is your current approximation to the solution, then the linear problem for the supposedly better solution z by Newton's method reads

$$Dz(0) = 0$$

$$w_0(x)z(x) + \varepsilon D^2 z(x) = b(x) \text{ for } x \in \text{ colsites}$$

$$z(1)=0$$

with $w_0(x)=2y(x), b(x)=(y(x))^2+1$. In fact, by choosing

$w_0(1){:} = 1, \ w_1(0){:} = 1$

$w_1(x){:} = 0, \ w_2(x){:} = \varepsilon \text{ for } x \in \text{ colsites}$

and choosing all other values of $w_0, w_1, w_2, \ b$ not yet specified to be zero, you can give your system the uniform shape

$w_0(x)z(x) + w_1(x)Dz(x) + w_2(x)D^2 z(x) = b(x), \text{ for } x \in \text{ sites}$

with

```
sites = [0,colsites,1];
```

Linear System to Be Solved

Because $z \in S_{4,\text{knots}}$, convert this last system into a system for the B-spline coefficients of z. This requires the values, first, and second derivatives at every $x \in sites$ and for all the relevant B-splines. The command `spcol` was expressly written for this purpose.

Use `spcol` to supply the matrix

```
colmat = ...
spcol(knots,k,brk2knt(sites,3));
```

From this, you get the collocation matrix by combining the row triple of `colmat` for *x* using the weights $w_0(x),w_1(x),w_2(x)$ to get the row for *x* of the actual matrix. For this, you need a current approximation *y*. Initially, you get it by interpolating some reasonable initial guess from your piecewise-polynomial space at the `sites`. Use the parabola x^2-1, which satisfies the end conditions as the initial guess, and pick the matrix from the full matrix `colmat`. Here it is, in several cautious steps:

```
intmat = colmat([2 1+(1:(n-2))*3,1+(n-1)*3],:);
coefs = intmat\[0 colsites.*colsites-1 0].';
y = spmak(knots,coefs.');
```

Plot the initial guess, and turn hold on for subsequent plotting:

```
fnplt(y,'g');
legend('Initial Guess (x^2-1)','location','NW');
axis([-0.01 1.01 -1.01 0.01]);
hold on
```

Iteration

You can now complete the construction and solution of the linear system for the improved approximate solution *z* from your current guess *y*. In fact, with the initial guess *y* available, you now set up an iteration, to be terminated when the change *z*–*y* is *small enough*. Choose a relatively mild ε = .1.

```
tolerance = 6.e-9;
epsilon = .1;
while 1
   vtau = fnval(y,colsites);
   weights=[0 1 0;
            [2*vtau.' zeros(n-2,1) repmat(epsilon,n-2,1)];
            1 0 0];
   colloc = zeros(n,n);
   for j=1:n
      colloc(j,:) = weights(j,:)*colmat(3*(j-1)+(1:3),:);
   end
   coefs = colloc\[0 vtau.*vtau+1 0].';
   z = spmak(knots,coefs.');
   fnplt(z,'k');
   maxdif = max(max(abs(z.coefs-y.coefs)));
   fprintf('maxdif = %g\n',maxdif)
   if (maxdif<tolerance), break, end
% now reiterate
   y = z;
end
legend({'Initial Guess (x^2-1)' 'Iterates'},'location','NW');
```

The resulting printout of the errors is:

```
maxdif = 0.206695
maxdif = 0.01207
maxdif = 3.95151e-005
maxdif = 4.43216e-010
```

If you now decrease ε, you create more of a boundary layer near the right endpoint, and this calls for a nonuniform mesh.

Use `newknt` to construct an appropriate finer mesh from the current approximation:

```
knots = newknt(z, ninterv+1); breaks = knt2brk(knots);
knots = augknt(breaks,4,2);
n = length(knots)-k;
```

From the new break sequence, you generate the new collocation site sequence:

```
ninterv = length(breaks)-1;
temp = ((breaks(2:ninterv+1)+breaks(1:ninterv))/2);
temp = temp([1 1], :) + gauss*diff(breaks);
colpnts = temp(:).';
sites = [0,colpnts,1];
```

Use `spcol` to supply the matrix

```
colmat = spcol(knots,k,sort([sites sites sites]));
```

and use your current approximate solution `z` as the initial guess:

```
intmat = colmat([2 1+(1:(n-2))*3,1+(n-1)*3],:);
y = spmak(knots,[0 fnval(z,colpnts) 0]/intmat.');
```

Thus set up, divide ε by 3 and repeat the earlier calculation, starting with the statements

```
tolerance=1.e-9;
while 1
  vtau=fnval(y,colpnts);
  .
  .
  .
```

Repeated passes through this process generate a sequence of solutions, for ε = 1/10, 1/30, 1/90, 1/270, 1/810. The resulting solutions, ever flatter at 0 and ever steeper at 1, are shown in the example plot. The plot also shows the final break sequence, as a sequence of vertical bars. To view the plots, run the example "Solving a Nonlinear ODE with a Boundary Layer by Collocation".

In this example, at least, `newknt` has performed satisfactorily.

Construction of the Chebyshev Spline

This section discusses these aspects of the Chebyshev spline construction:

- "What Is a Chebyshev Spline?" on page 11-10
- "Choice of Spline Space" on page 11-10
- "Initial Guess" on page 11-10
- "Remez Iteration" on page 11-11

What Is a Chebyshev Spline?

The *Chebyshev spline* $C=C_t=C_{k,t}$ of order k for the knot sequence $t=(t_i: i=1:n+k)$ is the unique element of $S_{k,t}$ of max-norm 1 that maximally oscillates on the interval $[t_k..t_{n+1}]$ and is positive near t_{n+1}. This means that there is a unique strictly increasing n-sequence τ so that the function $C=C_t \epsilon S_{k,t}$ given by $C(\tau_i)=(-1)^{n-1}$, all i, has max-norm 1 on $[t_k..t_{n+1}]$. This implies that $\tau_1=t_k, \tau_n=t_{n+1}$, and that $t_i < \tau_i < t_{k+i}$, for all i. In fact, $t_{i+1} \le \tau_i \le t_{i+k-1}$, all i. This brings up the point that the knot sequence is assumed to make such an inequality possible, i.e., the elements of $S_{k,t}$ are assumed to be continuous.

In short, the Chebyshev spline C looks just like the Chebyshev polynomial. It performs similar functions. For example, its extreme sites τ are particularly good sites to interpolate at from $S_{k,t}$ because the norm of the resulting projector is about as small as can be; see the toolbox command `chbpnt`.

You can run the example Construction of a Chebyshev Spline to construct C for a particular knot sequence t.

Choice of Spline Space

You deal with cubic splines, i.e., with order

```
k = 4;
```

and use the break sequence

```
breaks = [0 1 1.1 3 5 5.5 7 7.1 7.2 8];
lp1 = length(breaks);
```

and use simple interior knots, i.e., use the knot sequence

```
t = breaks([ones(1,k) 2:(lp1-1) lp1(:,ones(1,k))]);
n = length(t)-k;
```

Note the quadruple knot at each end. Because `k = 4`, this makes $[0..8] =$ `[breaks(1)..breaks(lp1)]` the interval $[t_k..t_{n+1}]$ of interest, with `n = length(t)-k` the dimension of the resulting spline space $S_{k,t}$. The same knot sequence would have been supplied by

```
t=augknt(breaks,k);
```

Initial Guess

As the initial guess for the τ, use the knot averages

$$t_i = (t_{i+1} + \dots + t_{i+k-1})/(k-1)$$

recommended as good interpolation site choices. These are supplied by

```
tau=aveknt(t,k);
```

Plot the resulting first approximation to C, i.e., the spline c that satisfies $c(\tau_i)=(-1)^{n-i}$, all i:

```
b = cumprod(repmat(-1,1,n)); b = b*b(end);
c = spapi(t,tau,b);
fnplt(c,'-.')
grid
```

Here is the resulting plot.
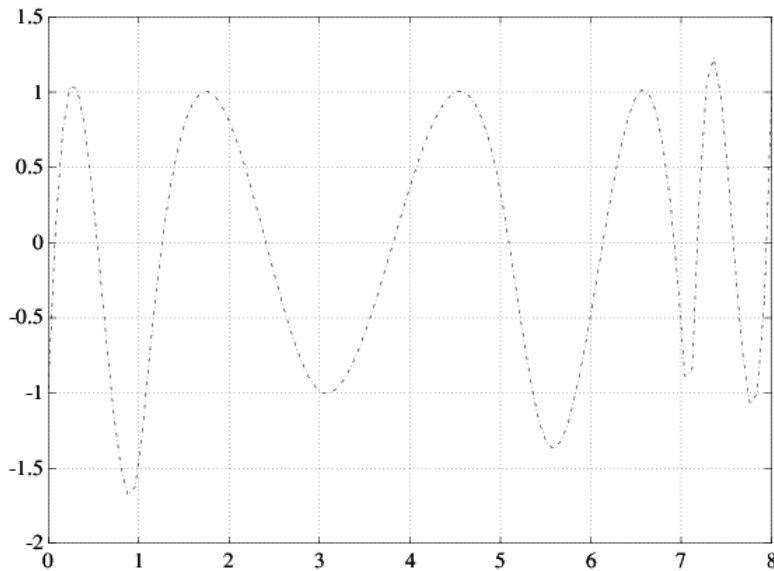
First Approximation to a Chebyshev Spline

Remez Iteration

Starting from this approximation, you use the Remez algorithm to produce a sequence of splines converging to C. This means that you construct new τ as the extrema of your current approximation c to C and try again. Here is the entire loop.

You find the new interior τ_i as the zeros of Dc, i.e., the first derivative of c, in several steps. First, differentiate:

```
Dc = fnder(c);
```

Next, take the zeros of the control polygon of Dc as your first guess for the zeros of Dc. For this, you must take apart the spline Dc.

```
[knots,coefs,np,kp] = fnbrk(Dc,'knots','coefs','n','order');
```

The control polygon has the vertices (`tstar(i)`,`coefs(i)`), with `tstar` the knot averages for the spline, provided by `aveknt`:

```
tstar = aveknt(knots,kp);
```

Here are the zeros of the resulting control polygon of `Dc`:

```
npp = (1:np-1);
guess = tstar(npp) -coefs(npp).*(diff(tstar)./diff(coefs));
```

This provides already a very good first guess for the actual zeros.

Refine this estimate for the zeros of *Dc* by two steps of the secant method, taking `tau` and the resulting `guess` as your first approximations. First, evaluate *Dc* at both sets:

```
sites = tau(ones(4,1),2:n-1);
sites(1,:) = guess;
values = zeros(4,n-2);
values(1:2,:) = reshape(fnval(Dc,sites(1:2,:)),2,n-2);
```

Now come two steps of the secant method. You guard against division by zero by setting the function value difference to 1 in case it is zero. Because *Dc* is strictly monotone near the sites sought, this is harmless:

```
for j=2:3
  rows = [j,j-1];Dcd=diff(values(rows,:));
  Dcd(find(Dcd==0)) = 1;
  sites(j+1,:) = sites(j,:) ...
           -values(j,:).*(diff(sites(rows,:))./Dcd);
  values(j+1,:) = fnval(Dc,sites(j+1,:));
end
```

The check

```
max(abs(values.'))
  ans = 4.1176 5.7789 0.4644 0.1178
```

shows the improvement.

Now take these sites as your new `tau`,

```
tau = [tau(1) sites(4,:) tau(n)];
```

and check the extrema values of your current approximation there:

```
extremes = abs(fnval(c, tau));
```

The difference

```
max(extremes)-min(extremes)
  ans = 0.6905
```

is an estimate of how far you are from total leveling.

Construct a new spline corresponding to the new choice of `tau` and plot it on top of the old:

```
c = spapi(t,tau,b);
sites = sort([tau (0:100)*(t(n+1)-t(k))/100]);
values = fnval(c,sites);
hold on, plot(sites,values)
```

The following code turns on the grid and plots the locations of the extrema.

```
grid on
plot( tau(2:end-1), zeros( 1, np-1 ), 'o' )
hold off
legend( 'Initial Guess', 'Current Guess', 'Extreme Locations',...
 'location', 'NorthEastOutside' );
```

Following is the resulting figure (legend not shown).
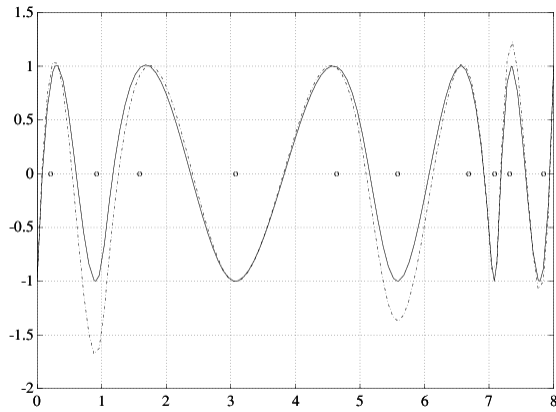
A More Nearly Level Spline

If this is not close enough, one simply reiterates the loop. For this example, the next iteration already produces C to graphic accuracy.

Approximation by Tensor Product Splines

Because the toolbox can handle splines with *vector* coefficients, it is easy to implement interpolation or approximation to gridded data by tensor product splines, as the following illustration is meant to show. You can also run the example "Bivariate Tensor Product Splines".

To be sure, most tensor product spline approximation to gridded data can be obtained directly with one of the spline construction commands, like `spapi` or `csape`, in this toolbox, without concern for the details discussed in this example. Rather, this example is meant to illustrate the theory behind the tensor product construction, and this will be of help in situations not covered by the construction commands in this toolbox.

This section discusses these aspects of the tensor product spline problem:

- "Choice of Sites and Knots" on page 11-14
- "Least Squares Approximation as Function of y" on page 11-14
- "Approximation to Coefficients as Functions of x" on page 11-15
- "The Bivariate Approximation" on page 11-19
- "Switch in Order" on page 11-17
- "Approximation to Coefficients as Functions of y" on page 11-18
- "The Bivariate Approximation" on page 11-19
- "Comparison and Extension" on page 11-20

Choice of Sites and Knots

Consider, for example, least squares approximation to given data $z(i,j)=f(x(i),y(j)),i=1:Nx,j=1:Ny$. You take the data from a function used extensively by Franke for the testing of schemes for surface fitting (see R. Franke, "A critical comparison of some methods for interpolation of scattered data," *Naval Postgraduate School Techn. Rep. NPS-53-79-003*, March 1979). Its domain is the unit square. You choose a few more data sites in the *x*-direction than the *y*-direction; also, for a better definition, you use higher data density near the boundary.

```
x = sort([(0:10)/10,.03 .07, .93 .97]);
y = sort([(0:6)/6,.03 .07, .93 .97]);
[xx,yy] = ndgrid(x,y); z = franke(xx,yy);
```

Least Squares Approximation as Function of y

Treat these data as coming from a vector-valued function, namely, the function of *y* whose value at $y(j)$ is the vector $z(:,j)$, all *j*. For no particular reason, choose to approximate this function by a vector-valued parabolic spline, with three uniformly spaced interior knots. This means that you choose the spline order and the knot sequence for this vector-valued spline as

```
ky = 3; knotsy = augknt([0,.25,.5,.75,1],ky);
```

and then use `spap2` to provide the least squares approximant to the data:

```
sp = spap2(knotsy,ky,y,z);
```

In effect, you are finding simultaneously the discrete least squares approximation from $S_{ky,knotsy}$ to each of the Nx data sets

$$(y(j), z(i, j))_{j=1}^{Ny}, \ i = 1 : Nx$$

In particular, the statements

```
yy = -.1:.05:1.1;
vals = fnval(sp,yy);
```

provide the array `vals`, whose entry `vals(i,j)` can be taken as an approximation to the value $f(x(i), yy(j))$ of the underlying function f at the mesh-point $x(i), yy(j)$ because `vals(:,j)` is the value at $yy(j)$ of the approximating spline curve in `sp`.

This is evident in the following figure, obtained by the command:

```
mesh(x,yy,vals.'), view(150,50)
```

Note the use of `vals.'`, in the `mesh` command, needed because of the MATLAB matrix-oriented view when plotting an array. This can be a serious problem in bivariate approximation because there it is customary to think of $z(i, j)$ as the function value at the point $(x(i), y(j))$, while MATLAB thinks of $z(i, j)$ as the function value at the point $(x(j), y(i))$.
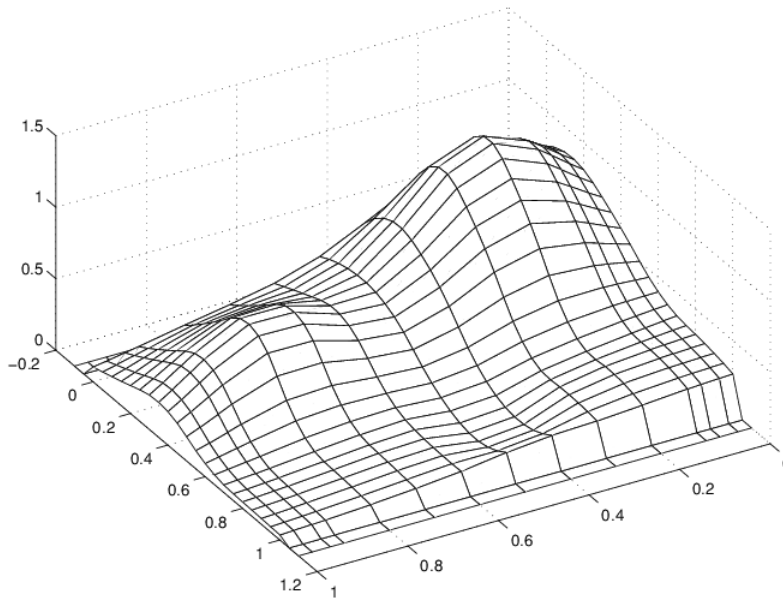
A Family of Smooth Curves Pretending to Be a Surface

Note that both the first two and the last two values on each smooth curve are actually zero because both the first two and the last two sites in `yy` are outside the basic interval for the spline in `sp`.

Note also the ridges. They confirm that you are plotting smooth curves in one direction only.

Approximation to Coefficients as Functions of x

To get an actual surface, you now have to go a step further. Look at the coefficients `coefsy` of the spline in `sp`:

```
coefsy = fnbrk(sp,'coefs');
```

Abstractly, you can think of the spline in `sp` as the function

$$y \Big| \rightarrow \sum_r coefsy(:,r)B_{r,ky}(y)$$

with the *i*th entry `coefsy(i,r)` of the vector coefficient `coefsy(:,r)` corresponding to *x*(*i*), for all *i*. This suggests approximating each coefficient vector `coefsy(q,:)` by a spline of the same order `kx` and with the same appropriate knot sequence `knotsx`. For no particular reason, this time use *cubic* splines with *four* uniformly spaced interior knots:

```
kx = 4; knotsx = augknt([0:.2:1],kx);
sp2 = spap2(knotsx,kx,x,coefsy.');
```

Note that `spap2(knots,k,x,fx)` expects `fx(:,j)` to be the datum at `x(j)`, i.e., expects each *column* of `fx` to be a function value. To fit the datum `coefsy(q, :)` at *x*(*q*), for all *q*, present `spap2` with the *transpose* of `coefsy`.

The Bivariate Approximation

Now consider the transpose of the coefficients `cxy` of the resulting spline *curve*:

```
coefs = fnbrk(sp2,'coefs').';
```

It provides the *bivariate* spline approximation

$$(x,y) \Big| \rightarrow \sum_q \sum_r coefs(q,r)B_{q,kx}(x)B_{r,ky}(y)$$

to the original data

$$(x(i), y(j)) \Big| \rightarrow z(x(i), y(j)), i = 1{:}Nx, j = 1{:}Ny$$

To plot this spline surface over a grid, e.g., the grid

```
xv = 0:.025:1; yv = 0:.025:1;
```

you can do the following:

```
values = spcol(knotsx,kx,xv)*coefs*spcol(knotsy,ky,yv).';
mesh(xv,yv,values.'), view(150,50);
```

This results in the following figure.
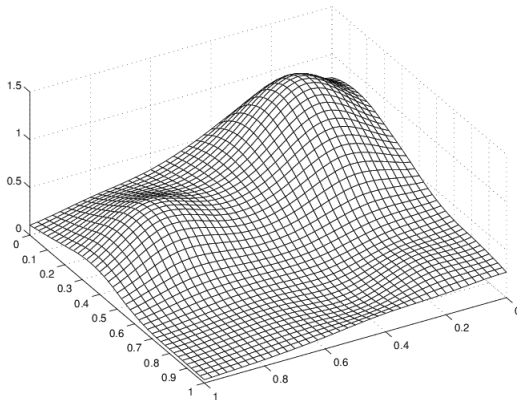
Spline Approximation to Franke's Function

This makes good sense because `spcol(knotsx,kx,xv)` is the matrix whose (i,q)th entry equals the value $B_{q,kx}(xv(i))$ at $xv(i)$ of the qth B-spline of order `kx` for the knot sequence `knotsx`.

Because the matrices `spcol(knotsx,kx,xv)` and `spcol(knotsy,ky,yv)` are banded, it may be more efficient, though perhaps more memory-consuming, for *large* `xv` and `yv` to make use of `fnval`, as follows:

```
value2 = ...
  fnval(spmak(knotsx,fnval(spmak(knotsy,coefs),yv).'),xv).';
```

This is, in fact, what happens internally when `fnval` is called directly with a tensor product spline, as in

```
value2 = fnval(spmak({knotsx,knotsy},coefs),{xv,yv});
```

Here is the calculation of the relative error, i.e., the difference between the given data and the value of the approximation at those data sites as compared with the magnitude of the given data:

```
errors = z - spcol(knotsx,kx,x)*coefs*spcol(knotsy,ky,y).';
disp( max(max(abs(errors)))/max(max(abs(z))) )
```

The output is `0.0539`, perhaps not too impressive. However, the coefficient array was only of size 8 6

```
disp(size(coefs))
```

to fit a data array of size 15 11.

```
disp(size(z))
```

Switch in Order

The approach followed here seems *biased*, in the following way. First think of the given data `z` as describing a vector-valued function of *y*, and then treat the matrix formed by the vector coefficients of the approximating curve as describing a vector-valued function of *x*.

What happens when you take things in the opposite order, i.e., think of z as describing a vector-valued function of *x*, and then treat the matrix made up from the vector coefficients of the approximating curve as describing a vector-valued function of *y*?

Perhaps surprisingly, the final approximation is the same, up to round-off. Here is the numerical experiment.

Least Squares Approximation as Function of x

First, fit a spline curve to the data, but this time with *x* as the independent variable, hence it is the *rows* of z that now become the data values. Correspondingly, you must supply z.', rather than z, to spap2,

```
spb = spap2(knotsx,kx,x,z.');
```

thus obtaining a spline approximation to all the curves $(x ; z (:, j))$. In particular, the statement

```
valsb = fnval(spb,xv).';
```

provides the matrix valsb, whose entry valsb(i, j) can be taken as an approximation to the value $f(xv(i),y(j))$ of the underlying function *f* at the mesh-point $(xv(i),y(j))$. This is evident when you plot valsb using mesh:

```
mesh(xv,y,valsb.'), view(150,50)
```
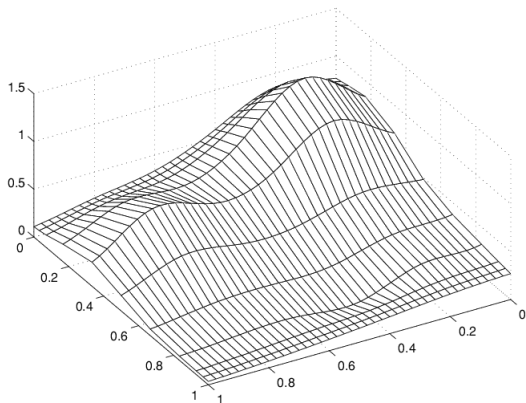

Another Family of Smooth Curves Pretending to Be a Surface

Note the ridges. They confirm that you are, once again, plotting smooth curves in one direction only. But this time the curves run in the other direction.

Approximation to Coefficients as Functions of y

Now comes the second step, to get the actual surface. First, extract the coefficients:

```
coefsx = fnbrk(spb,'coefs');
```

Then fit each coefficient vector coefsx(r,:) by a spline of the same order ky and with the same appropriate knot sequence knotsy:

```
spb2 = spap2(knotsy,ky,y,coefsx.');
```

Note that, once again, you need to transpose the coefficient array from `spb`, because `spap2` takes the columns of its last input argument as the data values.

Correspondingly, there is now no need to transpose the coefficient array `coefsb` of the resulting *curve*:

```
coefsb = fnbrk(spb2,'coefs');
```

The Bivariate Approximation

The claim is that `coefsb` equals the earlier coefficient array `coefs`, up to round-off, and here is the test:

```
disp( max(max(abs(coefs - coefsb))) )
```

The output is `1.4433e-15`.

The explanation is simple enough: The coefficients `c` of the spline *s* contained in `sp = spap2(knots,k,x,y)` depend *linearly* on the input values *y*. This implies, given that both `c` and `y` are 1-row matrices, that there is some matrix $A = A_{\mathrm{knots},k,x}$ so that

$$c = yA_{\mathrm{knots},\,k,\,x}$$

for any data `y`. This statement even holds when `y` is a *matrix*, of size *d*-by-*N*, say, in which case each datum *y*(:,*j*) is taken to be a point in R^d, and the resulting spline is correspondingly *d*-vector-valued, hence its coefficient array `c` is of size *d*-by-n, with `n = length(knots)-k`.

In particular, the statements

```
sp = spap2(knotsy,ky,y,z);
coefsy =fnbrk(sp,'coefs');
```

provide us with the matrix `coefsy` that satisfies

$$\mathrm{coefs}y = z\,.\,A_{\mathrm{knotsy},ky,y}$$

The subsequent computations

```
 sp2 = spap2(knotsx,kx,x,coefsy.');
 coefs = fnbrk(sp2,'coefs').';
```

generate the coefficient array `coefs`, which, taking into account the two transpositions, satisfies

$$\mathrm{coefs} = \left(\left(zA_{\mathrm{knotsy},ky,y}\right)' . A_{\mathrm{knotsx},kx,x}\right)'$$
$$= \left(A_{\mathrm{knotsx},kx,x}\right)' . z . A_{\mathrm{knotsy},ky,y}$$

In the second, alternative, calculation, you first computed

```
 spb = spap2(knotsx,kx,x,z.');
 coefsx = fnbrk(spb,'coefs');
```

hence `coefsx`=z'.$A_{\mathrm{knotsx},kx,x}$. The subsequent calculation

```
 spb2 = spap2(knotsy,ky,y,coefsx.');
 coefsb = fnbrk(spb,'coefs');
```

then provided

$$\text{coefsb} = \text{coefsx} \,.'. \, A_{\text{knotsy,ky,y}} = \left(A_{\text{knotsx,kx,x}}\right).'. z \,. \, A_{\text{knotsy,ky,y}}$$

Consequently, `coefsb` = `coefs`.

Comparison and Extension

The second approach is more symmetric than the first in that transposition takes place in each call to `spap2` and nowhere else. This approach can be used for approximation to gridded data in any number of variables.

If, for example, the given data over a *three*-dimensional grid are contained in some three-dimensional array `v` of size `[Nx,Ny,Nz]`, with `v(i,j,k)` containing the value $f(x(i),y(j),z(k))$, then you would start off with

```
coefs = reshape(v,Nx,Ny*Nz);
```

Assuming that `nj = knotsj - kj`, for `j = x,y,z`, you would then proceed as follows:

```
sp = spap2(knotsx,kx,x,coefs.');
coefs = reshape(fnbrk(sp,'coefs'),Ny,Nz*nx);
sp = spap2(knotsy,ky,y,coefs.');
coefs = reshape(fnbrk(sp,'coefs'),Nz,nx*ny);
sp = spap2(knotsz,kz,z,coefs.');
coefs = reshape(fnbrk(sp,'coefs'),nx,ny*nz);
```

See Chapter 17 of *PGS* or [C. de Boor, "Efficient computer manipulation of tensor products," *ACM Trans. Math. Software 5* (1979), 173–182; Corrigenda, 525] for more details. The same references also make clear that there is nothing special here about using least squares approximation. Any approximation process, including spline interpolation, whose resulting approximation has coefficients that depend linearly on the given data, can be extended in the same way to a multivariate approximation process to gridded data.

This is exactly what is used in the spline construction commands `csapi`, `csape`, `spapi`, `spaps`, and `spap2`, when gridded data are to be fitted. It is also used in `fnval`, when a tensor product spline is to be evaluated on a grid.

Examples

Polynomial Curve Fitting

This example shows how to fit polynomials up to sixth degree to some census data using Curve Fitting Toolbox™. It also shows how to fit a single-term exponential equation and compare this to the polynomial models.

The steps show how to:

- Load data and create fits using different library models.
- Search for the best fit by comparing graphical fit results, and by comparing numerical fit results including the fitted coefficients and goodness of fit statistics.

Load and Plot the Data

The data for this example is the file `census.mat`.

```
load census
```

The workspace contains two new variables:

- `cdate` is a column vector containing the years 1790 to 1990 in 10-year increments.
- `pop` is a column vector with the U.S. population figures that correspond to the years in `cdate`.

```
whos cdate pop
```

```
  Name        Size            Bytes  Class     Attributes

  cdate       21x1              168  double
  pop         21x1              168  double
```

```
plot(cdate,pop,'o')
```
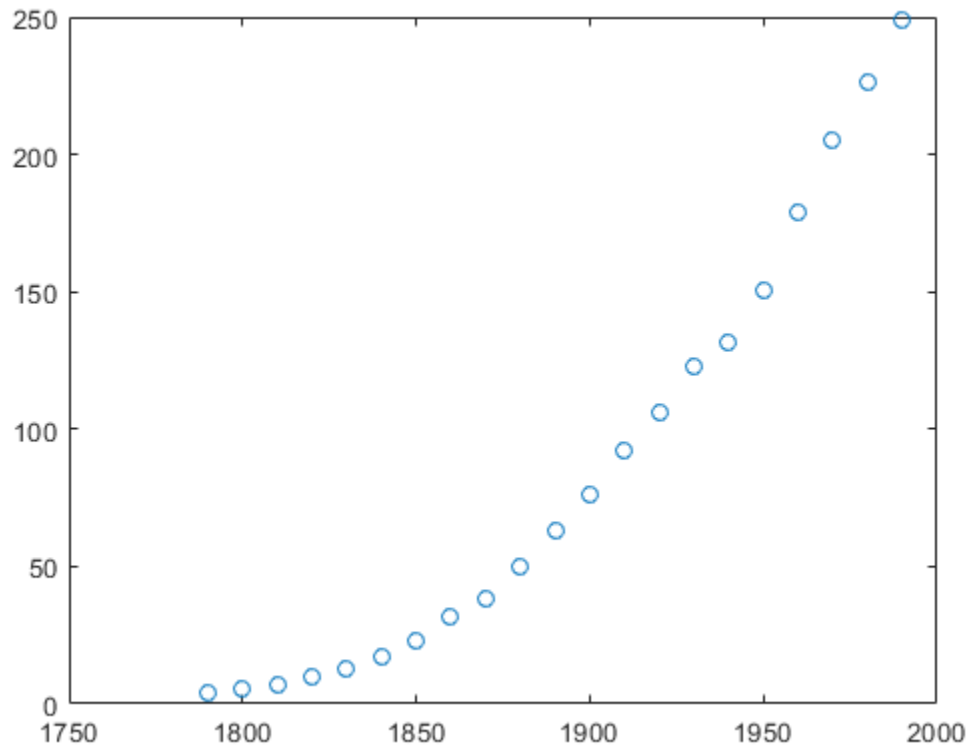
Create and Plot a Quadratic

Use the `fit` function to fit a polynomial to data. You specify a quadratic, or second-degree polynomial, with the string `'poly2'`. The first output from `fit` is the polynomial, and the second output, `gof`, contains the goodness of fit statistics you will examine in a later step.

```
[population2,gof] = fit(cdate,pop,'poly2');
```

To plot the fit, use the `plot` function. Add a legend in the top left corner.

```
plot(population2,cdate,pop);
legend('Location','NorthWest');
```

Create and Plot a Selection of Polynomials

To fit polynomials of different degrees, change the fittype string, e.g., for a cubic or third-degree polynomial use `'poly3'`. The scale of the input, `cdate`, is quite large, so you can obtain better results by centering and scaling the data. To do this, use the `'Normalize'` option.

```
population3 = fit(cdate,pop,'poly3','Normalize','on');
population4 = fit(cdate,pop,'poly4','Normalize','on');
population5 = fit(cdate,pop,'poly5','Normalize','on');
population6 = fit(cdate,pop,'poly6','Normalize','on');
```

A simple model for population growth tells us that an exponential equation should fit this census data well. To fit a single term exponential model, use `'exp1'` as the fittype.

```
populationExp = fit(cdate,pop,'exp1');
```

Plot all the fits at once, and add a meaningful legend in the top left corner of the plot.

```
hold on
plot(population3,'b');
plot(population4,'g');
plot(population5,'m');
plot(population6,'b--');
plot(populationExp,'r--');
hold off
legend('cdate v pop','poly2','poly3','poly4','poly5','poly6','exp1', ...
    'Location','NorthWest');
```

Plot the Residuals to Evaluate the Fit

To plot residuals, specify `'residuals'` as the plot type in the `plot` function.

```
plot(population2,cdate,pop,'residuals');
```

The fits and residuals for the polynomial equations are all similar, making it difficult to choose the best one.

If the residuals display a systematic pattern, it is a clear sign that the model fits the data poorly.

```
plot(populationExp,cdate,pop,'residuals');
```

The fit and residuals for the single-term exponential equation indicate it is a poor fit overall. Therefore, it is a poor choice and you can remove the exponential fit from the candidates for best fit.

Examine Fits Beyond the Data Range

Examine the behavior of the fits up to the year 2050. The goal of fitting the census data is to extrapolate the best fit to predict future population values.

By default, the fit is plotted over the range of the data. To plot a fit over a different range, set the x-limits of the axes before plotting the fit. For example, to see values extrapolated from the fit, set the upper x-limit to 2050.

```
plot(cdate,pop,'o');
xlim([1900, 2050]);
hold on
plot(population6);
hold off
```

Examine the plot. The behavior of the sixth-degree polynomial fit beyond the data range makes it a poor choice for extrapolation and you can reject this fit.

Plot Prediction Intervals

To plot prediction intervals, use `'predobs'` or `'predfun'` as the plot type. For example, to see the prediction bounds for the fifth-degree polynomial for a new observation up to year 2050:

```
plot(cdate,pop,'o');
xlim([1900,2050])
hold on
plot(population5,'predobs');
hold off
```

Plot prediction intervals for the cubic polynomial up to year 2050:

```
plot(cdate,pop,'o');
xlim([1900,2050])
hold on
plot(population3,'predobs')
hold off
```

Examine Goodness-of-Fit Statistics

The struct gof shows the goodness-of-fit statistics for the `'poly2'` fit. When you created the `'poly2'` fit with the `fit` function in an earlier step, you specified the gof output argument.

gof

```
gof = struct with fields:
          sse: 159.0293
      rsquare: 0.9987
          dfe: 18
   adjrsquare: 0.9986
         rmse: 2.9724
```

Examine the sum of squares due to error (SSE) and the adjusted R-square statistics to help determine the best fit. The SSE statistic is the least-squares error of the fit, with a value closer to zero indicating a better fit. The adjusted R-square statistic is generally the best indicator of the fit quality when you add additional coefficients to your model.

The large SSE for `'exp1'` indicates it is a poor fit, which you already determined by examining the fit and residuals. The lowest SSE value is associated with `'poly6'`. However, the behavior of this fit beyond the data range makes it a poor choice for extrapolation, so you already rejected this fit by examining the plots with new axis limits.

The next best SSE value is associated with the fifth-degree polynomial fit, `'poly5'`, suggesting it might be the best fit. However, the SSE and adjusted R-square values for the remaining polynomial fits are all very close to each other. Which one should you choose?

Compare the Coefficients and Confidence Bounds to Determine the Best Fit

Resolve the best fit issue by examining the coefficients and confidence bounds for the remaining fits: the fifth-degree polynomial and the quadratic.

Examine `population2` and `population5` by displaying the models, the fitted coefficients, and the confidence bounds for the fitted coefficients:

`population2`

```
population2 =
     Linear model Poly2:
     population2(x) = p1*x^2 + p2*x + p3
     Coefficients (with 95% confidence bounds):
       p1 =    0.006541  (0.006124, 0.006958)
       p2 =      -23.51  (-25.09, -21.93)
       p3 =   2.113e+04  (1.964e+04, 2.262e+04)
```

`population5`

```
population5 =
     Linear model Poly5:
     population5(x) = p1*x^5 + p2*x^4 + p3*x^3 + p4*x^2 + p5*x + p6
       where x is normalized by mean 1890 and std 62.05
     Coefficients (with 95% confidence bounds):
       p1 =      0.5877  (-2.305, 3.48)
       p2 =      0.7047  (-1.684, 3.094)
       p3 =     -0.9193  (-10.19, 8.356)
       p4 =       23.47  (17.42, 29.52)
       p5 =       74.97  (68.37, 81.57)
       p6 =       62.23  (59.51, 64.95)
```

You can also get the confidence intervals by using `confint`:

```
ci = confint(population5)

ci = 2×6

   -2.3046    -1.6841   -10.1943    17.4213    68.3655    59.5102
    3.4801     3.0936     8.3558    29.5199    81.5696    64.9469
```

The confidence bounds on the coefficients determine their accuracy. Check the fit equations (e.g. `f(x)=p1*x+p2*x...`) to see the model terms for each coefficient. Note that `p2` refers to the `p2*x` term in `'poly2'` and the `p2*x^4` term in `'poly5'`. Do not compare normalized coefficients directly with non-normalized coefficients.

The bounds cross zero on the `p1`, `p2`, and `p3` coefficients for the fifth-degree polynomial. This means you cannot be sure that these coefficients differ from zero. If the higher order model terms may have coefficients of zero, they are not helping with the fit, which suggests that this model over fits the census data.

The fitted coefficients associated with the constant, linear, and quadratic terms are nearly identical for each normalized polynomial equation. However, as the polynomial degree increases, the coefficient bounds associated with the higher degree terms cross zero, which suggests over fitting.

However, the small confidence bounds do not cross zero on p1, p2, and p3 for the quadratic fit, indicating that the fitted coefficients are known fairly accurately.

Therefore, after examining both the graphical and numerical fit results, you should select the quadratic population2 as the best fit to extrapolate the census data.

Evaluate the Best Fit at New Query Points

Now you have selected the best fit, population2, for extrapolating this census data, evaluate the fit for some new query points:

```
cdateFuture = (2000:10:2020).';
popFuture = population2(cdateFuture)
```

```
popFuture = 3×1

  274.6221
  301.8240
  330.3341
```

To compute 95% confidence bounds on the prediction for the population in the future, use the predint method:

```
ci = predint(population2,cdateFuture,0.95,'observation')
```

```
ci = 3×2

  266.9185   282.3257
  293.5673   310.0807
  321.3979   339.2702
```

Plot the predicted future population, with confidence intervals, against the fit and data.

```
plot(cdate,pop,'o');
xlim([1900,2040])
hold on
plot(population2)
h = errorbar(cdateFuture,popFuture,popFuture-ci(:,1),ci(:,2)-popFuture,'.');
hold off
legend('cdate v pop','poly2','prediction', ...
    'Location','NorthWest')
```
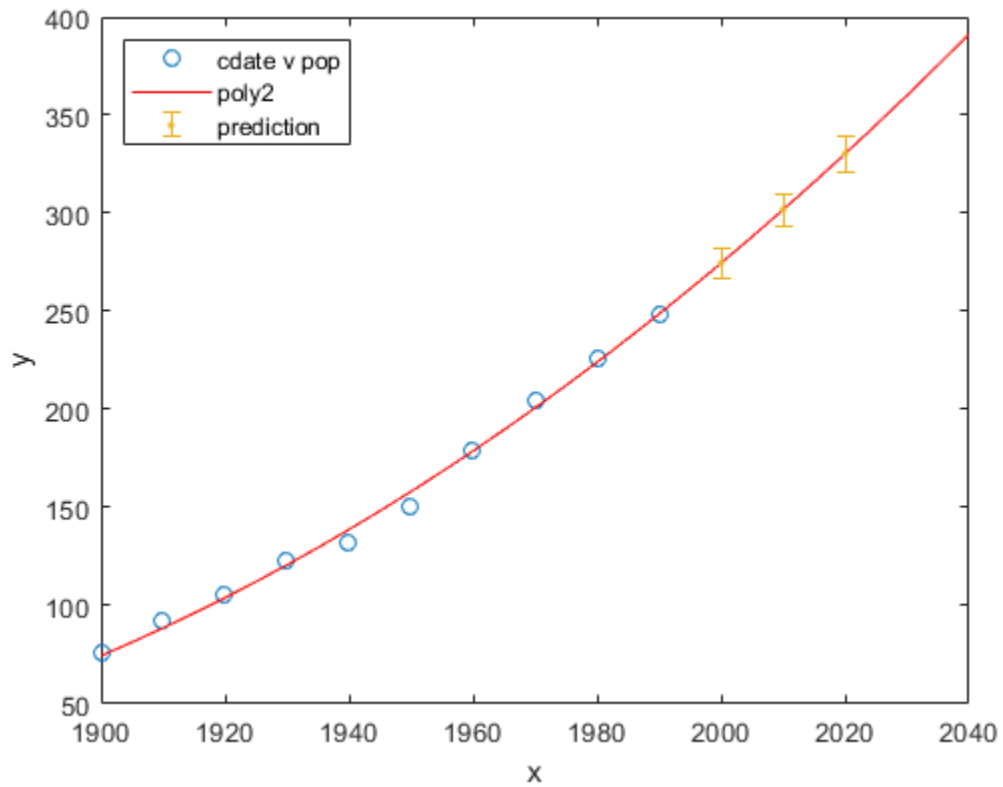
For more information, see "Polynomial Models" on page 4-14.

Surface Fitting With Custom Equations to Biopharmaceutical Data

This example shows how to use Curve Fitting Toolbox™ to fit response surfaces to some anesthesia data to analyze drug interaction effects. Response surface models provide a good method for understanding the pharmacodynamic interaction behavior of drug combinations.

This data is based on the results in this paper: Kern SE, Xie G, White JL, Egan TD. Opioid-hypnotic synergy: A response surface analysis of propofol-remifentanil pharmacodynamic interaction in volunteers. Anesthesiology 2004; 100: 1373-81.

Anesthesia is typically at least a two-drug process, consisting of an opioid and a sedative hypnotic. This example uses Propofol and Reminfentanil as drug class prototypes. Their interaction is measured by four different measures of the analgesic and sedative response to the drug combination. Algometry, Tetany, Sedation, and Laryingoscopy comprise the four measures of surrogate drug effects at various concentration combinations of Propofol and Reminfentanil.

The following code, using Curve Fitting Toolbox methods, reproduces the interactive surface building with the Curve Fitting Tool described in "Surface Fitting to Biopharmaceutical Data" on page 5-30.

Load Data

Load the data from file.

```
data = importdata( 'OpioidHypnoticSynergy.txt' );
Propofol      = data.data(:,1);
Remifentanil  = data.data(:,2);
Algometry     = data.data(:,3);
Tetany        = data.data(:,4);
Sedation      = data.data(:,5);
Laryingoscopy = data.data(:,6);
```

Create the Model Fit Type

You can use the `fittype` function to define the model from the paper, where `CA` and `CB` are the drug concentrations, and `IC50A`, `IC50B`, `alpha`, and `n` are the coefficients to be estimated. Create the model fit type.

```
ft = fittype( 'Emax*( CA/IC50A + CB/IC50B + alpha*( CA/IC50A ) * ( CB/IC50B ) )^n /(( CA/IC50A +
    'independent', {'CA', 'CB'}, 'dependent', 'z', 'problem', 'Emax' )

ft =
    General model:
    ft(IC50A,IC50B,alpha,n,Emax,CA,CB) = Emax*( CA/IC50A + CB/IC50B + alpha*(
                  CA/IC50A ) * ( CB/IC50B ) )^n /(( CA/IC50A + CB/IC50B
                  + alpha*( CA/IC50A ) * ( CB/IC50B ) )^n  + 1 )
```

Assume `Emax = 1` because the effect output is normalized.

```
Emax = 1;
```

Set Fit Options

Set fit options for robust fitting, bounds, and start points.

```
opts = fitoptions( ft );
opts.Lower = [0, 0, -5, -0];
```

```
opts.Robust = 'LAR';
opts.StartPoint = [0.0089, 0.706, 1.0, 0.746];
```

Fit and Plot a Surface for Algometry

```
[f, gof] = fit( [Propofol, Remifentanil], Algometry, ft,...
    opts, 'problem', Emax )
```

Success, but fitting stopped because change in residuals less than tolerance (TolFun).

```
    General model:
    f(CA,CB) = Emax*( CA/IC50A + CB/IC50B + alpha*( CA/IC50A ) * ( CB/IC50B
                    ) )^n /(( CA/IC50A + CB/IC50B + alpha*( CA/IC50A )
                    * ( CB/IC50B ) )^n  + 1 )
    Coefficients (with 95% confidence bounds):
      IC50A =         4.149  (4.123, 4.174)
      IC50B =         9.045  (8.971, 9.118)
      alpha =         8.502  (8.316, 8.688)
      n =       8.288  (8.131, 8.446)
    Problem parameters:
      Emax =              1

gof = struct with fields:
          sse: 0.0842
      rsquare: 0.9991
          dfe: 393
    adjrsquare: 0.9991
         rmse: 0.0146
```

```
plot( f, [Propofol, Remifentanil], Algometry );
```
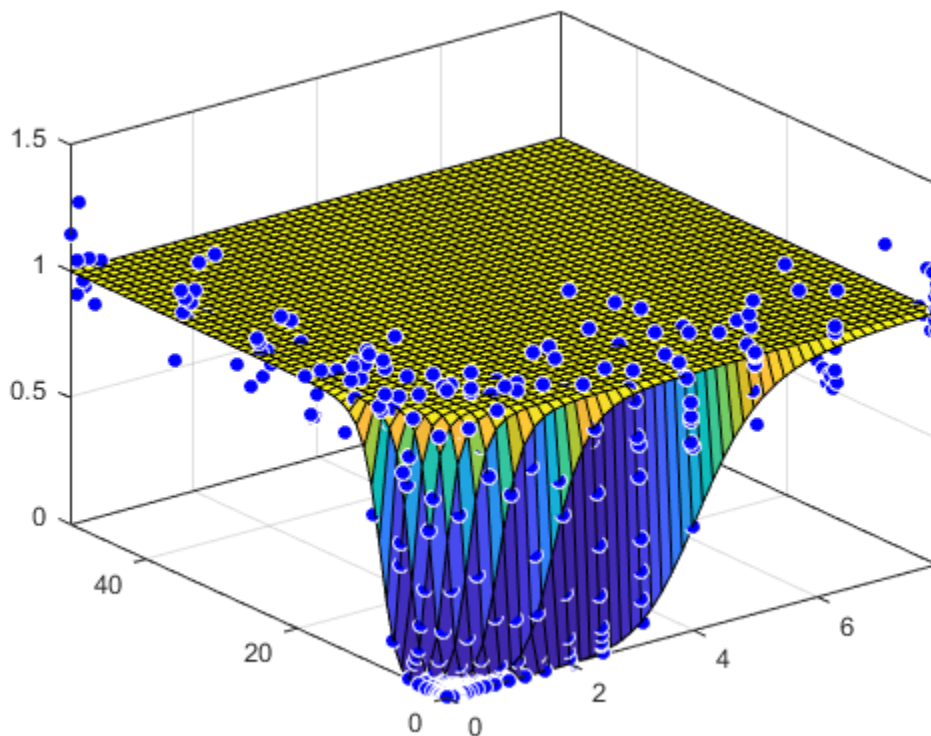
Fit a Surface to Tetany

Reuse the same `fittype` to create a response surface for tetany.

```
[f, gof] = fit( [Propofol, Remifentanil], Tetany, ft, opts, 'problem', Emax )

    General model:
    f(CA,CB) = Emax*( CA/IC50A + CB/IC50B + alpha*( CA/IC50A ) * ( CB/IC50B
                  ) )^n /(( CA/IC50A + CB/IC50B + alpha*( CA/IC50A )
                * ( CB/IC50B ) )^n  + 1 )
    Coefficients (with 95% confidence bounds):
      IC50A =        4.544  (4.522, 4.567)
      IC50B =        21.22  (21.04, 21.4)
      alpha =        14.94  (14.67, 15.21)
      n =        6.132  (6.055, 6.209)
    Problem parameters:
      Emax =             1

gof = struct with fields:
         sse: 0.0537
     rsquare: 0.9993
         dfe: 393
   adjrsquare: 0.9993
        rmse: 0.0117


plot( f, [Propofol, Remifentanil], Tetany );
```
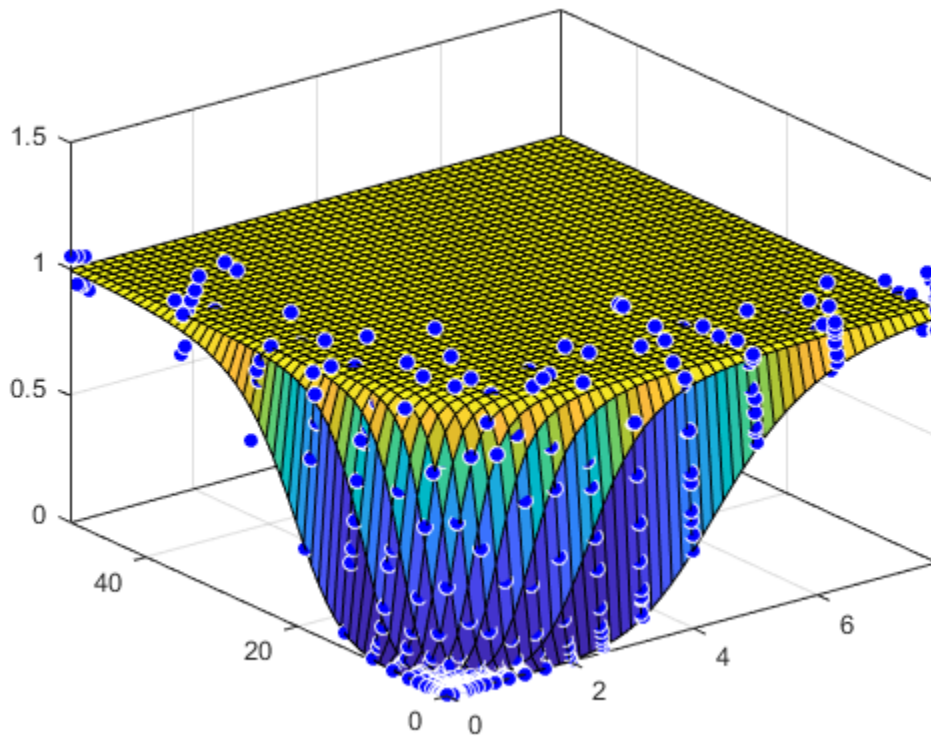
Fit a Surface to Sedation

```
[f, gof] = fit( [Propofol, Remifentanil], Sedation, ft, opts, 'problem', Emax )
```

```
     General model:
     f(CA,CB) = Emax*( CA/IC50A + CB/IC50B + alpha*( CA/IC50A ) * ( CB/IC50B
                   ) )^n /(( CA/IC50A + CB/IC50B + alpha*( CA/IC50A )
               * ( CB/IC50B ) )^n  + 1 )
     Coefficients (with 95% confidence bounds):
       IC50A =        1.843  (1.838, 1.847)
       IC50B =         13.7  (13.67, 13.74)
       alpha =        1.986  (1.957, 2.015)
       n =        44.27  (42.56, 45.98)
     Problem parameters:
       Emax =           1

gof = struct with fields:
          sse: 0.0574
      rsquare: 0.9994
          dfe: 393
    adjrsquare: 0.9994
         rmse: 0.0121
```

```
plot( f, [Propofol, Remifentanil], Sedation );
```
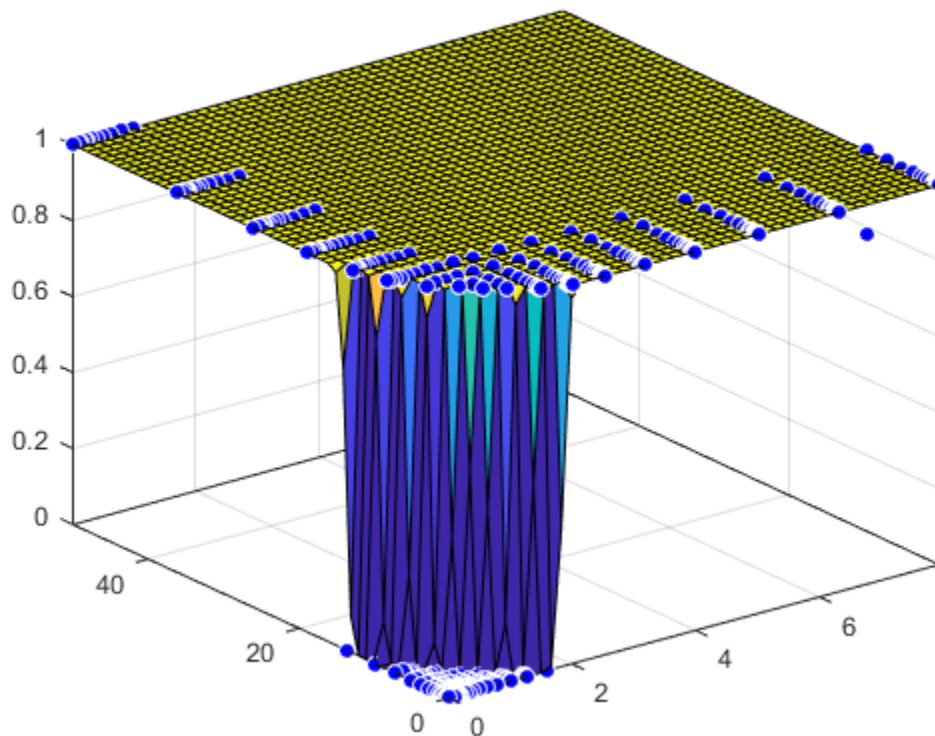
Fit a Surface to Laryingoscopy

```
[f, gof] = fit( [Propofol, Remifentanil], Laryingoscopy, ft, opts, 'problem', Emax )

    General model:
    f(CA,CB) = Emax*( CA/IC50A + CB/IC50B + alpha*( CA/IC50A ) * ( CB/IC50B
                ) )^n /(( CA/IC50A + CB/IC50B + alpha*( CA/IC50A )
                * ( CB/IC50B ) )^n  + 1 )
    Coefficients (with 95% confidence bounds):
      IC50A =       5.192  (5.177, 5.207)
      IC50B =       37.77  (37.58, 37.97)
      alpha =       19.67  (19.48, 19.86)
      n =           37  (35.12, 38.87)
    Problem parameters:
      Emax =            1

gof = struct with fields:
        sse: 0.1555
    rsquare: 0.9982
        dfe: 393
  adjrsquare: 0.9982
       rmse: 0.0199


plot( f, [Propofol, Remifentanil], Laryingoscopy );
```
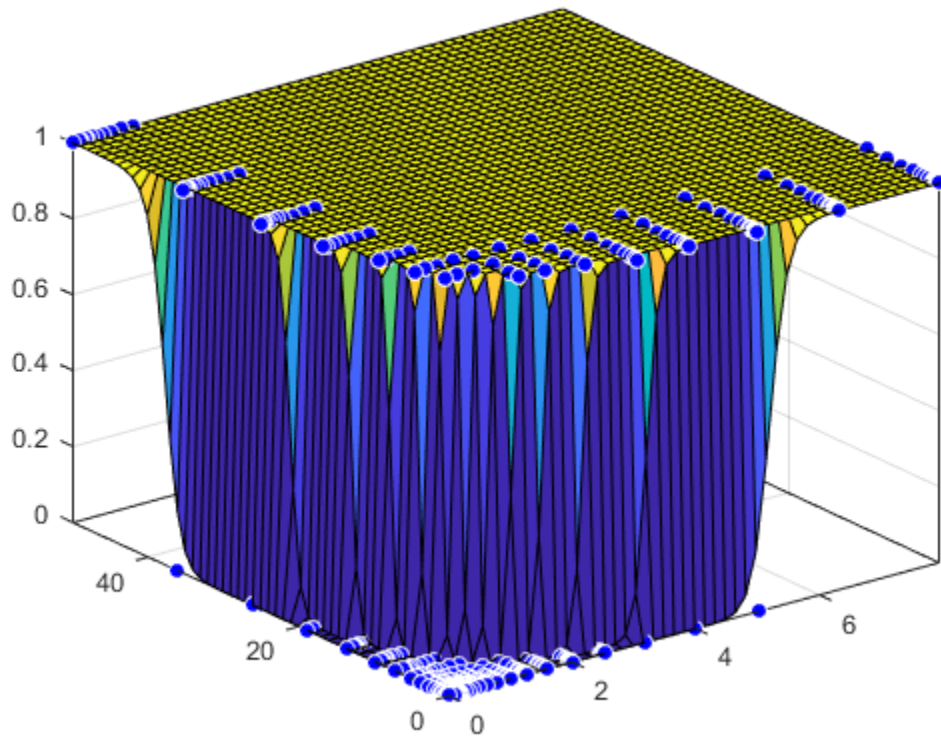
How to Construct Splines

This example shows how to construct splines in various ways using the spline functions in Curve Fitting Toolbox™.

Interpolation

You can construct a *cubic spline interpolant* that matches the cosine function at the following sites x, using the `csapi` command.

```
x = 2*pi*[0 1 .1:.2:.9];
y = cos(x);
cs = csapi(x,y);
```

You can then view the interpolating spline by using `fnplt`.

```
fnplt(cs,2);
axis([-1 7 -1.2 1.2])
hold on
plot(x,y,'o')
hold off
```
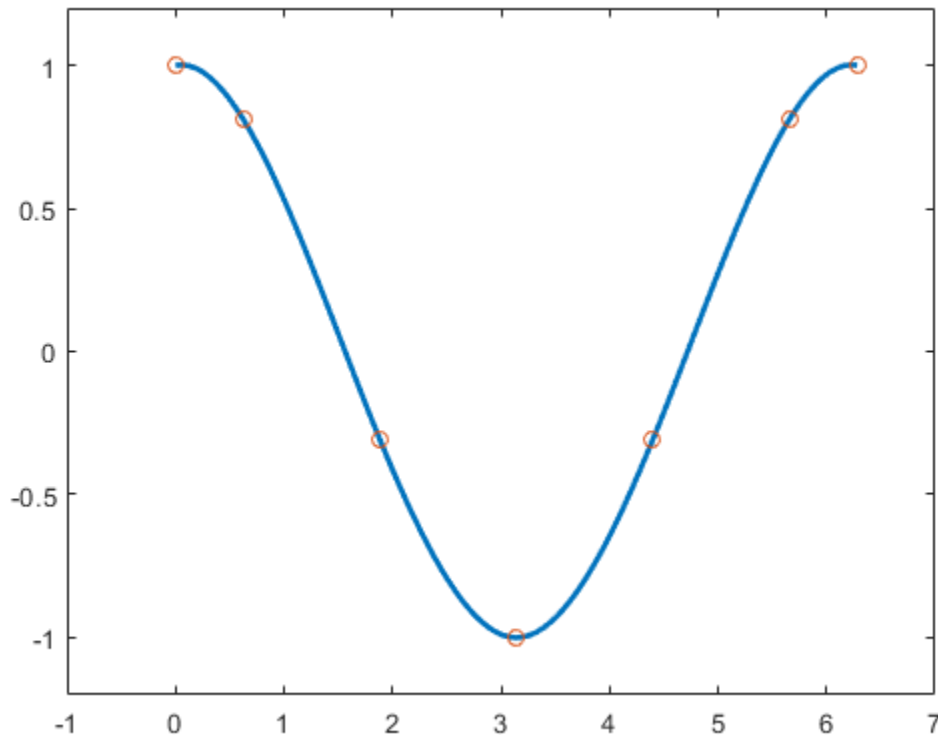

Checking the Interpolant

The cosine function is 2*pi-periodic. How well does our cubic spline interpolant do in that regard? One way to check is to compute the difference in the first derivative at the two endpoints.

```
diff( fnval( fnder(cs), [0 2*pi] ) )
```

ans = -0.1375

To enforce periodicity, use `csape` instead of `csapi`.

```
csp = csape( x, y, 'periodic' );
hold on
fnplt(csp,'g')
hold off
```
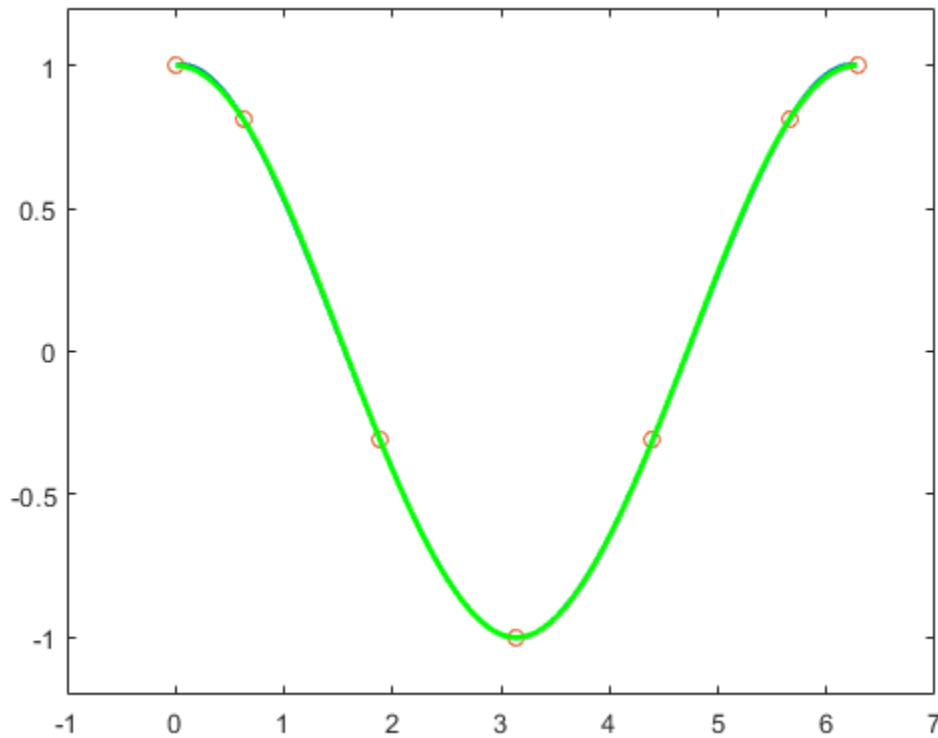

Now the check gives

```
diff( fnval( fnder(csp), [0 2*pi] ) )
```

ans = -2.2806e-17

Even the second derivative now matches at the endpoints.

```
diff( fnval( fnder(csp, 2), [0 2*pi] ) )
```

ans = -2.2204e-16

The *piecewise linear interpolant* to the same data is available via `spapi`. Here we add it to the previous plot, in red.

```
pl = spapi(2, x, y);
hold on
```

```
fnplt(pl, 'r', 2)
hold off
```
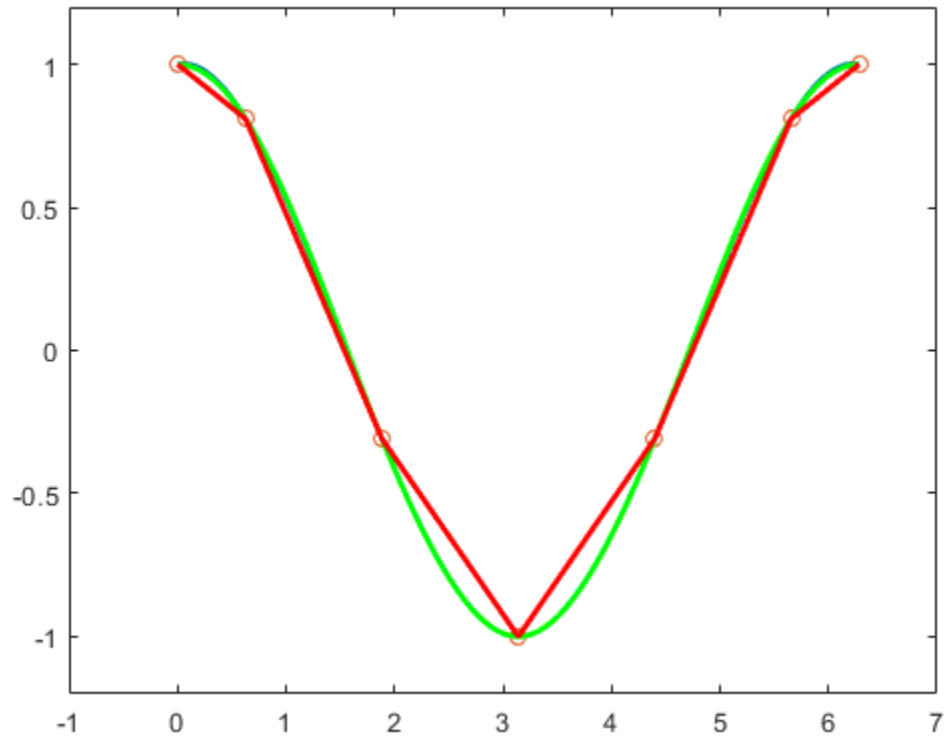

Smoothing

If the data are noisy, you usually want to approximate rather than interpolate. For example, with these data

```
x = linspace(0,2*pi,51);
noisy_y = cos(x) + .2*(rand(size(x))-.5);
plot(x,noisy_y,'x')
axis([-1 7 -1.2 1.2])
```
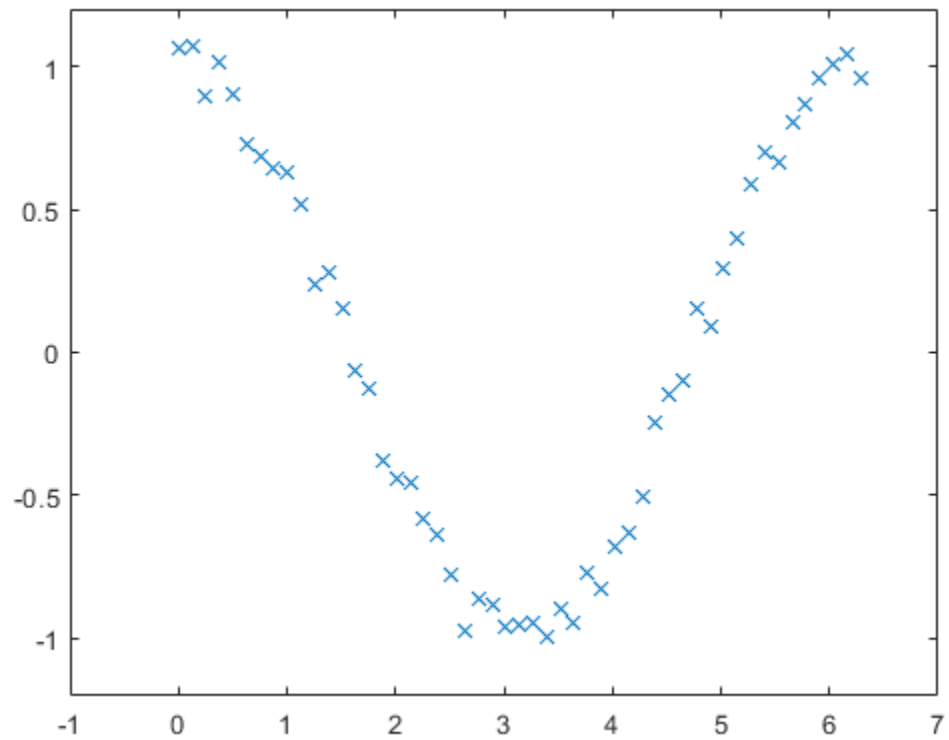
interpolation would give the wiggly interpolant shown below in blue.

```
hold on
fnplt( csapi(x, noisy_y) )
hold off
```
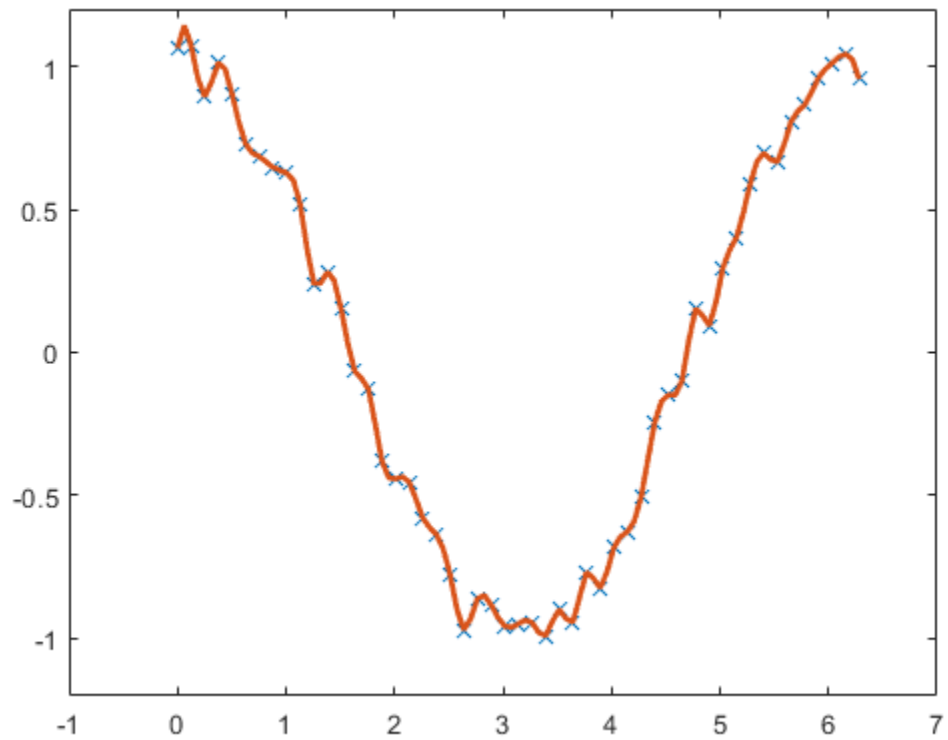
In contrast, smoothing with a proper tolerance

```
tol = (.05)^2*(2*pi)

tol = 0.0157
```

gives a smoothed approximation, shown below in red.

```
hold on
fnplt( spaps(x, noisy_y,  tol), 'r', 2 )
hold off
```
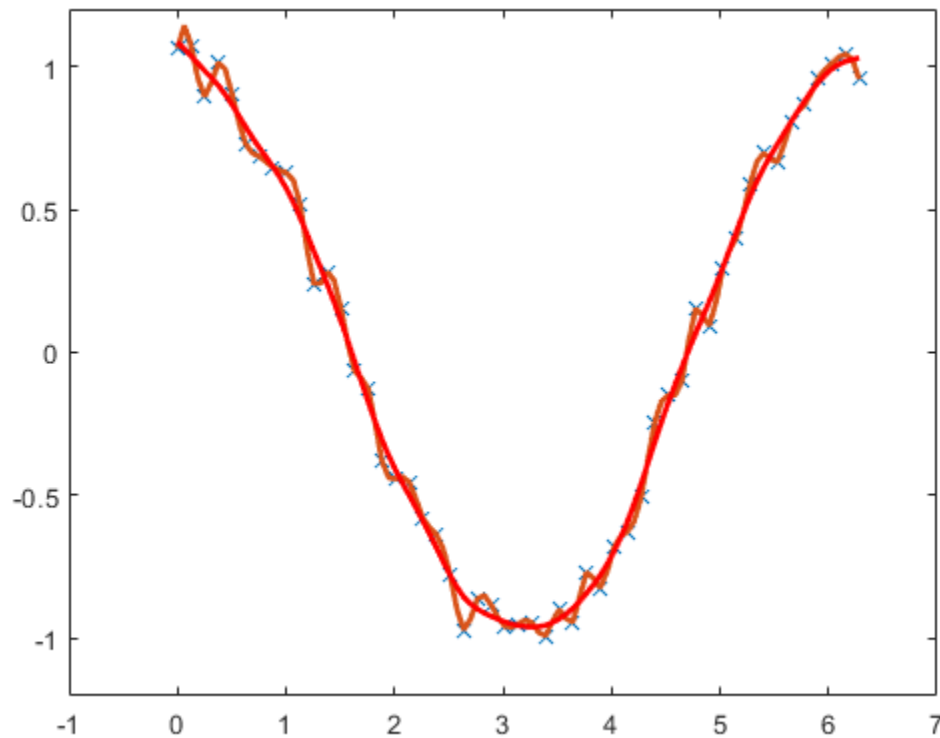
The approximation is much worse near the ends of the interval, and is far from periodic. To enforce periodicity, approximate to periodically extended data, then restrict the approximation to the original interval.

```
noisy_y([1 end]) = mean( noisy_y([1 end]) );
lx = length(x);
lx2 = round(lx/2);
range = [lx2:lx 2:lx 2:lx2];
sps = spaps([x(lx2:lx)-2*pi x(2:lx) x(2:lx2)+2*pi],noisy_y(range),2*tol);
```

This gives the more nearly periodic approximation, shown in black.

```
hold on
fnplt(sps, [0 2*pi], 'k', 2)
hold off
```
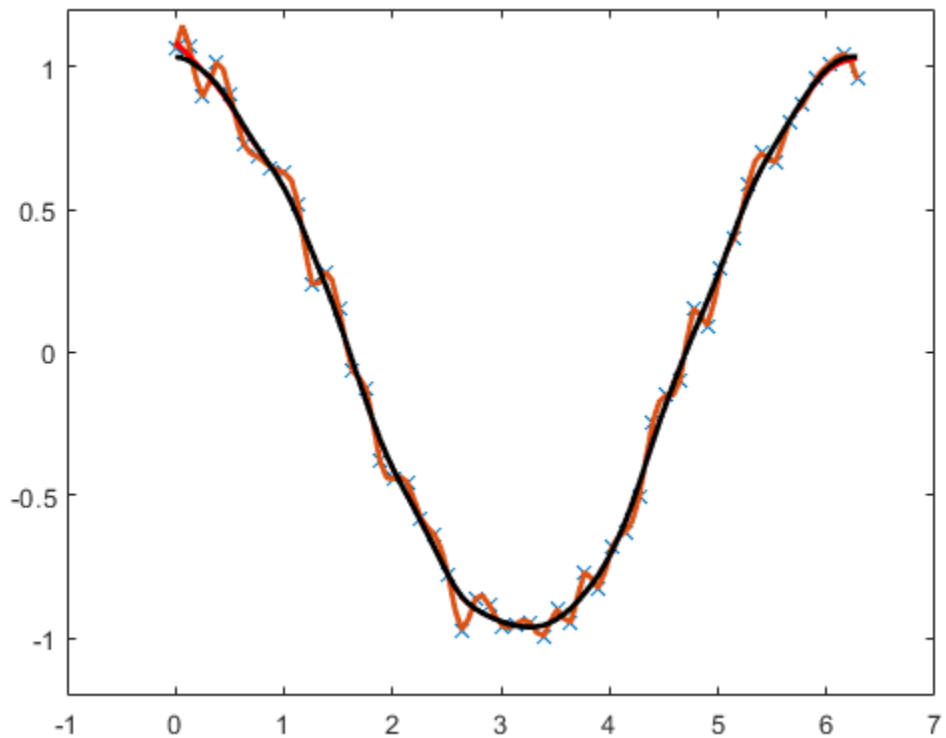
Least-Squares Approximation

Alternatively, you could use least-squares approximation to the noisy data by a spline with few degrees of freedom.

For example, you might try a cubic spline with just four pieces.

```
spl2 = spap2(4, 4, x, noisy_y);
fnplt(spl2,'b',2);
axis([-1 7 -1.2 1.2])
hold on
plot(x,noisy_y,'x')
hold off
```
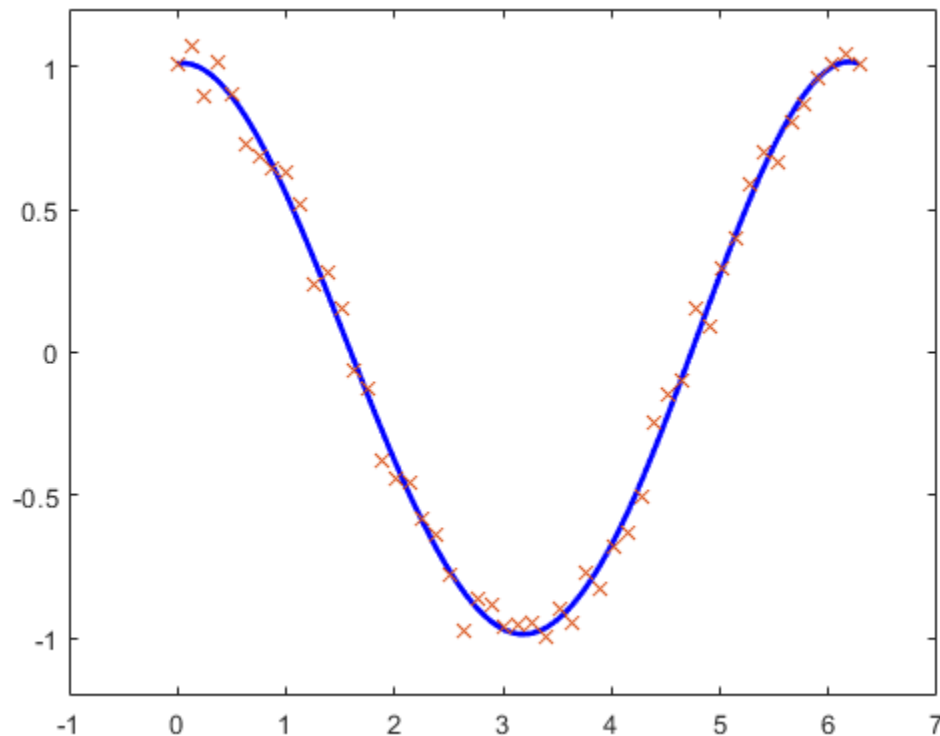
Knot Selection

When using `spapi` or `spap2`, you usually have to specify a particular spline space. This is done by specifying a *knot sequence* and an *order*, and this may be a bit of a problem. However, when doing spline interpolation to `x,y` data using a spline of order `k`, you can use the function `optknt` to supply a good knot sequence, as in the following example.

```
k = 5;    % order 5, i.e., we are working with quartic splines
x = 2*pi*sort([0 1 rand(1,10)]);
y = cos(x);
sp = spapi( optknt(x,k), x, y );
fnplt(sp,2,'g');
hold on
plot(x,y,'o')
hold off
axis([-1 7 -1.1 1.1])
```
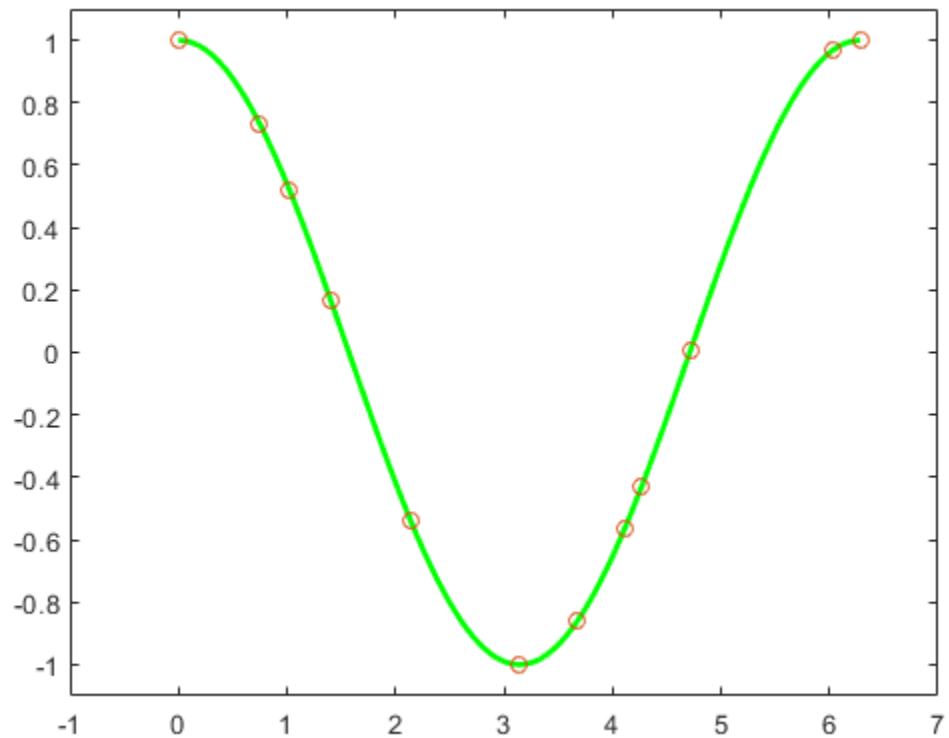
When doing least-squares approximation, you can use the current approximation to determine a possibly better knot selection with the aid of `newknt`. For example, the following approximation to the exponential function isn't all that good, as can be seen from its error, plotted in red.

```
x = linspace(0,10,101);
y = exp(x);
sp0 = spap2( augknt(0:2:10,4), 4, x, y );
plot(x,y-fnval(sp0,x),'r','LineWidth',2)
```
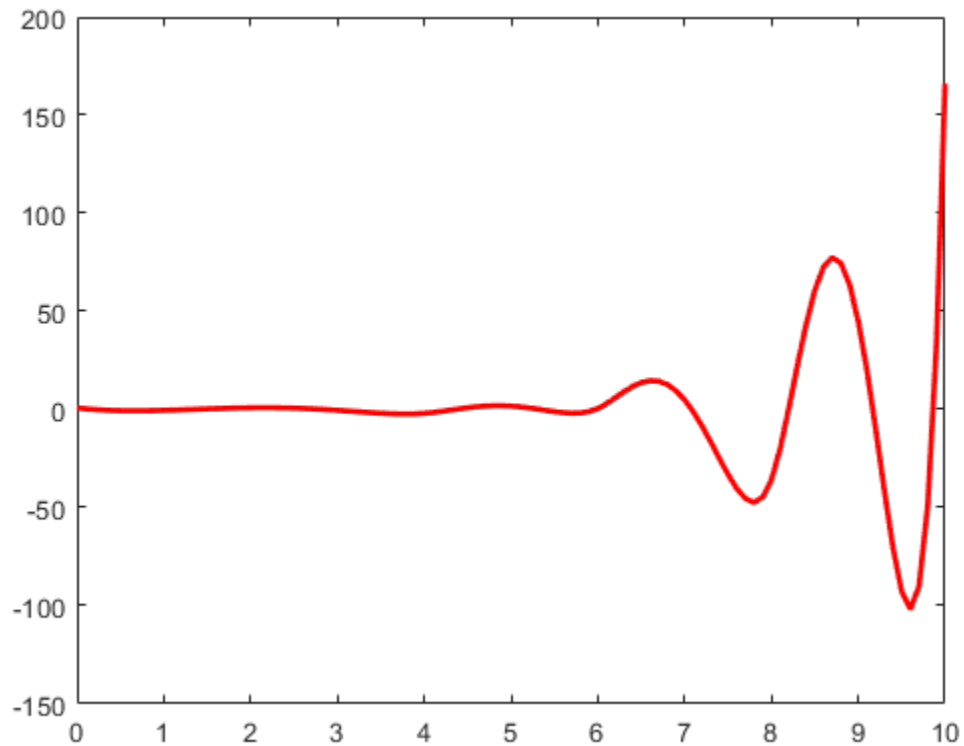
However, you can use that initial approximation to create another one with the *same* number of knots, but which are better distributed. Its error is plotted in black.

```
sp1 = spap2( newknt(sp0), 4, x, y );
hold on
plot(x,y-fnval(sp1,x),'k','LineWidth',2)
hold off
```
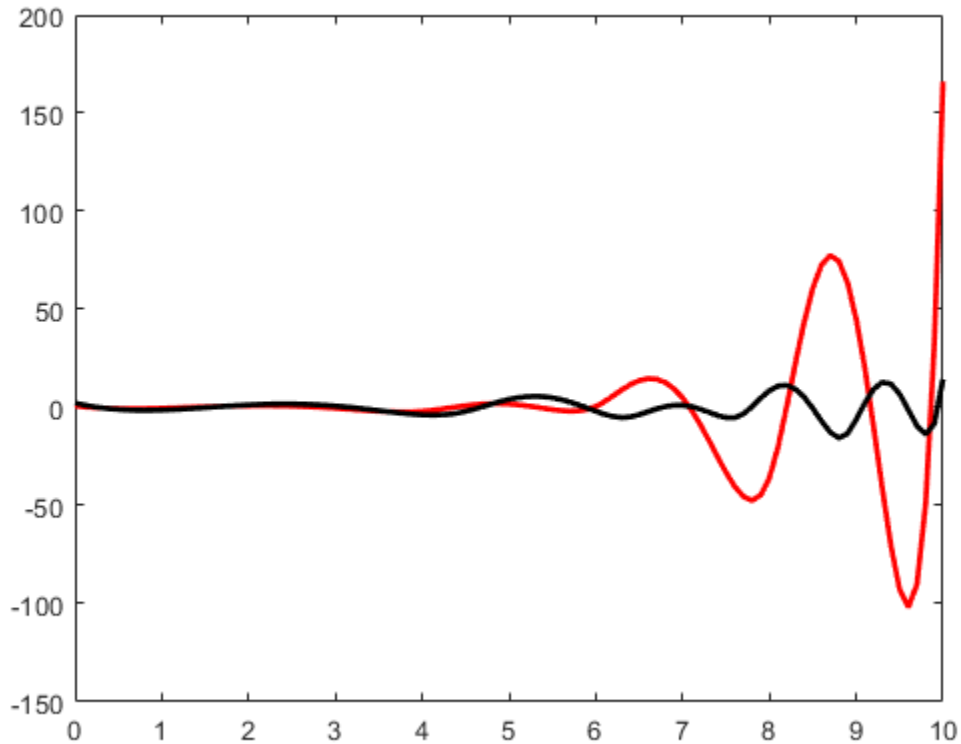
Gridded Data

All the spline interpolation and approximation commands in the Curve Fitting Toolbox can also handle gridded data, in any number of variables.

For example, here is a bicubic spline interpolant to the Mexican Hat function.

```
x =.0001+(-4:.2:4);
y = -3:.2:3;
[yy,xx] = meshgrid(y,x);
r = pi*sqrt(xx.^2+yy.^2);
z = sin(r)./r;
bcs = csapi({x,y}, z);
fnplt(bcs)
axis([-5 5 -5 5 -.5 1])
```
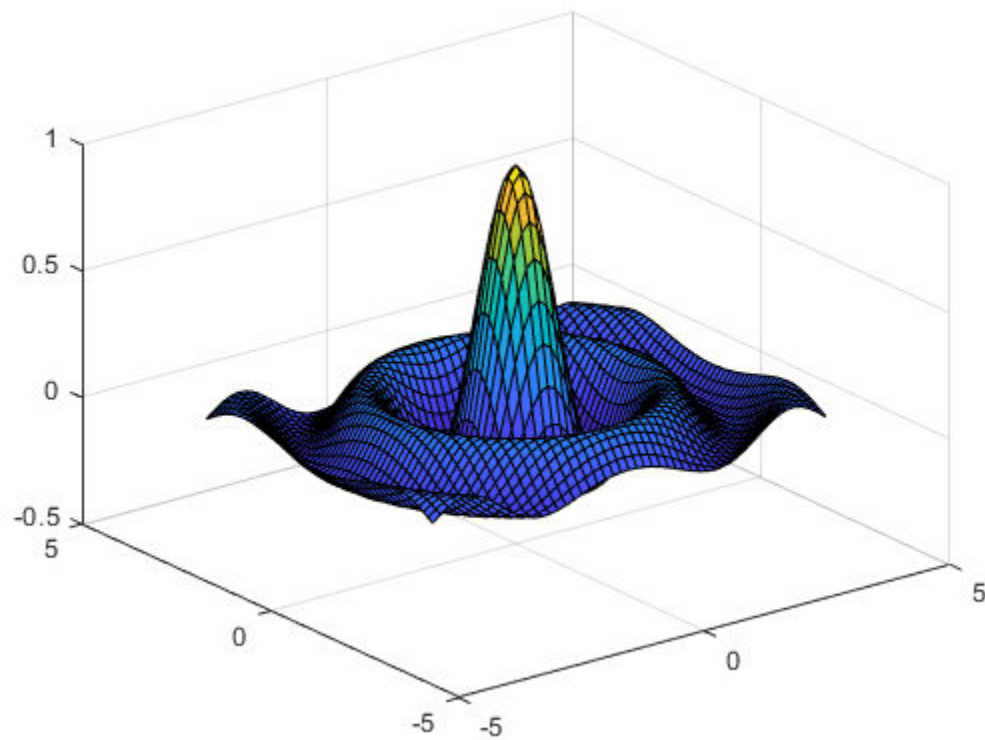
Here is the *least-squares* approximation to noisy values of that same function on the same grid.

```
knotsx = augknt(linspace(x(1), x(end), 21), 4);
knotsy = augknt(linspace(y(1), y(end), 15), 4);
bsp2 =  spap2({knotsx,knotsy},[4 4], {x,y},z+.02*(rand(size(z))-.5));
fnplt(bsp2)
axis([-5 5 -5 5 -.5 1])
```
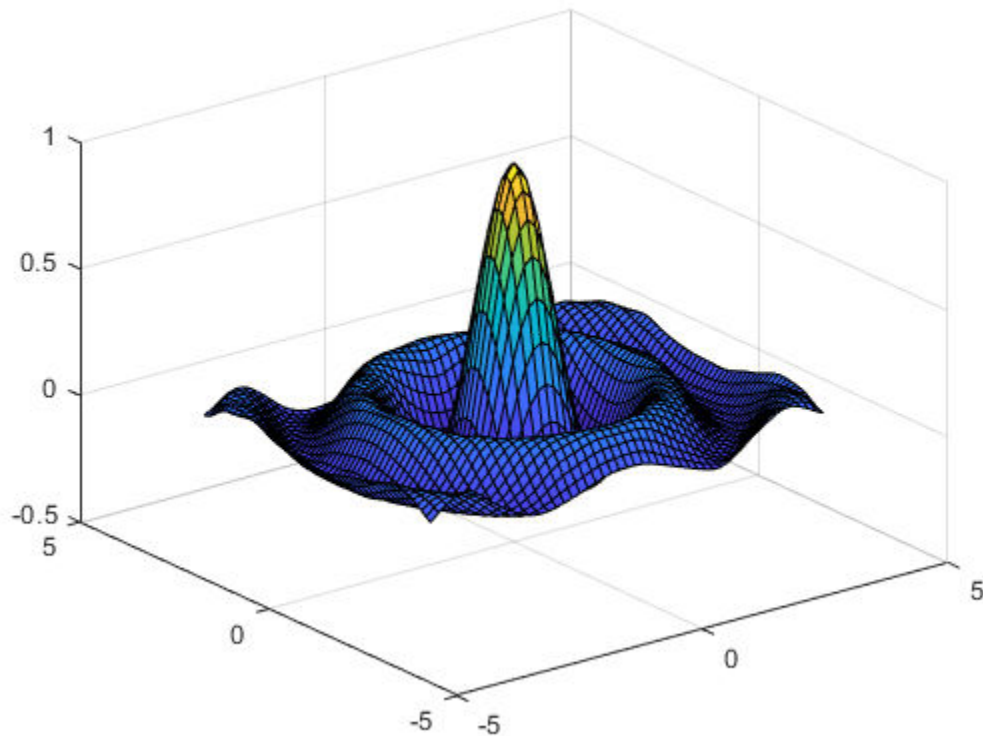
Curves

Gridded data can be handled easily because Curve Fitting Toolbox can deal with *vector-valued* splines. This also makes it easy to work with parametric curves.

Here, for example, is an approximation to infinity, obtained by putting a cubic spline curve through the points marked in the following figure.

```
t = 0:8;
xy = [0 0;1 1; 1.7 0;1 -1;0 0; -1 1; -1.7 0; -1 -1; 0 0].';
infty = csape(t, xy, 'periodic');
fnplt(infty, 2)
axis([-2 2 -1.1 1.1])
hold on
plot(xy(1,:),xy(2,:),'o')
hold off
```
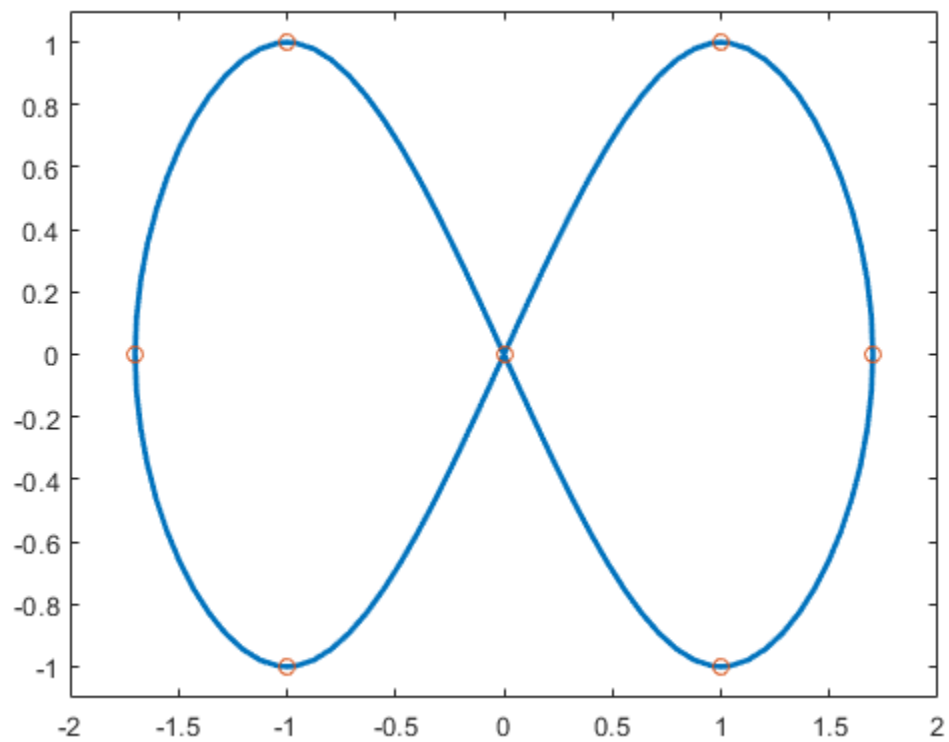
Here is the same curve, but with motion in a third dimension.

```
roller = csape( t , [ xy ;0 1/2 1 1/2 0 1/2 1 1/2 0], 'periodic');
fnplt( roller , 2, [0 4],'b' )
hold on
fnplt( roller, 2, [4 8], 'r')
plot3(0,0,0,'o')
hold off
```
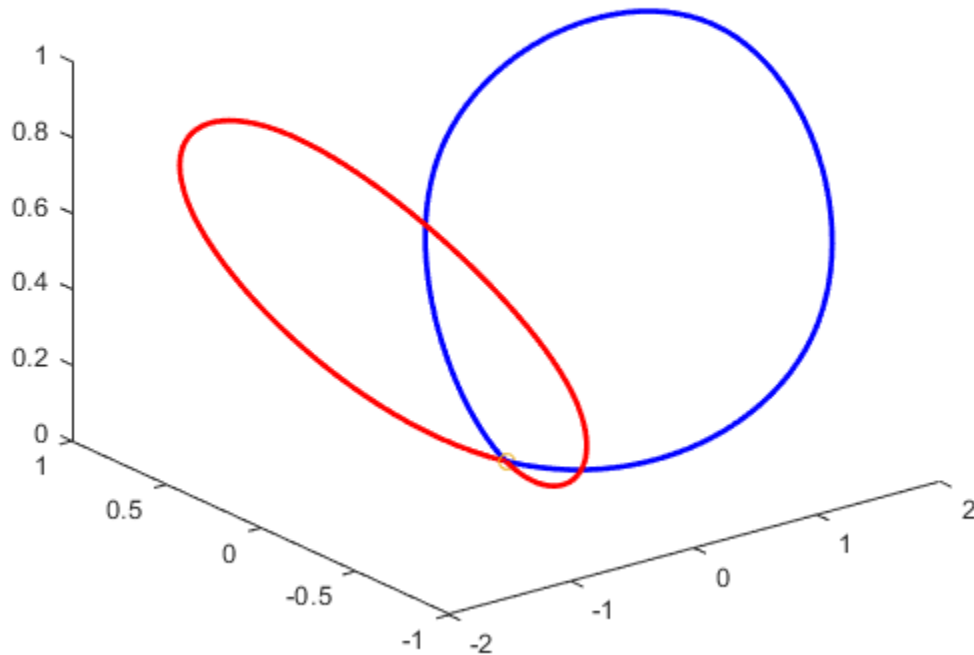
The two halves of the curve are plotted in different colors and the origin is marked, as an aid to visualizing this two-winged space curve.

Surfaces

Bivariate tensor-product splines with values in R^3 give surfaces. For example, here is a good approximation to a torus.

```
x = 0:4;
y = -2:2;
R = 4;
r = 2;
v = zeros(3,5,5);
v(3,:,:) = [0 (R-r)/2 0 (r-R)/2 0].'*[1 1 1 1 1];
v(2,:,:) = [R (r+R)/2 r (r+R)/2 R].'*[0 1 0 -1 0];
v(1,:,:) = [R (r+R)/2 r (r+R)/2 R].'*[1 0 -1 0 1];
dough0 = csape({x,y},v,'periodic');
fnplt(dough0)
axis equal, axis off
```
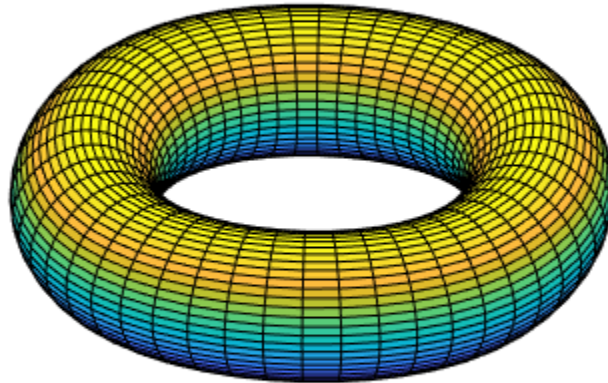
Here is a crown of normals to that surface.

```
nx = 43;
xy = [ones(1,nx); linspace(2,-2,nx)];
points = fnval(dough0,xy)';
ders = fnval(fndir(dough0,eye(2)),xy);
normals = cross(ders(4:6,:),ders(1:3,:));
normals = (normals./repmat(sqrt(sum(normals.*normals)),3,1))';
pn = [points;points+normals];
hold on
for j=1:nx
    plot3(pn([j,j+nx],1),pn([j,j+nx],2),pn([j,j+nx],3))
end
hold off
```
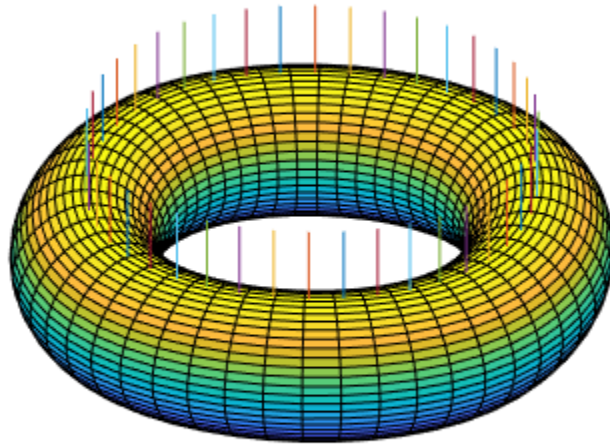
Finally, here is its projection onto the (x,y)-plane.

```
fnplt(fncmb(dough0, [1 0 0; 0 1 0]))
axis([-5.25 5.25 -4.14 4.14]), axis off
```
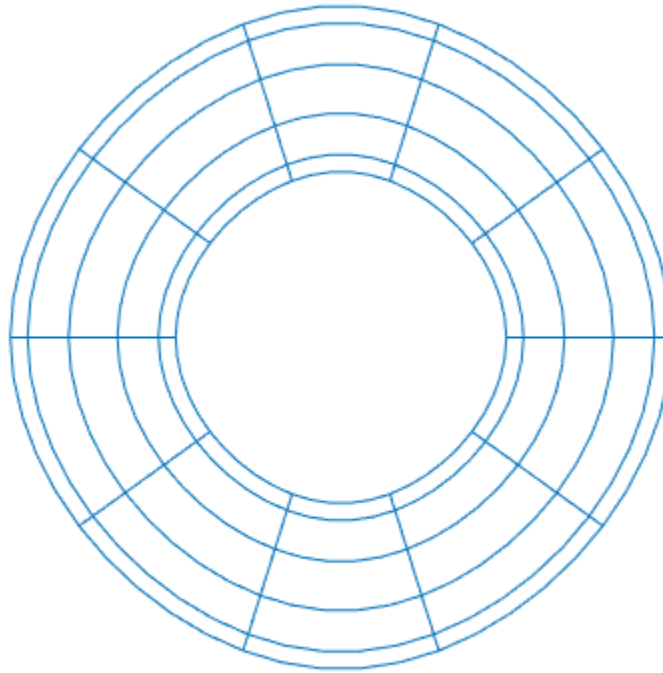
Scattered Data

It is also possible to interpolate to values given at ungridded data sites in the plane. Consider, for example, the task of mapping the unit square smoothly to the unit disk. We construct the data values, marked as circles, and the corresponding data sites, marked as x's. Each data site is connected to its associated value by an arrow.

```
n = 64;
t = linspace(0,2*pi,n+1); t(end) = [];
values = [cos(t); sin(t)];
plot(values(1,:),values(2,:),'or')
axis equal, axis off

sites = values./repmat(max(abs(values)),2,1);
hold on
plot(sites(1,:),sites(2,:),'xk')
quiver(sites(1,:),sites(2,:), ...
       values(1,:)-sites(1,:), values(2,:)-sites(2,:))
hold off
```
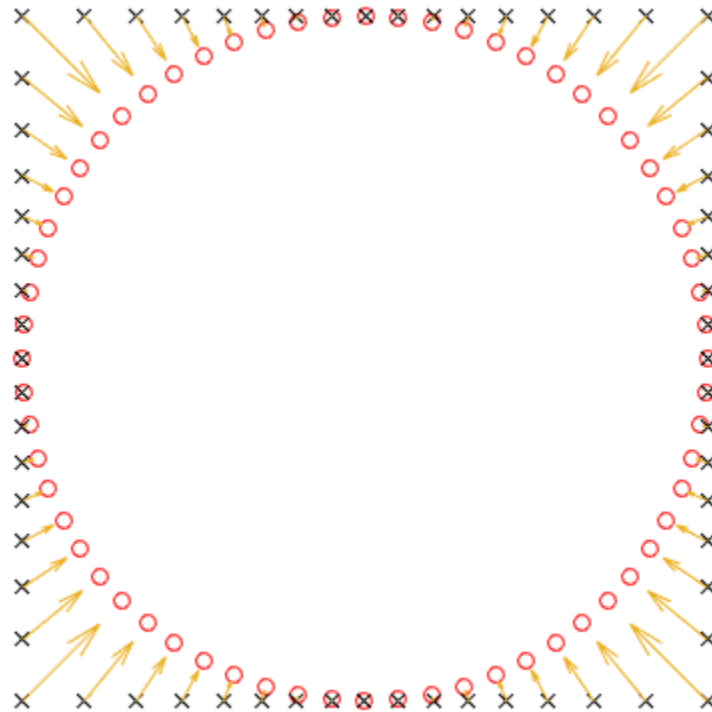
Then use `tpaps` to construct a bivariate interpolating vector-valued thin-plate spline.

```
st = tpaps(sites, values, 1);
```

The spline does indeed map the unit square smoothly (approximately) to the unit disk, as its plot via `fnplt` indicates. The plot shows the image of a uniformly-spaced square grid under the spline map in `st`.

```
hold on
fnplt(st)
hold off
```
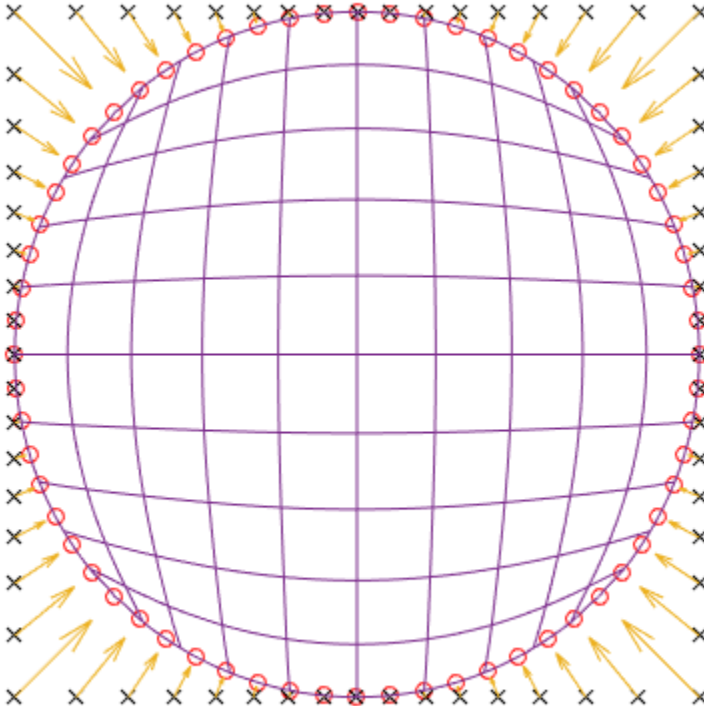
Construct and Work with the B-form

This example shows how to construct and work with the B-form of a spline in Curve Fitting Toolbox™.

Introduction

In Curve Fitting Toolbox, a piecewise polynomial, or pp, function in B-form is often called a *spline*.

The *B-form* of a (univariate) pp is specified by its (nondecreasing) *knot* sequence t and by its B-spline *coefficient* sequence a.

Given the knot sequence and coefficient sequence of a pp, the command spmak returns the corresponding B-form, for use in commands such as fnval (evaluate a function), fnplt (plot a function), fnder (differentiate a function), and other related commands.

The resulting spline is of *order* k := length(t) - size(a,2). This means that all its polynomial pieces have degree < k.

To say that a spline s has knots t and coefficients a means that

n

s(x) := sum B_{j,k}(x) * a(:,j),

j=1

where B_(j,k) = B(. | t_j, ..., t_{j+k}) is the j-th *B-spline* of *order* k for the given *knot sequence* t, i.e., the B-spline with knots t_j, ..., t_{j+k}. For example,

```
t = [.1 .4 .5 .8 .9];
a = 1;
fnplt(spmak(t,a),2.5);
tmp = repmat(t,3,1);
ty = repmat(.1*[1;-1;NaN],1,5);
hold on
plot(tmp(:),ty(:))
text(.65,.5,'B( \cdot | .1, .4, .5, .8, .9)','FontSize',12)
text(.05,1.,'s(x)  =  \Sigma_j  B( x | t_j , \ldots, t_{j+k} ) a(:,j)', ...
     'FontSize',16,'Color','r')
axis([0 1 -.2 1.2])
title('A B-spline of Order 4')
hold off
```

A B-spline of Order 4

$$s(x) = \Sigma_j \ B(\ x \ | \ t_j \ , \ ..., \ t_{j+k} \ ) \ a(:,j)$$

$B(\ \cdot \ | \ .1, \ .4, \ .5, \ .8, \ .9)$
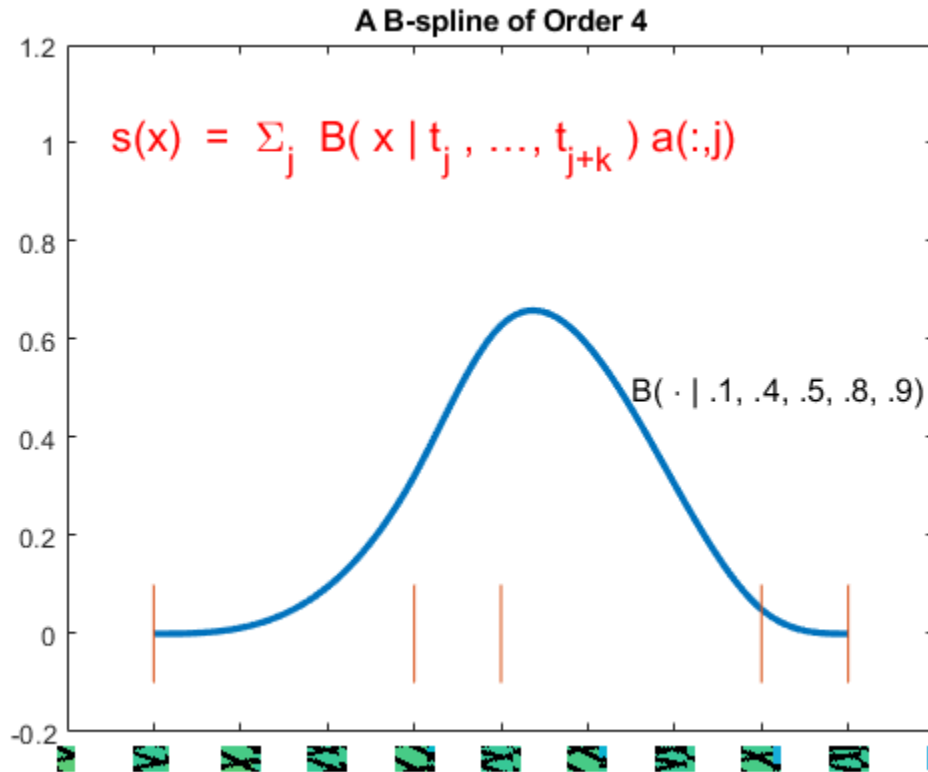
The Local Partition of Unity

The value of the spline

```
s(x) = sum B_{j,k}(x) a(:,j)

j
```

at any x in the knot interval [t_i .. t_{i+1}] is a *convex* combination of the k coefficients a(:.i-k+1), ..., a(:,i) since, on that interval, only the k B-splines B_{i-k+1,k}, ..., B_{i,k} are nonzero, and they are nonnegative there and sum to 1, as is illustrated in the next figure.

This is often summarized by saying that the B-splines provide a *local* (nonnegative) *partition of unity*.

```
k = 3;
n = 3;
t = [1 1.7 3.2 4.2 4.8 6];
tt = (10:60)/10;
vals = fnval(spmak(t,eye(k)),tt);
plot(tt.',vals.');
hold on
ind = find(tt>=t(3)&tt<=t(4));
plot(tt(ind).',vals(:,ind).','LineWidth',3)
plot(t([3 4]),[1 1],'k','LineWidth',3)
ty = repmat(.1*[1;-1;NaN],1,6);
plot([0 0 -.2 0 0 -.2 0 0],[-.5 0 0 0 1 1 1 1.5],'k')
```

12-41

```
text(-.5,0,'0','FontSize',12)
text(-.5,1,'1','FontSize',12)
tmp = repmat(t,3,1);
plot(tmp(:),ty(:),'k');
yd = -.25;
text(t(1),yd,'t_{i-2}','FontSize',12);
text(t(3),yd,'t_i','FontSize',12);
text(t(4),yd,'t_{i+1}','FontSize',12);
text(1.8,.5,'B_{i-2,3}','FontSize',12);
text(5,.45,'B_{i,3}','FontSize',12);
axis([-.5 7 -.5 1.5])
title('B-splines Form a Local Partition of Unity')
axis off
hold off
```
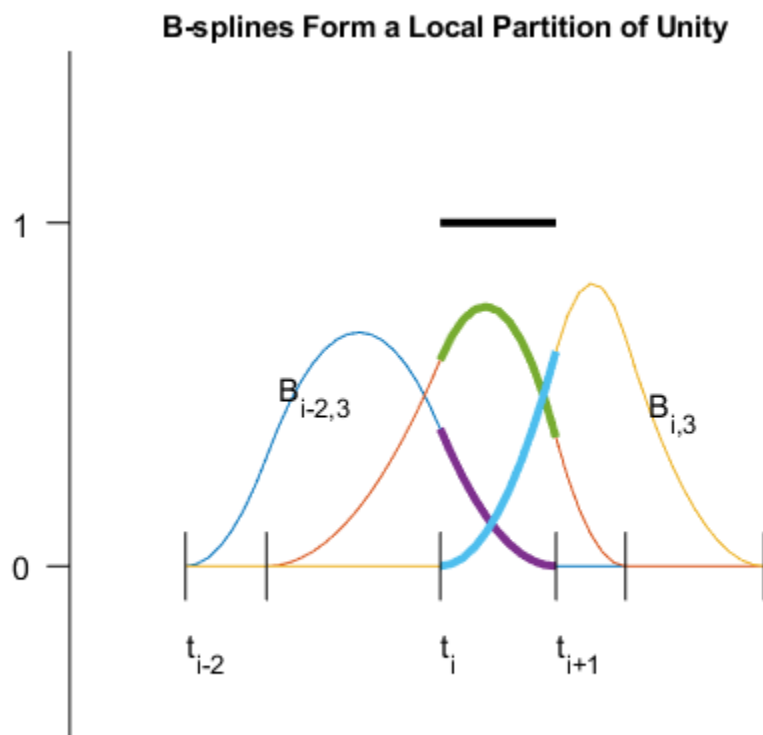

The Convex Hull Property and the Control Polygon

When the coefficients are points in the plane and, correspondingly, the spline

```
s(x) = sum B_{j,k}(x) a(:,j)

j
```

traces out a curve, this means that the curve piece

```
{ s(x) : t_i <= x <= t_{i+1} }
```

highlighted in the figure below by a larger LineWidth, lies in the convex hull, shown in yellow in the figure below, of the k points `a(:,i-k+1), ... a(:,i)`.

```
t = 1:9;
c = [2 1.4;1 .5; 2 -.4; 5 1.4; 6 .5; 5 -.4].';
sp = spmak(t,c);
fill(c(1,3:5),c(2,3:5),'y','EdgeColor','y');
hold on
fnplt(sp,t([3 7]),1.5)
fnplt(sp,t([5 6]),3)
plot(c(1,:),c(2,:),':ok')
text(2,-.55,'a(:,i-2)','FontSize',12)
text(5,1.6,'a(:,i-1)','FontSize',12)
text(6.1,.5,'a(:,i)','FontSize',12)
title('The Convex-Hull Property')
axis([.5 7 -.8 1.8])
axis off
hold off
```
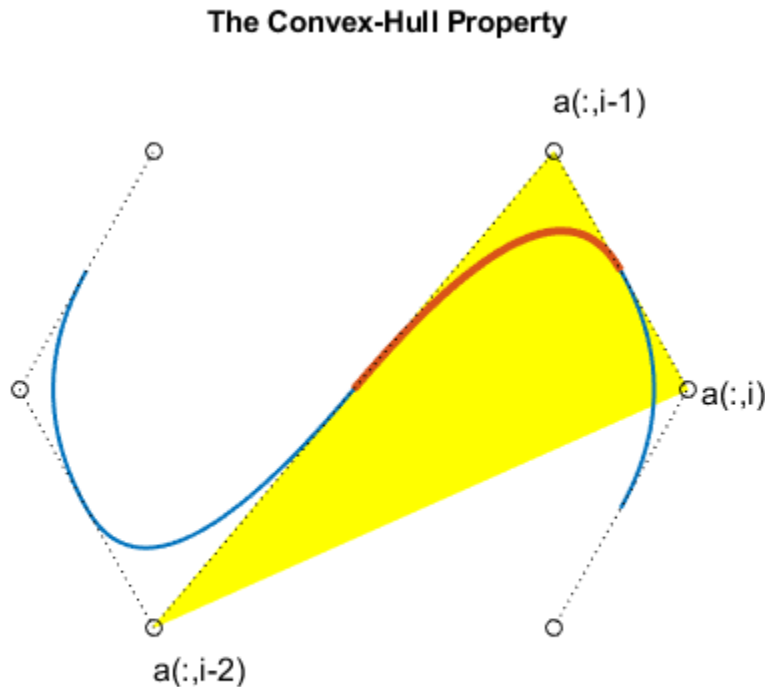

For a quadratic spline (i.e., k = 3), as shown here, it even means that the curve is tangent to the *control polygon* (shown as dotted lines). This is the broken line that connects the coefficients, which are called *control points* in this connection (shown here as open circles).

The Control Polygon for Scalar-Valued Splines

We can think of the graph of the *scalar*-valued spline

```
s = sum B_{j,k}*a(j)
```

j

as the curve x |--> (x,s(x)). Since

x = sum B_{j,k}(x) t^*_j

j

for x in the interval [t_k .. t_{n+1}], where

t^*_i := (t_{i+1} + ... + t_{i+k-1})/(k-1) for i = 1:n

are the knot averages obtainable using the aveknt command, the *control polygon* for a scalar-valued spline is the broken line with vertices (t^*_i, a(i)), i=1:n.

The example below shows a *cubic* spline (k = 4), with 4-fold end knots, hence

t^*_1 = t_1 and t^*_n = t_{n+k}.

```matlab
t = [0 .2 .35 .47 .61 .84 1]*(2*pi);
s = t([1 3 4 5 7]);
knots = augknt(s,4);
sp = spapi(knots,t,sin(t)+1.8);
fnplt(sp,2);
hold on
c = fnbrk(sp,'c');
ts = aveknt(knots,4);
plot(ts,c,':ok');
tt = [s;s;NaN(size(s))];
ty = repmat(.25*[-1;1;NaN], size(s));
plot(tt(:),ty(:),'r')
plot(ts(1,:),zeros(size(ts)),'*')
text(knots(5),-.5,'t_5','FontSize',12)
text(ts(2),-.45,'t^*_2','FontSize',12)
text(knots(1)-.28,-.5,'t_1=t_4','FontSize',12)
text(knots(end)-.65,-.45,'t_{n+1}=t^*_n=t_{n+4}','FontSize',12)
title('A Cubic Spline and its Control Polygon')
axis([-.72 7 -.5 3.5])
axis off
hold off
```

A Cubic Spline and its Control Polygon
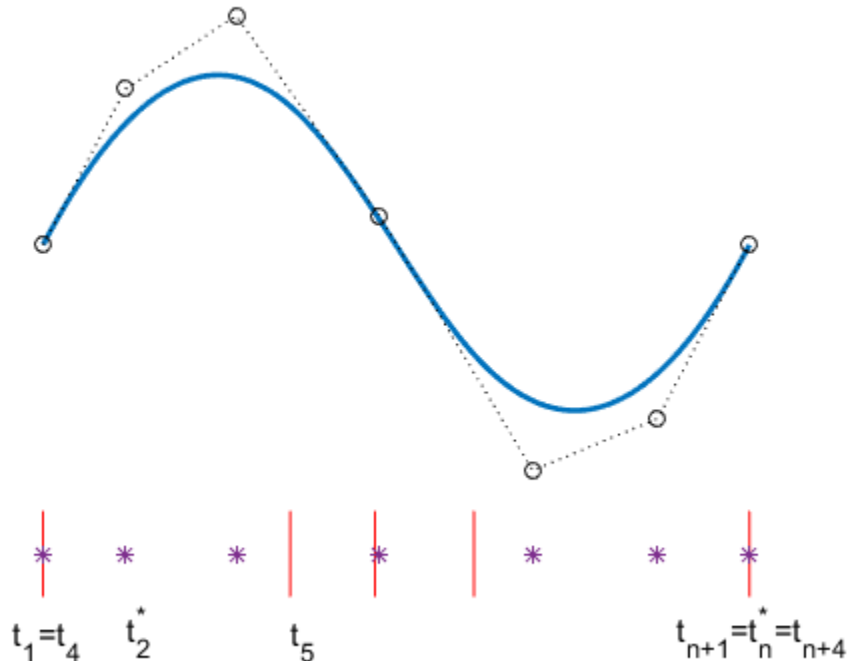

```
savesp = sp;
```

The essential parts of the B-form are the knot sequence t and the B-spline coefficient sequence a. Other parts are the *number* n of the B-splines or coefficients involved, the *order* k of its polynomial pieces, and the *dimension* d of its coefficients a. In particular, size(a) equals [d,n].

There is one more part, namely the *basic interval*, [t(1) .. t(end)]. It is used as the default interval when plotting the function. Also, a spline is taken to be continuous from the right everywhere except at the right endpoint of the basic interval, where it is taken to be continuous from the left. This is illustrated in the example below, for a spline created using spmak.

```
b = 0:3;
sp = spmak(augknt(b,3),[-1,0,1,0,-1]);
x = linspace(-1,4,51);
plot(x,fnval(sp,x),'x')
hold on
axis([-2 5,-1.5,1])
tx = repmat(b,3,1);
ty = repmat(.5*[1;-1;NaN],1,length(b));
plot(tx(:),ty(:),'-r')
legend({'Spline Values' 'Knots'})
hold off
title({'A Spline in B-form is Right(Left)-Continuous ';...
        'at Left(Right) Endpoint of its Basic Interval'})
```

A Spline in B-form is Right(Left)-Continuous at Left(Right) Endpoint of its Basic Interval
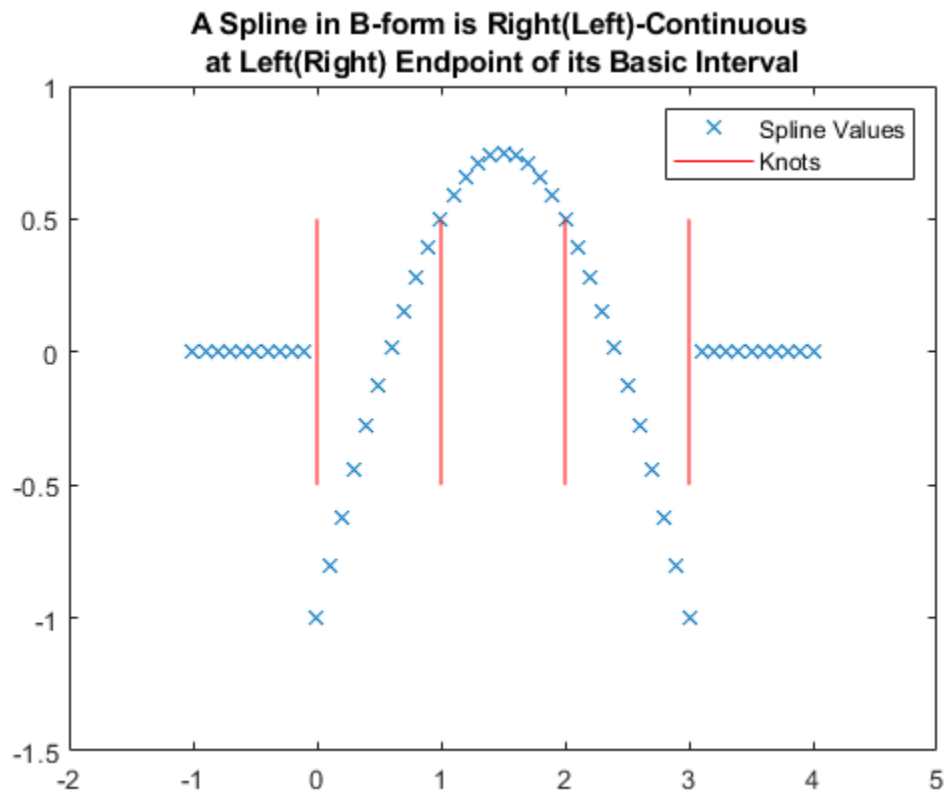
`fnbrk` can be used to obtain any or all parts of a B-form. For example, here is the output provided by `fnbrk` for the B-form of the spline shown above.

```
fnbrk(sp)
```

```
The input describes a B- form

knots(1:n+k)
     0     0     0     1     2     3     3     3

coefficients(d,n)
    -1     0     1     0    -1

number n of coefficients
     5

order k
     3

dimension d of target
     1
```

However, there is usually *no need* to know any of these parts. Rather, you use commands like `spapi` or `spaps` to construct the B-form of a spline from some data, then use commands like `fnval`, `fnplt`, `fnder`, etc., to make use of the spline constructed, without any need to look at its various parts.

The following sections give more detailed information about the B-splines, in particular about the important role played by knot *multiplicity*.

Knot Multiplicity

Here, for k = 2, 3, and 4, are B-splines of order k, and below them their first and second (piecewise) derivatives, to illustrate some facts about B-splines. Try out the `bspligui` tool if you want to experiment with examples of your own.

```matlab
cl = ['g','r','b','k','k'];
v = 5.4; d1 = 2.5; d2 = 0; s1 = 1; s2 = .5;
t1 = [0 .8 2];
t2 = [3 4.4 5  6];
t3 = [7  7.9  9.2 10 11];
tt = [t1 t2 t3];
ext = tt([1 end])+[-.5 .5];
plot(ext([1 2]),[v v],cl(5))
hold on
plot(ext([1 2]),[d1 d1],cl(5))
plot(ext([1 2]),[d2 d2],cl(5))
ts = [tt;tt;NaN(size(tt))];
ty = repmat(.2*[-1;0;NaN],size(tt));
plot(ts(:),ty(:)+v,cl(5))
plot(ts(:),ty(:)+d1,cl(5))
plot(ts(:),ty(:)+d2,cl(5))
b1 = spmak(t1,1);
p1 = [t1;0 1 0];
db1 = fnder(b1);
p11 = fnplt(db1,'j');
p12 = fnplt(fnder(db1));
lw = 2;
plot(p1(1,:),p1(2,:)+v,cl(2),'LineWidth',lw)
plot(p11(1,:),s1*p11(2,:)+d1,cl(2),'LineWidth',lw)
plot(p12(1,:),s2*p12(2,:)+d2,cl(2),'LineWidth',lw)
b1 = spmak(t2,1);
p1 = fnplt(b1);
db1 = fnder(b1);
p11 = [t2;fnval(db1,t2)];
p12 = fnplt(fnder(db1),'j');
plot(p1(1,:),p1(2,:)+v,cl(3),'LineWidth',lw)
plot(p11(1,:),s1*p11(2,:)+d1,cl(3),'LineWidth',lw)
plot(p12(1,:),s2*p12(2,:)+d2,cl(3),'LineWidth',lw)
b1 = spmak(t3,1);
p1 = fnplt(b1);
db1 = fnder(b1);
p11 = fnplt(db1);
p12=[t3;fnval(fnder(db1),t3)];
plot(p1(1,:),p1(2,:)+v,cl(4),'LineWidth',lw)
plot(p11(1,:),s1*p11(2,:)+d1,cl(4),'LineWidth',lw)
plot(p12(1,:),s2*p12(2,:)+d2,cl(4),'LineWidth',lw)
tey = v+1.5;
text(t1(2)-.5,tey,'linear','FontSize',12,'Color',cl(2))
text(t2(2)-.8,tey,'quadratic','FontSize',12,'Color',cl(3))
text(t3(3)-.5,tey,'cubic','FontSize',12,'Color',cl(4))
text(-2,v,'B','FontSize',12)
text(-2,d1,'DB','FontSize',12)
text(-2,d2,'D^2B')
axis([-1 12 -2 7.5])
title({'B-splines with Simple Knots and Their Derivatives'})
axis off
hold off
```

B-splines with Simple Knots and Their Derivatives
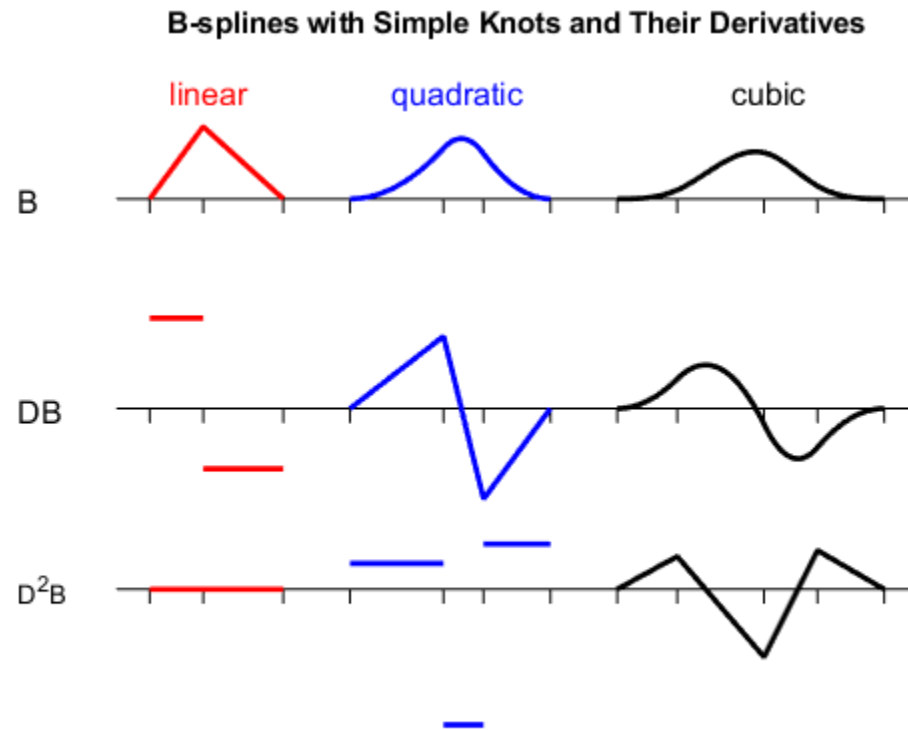
1. The B-spline B_{j,k} = B(. | t_j, ..., t_{j+k}) is a pp of order k with breaks at t_j, ..., t_{j+k} (and nowhere else). Actually, its nontrivial polynomial pieces are all of exact degree k-1.

For example, the rightmost B-spline above involves 5 knots, hence is of order 4, i.e., a *cubic* B-spline. Correspondingly, its second derivative is piecewise linear.

2. B_{j,k} is positive on the interval (t_j .. t_{j+k}) and is zero off that interval. It also vanishes at the endpoints of that interval, unless the endpoint is a knot of multiplicity k (see the rightmost example in the next figure).

3. Knot *multiplicity* determines the smoothness with which the two adjacent polynomials join across that knot. In shorthand, the rule is:

knot multiplicity + number of smoothness conditions = order

To illustrate this last point, the figure below shows four cubic B-splines and, below them, their first two derivatives. Each spline has a certain knot of multiplicity 1, 2, 3, 4, as indicated by the lengths of the knot lines.

```
d2 = -1;
t1 = [7 7.9 9.2 10  11]-7;
t2 = [7 7.9 7.9 9   10]-2;
t3 = [7 7.9 7.9 7.9 9]+2;
t4 = [7 7.9 7.9 7.9 7.9]+5;
[m,tt] = knt2mlt([t1 t2 t3 t4]);
```

```
ext = tt([1 end])+[-.5 .5];
plot(ext,[v v],cl(5))
hold on
plot(ext,[d1 d1],cl(5))
plot(ext,[d2 d2],cl(5))
ts = [tt;tt;NaN(size(tt))];
ty = .2*[-m-1;zeros(size(m));m];
plot(ts(:),ty(:)+v,cl(5))
plot(ts(:),ty(:)+d1,cl(5))
plot(ts(:),ty(:)+d2,cl(5))
b1 = spmak(t1,1);
p1 = fnplt(b1);
db1 = fnder(b1);
p11 = fnplt(db1);
p12 = [t1;fnval(fnder(db1),t1)];
plot(p1(1,:),p1(2,:)+v,cl(1),'LineWidth',lw)
plot(p11(1,:),s1*p11(2,:)+d1,cl(1),'LineWidth',lw)
plot(p12(1,:),s2*p12(2,:)+d2,cl(1),'LineWidth',lw)
text(-2,v,'B'), text(-2,d1,'DB'), text(-2,d2,'D^2B')
b1 = spmak(t2,1);
p1 = fnplt(b1);
db1 = fnder(b1);
p11 = fnplt(db1);
p12 = fnplt(fnder(db1),'j');
plot(p1(1,:),p1(2,:)+v,cl(2),'LineWidth',lw)
plot(p11(1,:),s1*p11(2,:)+d1,cl(2),'LineWidth',lw)
plot(p12(1,:),s2*s2*p12(2,:)+d2,cl(2),'LineWidth',lw)
b1 = spmak(t3,1);
p1 = fnplt(b1);
db1 = fnder(b1);
p11 = fnplt(db1,'j');
p12 = fnplt(fnder(db1),'j');
plot(p1(1,:),p1(2,:)+v,cl(3),'LineWidth',lw)
plot(p11(1,:),s1*s2*p11(2,:)+d1,cl(3),'LineWidth',lw)
plot(p12(1,:),s2*s2*p12(2,:)+d2,cl(3),'LineWidth',lw)
b1 = spmak(t4,1);
p1 = fnplt(b1);
db1 = fnder(b1);
p11 = fnplt(db1);
p12 = fnplt(fnder(db1));
plot(p1(1,:),p1(2,:)+v,cl(4),'LineWidth',lw)
plot(p11(1,:),s2*p11(2,:)+d1,cl(4),'LineWidth',lw)
plot(p12(1,:),s2*s2*p12(2,:)+d2,cl(4),'LineWidth',lw)
text(t2(2)-.5,tey,'2-fold','FontSize',12,'Color',cl(2))
text(t3(2)-.8,tey,'3-fold','FontSize',12,'Color',cl(3))
text(t4(3)-.8,tey,'4-fold','FontSize',12,'Color',cl(4))
axis([-1 14 -3 7.5])
title('Cubic B-splines With A Knot of Various Multiplicities')
axis off
hold off
```

Cubic B-splines With A Knot of Various Multiplicities
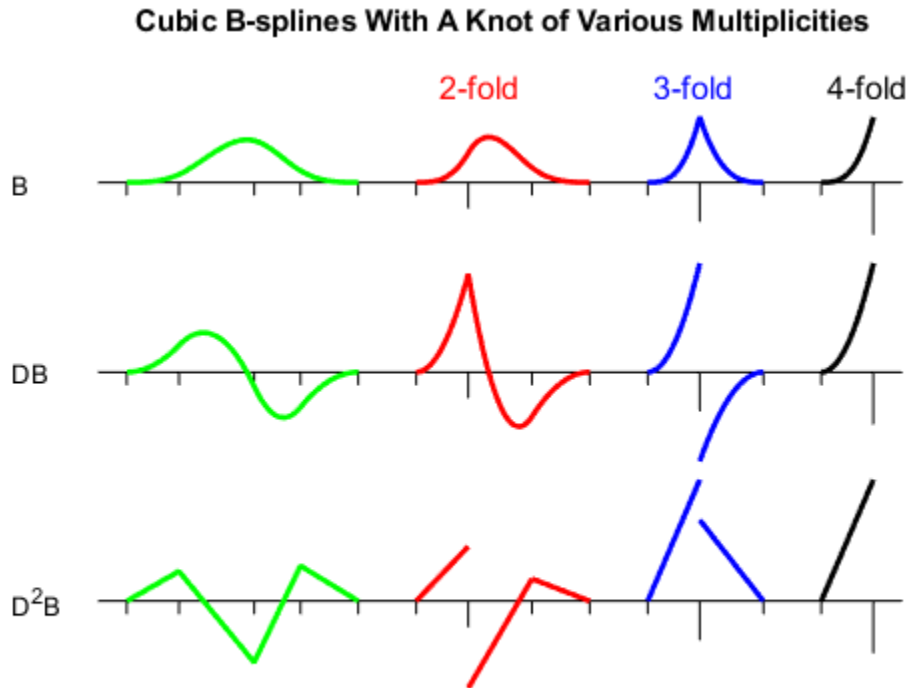
For example, since the order of a cubic B-spline is 4, a double knot means just 2 smoothness conditions, i.e., just continuity across that knot of the function and its first derivative.

Refinement and Knot Insertion

Any B-form can be *refined*, i.e., converted, by *knot insertion*, into the B-form for the same function, but for a finer knot sequence. The finer the knot sequence, the closer is the control polygon to the function being represented.

For example, this figure shows the original (in black) and refined (in red) control polygons for the cubic spline used earlier.

```
sp = savesp;
fnplt(sp,2.5);
hold on
c = fnbrk(sp,'c');
plot(aveknt(fnbrk(sp,'k'),4),c,':ok');
b = knt2brk(fnbrk(sp,'k'));
spref = fnrfn(sp,(b(2:end)+b(1:end-1))/2);
cr = fnbrk(spref,'c');
h2 = plot(aveknt(fnbrk(spref,'knots'),4),cr,':*r');
axis([-.72 7 -.5 3.5])
title('A Spline, its Control Polygon, and a Refined Control Polygon')
axis off
hold off
```

A Spline, its Control Polygon, and a Refined Control Polygon
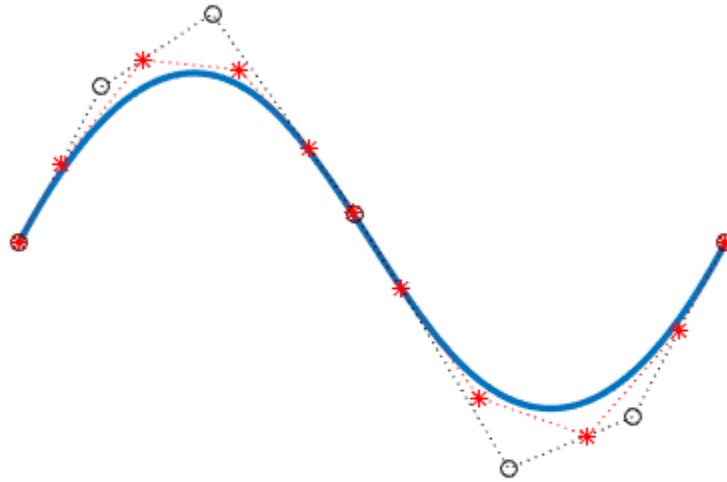
As a second example, we start with the vertices of the standard diamond as our control points, but run through the sequence twice.

```
ozmz = [1 0 -1 0];
c = [ozmz ozmz 1; 0 ozmz ozmz];
circle = spmak(-4:8,c);
fnplt(circle)
hold on
plot(c(1,:),c(2,:),':ok')
axis(1.1*[-1 1 -1 1])
axis equal, axis off
hold off
```
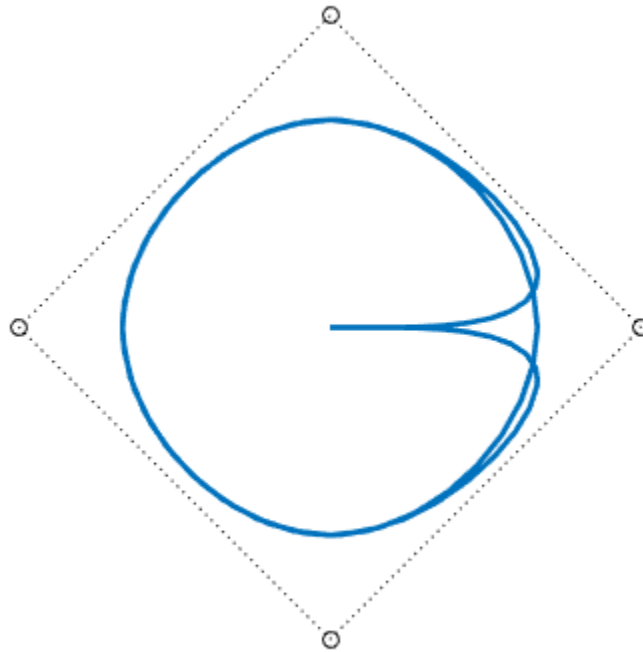
However, when we plot the resulting spline, we get a curve that begins and ends at the origin, due to the fact that we chose to make the knot sequence simple. Hence our spline vanishes at the endpoints of its basic interval, [-4 .. 8]. We really only want the part of the spline that corresponds to the interval [0 .. 4], plotted more boldly in the figure below.

```
fnplt(circle)
hold on
fnplt(circle,[0 4],4)
plot(c(1,:),c(2,:),':ok')
axis(1.1*[-1 1 -1 1])
title('A Circle as Part of a Spline Curve with a Diamond as Control Polygon')
axis equal, axis off
hold off
```

A Circle as Part of a Spline Curve with a Diamond as Control Polygon
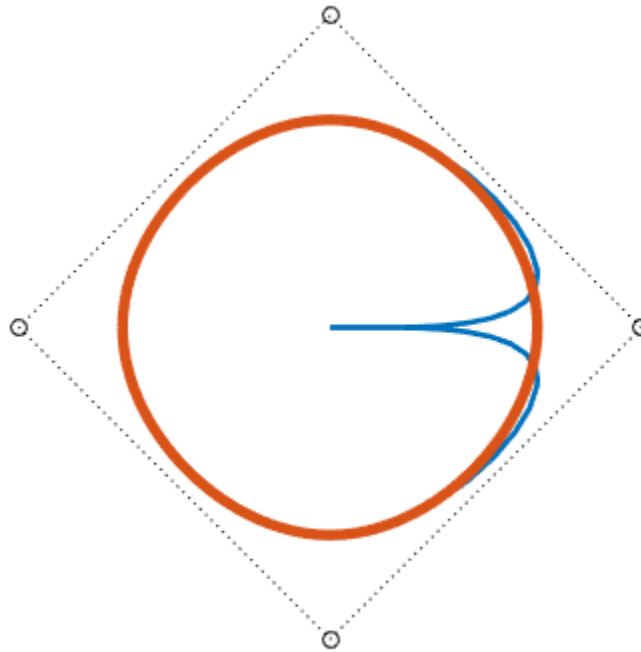
To get just the circle, we restrict our spline to the interval [0 .. 4]. We do this by converting to ppform, restricting to [0 .. 4], then converting to B-form.

```
circ = fn2fm(fnbrk(fn2fm(circle,'pp'),[0 4]),'B-');
fnplt(circ,2.5)
hold on
cc = fnbrk(circ,'c');
plot(cc(1,:),cc(2,:),':ok')
axis(1.1*[-1 1 -1 1])
axis equal, axis off
hold off
```
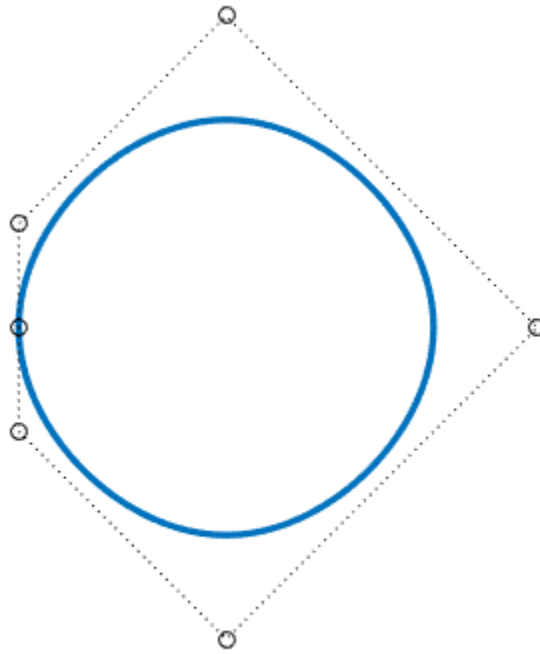
Refinement of the resulting knot sequence leads to a control polygon much closer to the circle.

```
ccc = fnbrk(fnrfn(circ,.5:4),'c');
hold on
plot(ccc(1,:),ccc(2,:),'-r*')
title('A Circle as a Spline Curve, its Control Polygon, and a Refinement')
hold off
```

A Circle as a Spline Curve, its Control Polygon, and a Refinement
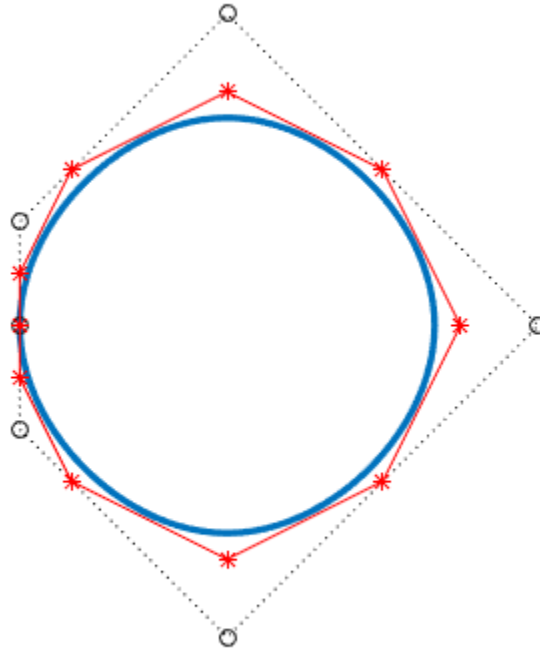
Multivariate Splines

A spline in Curve Fitting Toolbox can also be multivariate, namely, the tensor product of univariate splines. The B-form for such a function is only slightly more complicated, with the knots now a cell array containing the various univariate knot sequences, and the coefficient array suitably multidimensional.

For example, this random spline surface is cubic in the first variable (there are 11 knots and 7 coefficients in that variable), but only piecewise constant in the second variable ((2+5+2)-8 = 1).

```
fnplt( spmak({augknt(0:4,4),augknt(0:4,3)}, rand(7,8)) )
```
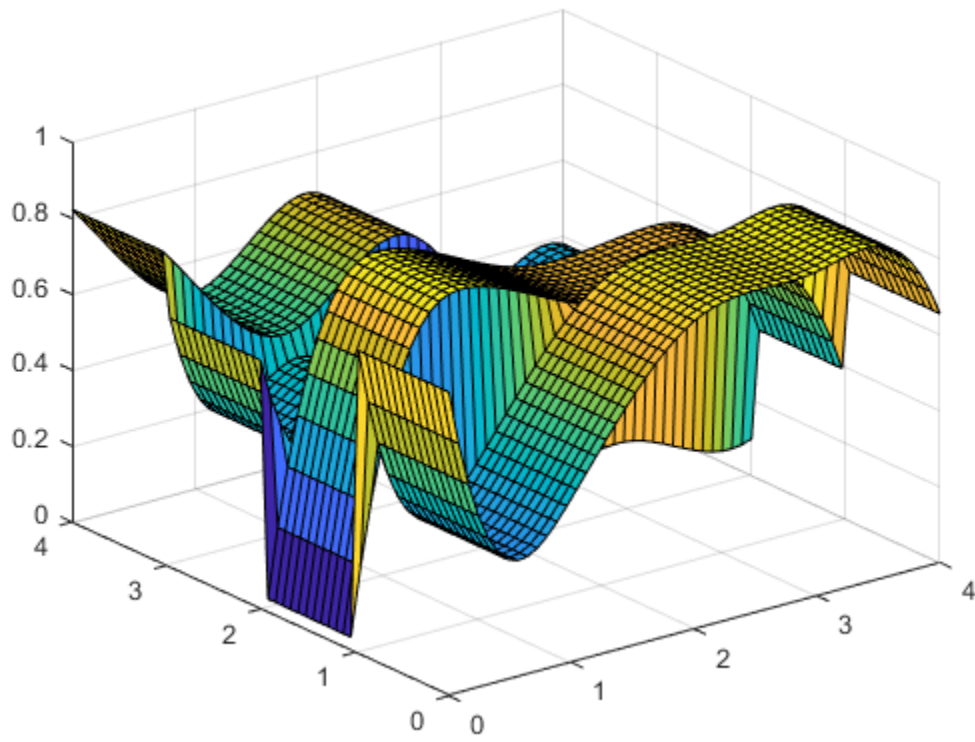
Construct and Work with the PPFORM

This example shows how to construct and work with the ppform of a spline in Curve Fitting Toolbox™.

Introduction

A (univariate) piecewise polynomial, or *pp* for short, is characterized by its *break sequence*, breaks say, and its *coefficient array*, coefs say, of the local power form of its polynomial pieces. The break sequence is assumed to be strictly increasing,

```
breaks(1) < breaks(2) < ... < breaks(l+1),
```

with l the number of polynomial pieces that make up the pp. In the figure below, breaks is [0,1,4,6], hence l is 3.

While these polynomials may be of varying degrees, they are all recorded as polynomials of the same *order* k, i.e., the coefficient array coefs is of size [l,k], with coefs(j,:) containing the k coefficients in the local power form for the j-th polynomial piece.

Here is an example of a pp made up of three quadratic polynomials, i.e., l = k = 3. The breaks are marked in red.

```
sp = spmak([0 1 4 4 6],[2 -1]);
pp = fn2fm(sp,'pp') ;
breaks = fnbrk(pp,'b');
coefs = fnbrk(pp,'c');
coefs(3,[1 2]) = [0 1];
pp = ppmak(breaks,coefs,1);
fnplt(pp,[breaks(1)-1 breaks(2)],'g',1.8)
hold on
fnplt(pp, breaks([2 3]),'b',1.8)
fnplt(pp,[breaks(3),breaks(4)+1],'m',1.8)
lp1 = length(breaks);
xb = repmat(breaks,3,1);
yb = repmat([2;-2.2;NaN],1,lp1);
plot(xb(:),yb(:),'r')
axis([-1 7 -2.5 2.3])
hold off
```
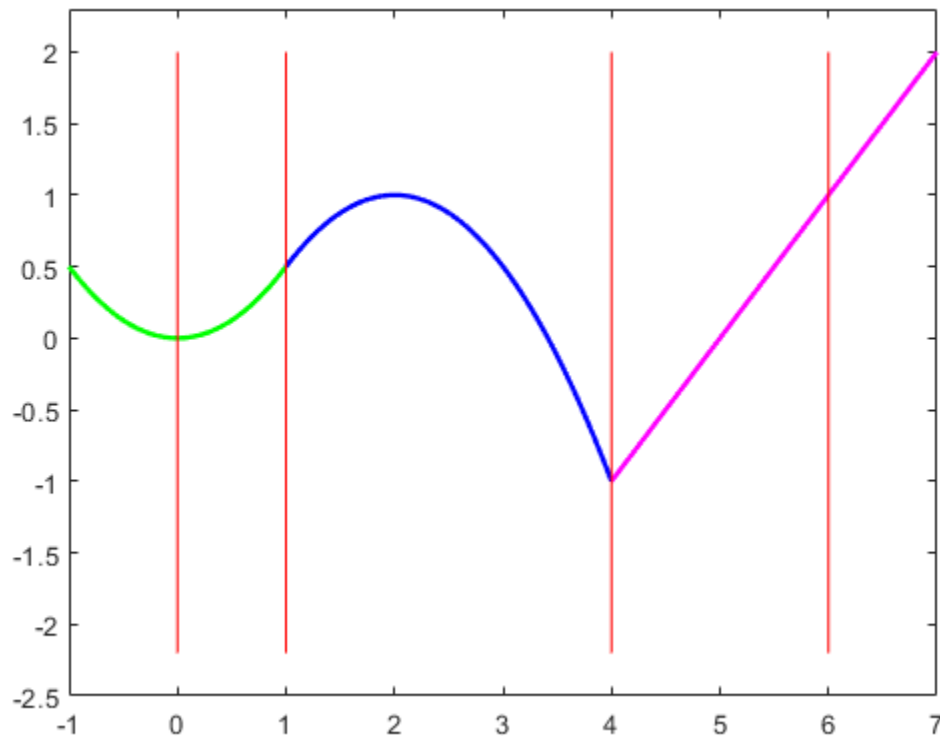
The precise description of the pp in terms of the break sequence `breaks` and the coefficient array `coefs` is

```
pp(t) = polyval(coefs(j,:), t-breaks(j))
```

$$\text{for breaks(j)} \ <= \ t \ < \ \text{breaks(j+1)}$$

where, to recall,

```
polyval(a,x) = a(1)*x^(k-1) + a(2)*x^(k-2) + ... + a(k)*x^0.
```

For the pp in the figure above, `breaks(1)` is 0, and `coefs(1,:)` is [-1/2 0 0], while `breaks(3)` is 4, and `coefs(3,:)` is [0 1 -1]. For `t` not in [`breaks(1)` .. `breaks(l+1)`], `pp(t)` is defined by extending the first or last polynomial piece.

A pp is usually constructed through a process of interpolation or approximation. But it is also possible to make one up in ppform from scratch, using the command `ppmak`. For example, the pp above can be obtained as

```
breaks = [0 1 4 6];
coefs = [1/2 0 0 -1/2 1 1/2 0 1 -1];
fn = ppmak(breaks,coefs)
```

```
fn =

  struct with fields:
```

```
    form: 'pp'
  breaks: [0 1 4 6]
   coefs: [3x3 double]
  pieces: 3
   order: 3
     dim: 1
```

This returns, in `fn`, a complete description of this pp function in the so-called ppform.

Various commands in Curve Fitting Toolbox can operate on this form. The remaining sections show some examples.

Operating on Piecewise Polynomials

To evaluate a pp, use the `fnval` command.

```
fnval(fn, -1:7)


ans =

  Columns 1 through 7

    0.5000         0    0.5000    1.0000    0.5000   -1.0000         0

  Columns 8 through 9

    1.0000    2.0000
```

To differentiate a pp, use the `fnder` command.

```
dfn = fnder ( fn );
hold on
fnplt(dfn, 'jumps','y', 3)
hold off
h1 = findobj(gca,'Color','y');
legend(h1,{'First Derivative'},'location','SW')
```
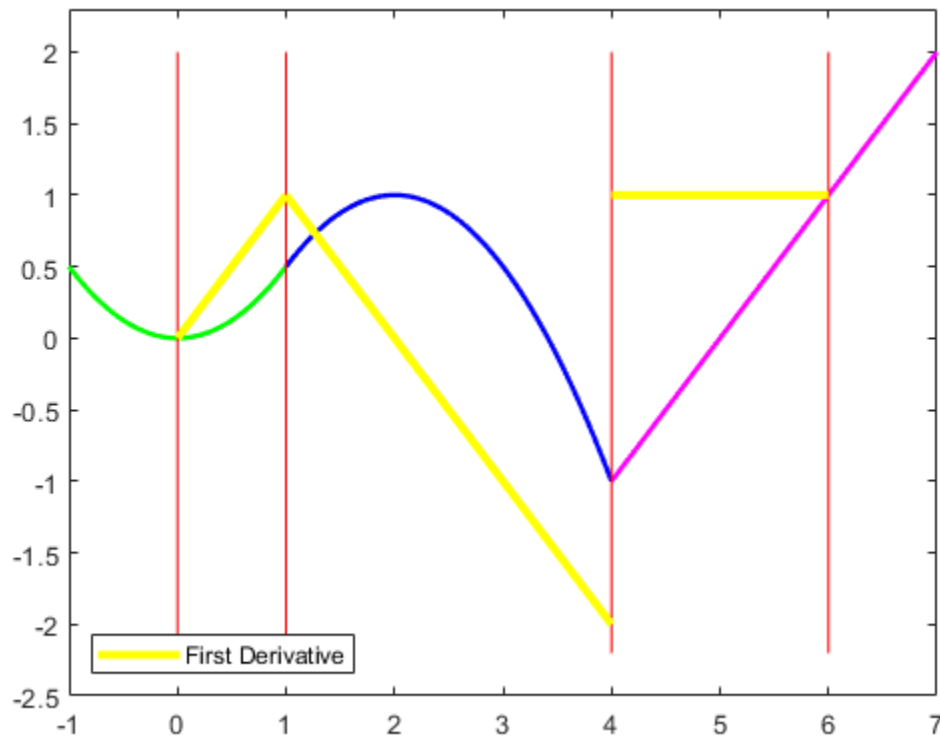
Note that the derivative of the example pp is continuous at 1 but discontinuous at 4. Also note that, by default, `fnplt` plots a ppform on its *basic interval*, i.e., on the interval `[breaks(1) .. breaks(end)]`.

You can also use `fnder` to take the second derivative of a pp.

```
ddfn = fnder(fn, 2);
hold on
fnplt( ddfn ,'j', 'k', 1.6)
hold off
h2 = findobj(gca,'Color','k');
legend([h1 h2],{'First Derivative' 'Second Derivative'},'location','SW')
```
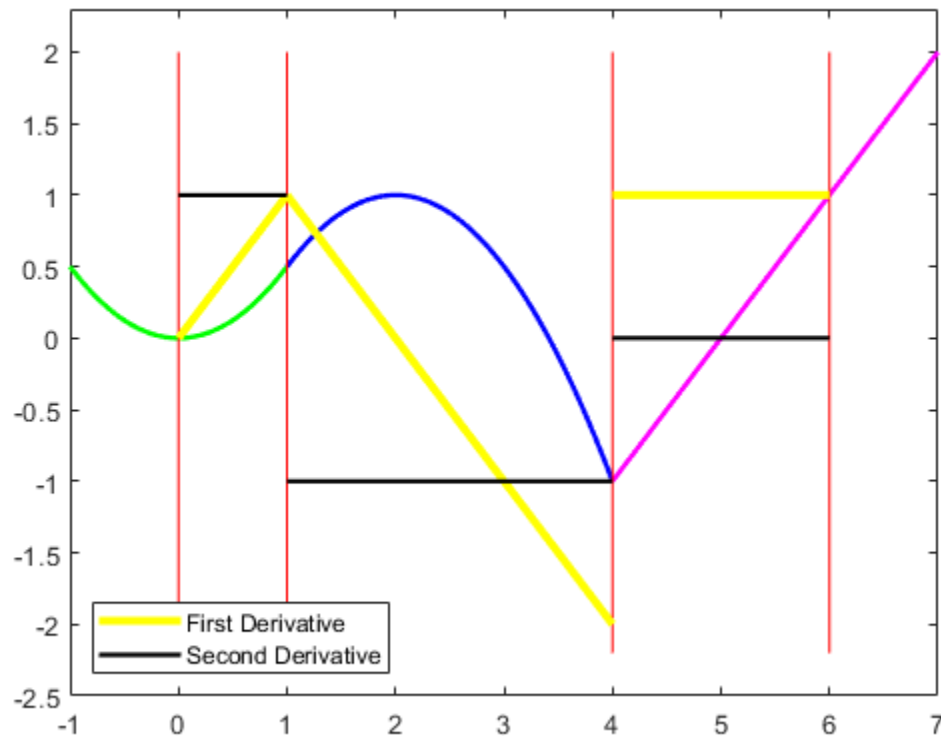
The second derivative is piecewise constant.

Note that differentiation via `fnder` is done separately for each polynomial piece. For example, although the first derivative has a jump discontinuity across 4, the second derivative is not infinite there. This has consequences when we integrate the second derivative.

To integrate a pp, use the `fnint` command.

```
iddfn = fnint(ddfn);
hold on
fnplt(iddfn, 'b', .5)
hold off
h3 = findobj(gca,'Color','b', 'LineWidth',.5);
legend([h1 h2 h3],{'First Derivative' 'Second Derivative' ...
                    'Integral of Second Derivative'},'location','SW')
```
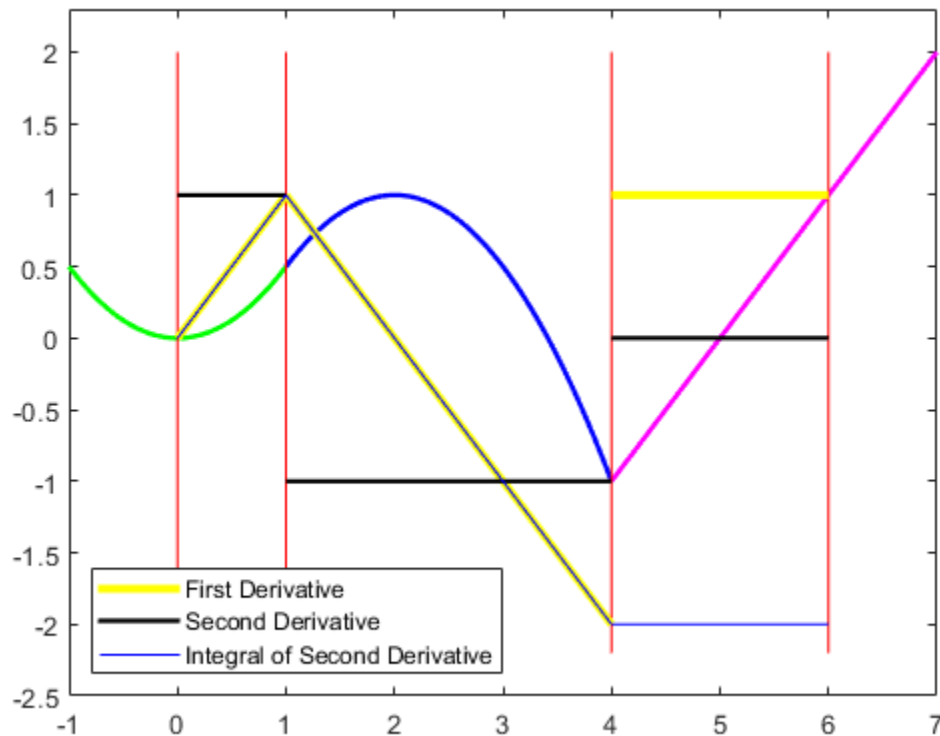
Note that integration of the second derivative does recover the first derivative, except for the jump across 4, which cannot be recovered, since the integral of any pp function is continuous.

You can obtain parts with the aid of the command `fnbrk`. For example

```
breaks = fnbrk(fn, 'breaks')
```

```
breaks =

     0    1    4    6
```

recovers the break sequence of the pp in `fn`, while

```
piece2 = fnbrk(fn, 2);
```

recovers the second polynomial piece, as this plot confirms.

```
fnplt(pp,[breaks(1)-1 breaks(2)],'g',1.8)
hold on
fnplt(piece2, 'b', 2.5, breaks([2 3])+[-1 .5])
fnplt(pp,[breaks(3),breaks(4)+1],'m',1.8)
plot(xb(:),yb(:),'r')
title('The Polynomial that Supplies the Second Polynomial Piece')
hold off
axis([-1 7 -2.5 2.3])
```

The Polynomial that Supplies the Second Polynomial Piece
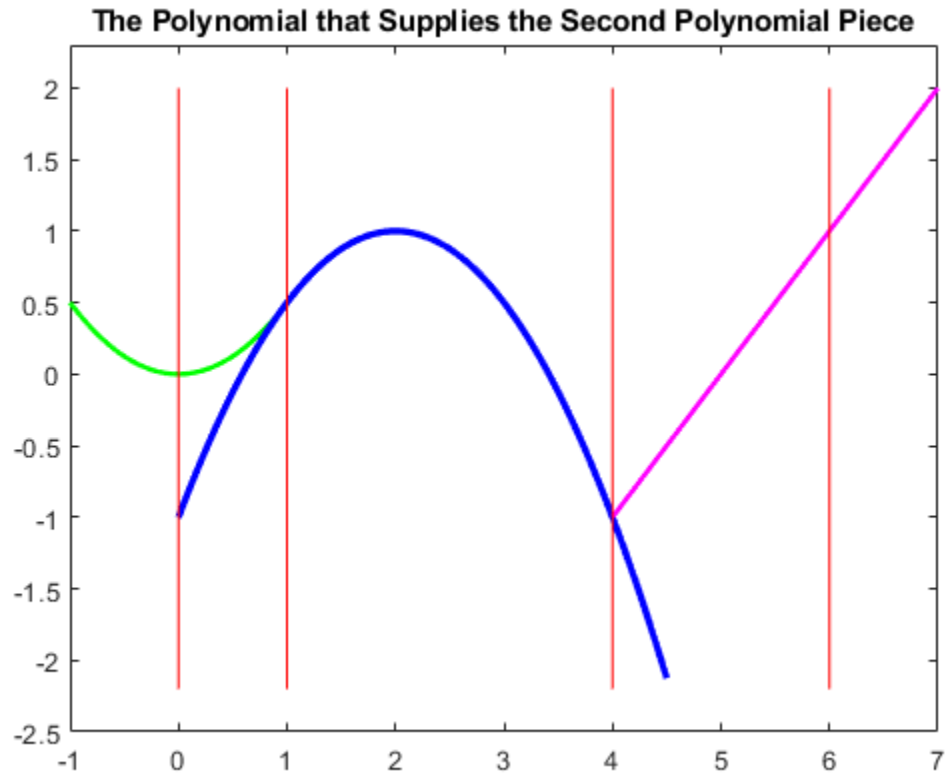
Vector-Valued Piecewise Polynomials

A pp can also be vector-valued, to describe a curve, in 2-space or 3-space. In that case, each local polynomial coefficient is a vector rather than a number, but nothing else about the ppform changes. There is one additional part of the ppform to record this, the *dimension* of its target.

For example, here is a 2-vector-valued pp describing the unit square, as its plot shows. It is a 2D-curve, hence its dimension is 2.

```
square = ppmak(0:4, [1 0  0 1  -1 1  0 0 ; 0 0  1 0  0 1  -1 1]);
fnplt(square,'r',2)
axis([-.5 1.5 -.5 1.5])
axis equal
title('A Vector-Valued PP that Describes a Square')
```

A Vector-Valued PP that Describes a Square
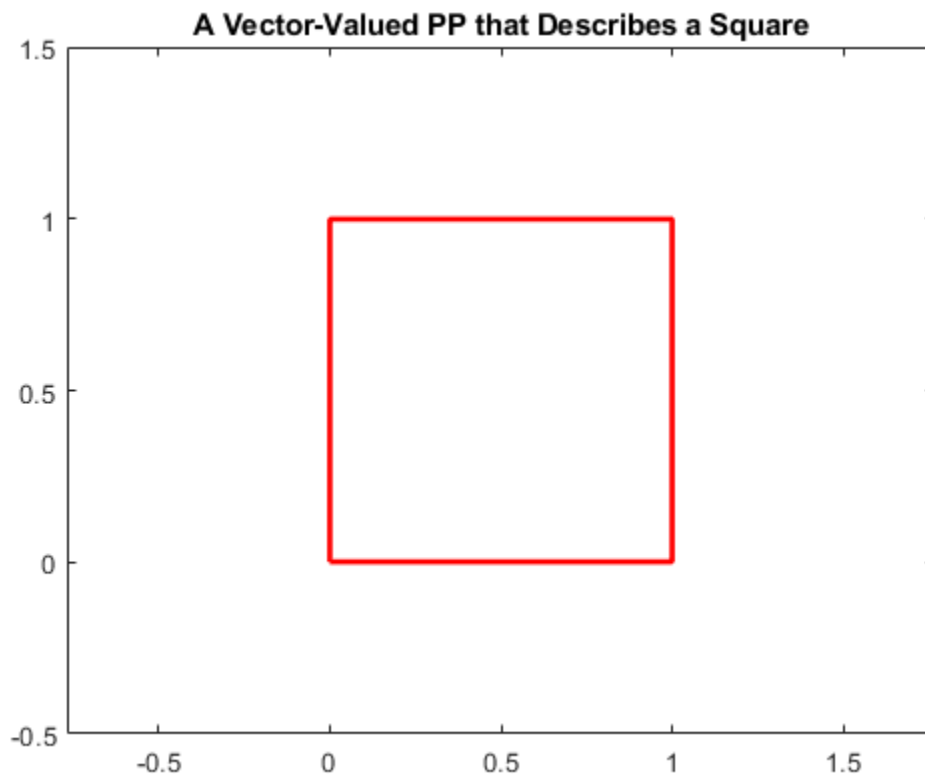
Multivariate Piecewise Polynomials

A pp in Curve Fitting Toolbox can also be multivariate, namely, a tensor product of univariate pp functions. The ppform of such a multivariate pp is only slightly more complicated, with `breaks` now a cell array containing the break sequence for each variable, and `coefs` now a multidimensional array. It is much harder to make up a non-random such function from scratch, so we won't try that here, particularly since the toolbox is meant to help with the construction of pp functions from some conditions about them. For example, the sphere in this figure is constructed with the aid of `csape`.

```
x = 0:4;
y = -2:2;
s2 = 1/sqrt(2);
v = zeros(3,7,5);
v(3,:,:) = [0 1 s2 0 -s2 -1 0].'*[1 1 1 1 1];
v(2,:,:) = [1 0 s2 1 s2 0 -1].'*[0 1 0 -1 0];
v(1,:,:) = [1 0 s2 1 s2 0 -1].'*[1 0 -1 0 1];
sph = csape({x,y},v,{'clamped','periodic'});
fnplt(sph)
axis equal
axis off
title('A Sphere Described by a Bicubic 3-Vector-Valued Spline')
```

A Sphere Described by a Bicubic 3-Vector-Valued Spline
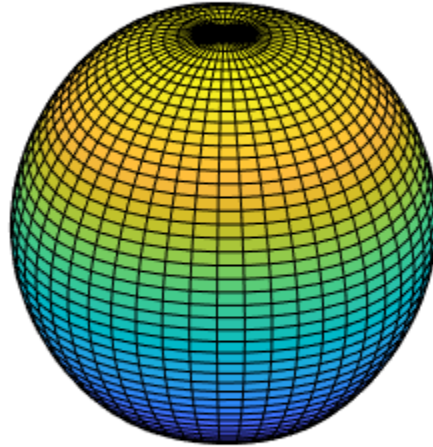
While the ppform of a spline is efficient for *evaluation*, the *construction* of a spline from some data is usually handled more efficiently by first determining its *B-form*, i.e., its representation as a linear combination of B-splines. For more information, see "Construct and Work with the B-form" on page 12-40.

How to Choose Knots

This example shows how to select and optimize knots using the `optknt` and `newknt` commands from Curve Fitting Toolbox™.

Sample Data

Here are some sample data, much used for testing spline approximation with variable knots, the so-called Titanium Heat Data. They record some property of titanium measured as a function of temperature.

```
[xx,yy] = titanium;
plot(xx,yy,'x');
axis([500 1100 .55 2.25]);
title('The Titanium Heat Data');
hold on
```
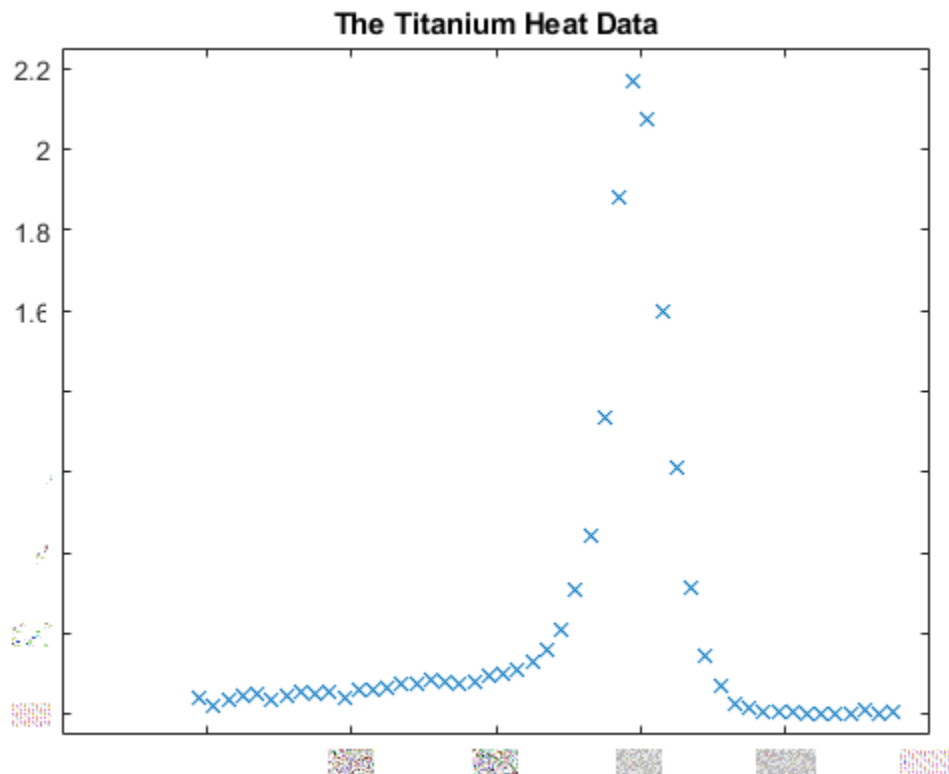

Notice the rather sharp peak. We'll use these data to illustrate some methods for knot selection.

First, we pick a few data points from these somewhat rough data. We will interpolate using this subset, then compare results to the full dataset.

```
pick = [1 5 11 21 27 29 31 33 35 40 45 49];
tau = xx(pick);
y = yy(pick);
plot(tau,y,'ro');
legend({'Full Dataset' 'Subsampled Data'}, 'location','NW');
```
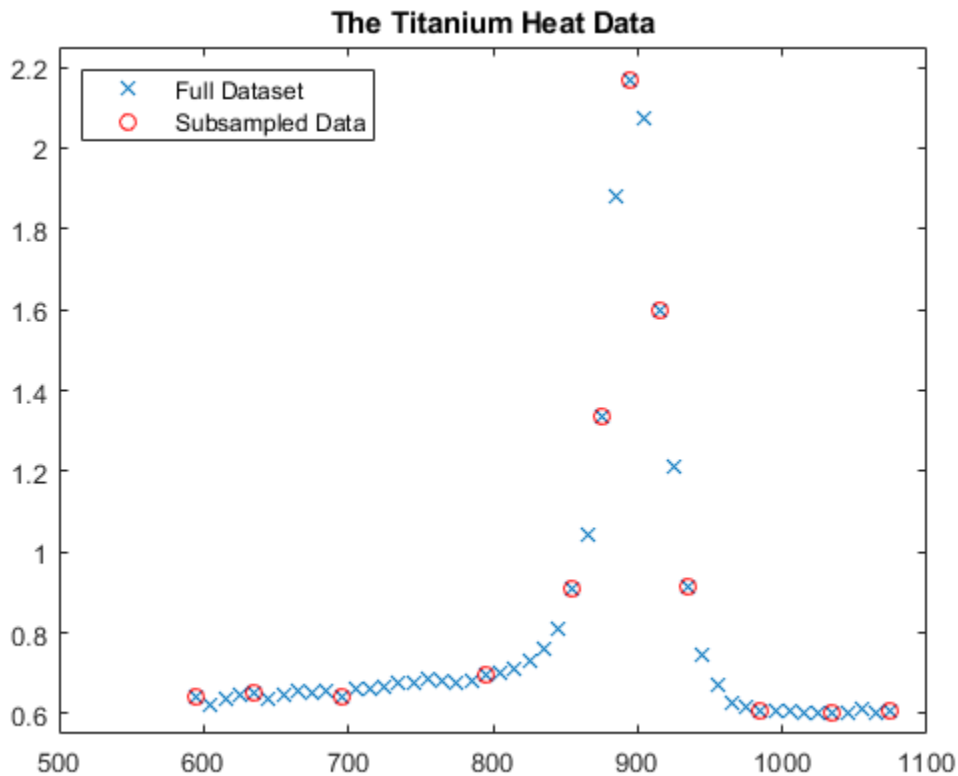
General Considerations

A spline of order `k` with `n+k` knots has `n` degrees of freedom. Since we have 12 data sites, `tau(1) < ... < tau(12)`, a fit with a cubic spline, i.e., a fourth order spline, requires a knot sequence `t` of length 12+4.

Moreover, the knot sequence `t` must satisfy the *Schoenberg-Whitney conditions*, i.e., must be such that the i-th data site lies in the support of the i-th B-spline, i.e.,

```
t(i) < tau(i) < t(i+k) for all i,
```

with equality allowed only in case of a knot of multiplicity `k`.

One way to choose a knot sequence satisfying all these conditions is as the optimal knots, of Gaffney/ Powell and Micchelli/Rivlin/Winograd.

Optimal Knots

In *optimal* spline interpolation, to values at sites

```
tau(1), ..., tau(n)
```

say, the knots are chosen so as to minimize the constant in a standard error formula. Specifically, the first and the last data site are chosen as k-fold knots. The remaining `n-k` knots are supplied by `optknt`.

Here is the beginning of the help from `optknt`:

OPTKNT Optimal knot distribution.

OPTKNT(TAU,K) returns an `optimal' knot sequence for

interpolation at data sites TAU(1), ..., TAU(n) by splines of

order K. TAU must be an increasing sequence, but this is not

checked.

OPTKNT(TAU,K,MAXITER) specifies the number MAXITER of iterations

to be tried, the default being 10.

The interior knots of this knot sequence are the n-K

sign-changes in any absolutely constant function h ~= 0 that

satisfies

integral{ f(x)h(x) : TAU(1) < x < TAU(n) } = 0

for all splines f of order K with knot sequence TAU.

Trying OPTKNT

We try using optknt for interpolation on our example, interpolating by cubic splines to data

(tau(i), y(i)), for i = 1, ..., n.

```
k = 4;
osp = spapi( optknt(tau,k), tau,y);

fnplt(osp,'r');
hl = legend({'Full Dataset' 'Subsampled Data' ...
        'Cubic Spline Interpolant Using Optimal knots'}, ...
        'location','NW');
hl.Position = hl.Position-[.14,0,0,0];
```

The Titanium Heat Data
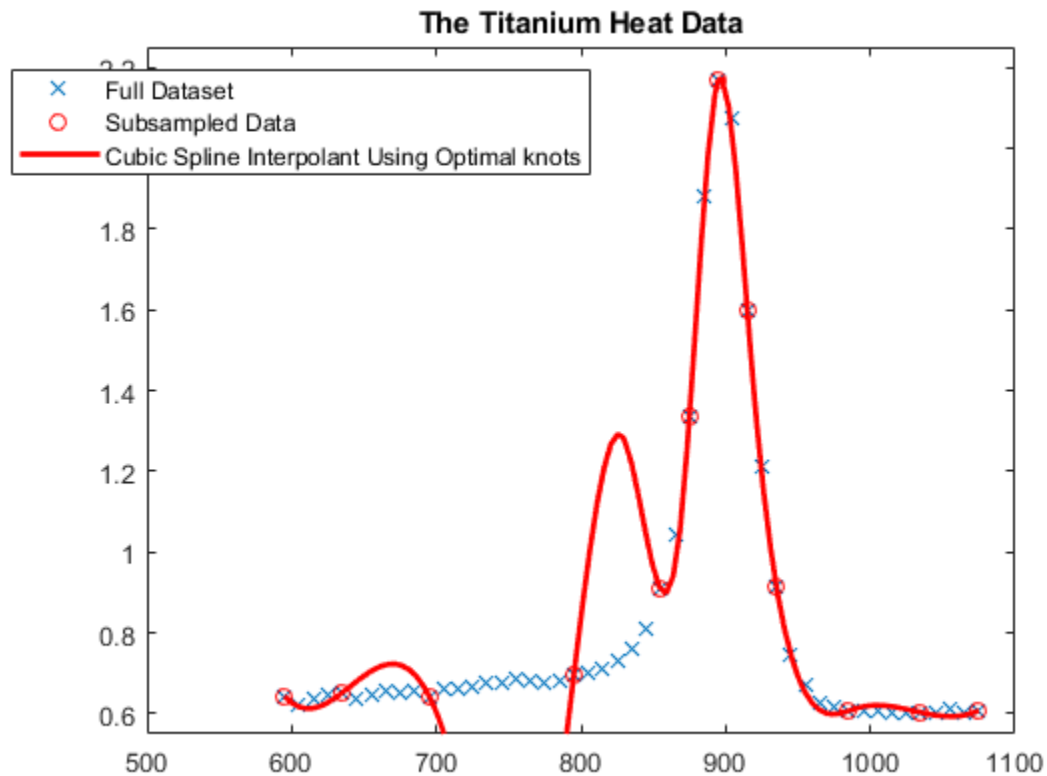
This is a bit disconcerting!

Here, marked by stars, are also the interior optimal knots:

```
xi = fnbrk(osp,'knots');
xi([1:k end+1-(1:k)]) = [];
plot(xi,repmat(1.4, size(xi)),'*');
hl = legend({'Full Dataset' 'Subsampled Data' ...
        'Cubic Spline Interpolant Using Optimal knots' ...
        'Optimal Knots'}, 'location','NW');
hl.Position = hl.Position-[.14,0,0,0];
```

The Titanium Heat Data
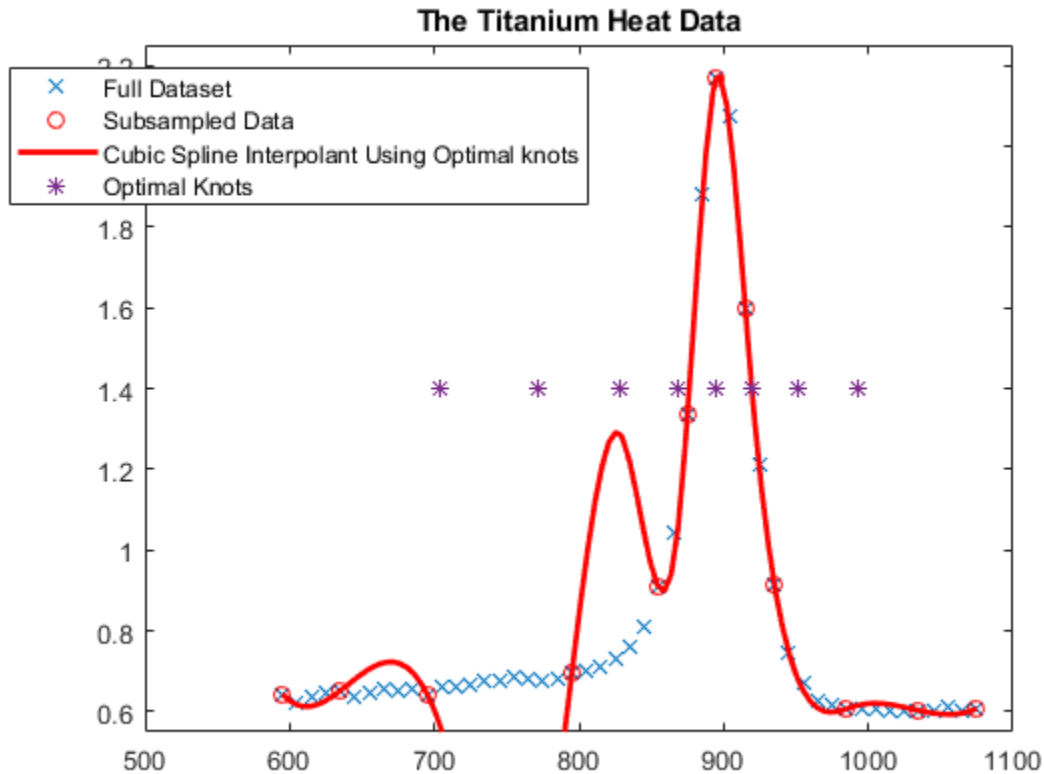
What Happened?

The knot choice for optimal interpolation is designed to make the maximum over *all* functions f of the ratio

```
norm(f - If) / norm(D^k f)
```

as small as possible, where the numerator is the norm of the interpolation error, f - If, and the denominator is the norm of the k-th derivative of the interpolant, D^k f. Since our data imply that D^k f is rather large, the interpolation error near the flat part of the data is of acceptable size for such an `optimal' scheme.

Actually, for these data, the ordinary cubic spline interpolant provided by csapi does quite well:

```
cs = csapi(tau,y);
fnplt(cs,'g',2);
hl = legend({'Full Dataset' 'Subsampled Data' ...
        'Cubic Spline Interpolant Using Optimal knots' ...
        'Optimal Knots' 'Cubic Spline Interpolant Using CSAPI'}, ...
        'location','NW');
hl.Position = hl.Position-[.14,0,0,0];
hold off
```

The Titanium Heat Data
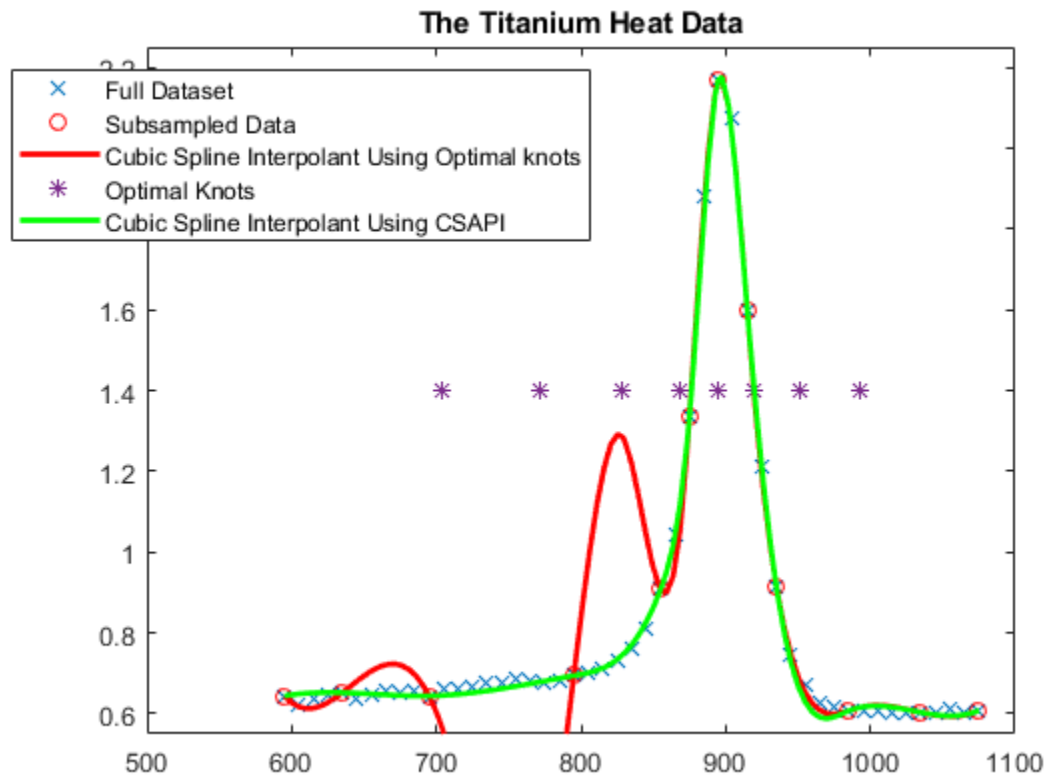
Knot Choice for Least Squares Approximation

Knots must be selected when doing least-squares approximation by splines. One approach is to use equally-spaced knots to begin with, then use `newknt` with the approximation obtained for a better knot distribution.

The next sections illustrate these steps with the full titanium heat data set.

Least Squares Approximation with Uniform Knot Sequence

We start with a uniform knot sequence.

```matlab
unif = linspace(xx(1), xx(end), 2+fix(length(xx)/4));
sp = spap2(augknt(unif, k), k, xx, yy);
plot(xx,yy,'x');
hold on
fnplt(sp,'r');
axis([500 1100 .55 2.25]);
title('The Titanium Heat Data');
hl = legend({'Full Dataset' ...
        'Least Squares Cubic Spline Using Uniform Knots'}, ...
        'location','NW');
hl.Position = hl.Position-[.14,0,0,0];
```

The Titanium Heat Data
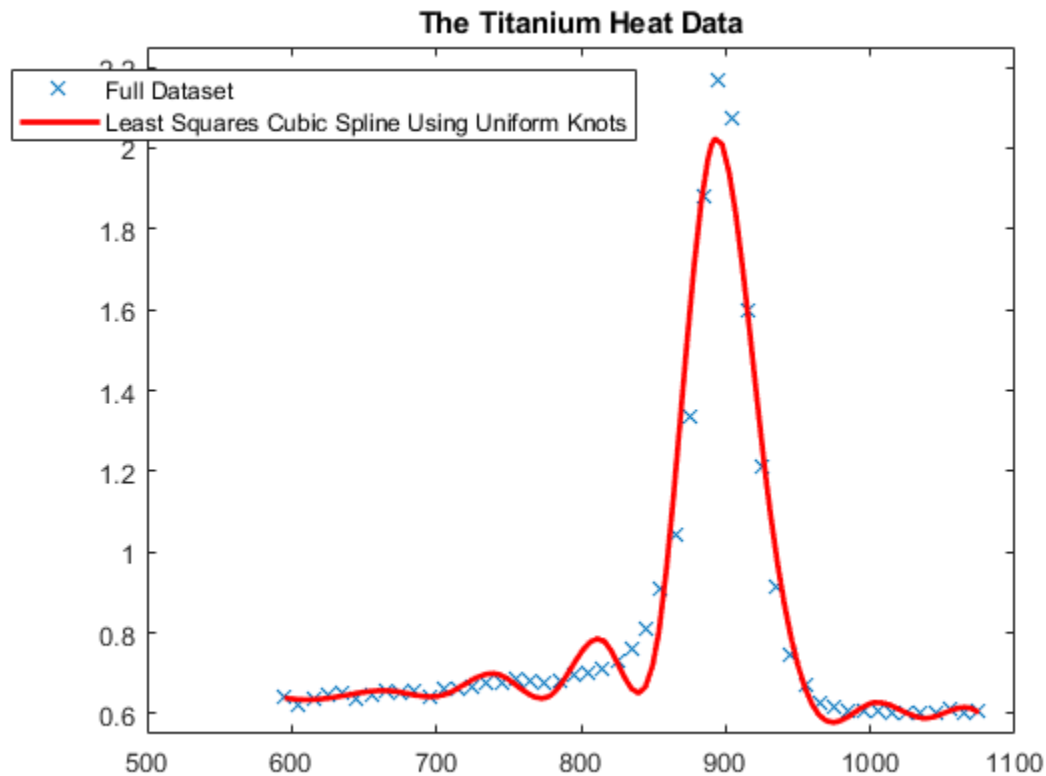
This is not at all satisfactory. So we use `newknt` for a spline approximation of the same order and with the same number of polynomial pieces, but the breaks better distributed.

Using NEWKNT to Improve the Knot Distribution

```
spgood = spap2(newknt(sp), k, xx,yy);
fnplt(spgood,'g',1.5);
hl = legend({'Full Dataset' ...
        'Least Squares Cubic Spline Using Uniform Knots' ...
        'Least Squares Cubic Spline Using NEWKNT'}, ...
        'location','NW');
hl.Position = hl.Position-[.14,0,0,0];
hold off
```

The Titanium Heat Data
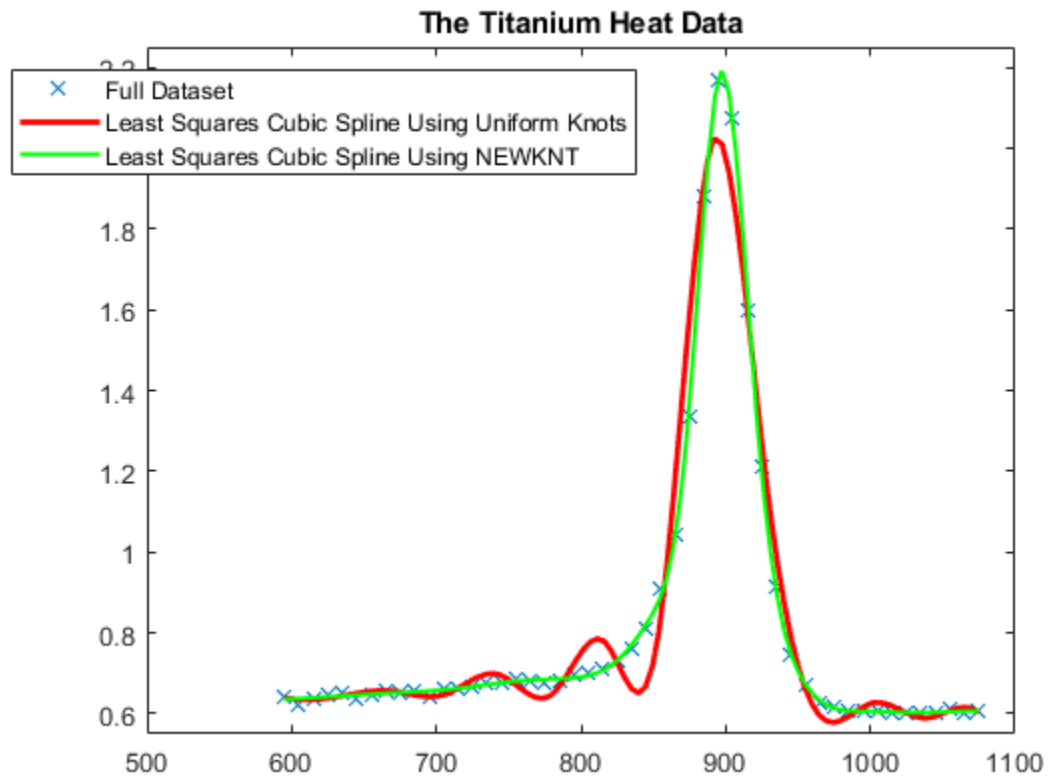
This is quite good. Incidentally, even one interior knot fewer would not have sufficed in this case.

Cubic Spline Interpolation

This example shows how to use the `csapi` and `csape` commands from Curve Fitting Toolbox™ to construct cubic spline interpolants.

The CSAPI Command

The command

```
values = csapi(x,y,xx)
```

returns the values at `xx` of the cubic spline interpolant to the given data (`x,y`), using the not-a-knot end condition. This interpolant is a piecewise cubic function, with break sequence `x`, whose cubic pieces join together to form a function with two continuous derivatives. The "not-a-knot" end condition means that, at the first and last interior break, even the third derivative is continuous (up to round-off error).

Specifying only two data points results in a straight line interpolant.

```
x = [0 1];
y = [2 0];
xx = linspace(0,6,121);
plot(xx,csapi(x,y,xx),'k-',x,y,'ro')
title('Interpolant to Two Points')
```
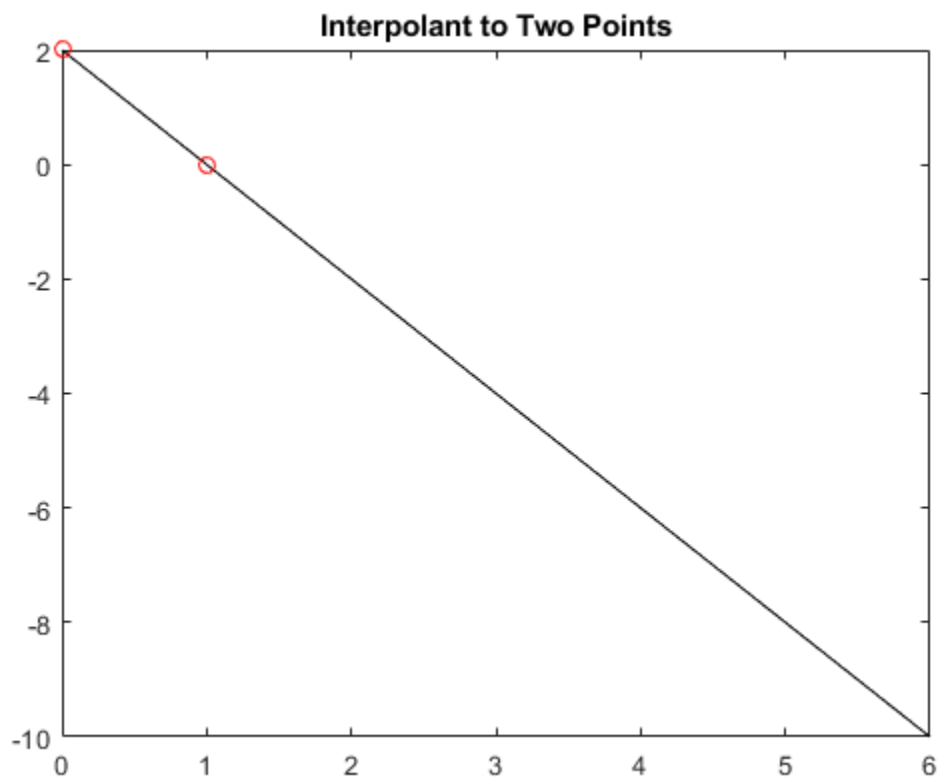

Specifying three data points gives a parabola.

```
x = [2 3 5];
y = [1 0 4];
plot(xx,csapi(x,y,xx),'k-',x,y,'ro')
title('Interpolant to Three Points')
```


Interpolant to Three Points
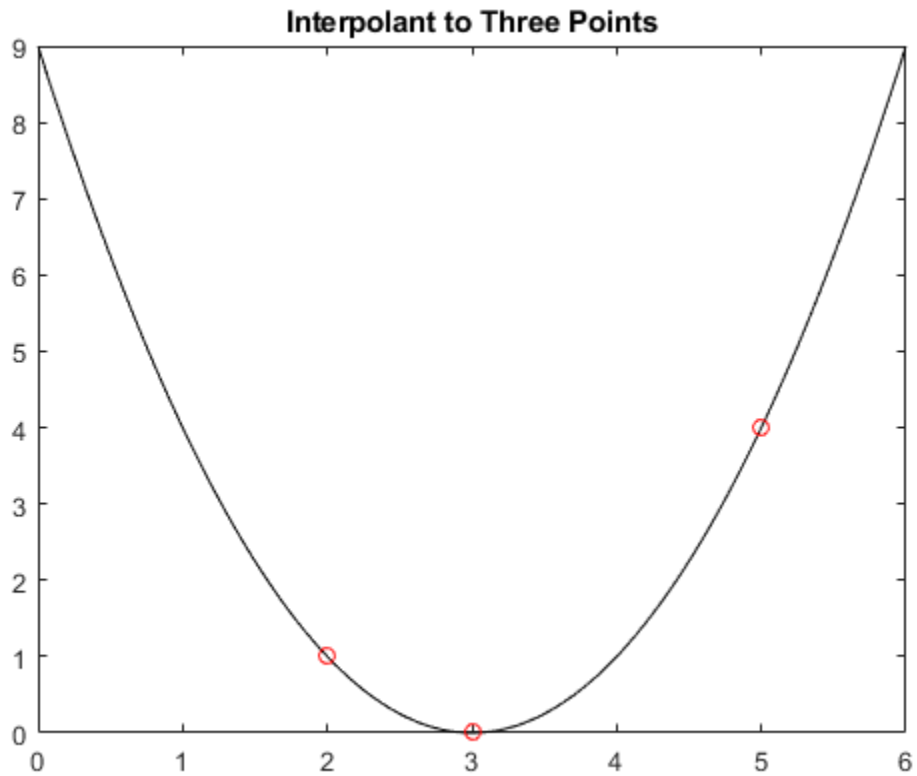
More generally, four or more data points give a cubic spline.

```
x = [1 1.5 2 4.1 5];
y = [1 -1 1 -1 1];
plot(xx,csapi(x,y,xx),'k-',x,y,'ro')
title('Cubic Spline Interpolant to Five Points')
```

Cubic Spline Interpolant to Five Points
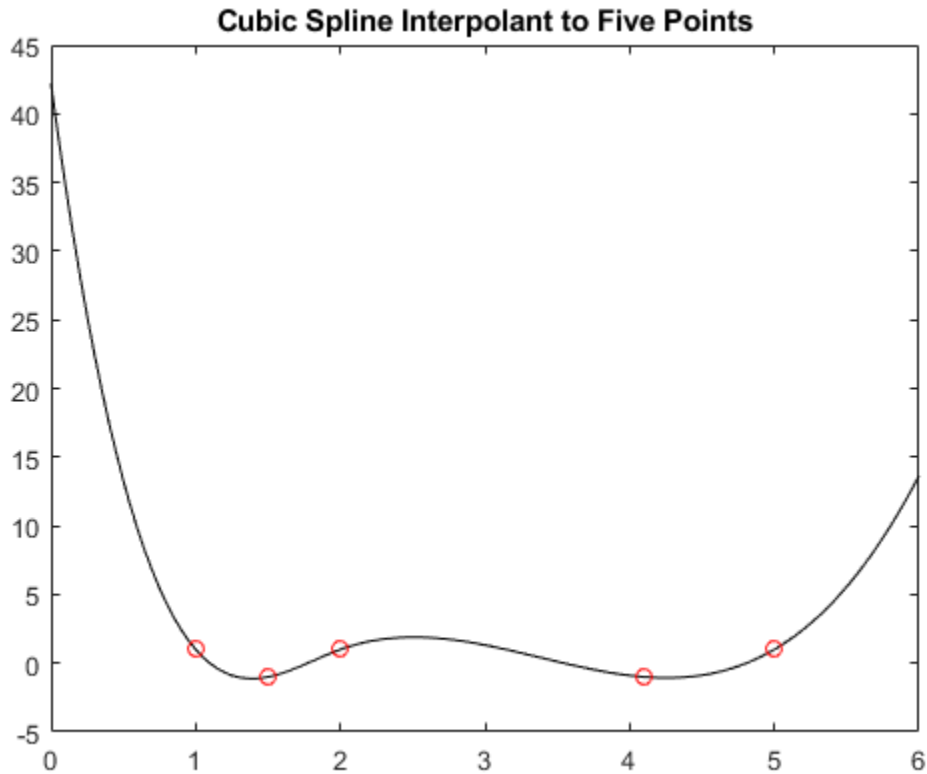
How to Check the Output of CSAPI

These look like nice interpolants, but how do we check that `csapi` performs as advertised?

We already saw that `csapi` interpolates, because we plotted the data points and the interpolant went right through those points. But to be sure that we get a cubic spline, it is best to start with data from a cubic spline of the expected sort and check whether `csapi` *reproduces* that cubic spline, i.e., gives back that cubic spline from which the data were taken.

Example: The Truncated Power Function

One simple example of a cubic spline function to check against is the truncated third power, i.e., the function

$$f(x) = ((x - xi)_+)^3,$$

where `xi` is one of the breaks and the "+" subscript indicates the *truncation function*, provided by the command `subplus`:

```
help subplus
```

```
 SUBPLUS Positive part.

                                    x , if  x>=0
      y  = subplus(x) := (x)_{+}  =                  ,
                                    0 , if  x<=0
```

```
     returns the positive part of X. Used for computing truncated powers.
```

The truncated 3rd power is plotted below for the particular choice `xi = 2`. As expected, it is zero to the left of 2, and rises like (x-2)^3 to the right of 2.

```
plot(xx, subplus(xx-2).^3,'y','LineWidth',3)
axis([0,6,-10,70])
```
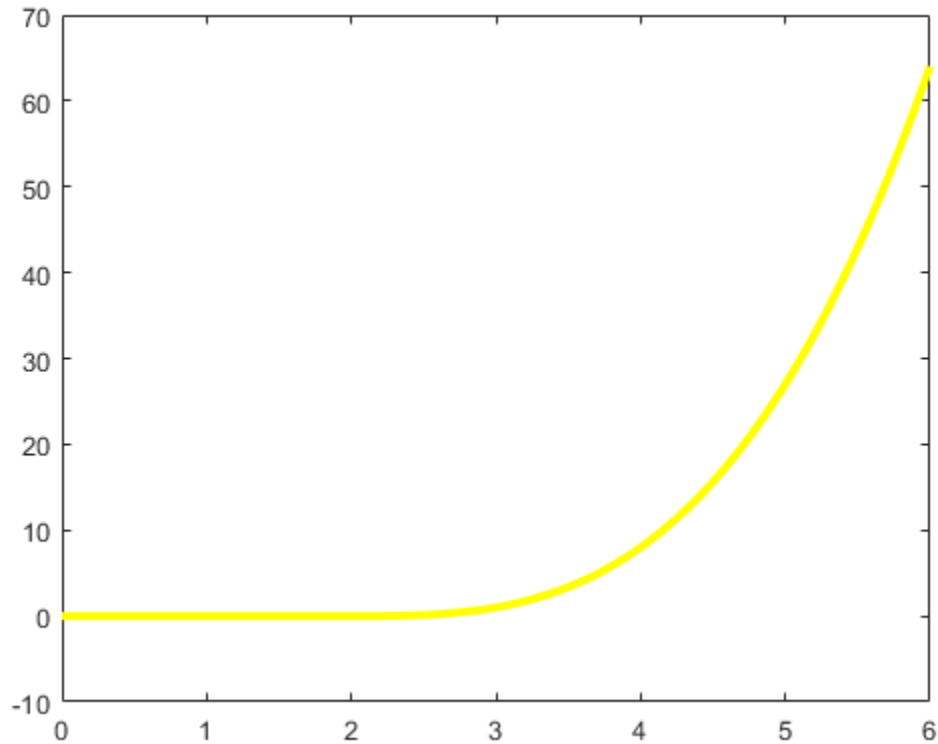

Now we interpolate this particular cubic spline at the data sites 0:6, and plot the interpolant on top of the spline, in black.

```
x = 0:6;
y = subplus(x-2).^3;
values = csapi(x,y,xx);
hold on
plot(xx,values,'k',x,y,'ro')
hold off
title('Interpolant to ((x-2)_+)^3')
```
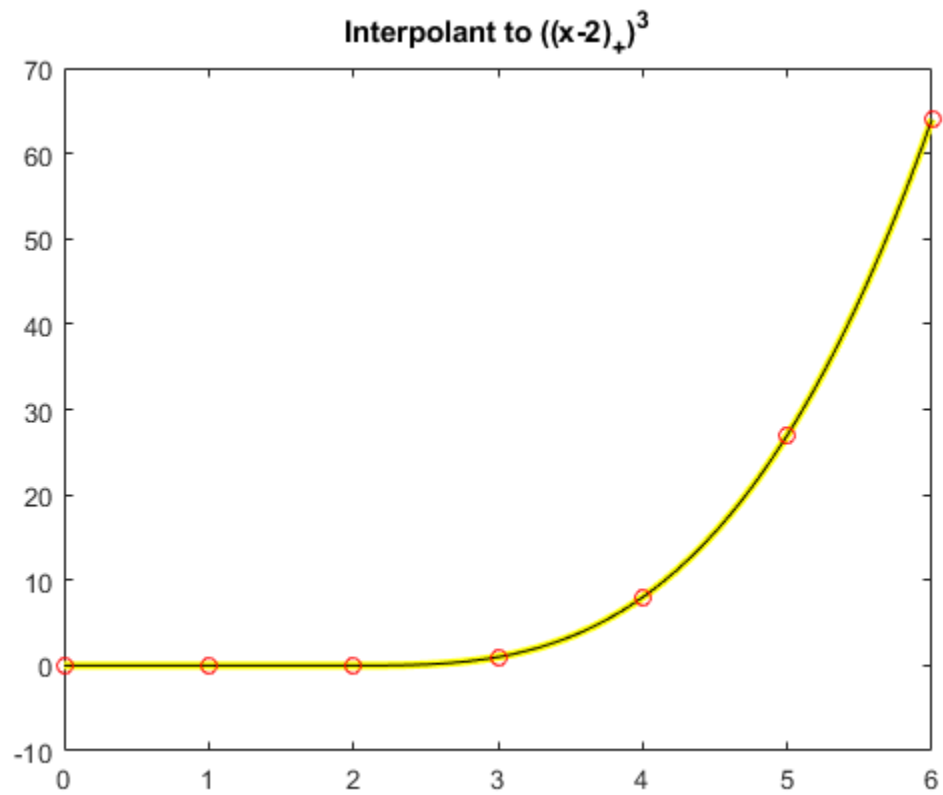
Interpolant to $((x-2)_+)^3$

The Interpolation Error

When comparing two functions, it is usually much more informative to plot their difference.

```
plot(xx, values - subplus(xx-2).^3)
title('Error in Cubic Spline Interpolation to ((x-2)_+)^3')
```

Error in Cubic Spline Interpolation to $((x-2)_+)^3$
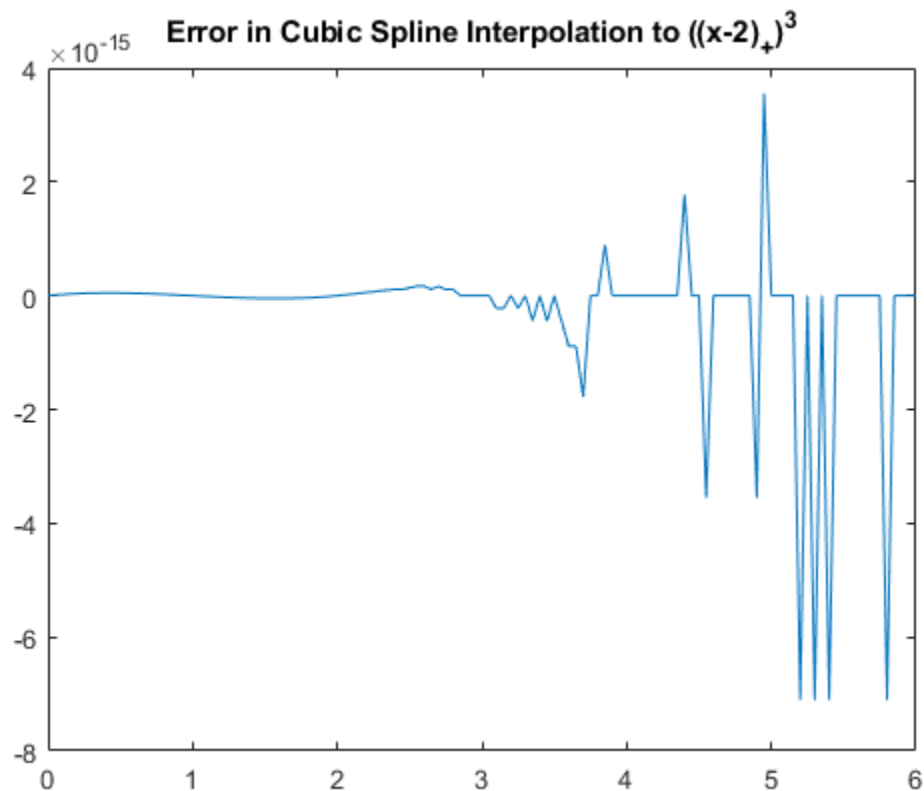
To put the size of their difference into context, you can also compute the maximum data value. This shows the error to be no worse than the inevitable round-off.

```
max_y = max(abs(y))
```

```
max_y = 64
```

A Truncated Power That Cannot be Reproduced

As a further test, we interpolate a truncated power whose `csapi`-produced interpolant at the sites 0:6 cannot coincide with it. For example, the first interior break of the interpolating spline is not really a knot since `csapi` uses the "not-a-knot" condition, hence the interpolant has three continuous derivatives at that site. This implies that we should not be able to reproduce the truncated 3rd power centered at that site since its third derivative is discontinuous across that site.

```
values = csapi(x,subplus(x-1).^3,xx);
plot(xx, values - subplus(xx-1).^3)
title('Error in Not-a-Knot Interpolant to ((x-1)_+)^3')
```

Error in Not-a-Knot Interpolant to $((x-1)_+)^3$
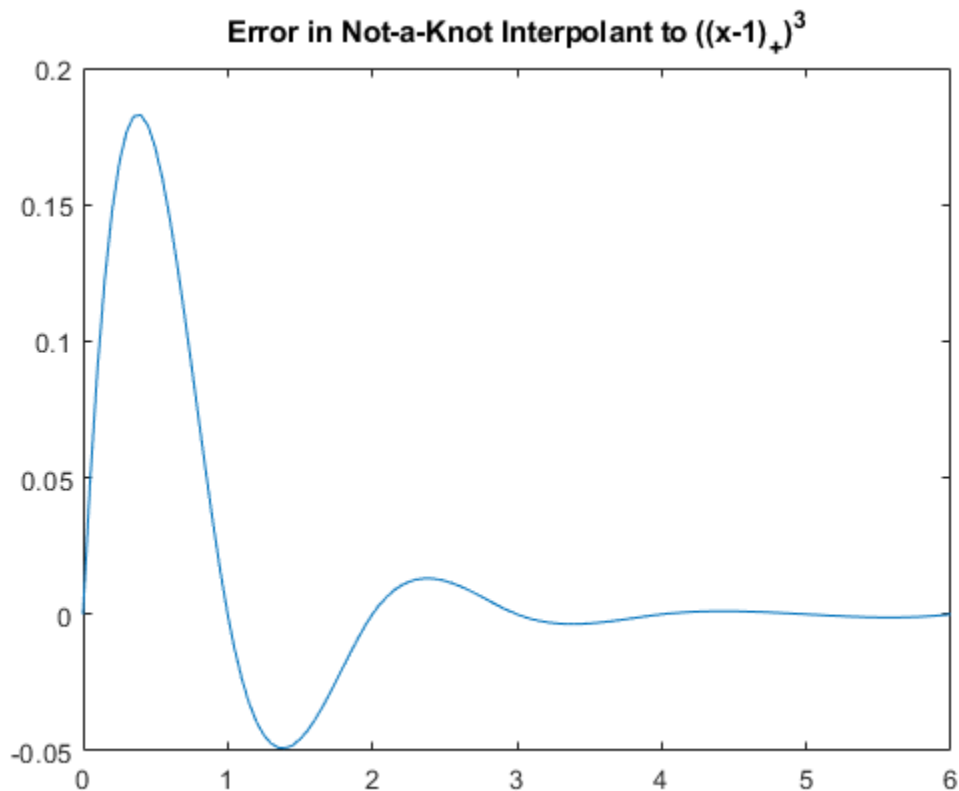
Since 1 is a first interior knot, it is not active for this interpolant.

The difference is as large as .18, but decays rapidly as we move away from 1. This illustrates that *cubic spline interpolation is essentially local*.

Using the ppform Instead of Values

It is possible to retain the interpolating cubic spline in a form suitable for subsequent evaluation, or for calculating its derivatives, or for other manipulations. This is done by calling `csapi` in the form

```
pp = csapi(x,y)
```

which returns the ppform of the interpolant. You can evaluate this form at some new points xx by the command

```
values = fnval(pp,xx)
```

You can differentiate the interpolant by the command

```
dpp = fnder(pp)
```

or integrate it by the command

```
ipp = fnint(pp)
```

which return the ppform of the derivative or the integral, respectively.

Example: Differentiating and Integrating the Interpolant

To show differentiation of an interpolant, we plot the derivative of this truncated power

$$f_2'(x) = 3((x-2)_+)^2,$$

(again in yellow) and then, on top of it, the derivative of our interpolant to the original truncated third power function (again in black).

```
plot(xx,3*subplus(xx-2).^2,'y','LineWidth',3)
pp = csapi(x,subplus(x-2).^3);
dpp = fnder(pp);
hold on
plot(xx,fnval(dpp,xx),'k')
hold off
title('Derivative of Interpolant to ((x-2)_+)^3')
```
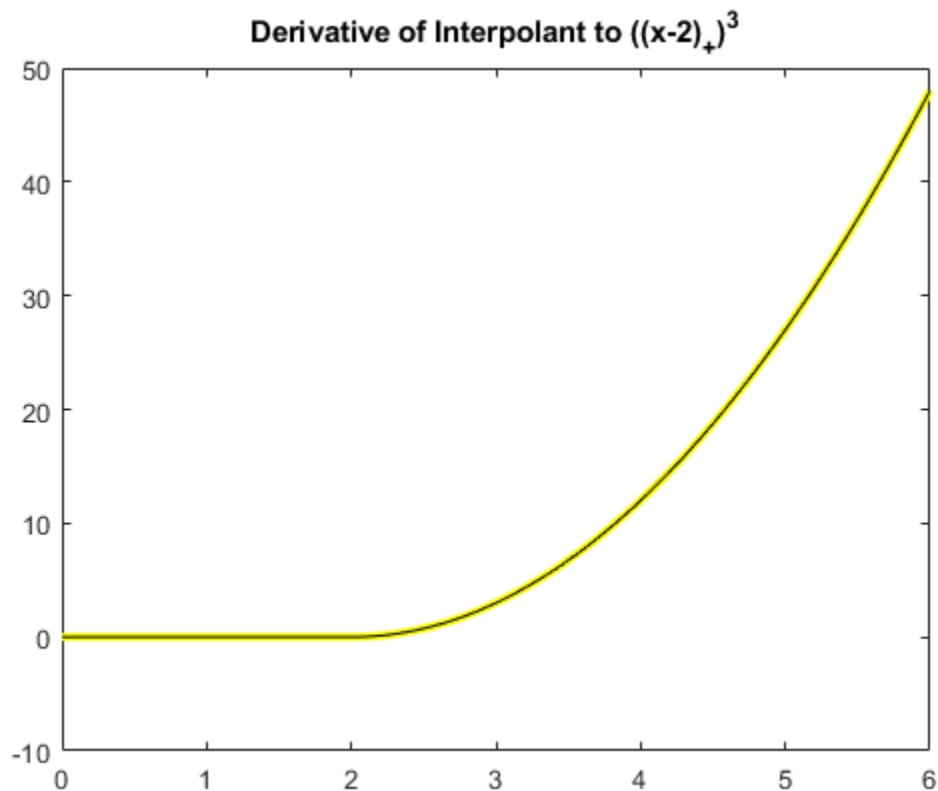

Again, the more informative comparison is to plot their difference, and as before this is no bigger than round-off.

```
plot(xx, fnval(dpp,xx) - 3*subplus(xx-2).^2)
title('Error in Derivative of interpolant to ((x-2)_+)^3')
```
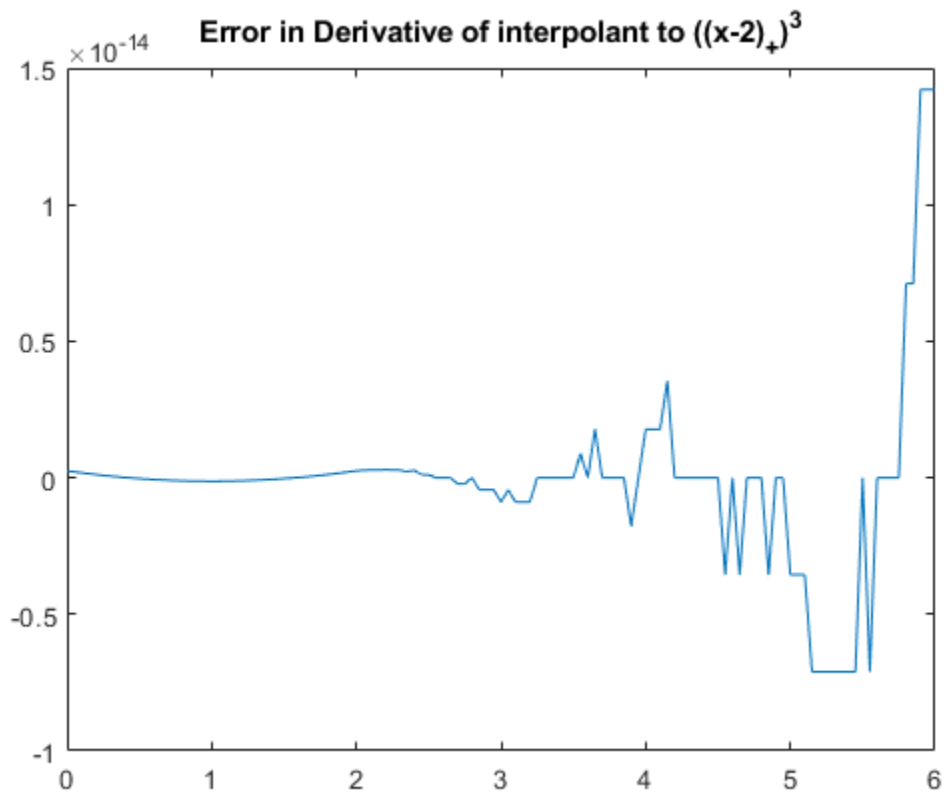
The second derivative of the truncated power is

$$f_2''(x) = 6(x - 2)_+$$

A plot of the difference between this function and the second derivative of the interpolant to the original function shows that there are now jumps, but they are still within roundoff.

```
ddpp = fnder(dpp);
plot(xx, fnval(ddpp,xx) - 6*subplus(xx-2))
title('Error in Second Derivative of Interpolant to ((x-2)_+)^3')
```
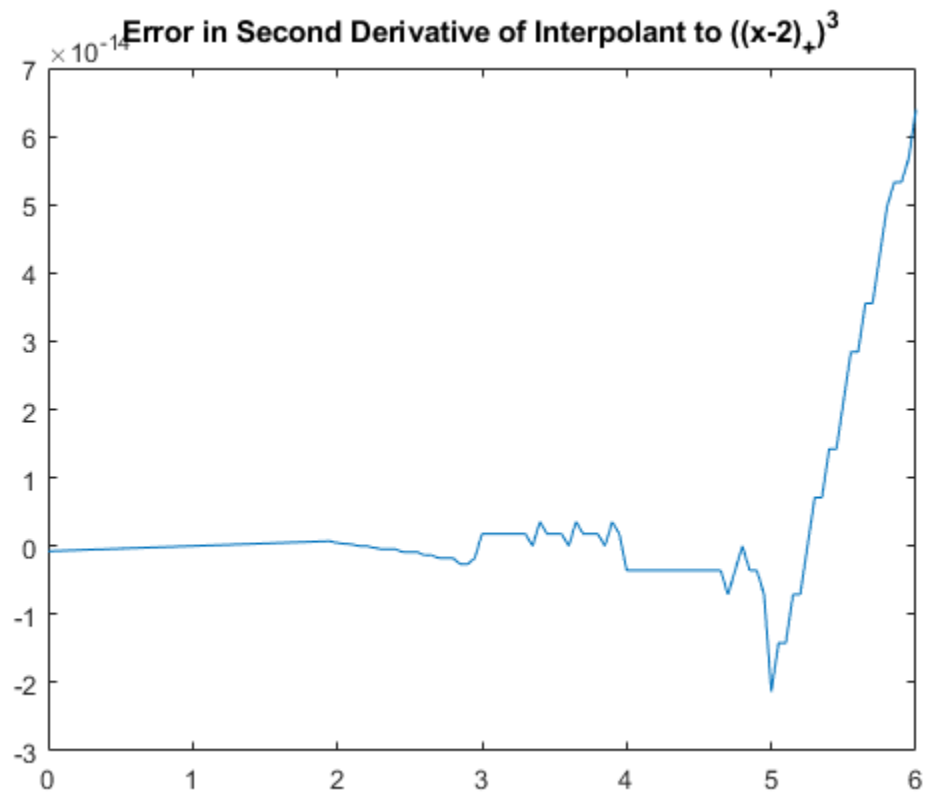
The integral of the truncated power is

$$F_2(x) = ((x-2)_+)^4/4.$$

A plot of the difference between this function and the integral of the interpolant to the original function again shows that the errors are within round-off.

```
ipp = fnint(pp);
plot(xx, fnval(ipp,xx) - subplus(xx-2).^4/4)
title('Error in Integral of Interpolant to ((x-2)_+)^3')
```

Error in Integral of Interpolant to $((x\text{-}2)_+)^3$
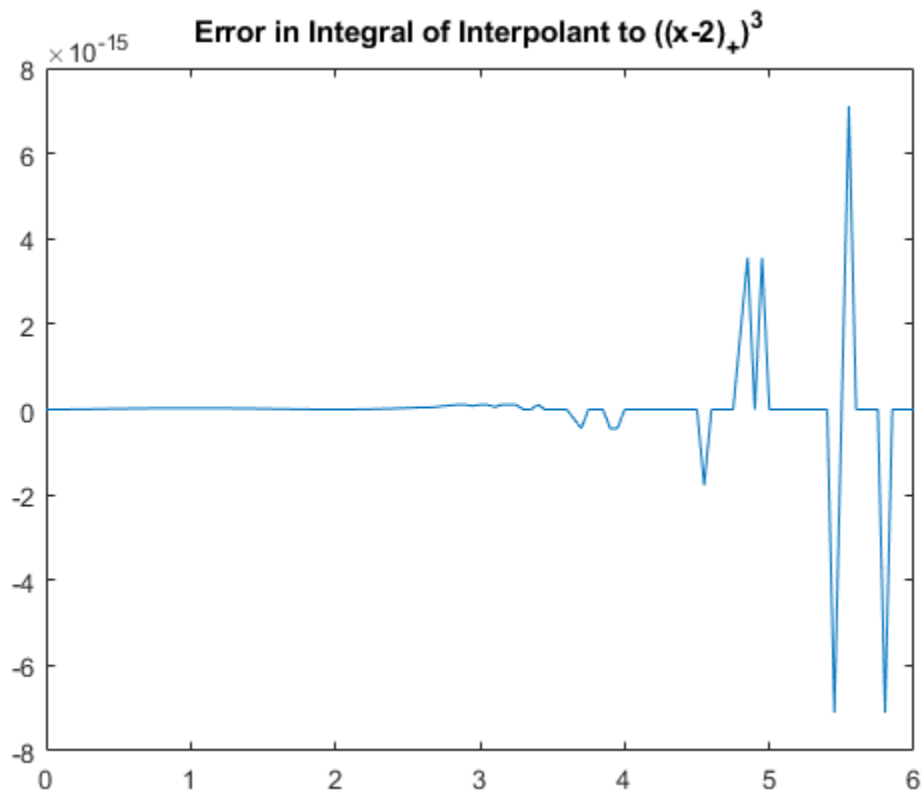
The CSAPE Command

Like `csapi`, the `csape` command provides a cubic spline interpolant to given data. However, it permits various additional end conditions. Its simplest version,

```
pp = csape(x,y)
```

uses the Lagrange end condition, which is a common alternative to the not-a-knot condition used by `csapi`. `csape` does not directly return values of the interpolant, but only its ppform.

For example, consider again interpolation to the function

$$f_1(x) = ((x - 1)_+)^3,$$

which `csapi` fails to reproduce well. We plot the error of the not-a-knot interpolant returned by `csapi` (in black), along with the error of the interpolant obtained from `csape` (in red).

```
exact = subplus(xx-1).^3;
plot(xx, fnval(csapi(x,subplus(x-1).^3),xx) - exact,'k')
hold on
plot(xx, fnval(csape(x,subplus(x-1).^3),xx) - exact,'r')
title('Error in Not-a-Knot vs. Lagrange End Conditions')
legend({'Not-a-Knot' 'Lagrange'});
hold off
```
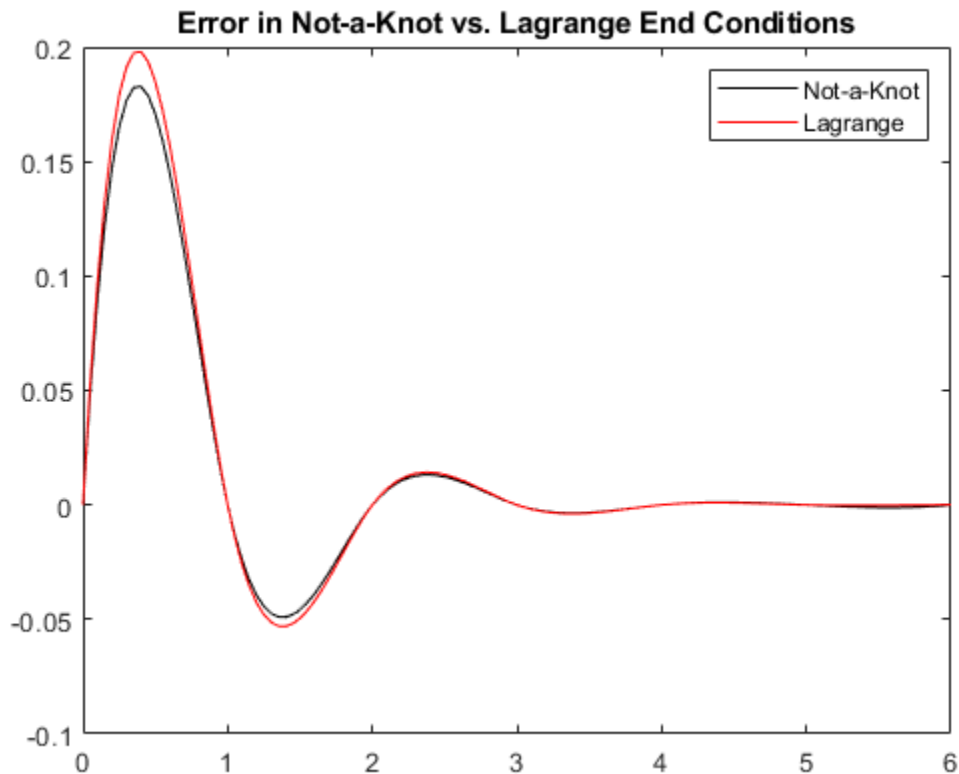
There is not much difference between the two interpolants in this case.

Other End Conditions: The `Natural' Spline Interpolant

The `csape` command also provides ways to specify several other types of end conditions for an interpolating cubic spline. For example, the command

```
pp = csape(x,y,'variational')
```

uses the so-called `natural' end conditions. This means that the second derivative is zero at the two extreme breaks.

This step shows how to apply `natural' cubic spline interpolation to the function

$$f_2(x) = ((x - 2)_+)^3,$$

and plot the error. The code below computes the `natural' spline interpolant with an alternative argument syntax that is equivalent to the 'variational' string argument: using the string 'second' specifies that `csape` should set the second derivative at the extreme data sites to the default value of 0.

```
pp = csape(x,subplus(x-2).^3,'second');
plot(xx, fnval(pp,xx) - subplus(xx-2).^3)
title('Error in ''Natural'' Spline Interpolation to ((x-2)_+)^3')
```

Error in 'Natural' Spline Interpolation to $((x-2)_+)^3$
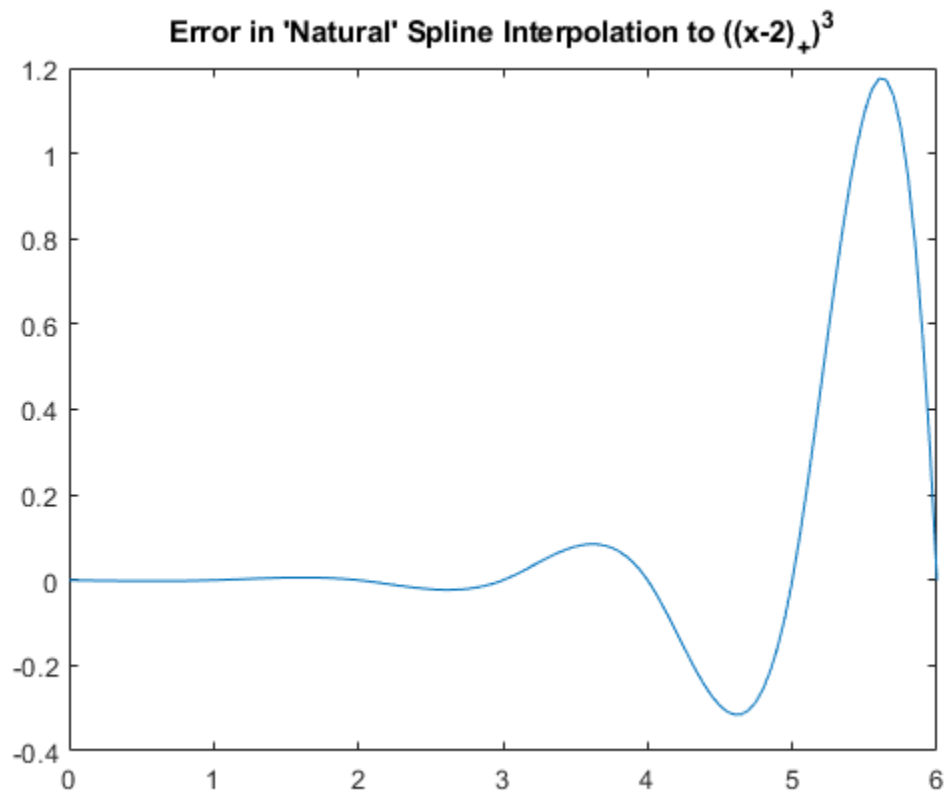
Note the large error near the right end. This is due to the fact that the 'natural' end conditions implicitly insist on having a zero second derivative there.

Other End Conditions: Prescribing Second Derivatives

We can also explicitly use the correct second derivatives to get a small error. First, we compute the correct second derivative values of the truncated power at the endpoints.

```
endcond = 6*subplus(x([1 end])-2);
```

Then we create the interpolant, specifying that second derivatives at the endpoints are to be matched to the second derivative values we just computed. We do this by providing `endcond(1)` for the left endpoint condition, and `endcond(2)` for the right, along with the data values.

```
pp = csape(x,[endcond(1) subplus(x-2).^3 endcond(2)], 'second');
plot(xx, fnval(pp,xx) - subplus(xx-2).^3,'r')
title(['Error in Spline Interpolation to ((x-1)_+)^3'; ...
        '  When Matching the 2nd Derivative at Ends  '])
```

Error in Spline Interpolation to $((x-1)_+)^3$ When Matching the 2nd Derivative at Ends
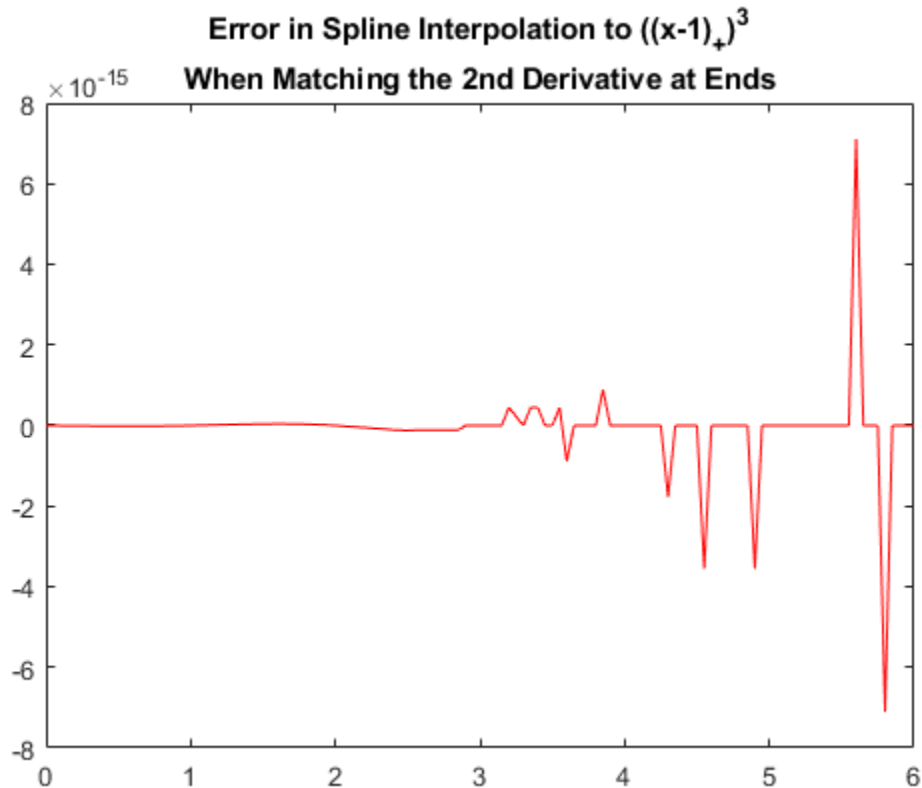
Other End Conditions: Prescribing Slopes

`csape` also permits specification of endpoint *slopes*. This is the *clamped* (or, *complete*) cubic spline interpolant. The statement

```
pp = csape(x,[sl,y,sr],'clamped')
```

creates the cubic spline interpolant to the data (x, y) that also has slope `sl` at the leftmost data site and slope `sr` at the rightmost data site.

Other End Conditions: Mixed End Conditions

It is even possible to mix these conditions. For example, our much-exercised truncated power function

$$f_1(x) = ((x-1)_+)^3$$

has slope 0 at x=0 and second derivative 30 at x=6 (the last data site).

Therefore, by matching the slope at the left end and the curvature at the right, we expect no error in the resulting interpolant.

```
pp = csape(x, [0 subplus(x-1).^3 30], [1 2]);
plot(xx, fnval(pp,xx) - subplus(xx-1).^3)
title(['Error in Spline Interpolation to ((x-1)_+)^3'; ...
'          with Mixed End Conditions.          '])
```

Error in Spline Interpolation to $((x-1)_+)^3$ with Mixed End Conditions.
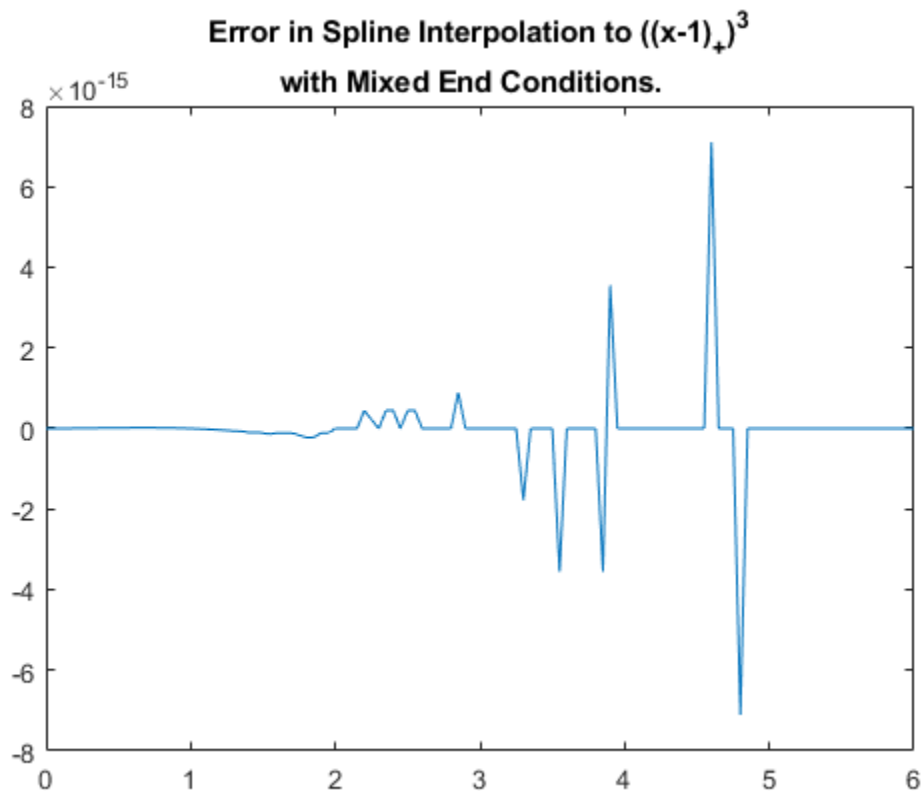
Other End Conditions: Periodic Conditions

It is also possible to prescribe *periodic* end conditions. For example, the sine function is 2*pi-periodic and has the values `[0 -1 0 1 0]` at the sites `(pi/2)*(-2:2)`. The difference, between the sine function and its periodic cubic spline interpolant at these sites, is only 2 percent. Not bad.

```
x = (pi/2)*(-2:2);
y = [0 -1 0 1 0];
pp = csape(x,y, 'periodic' );
xx = linspace(-pi,pi,201);
plot(xx, sin(xx) - fnval(pp,xx), 'x')
title('Error in Periodic Cubic Spline Interpolation to sin(x)')
```

Error in Periodic Cubic Spline Interpolation to sin(x)
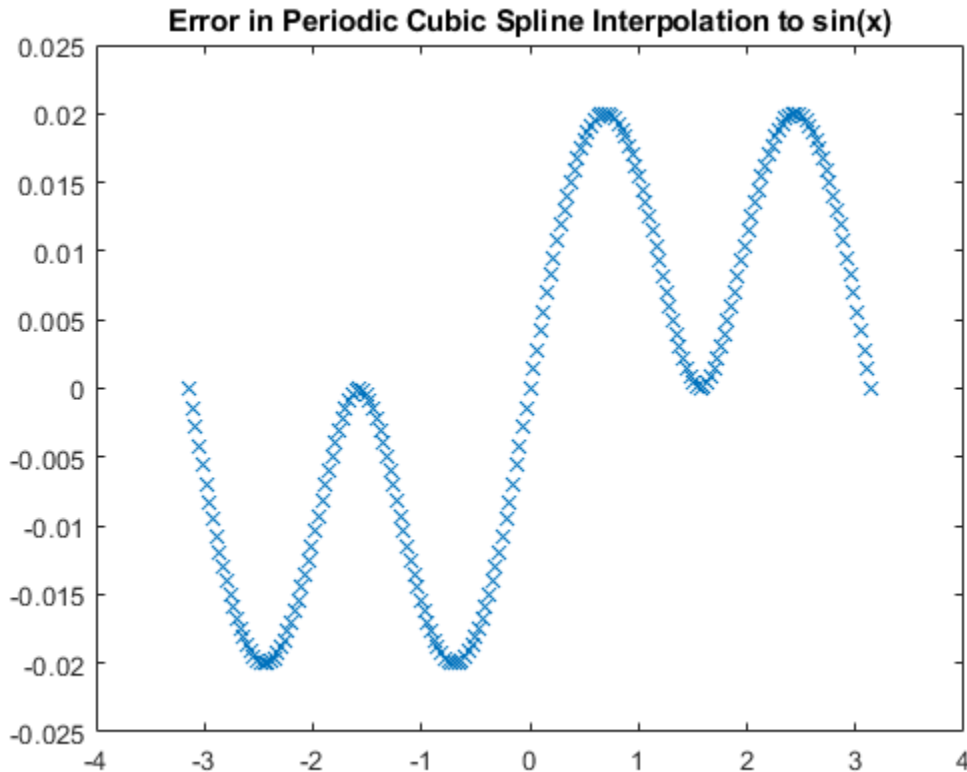
End Conditions Not Explicitly Covered by CSAPI or CSAPE

Any end condition not covered explicitly by `csapi` or `csape` can be handled by constructing the interpolant with the `csape` default side conditions, and then adding to it an appropriate scalar multiple of an interpolant to zero values and some side conditions. If there are two `nonstandard' side conditions to be satisfied, you may have to solve a 2-by-2 linear system first.

For example, suppose that you want to compute the cubic spline interpolant `s` to the data

```
x = 0:.25:3;
q = @(x) x.*(-1 + x.*(-1+x.*x/5));
y = q(x);
```

and enforce the condition

```
lambda(s) := a * (Ds)(e) + b * (D^2 s)(e) = c
```

on the first and second derivatives of `s` at the point `e`.

The data were generated from a quartic polynomial that happens to satisfy this side condition with specific parameters

```
e = x(1);
a = 2; b = -3; c = 4;
```

To construct the interpolant that satisfies this specific condition, we first construct the interpolant with the default end conditions

```
pp1 = csape(x,y);
```

and the first derivative of its first polynomial piece.

```
dp1 = fnder(fnbrk(pp1,1));
```

In addition, we construct the cubic spline interpolant to zero data values, specifying that it have a slope of 1 at e,

```
pp0 = csape(x,[1,zeros(size(y)),0], [1,0]);
```

as well as constructing the first derivative of its first polynomial piece.

```
dp0 = fnder(fnbrk(pp0,1));
```

Then we compute `lambda` for both `pp1` and `pp0`,

```
lam1 = a*fnval(dp1,e) + b*fnval(fnder(dp1),e);
lam0 = a*fnval(dp0,e) + b*fnval(fnder(dp0),e);
```

and construct the correct linear combination of `pp1` and `pp0` to get a cubic spline

```
s := pp1 + ((c - lambda(pp1))/lambda(pp0)) * pp0
```

that does satisfy the desired condition, as well as the default end condition at the right endpoint. We form this linear combination with the help of `fncmb`.

```
s = fncmb(pp0,(c-lam1)/lam0,pp1);
```

A plot of the interpolation error shows that `s` fits the quartic polynomial slightly better near e than the interpolant `pp1` with the default conditions does.

```
xx = (-.3):.05:.7; yy = q(xx);
plot(xx, fnval(pp1,xx) - yy, 'x')
hold on
plot(xx, fnval(s,xx) - yy, 'o')
hold off
legend({'Default conditions' 'Nonstandard conditions'},'location','SE')
```
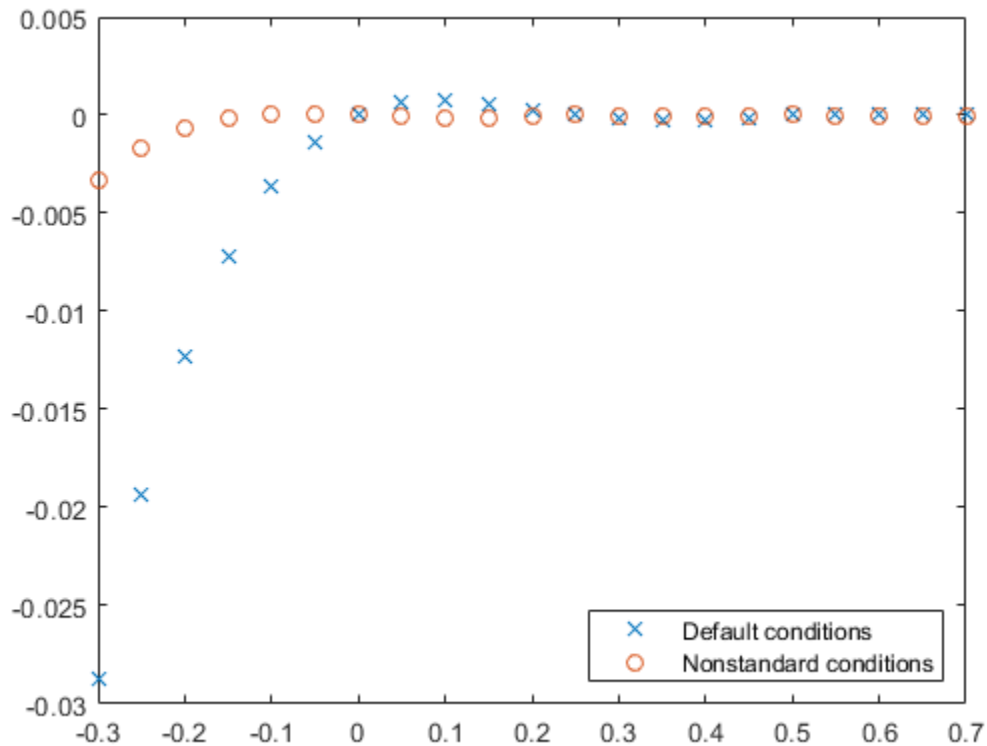
If we want to enforce the condition

```
mu(s) := (D^3 s)(3) = 14.6
```

on the third derivative of the interpolant (the quartic satisfies this condition), then we construct an additional cubic spline interpolating to zero values, and with zero first derivative at the left endpoint, hence certain to be independent from `pp0`.

```
pp2 = csape(x,[0,zeros(size(y)),1],[0,1]);
```

Then we find the coefficients `d0` and `d2` in the linear combination

```
s := pp1 + d0*pp0 + d2*pp2
```

that solves the linear system

```
lambda(s) = c
```

```
mu(s) = 14.6
```

Note that both `pp0` and `pp2` vanish at all interpolation sites, hence `s` will match the given data for any choice of `d0` and `d2`.

For amusement, we use the MATLAB® encoding facility to write a loop to compute `lambda(pp_j)` and `mu(pp_j)`, for `j=0:2`.

```
dd = zeros(2,3);
for j=0:2
```

```
    J = num2str(j);
    eval(['dpp',J,'=fnder(pp',J,');']);
    eval(['ddpp',J,'=fnder(dpp',J,');']);
    eval(['dd(1,1+',J,')=a*fnval(dpp',J,',e)+b*fnval(ddpp',J,',e);']);
    eval(['dd(2,1+',J,')=fnval(fnder(ddpp',J,'),3);']);
end
```

Given the values of `lambda` and `mu` for `pp0`, `pp1`, and `pp2`, we then solve for the coefficients that define the correct linear combination.

```
d = dd(:,[1,3])\([c;14.6]-dd(:,2));
s = fncmb(fncmb(pp0,d(1),pp2,d(2)),pp1);

xxx = 0:.05:3;
yyy = q(xxx);
plot(xxx, yyy - fnval(s,xxx),'x')
title('Error in Spline Interpolant to y = x*(-1 + x*(-1+x*x/5))')
```
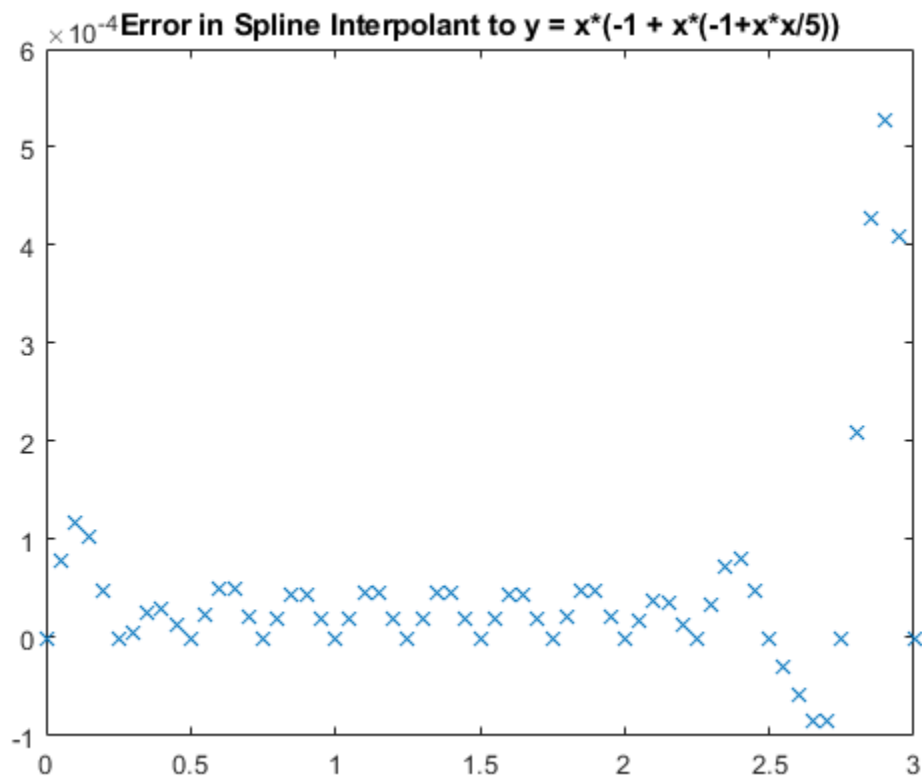

For reassurance, we compare this error with the one obtained in complete cubic spline interpolation to this function:

```
hold on
plot(xxx, yyy - fnval(csape(x,[-1,y,-7+(4/5)*27],'clamped'),xxx),'o')
hold off
legend({'Nonstandard conditions' 'Endslope conditions'})
```

Error in Spline Interpolant to y = x*(-1 + x*(-1+x*x/5))
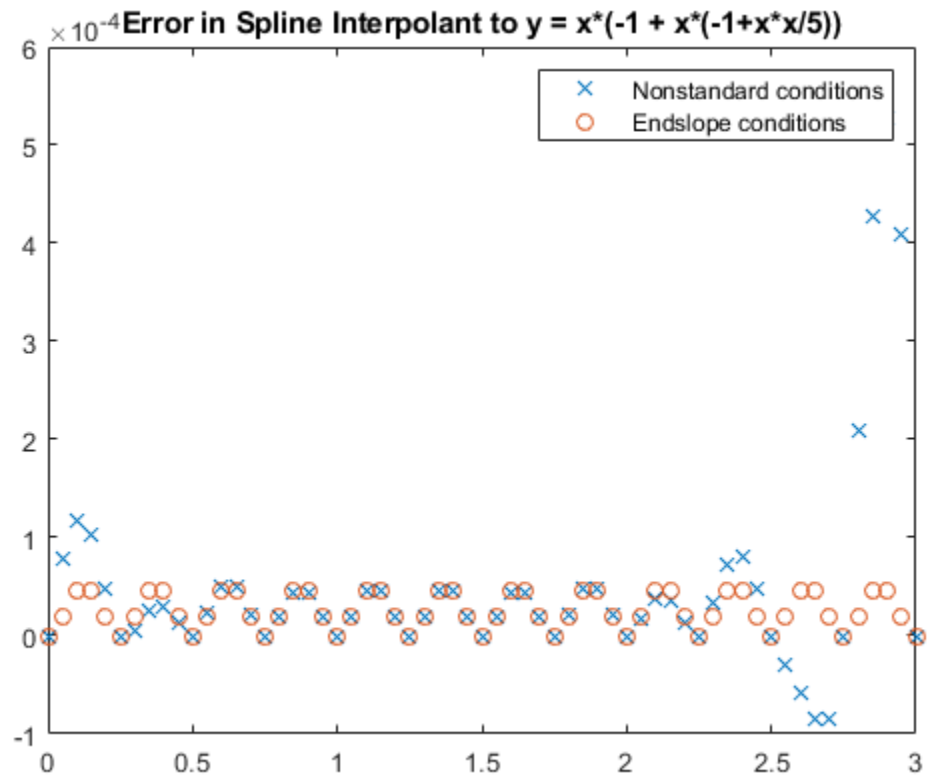
The errors differ (and not by much) only near the end points, testifying to the fact that both `pp0` and `pp2` are sizable only near their respective end points.

As a final check, we verify that `s` satisfies the desired third derivative condition at 3.

```
fnval(fnder(s,3),3)
```

```
ans = 14.6000
```

Cubic Smoothing Splines

This example shows how to use the `csaps` and `spaps` commands from Curve Fitting Toolbox™ to construct cubic smoothing splines.

The CSAPS Command

The command `csaps` provides the *smoothing* spline. This is a cubic spline that more or less follows the presumed underlying trend in noisy data. A smoothing parameter, to be chosen by you, determines just how closely the smoothing spline follows the given data. Here is the basic information, an abbreviated version of the documentation:

CSAPS Cubic smoothing spline.

VALUES = CSAPS(X, Y, P, XX)

Returns the values at XX of the cubic smoothing spline for the

given data (X,Y) and depending on the smoothing parameter P, chosen

from the interval [0 .. 1]. This smoothing spline f minimizes

P * sum_i W(i)(Y(i) - f(X(i)))^2 + (1-P) * integral (D^2 f)^2

Example: Noisy Data From a Cubic Polynomial

Here are some trial runs. We start with data from a simple cubic, `q(x) := x^3`, contaminate the values with some noise, and choose the value of the smoothing parameter to be .5. Then plot the resulting smoothed values, along with the underlying cubic, and the contaminated data.

```
xi = (0:.05:1);
q = @(x) x.^3;
yi = q(xi);
randomStream = RandStream.create( 'mcg16807', 'Seed', 23 );
ybad = yi+.3*(rand(randomStream, size(xi))-.5);
p = .5;
xxi = (0:100)/100;
ys = csaps(xi,ybad,p,xxi);
plot(xi,yi,':',xi,ybad,'x',xxi,ys,'r-')
title('Clean Data, Noisy Data, Smoothed Values')
legend( 'Exact', 'Noisy', 'Smoothed', 'Location', 'NorthWest' )
```

Clean Data, Noisy Data, Smoothed Values
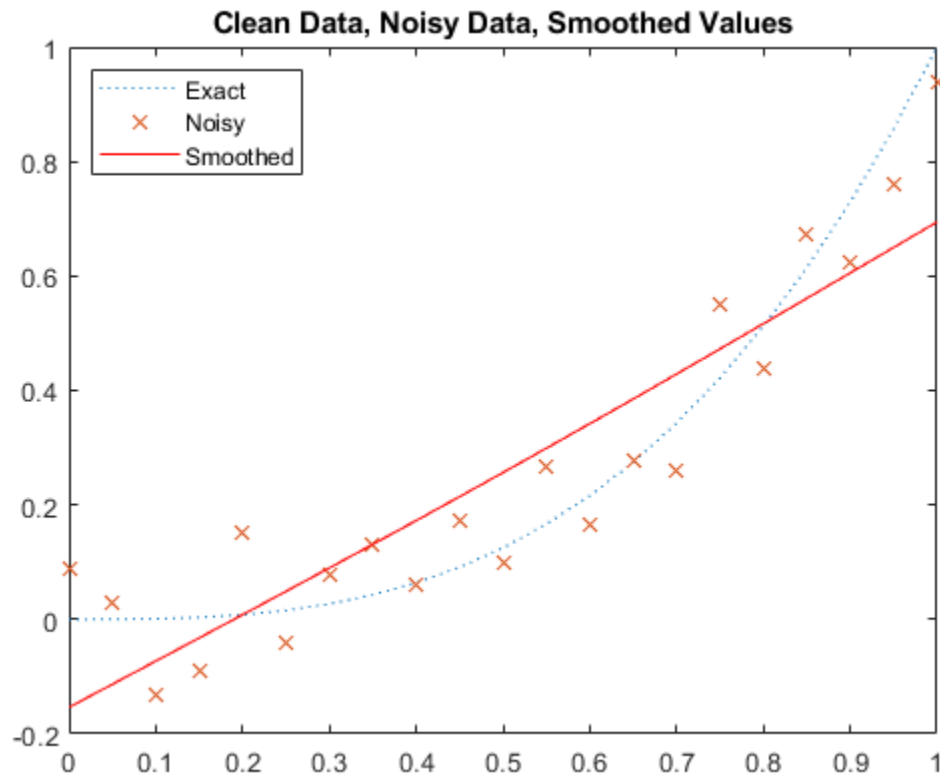
The smoothing is way overdone here. By choosing the smoothing parameter p closer to 1, we obtain a smoothing spline closer to the given data. We try `p = .6, .7, .8, .9, 1`, and plot the resulting smoothing splines.

```
yy = zeros(5,length(xxi));
p = [.6 .7 .8 .9 1];
for j=1:5
   yy(j,:) = csaps(xi,ybad,p(j),xxi);
end
hold on
plot(xxi,yy);
hold off
title('Smoothing Splines for Various Values of the Smoothing Parameter')
legend({'Exact','Noisy','p = 0.5','p = 0.6','p = 0.7','p = 0.8', ...
        'p = 0.9', 'p = 1.0'}, 'Location', 'NorthWest' )
```

Smoothing Splines for Various Values of the Smoothing Parameter
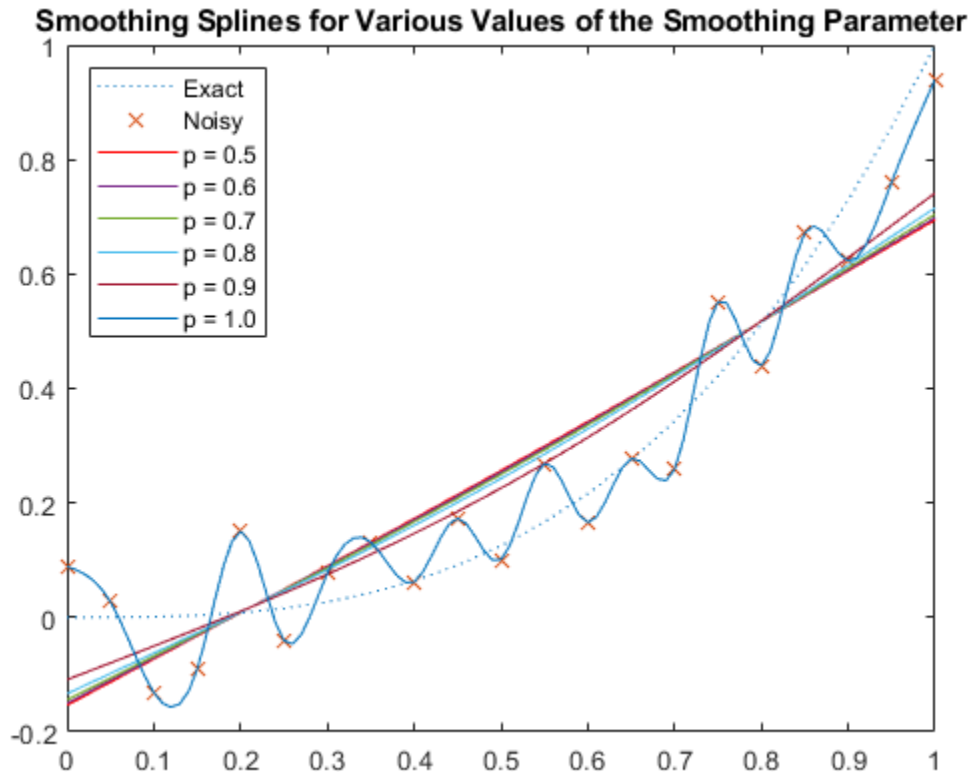
We see that the smoothing spline can be very sensitive to the choice of the smoothing parameter. Even for p = 0.9, the smoothing spline is still far from the underlying trend, while for p = 1, we get the interpolant to the (noisy) data.

In fact, the formulation used by `csapi` (p.235ff of *A Practical Guide to Splines*) is very sensitive to scaling of the independent variable. A simple analysis of the equations used shows that the sensitive range for `p` is around `1/(1+epsilon)`, with `epsilon := h^3/16`, and `h` the average difference between neighboring sites. Specifically, you would expect a close following of the data when `p = 1/(1+epsilon/100)` and some satisfactory smoothing when `p = 1/(1+epsilon*100)`.

The plot below shows the smoothing spline for values of `p` near this magic number `1/(1+epsilon)`. For this case, it is more informative to look at `1-p` since the magic number, `1/(1+epsilon)`, is very close to 1.

```
epsilon = ((xi(end)-xi(1))/(numel(xi)-1))^3/16;
1 - 1/(1+epsilon)
```

```
ans = 7.8124e-06
```

```
plot(xi,yi,':',xi,ybad,'x')
hold on
labels = cell(1,5);
for j=1:5
    p = 1/(1+epsilon*10^(j-3));
    yy(j,:) = csaps(xi,ybad,p,xxi);
    labels{j} = ['1-p= ',num2str(1-p)];
end
```

```
plot(xxi,yy)
title('Smoothing Splines for Smoothing Parameter Near Its ''Magic'' Value')
legend( [{'Exact', 'Noisy'}, labels], 'Location', 'NorthWest' )
hold off
```
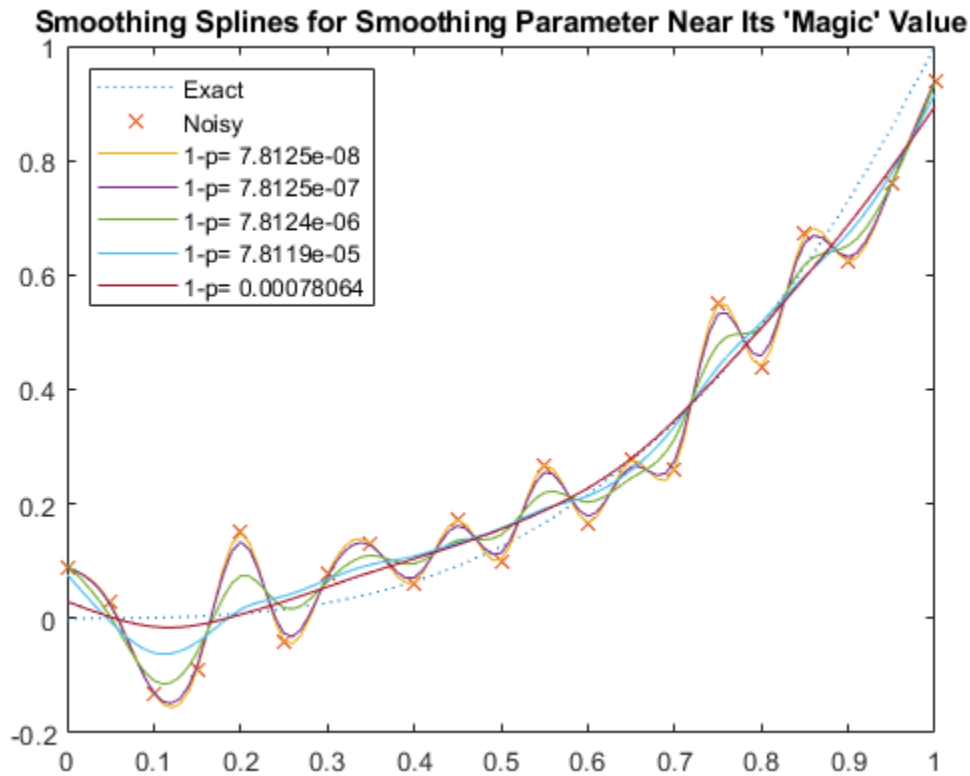

In this example, the smoothing spline is very sensitive to variation of the smoothing parameter near the magic number. The one farthest from 1 seems the best choice from these, but you may prefer the one beyond that.

```
p = 1/(1+epsilon*10^3);
yy = csaps(xi,ybad,p,xxi);
hold on
plot( xxi, yy, 'y', 'LineWidth', 2 )
title( sprintf( 'The Smoothing Spline For 1-p = %s is Added, in Yellow', num2str(1-p) ) )
hold off
```

The Smoothing Spline For 1-p = 0.0077519 is Added, in Yellow
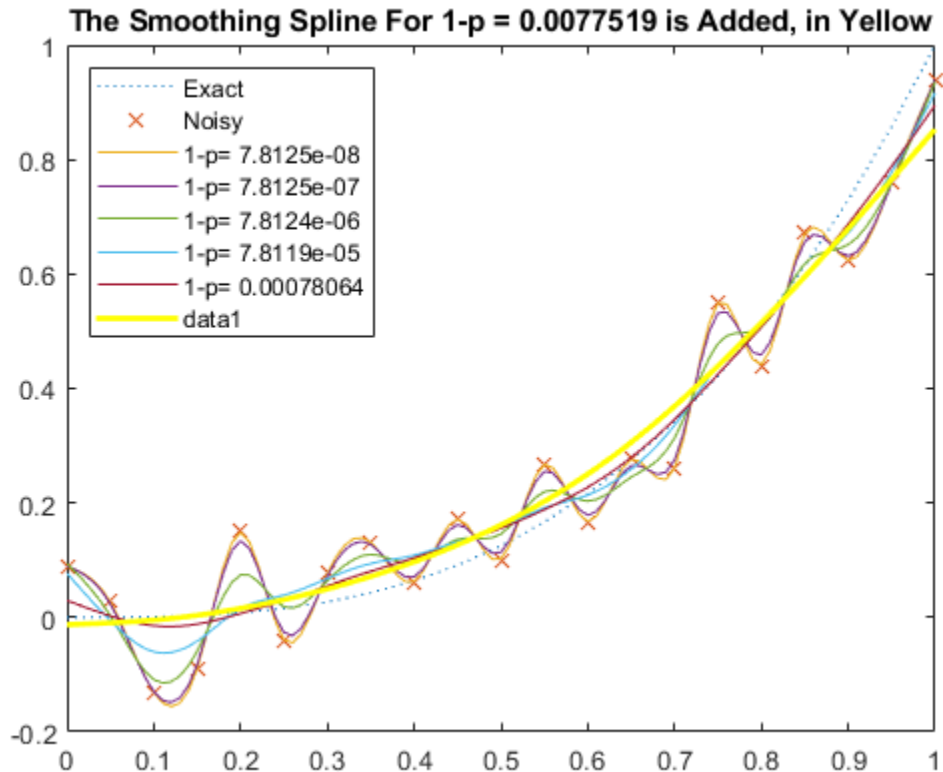
You can also supply `csaps` with error weights, to pay more attention to some data points than others. Also, if you do not supply the evaluation sites `xx`, then `csaps` returns the ppform of the smoothing spline.

Finally, `csaps` can also handle vector-valued data and even multivariate, gridded data.

The SPAPS Command

The cubic smoothing spline provided by the command `spaps` differs from the one constructed in `csaps` only in the way it is selected. Here is an abbreviated version of the documentation for `spaps`:

```
SPAPS Smoothing spline.

[SP,VALUES] = SPAPS(X,Y,TOL) returns the B-form and, if asked,

the values at X, of a cubic smoothing spline f for the given

data (X(i),Y(:,i)), i=1,2, ..., n.

The smoothing spline f minimizes the roughness measure

F(D^2 f) := integral ( D^2 f(t) )^2 dt on X(1) < t < X(n)

over all functions f for which the error measure

E(f) := sum_j { W(j)*( Y(:,j) - f(X(j)) )^2 : j=1,...,n }
```

is no bigger than the given TOL. Here, D^M f denotes the M-th

derivative of f. The weights W are chosen so that E(f) is the

Composite Trapezoid Rule approximation for F(y-f).

f is constructed as the unique minimizer of

rho*E(f) + F(D^2 f),

with the smoothing parameter RHO so chosen so that E(f) equals

TOL. Hence, FN2FM(SP,'pp') should be (up to roundoff) the same

as the output from CPAPS(X,Y,RHO/(1+RHO)).

Tolerance vs. Smoothing Parameter

It may be easier to supply a suitable tolerance for spaps than the smoothing parameter p required by csaps. In our earlier example, we added uniformly-distributed random noise from the interval 0.3*[-0.5 .. 0.5]. Hence, we can estimate a reasonable value for tol as the value of the error measure e at such noise.

```
tol = sum((.3*(rand(randomStream, size(yi))-.5)).^2);
```

This plot shows the resulting smoothing spline constructed by spaps. Note that the error weights are specified to be uniform, which is their default value in csaps.

```
[sp,ys,rho] = spaps(xi,ybad,tol,ones(size(xi)));
plot(xi,yi,':',xi,ybad,'x',xi,ys,'r-')
title( sprintf( 'Clean Data, Noisy Data, Smoothed Values (1-p = %s )', num2str(1/(1+rho)) ) );
legend( {'Exact','Noisy','Smoothed'}, 'location', 'NorthWest' )
```

Clean Data, Noisy Data, Smoothed Values (1-p = 0.013761)
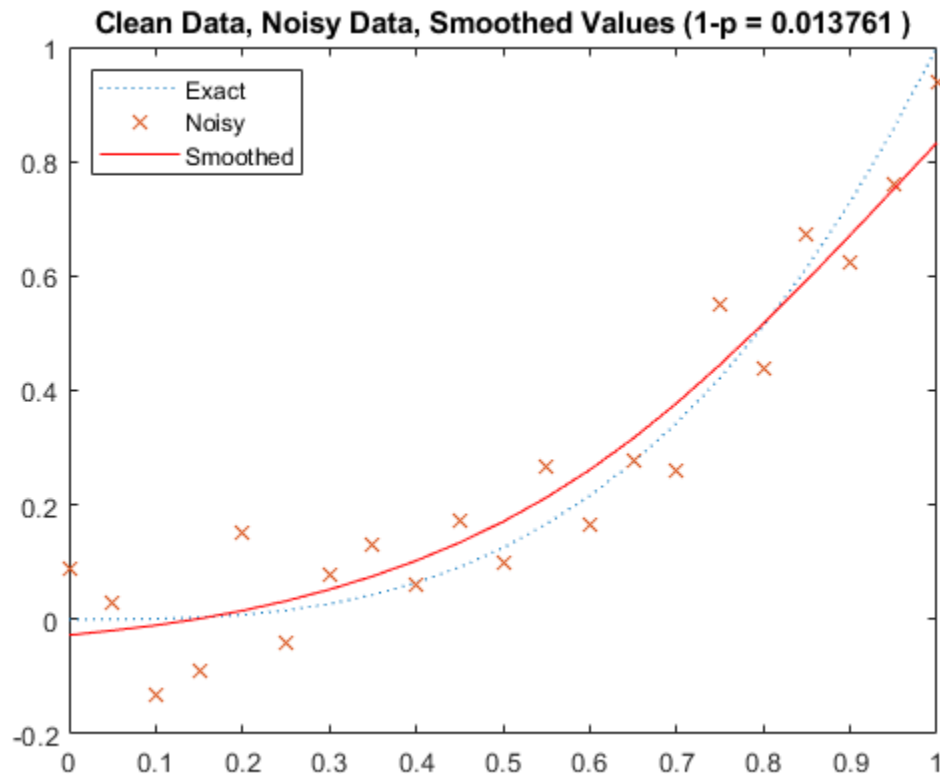
The figure title shows the value of `p` you would use in `csaps` to obtain exactly this smoothing spline for these data.

Here, in addition, is the smoothing spline provided by `csaps` when not given a smoothing parameter. In this case `csaps` chooses the parameter by a certain ad hoc procedure that attempts to locate the region where the smoothing spline is most sensitive to the smoothing parameter (similar to the earlier discussion).

```
hold on
plot(xxi,fnval(csaps(xi,ybad),xxi),'-')
title('Clean Data, Noisy Data, Smoothed Values')
legend({'Exact' 'Noisy' 'spaps, specified tolerance' ...
        'csaps, default smoothing parameter'}, 'Location', 'NorthWest' )
hold off
```

Clean Data, Noisy Data, Smoothed Values
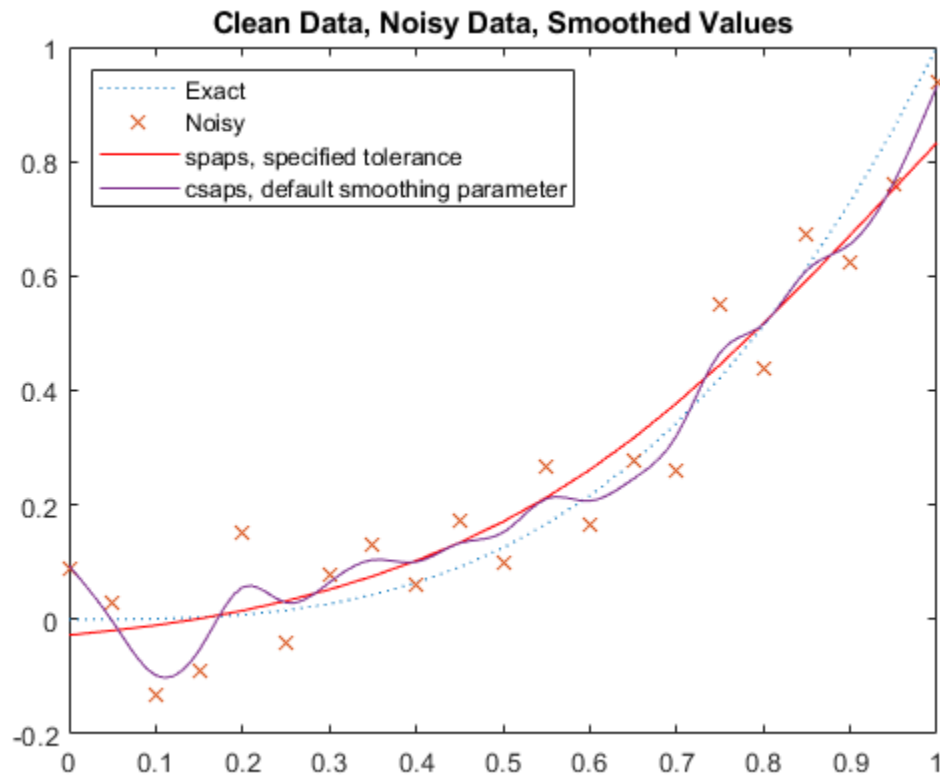
CSAPS vs. SPAPS

The `csaps` and `spaps` commands differ in the way in which you specify a particular smoothing spline, via a smoothing parameter vs. a tolerance. Another difference is that `spaps` can provide a linear or a quintic smoothing spline, in addition to the cubic smoothing spline.

The quintic smoothing spline is better than the cubic smoothing spline in the situation when you would like the second derivative to move as little as possible.

Fitting a Spline to Titanium Test Data

This example shows how to use commands from Curve Fitting Toolbox™ to fit a spline to titanium test data with manual and automatic selection of knots.

Manual Knot Choice for Spline Interpolation

Here are some data that record a certain property of titanium, measured as a function of temperature. We'll use it to illustrate some issues with spline interpolation.

```
[xx,yy] = titanium;
```

A plot of the data shows a rather sharp peak.

```
plot(xx,yy,'bx');
frame = [-10 10 -.1 .3]+[min(xx),max(xx),min(yy),max(yy)];
axis(frame);
```
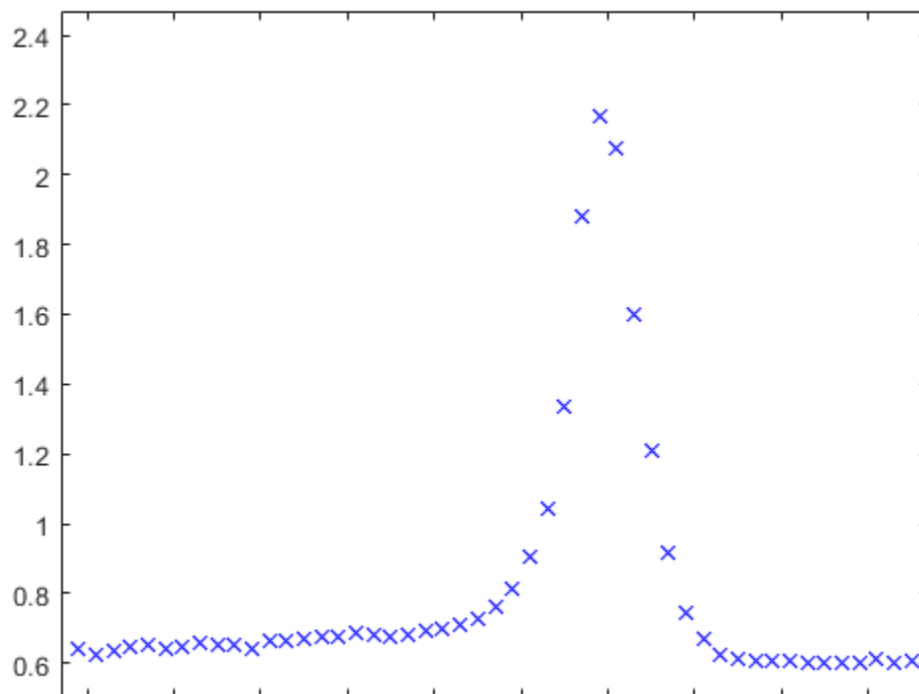

We pick a few data points from these somewhat rough data, since we want to interpolate. Here is a picture of the data, with the selected data points marked.

```
pick = [1 5 11 21 27 29 31 33 35 40 45 49];
tau = xx(pick);
y = yy(pick);
hold on
plot(tau,y,'ro');
hold off
```
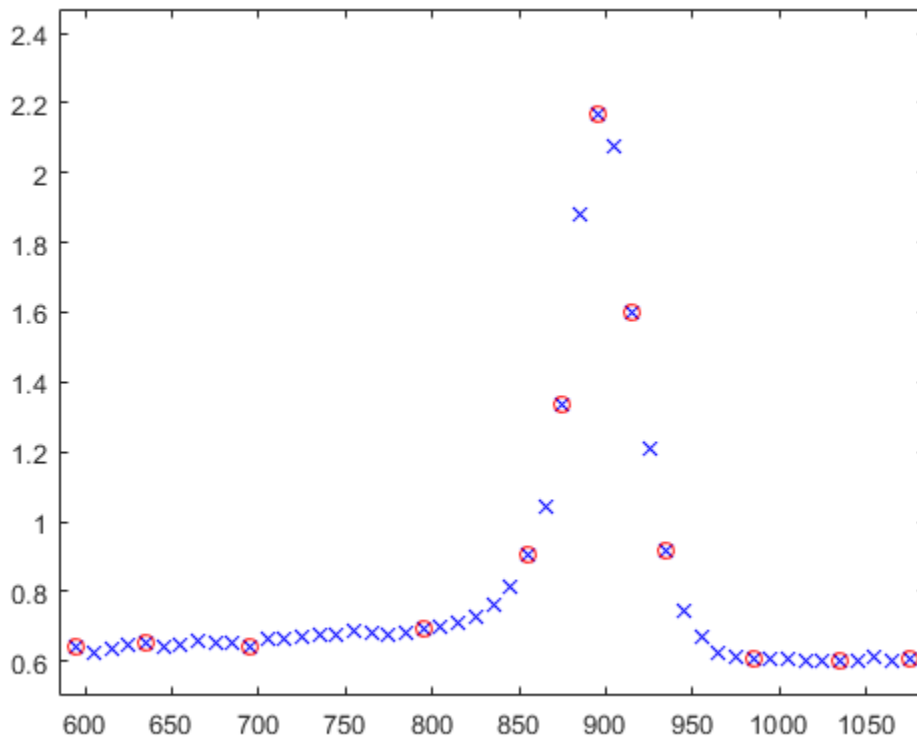
Since a spline of order k with n+k knots has n degrees of freedom, and we have 12 data points, a fit with a fourth order spline requires 12+4 = 16 knots. Moreover, this knot sequence t must be such that the i-th data site lies in the support of the i-th B-spline. We achieve this by using the data sites as knots, but add two simple knots at either end.

```
dl = tau(2) - tau(1);
dr = tau(end) - tau(end-1);
t = [tau(1)-dl*[2 1] tau tau(end)+dr*[1 2]];   % construct the knot sequence
plot(tau,y,'ro');
hold on
axis(frame+[-2*dl 2*dr 0 0])
plot(t,repmat(frame(3)+.03,size(t)),'kx')
hold off
legend({'Data Values' 'Knots'},'location','NW')
```
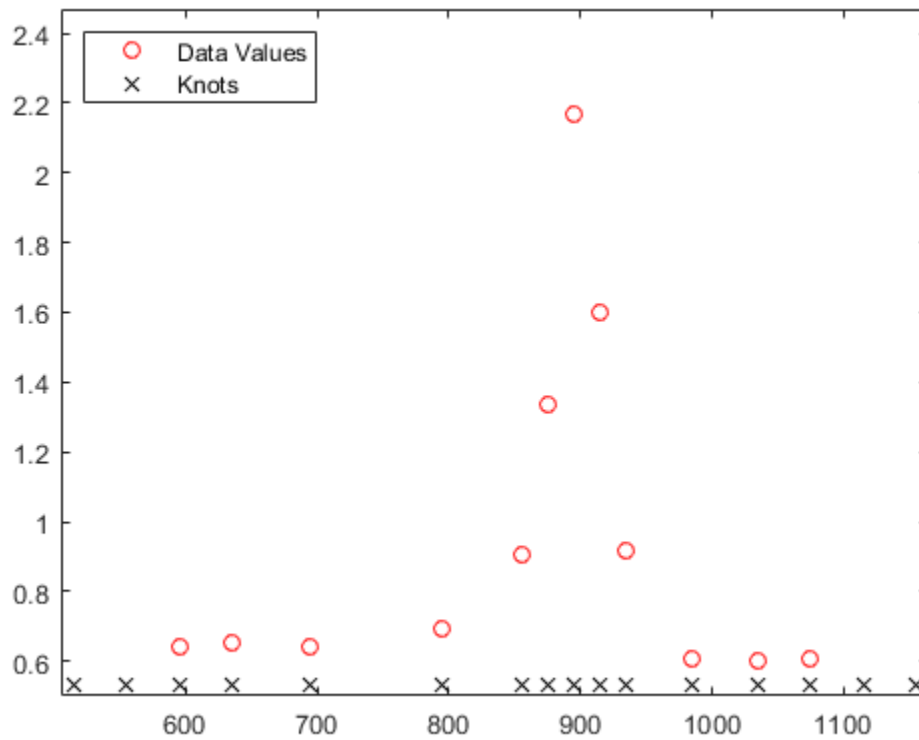
We use this knot sequence to construct an interpolating cubic spline.

```
sp = spapi(t,tau,y);
```

Now, for the plot. Since we do not care about the part of the spline outside the data interval, we restrict the plot to that interval.

```
plot(tau,y,'ro')
axis(frame)
hold on
fnplt(sp,[tau(1) tau(end)], 'k')
hold off
```
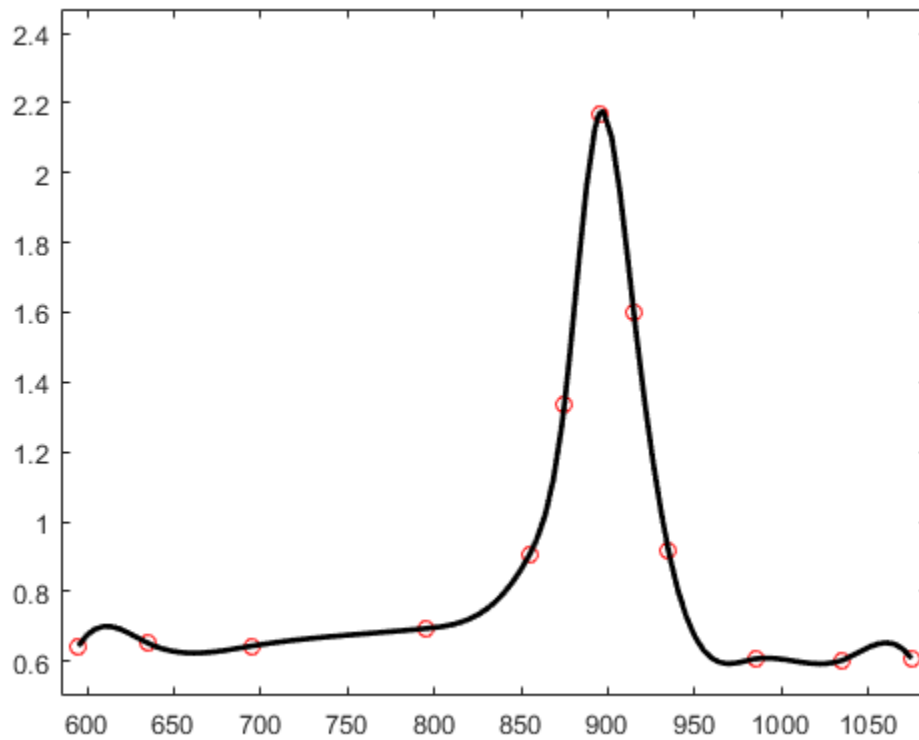
A closer look at the left part of the spline fit shows some undulations.

```
xxx = linspace(tau(1),tau(5),41);
plot(xxx, fnval(sp, xxx), 'k', tau, y, 'ro');
axis([tau(1) tau(5) 0.6 1.2]);
```
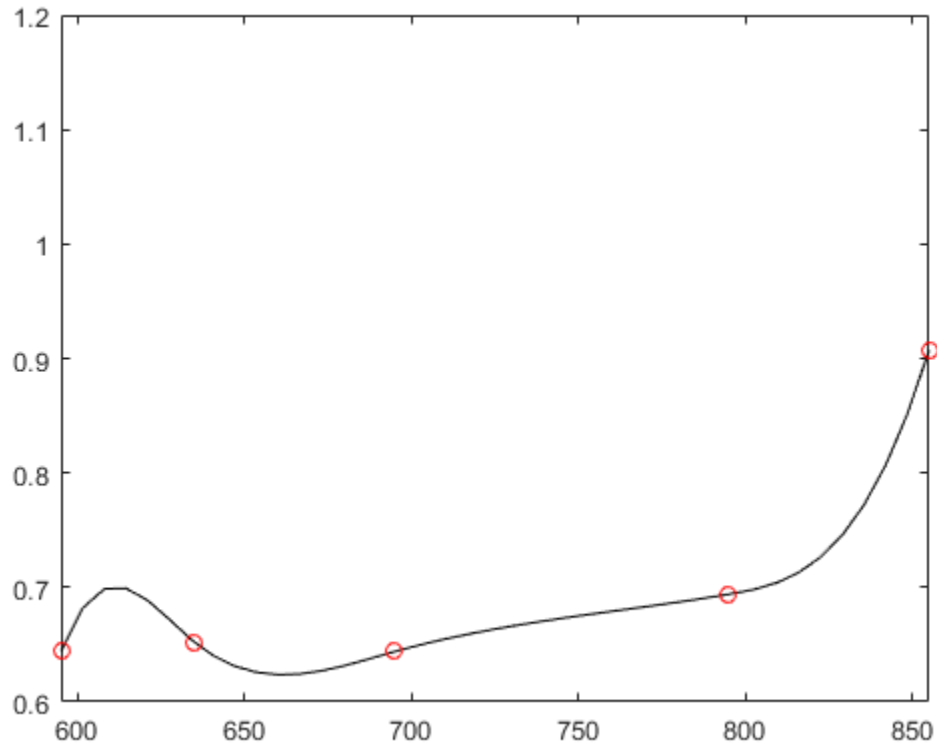
The unreasonable bump in the first interval stems from the fact that our spline goes smoothly to zero at its first knot. To see that, here is a picture of the entire spline, along with its knot sequence and the data points.

```
fnplt(sp,'k');
hold on
plot(tau,y,'ro', t,repmat(.1,size(t)),'kx');
hold off
legend({'Spline Interpolant' 'Data Values' 'Knots'},'location','NW')
```
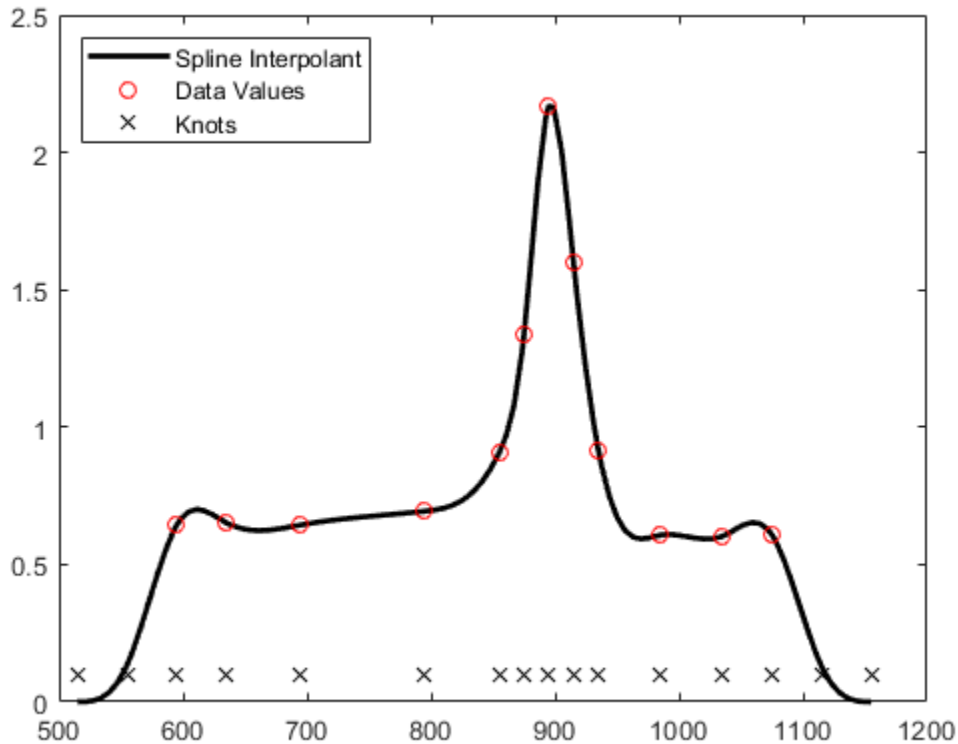
Here is a simple way to enforce a more reasonable boundary behavior. We add two more data points outside the given data interval and choose as our data there the values of the straight line through the first two data points.

```
tt = [tau(1)-[4 3 2 1]*dl tau tau(end)+[1 2 3 4]*dr];
xx = [tau(1)-[2 1]*dl tau tau(end)+[1 2]*dr];
yy = [y(1)-[2 1]*(y(2)-y(1)) y y(end)+[1 2]*(y(end)-y(end-1))];
sp2 = spapi(tt,xx,yy);
plot(tau,y,'ro', xx([1 2 end-1 end]),yy([1 2 end-1 end]),'bo');
axis(frame+[-2*dl 2*dr 0 0]);
hold on
fnplt(sp2,'b',tau([1 end]))
hold off
legend({'Original Data' 'Data Added for End Conditions' ...
        'Fit with Added Data'},'location','NW')
```
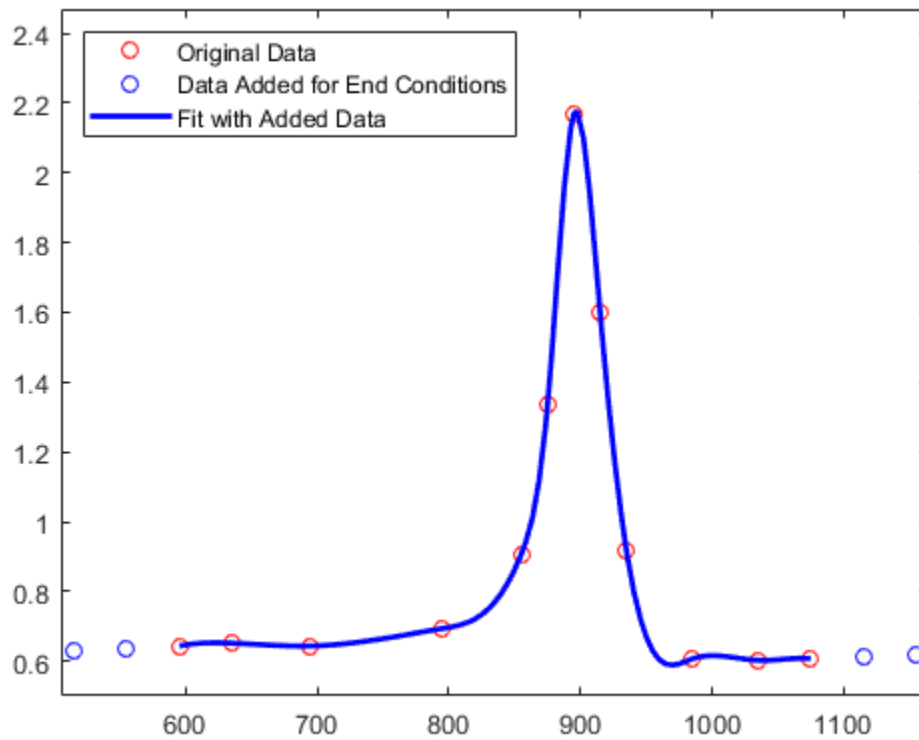
Here is a comparison of the two spline fits, to show the reduction of the undulation in the first and last interval.

```
hold on
fnplt(sp,'k',tau([1 end]))
hold off
legend({'Original Data' 'Data Added for End Conditions' ...
        'Fit with Added Data' 'Original Fit'},'location','NW')
```
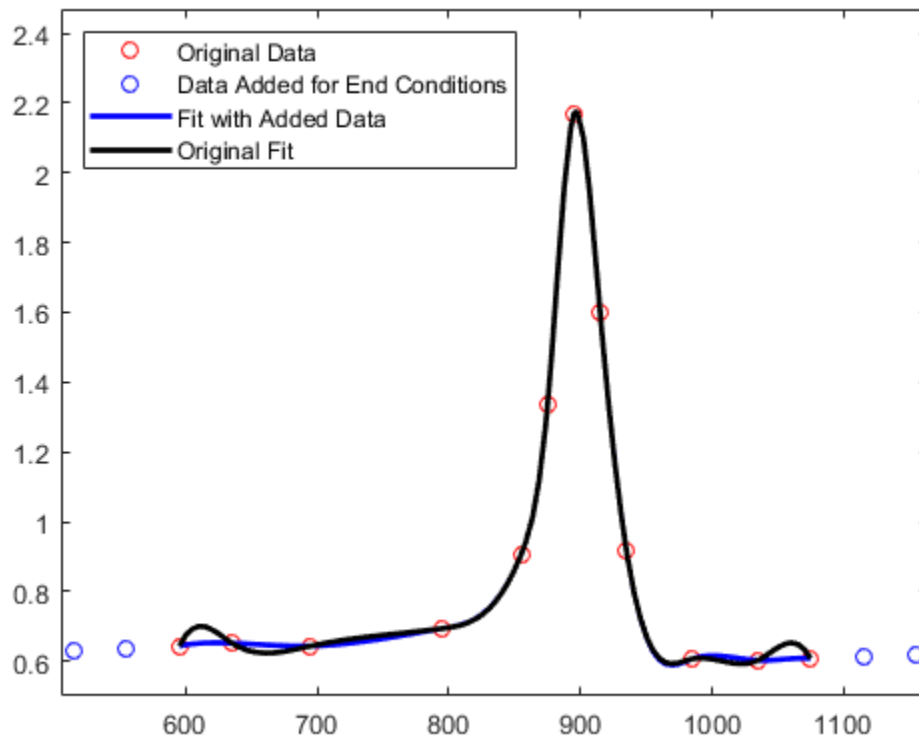
Finally, here is a closer look at the first four data intervals that shows more clearly the reduction of the undulation near the left end.

```
plot(tau,y,'ro',xxx,fnval(sp2,xxx),'b',xxx,fnval(sp,xxx),'k');
axis([tau(1) tau(5) .6 1.2]);
legend({'Original Data' 'Fit with Added Data' ...
        'Original Fit'},'location','NW')
```
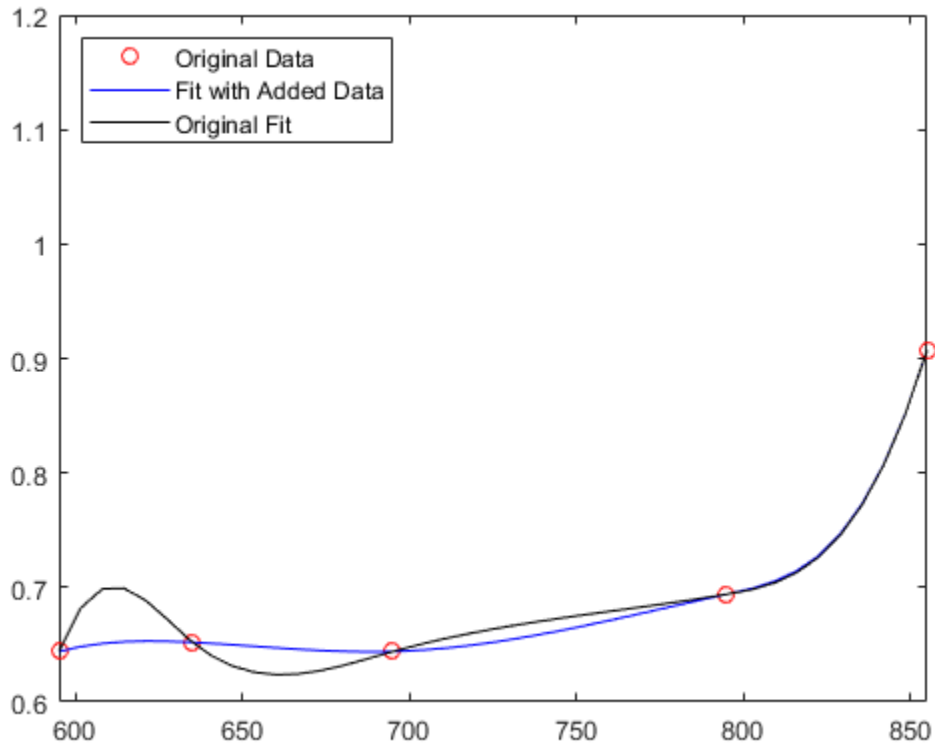
Automatic Knot Choice for Interpolation

If all this detail turns you off, let Curve Fitting Toolbox choose the knots for you. Specify the desired order of the interpolant as the first input argument to the spline interpolation command `spapi`, rather than a knot sequence.

```
autosp = spapi(4, tau, y);
knots = fnbrk(autosp,'knots');
plot(tau, y, 'ro')
hold on
fnplt(autosp,'g')
plot(knots, repmat(.5,size(knots)),'gx')
hold off
legend({'Data Values' 'Fit With Knots Chosen by SPAPI' ...
        'Knots Chosen by SPAPI'}, 'location','NW')
```
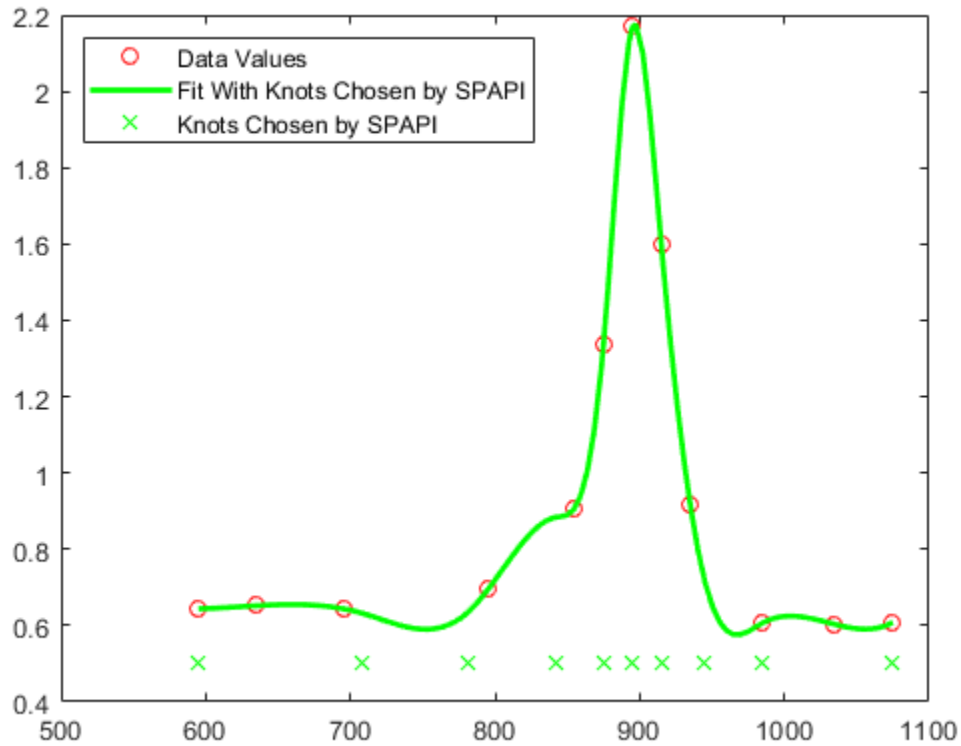
Below is the result of a much better knot choice, obtained by shifting the knot at 842 slightly to the right and the knot at 985 slightly to the left.

```
knots([7 12]) = [851, 971];
adjsp = spapi(knots, tau, y);
hold on
fnplt(adjsp,'r',2)
plot(knots, repmat(.54,size(knots)),'rx')
hold off
legend({'Data Values' 'Fit With Knots Chosen by SPAPI' ...
        'Knots Chosen by SPAPI' 'Fit With Knots Adjusted' ...
        'Adjusted Knots'}, 'location','NW')
```
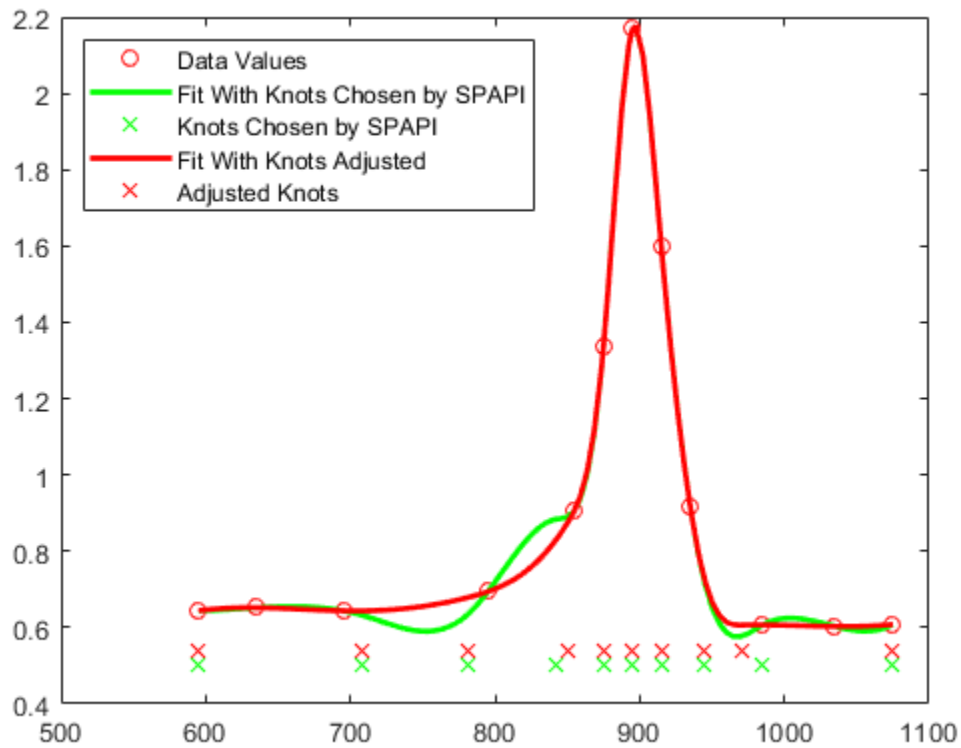
Or else, simply try the standard cubic spline interpolant, supplied by `csapi`. This amounts to a slightly different choice of knots.

```
autocs = csapi(tau, y);
plot(tau, y, 'ro')
hold on
fnplt(autocs,'c')
hold off
```
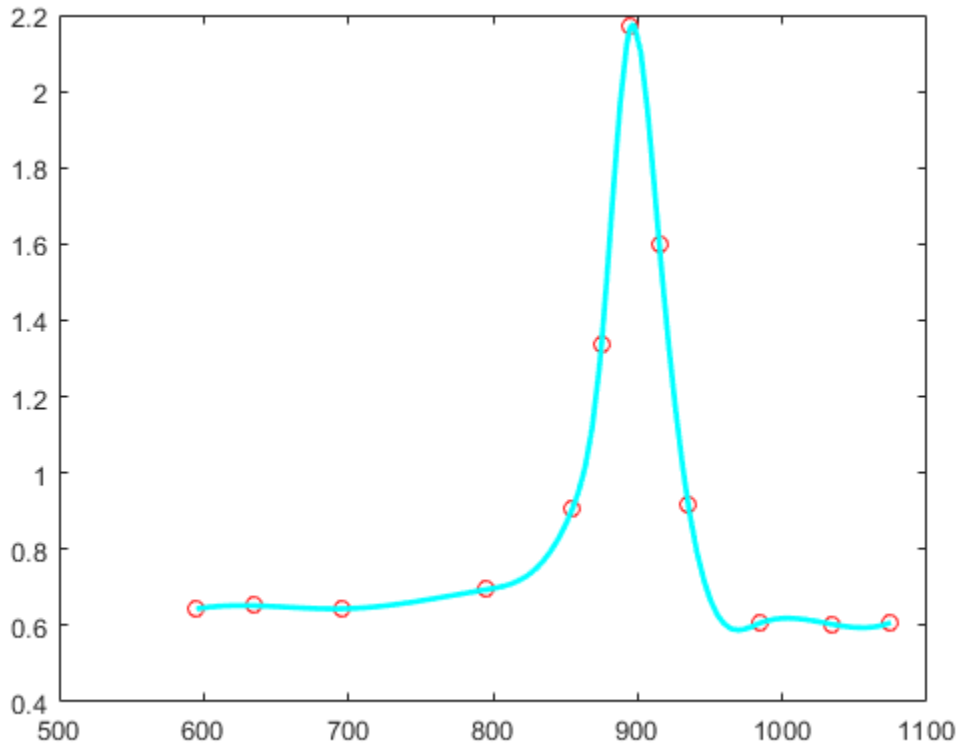
With such rapidly-varying data, it is hard to get agreement among all reasonable interpolants, even if each of them is a cubic spline. The plot below shows all five interpolants, for comparison.

```
plot(tau, y, 'ro')
hold on
fnplt(sp,'k',tau([1 end]))   % black: original
fnplt(sp2,'b',tau([1 end])) % blue:  with special end conditions
fnplt(autosp,'g')            % green: automatic knot choice by SPAPI
fnplt(autocs,'c')            % cyan:  automatic knot choice by CSAPI
fnplt(adjsp,'r',2)           % red:   knot choice by SPAPI slightly changed
hold off
legend({'Data Values' 'Original Fit' 'Special End Conditions' ...
        'With Knots Chosen by SPAPI' 'With Knots Chosen by CSAPI' ...
        'With Adjusted Knots'},'location','NW')
```
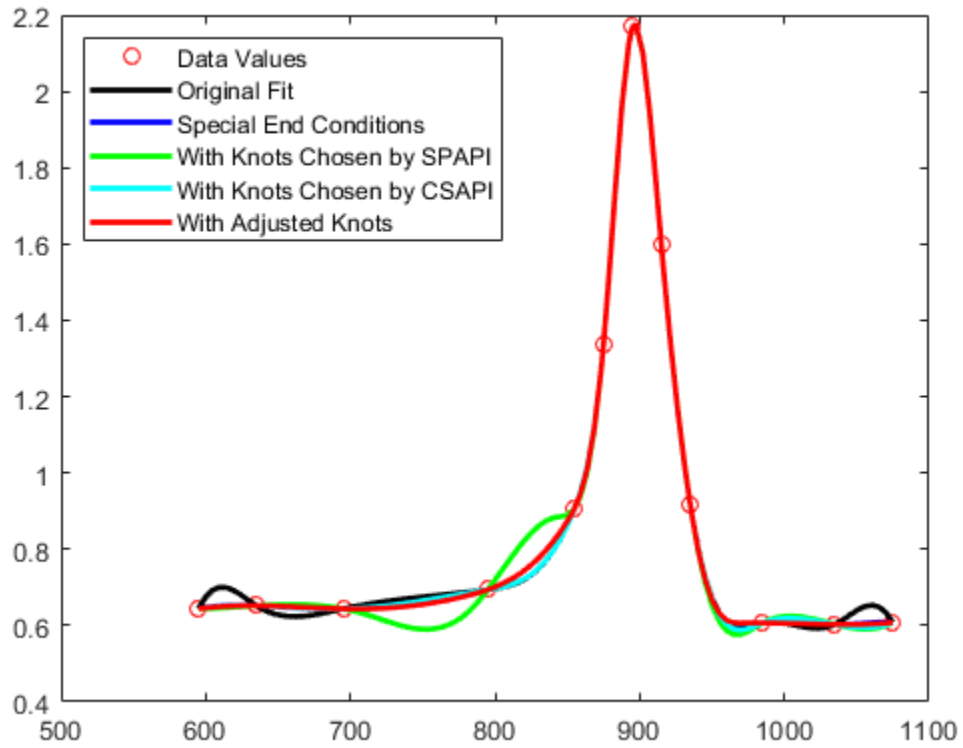
Splines in the Plane

This example shows how to use the `spmak`, `spcrv`, `cscvn` and `rscvn` commands from Curve Fitting Toolbox™ to construct spline curves in the plane. This includes plotting tangents and computing the area enclosed by a curve.

A Simple Spline Curve

Curve Fitting Toolbox can handle *vector-valued* splines. A d-vector-valued univariate spline provides a curve in d-space. In this mode, `d = 2` is most common, as it gives plane curves.

Here is an example, in which a spline with 2-dimensional coefficients is constructed and plotted.

```
knots = [1,1:9,9];
curve = spmak( knots, repmat([ 0 0; 1 0; 1 1; 0 1 ], 2,1).' );

t = linspace(2,8,121);
values = fnval(curve,t);
plot(values(1,:),values(2,:),'LineWidth',2);
axis([-.4 1.4 -.2 1.2]), axis equal
title('A Spline Curve');
hold on
```
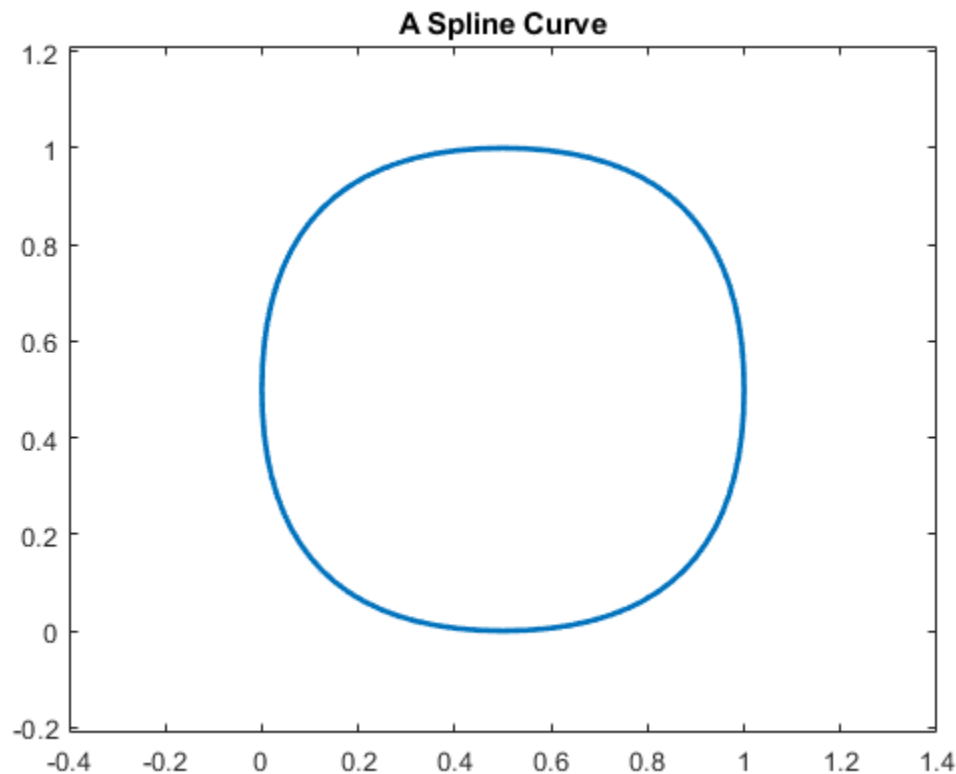

A Word of Caution

You may have noticed that this example did not use `fnplt` to plot the curve, but instead plotted some points on the curve obtained by `fnval`. Here is the code again:

```
t = linspace(2,8,121);
values = fnval(curve,t);
plot(values(1,:),values(2,:),'LineWidth',2)
```

Using `fnplt` directly with this particular spline curve gives the red curve in the figure below.

```
fnplt(curve,'r',.5);
title('The Full Spline Curve, in Red')
```
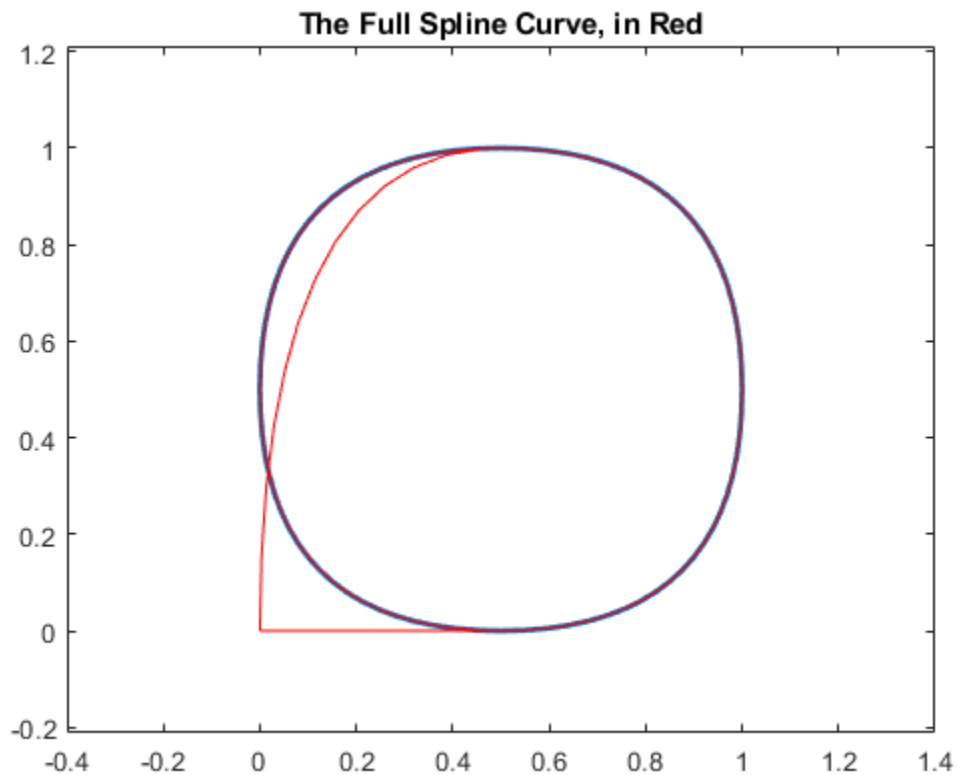

The explanation?

The spline is of order 4, yet the end knots in the knot sequence

`knots`

```
knots =

    1    1    2    3    4    5    6    7    8    9    9
```

only have multiplicity 2. Therefore, all the B-splines of order 4 for this knot sequence are 0 at the endpoints of the basic interval. This makes the curve start and stop at (0,0).

A Remedy

Since, in this case, we are really interested only in the curve segment corresponding to the parameter interval [3 .. 7], we can use `fnbrk` to extract that part, and then have no difficulty plotting it, in yellow, with `fnplt`.

```
mycv = fnbrk(curve,[3 7]);
fnplt(mycv,'y',2.5);
title('The Spline Curve of Interest, in Yellow')
```
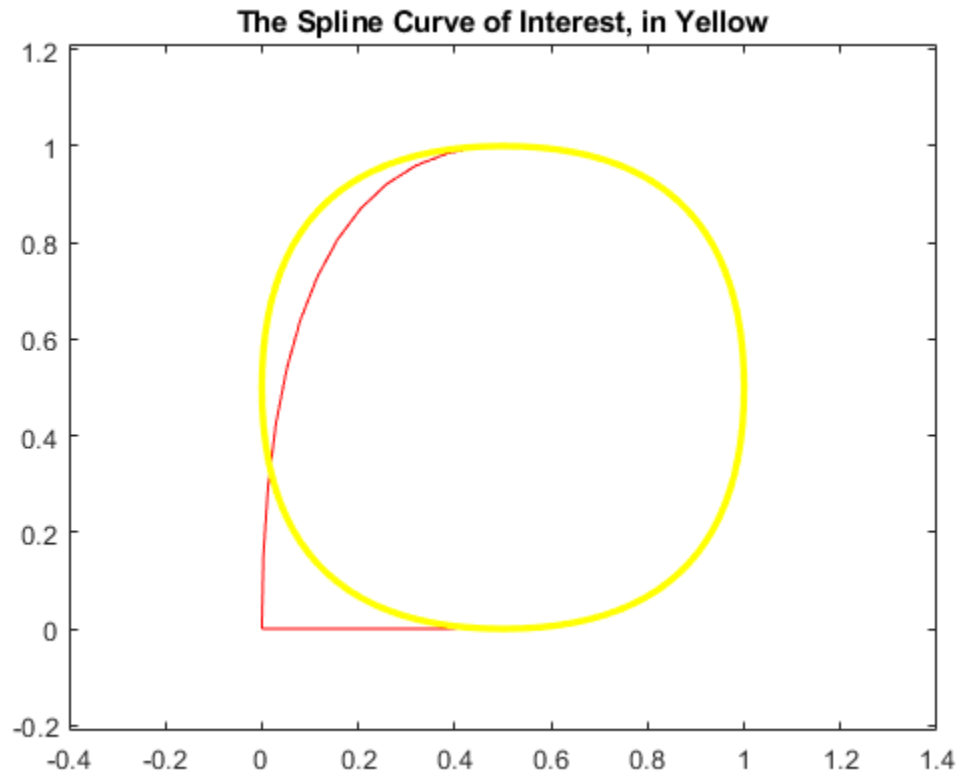

The Area Enclosed By This Curve

Since you now have a spline, namely `mycv`, that describes the curve (and nothing else), you can easily compute the area enclosed by this closed curve, as follows.

```
area = diff(fnval(fnint( ...
        fncmb(fncmb(mycv,[0 1]),'*',fnder(fncmb(mycv,[1 0])))) ...
                ),fnbrk(mycv,'interval')))
```

```
area =

   -0.8333
```

With a little effort, you can recognize this as the value of the integral

```
int y(t) d(x(t)) = int y(t) Dx(t) dt
```

over the basic interval of the spline `mycv`, with `(x(t),y(t)) := fnval(mycv,t)` the point on the curve corresponding to the parameter value `t`. Here, `fncmb(mycv,[1,0])`, `fncmb(mycv,[0,1])` describe the two components of the spline curve, i.e., the scalar-valued splines `x` and `y`.

Also, the curve is roughly a circle with radius 1/2. Hence, you would expect an area of, roughly,

```
disp(pi/4)
```

```
    0.7854
```

But why is the computed area *negative*? Because the area enclosed by the curve lies to the left as one travels on the curve with increasing `t`. To verify this, we draw some tangent vectors.

Add Some Tangent Vectors

We redraw the curve and also draw the tangent vector to the curve at some points.

```
hold off
fnplt(mycv,'y',2.5); hold on
t = 3:.4:6.2;
cv = fnval(curve, t);
cdv = fnval(fnder(curve), t);
quiver(cv(1,:),cv(2,:), cdv(1,:),cdv(2,:));
title('A Spline Curve With Some Tangents')
axis([-.4 1.4 -.2 1.2]), axis equal
```


A Spline Curve With Some Tangents
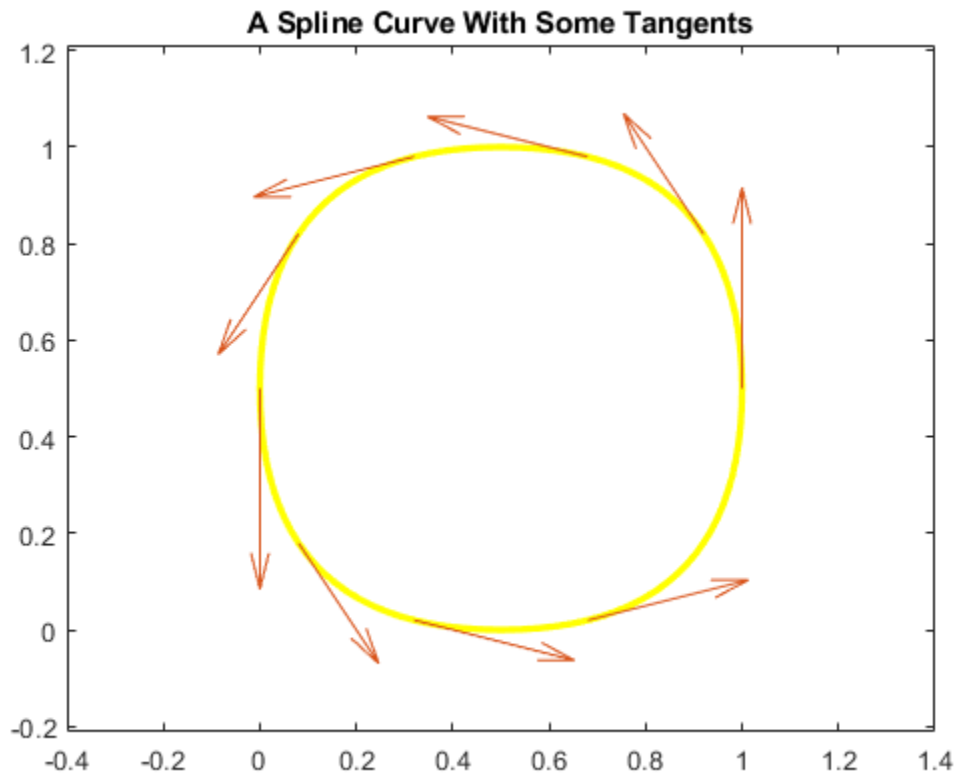
The Intersection of the Curve With a Straight Line

If you wanted to determine the points of intersection of this spline curve with the straight line y = x, the following code would give them to you, and plot the segment of that straight line inside the curve:

```
cuts = fnval(mycv, ...
    mean(fnzeros(fncmb(fncmb(mycv,[0,1]),'-',fncmb(mycv,[1,0])))));
plot(cuts(1,:), cuts(2,:),'y','LineWidth',2.5)
hold off
title('A Spline Curve With Some Tangents and a Cut Across')
```
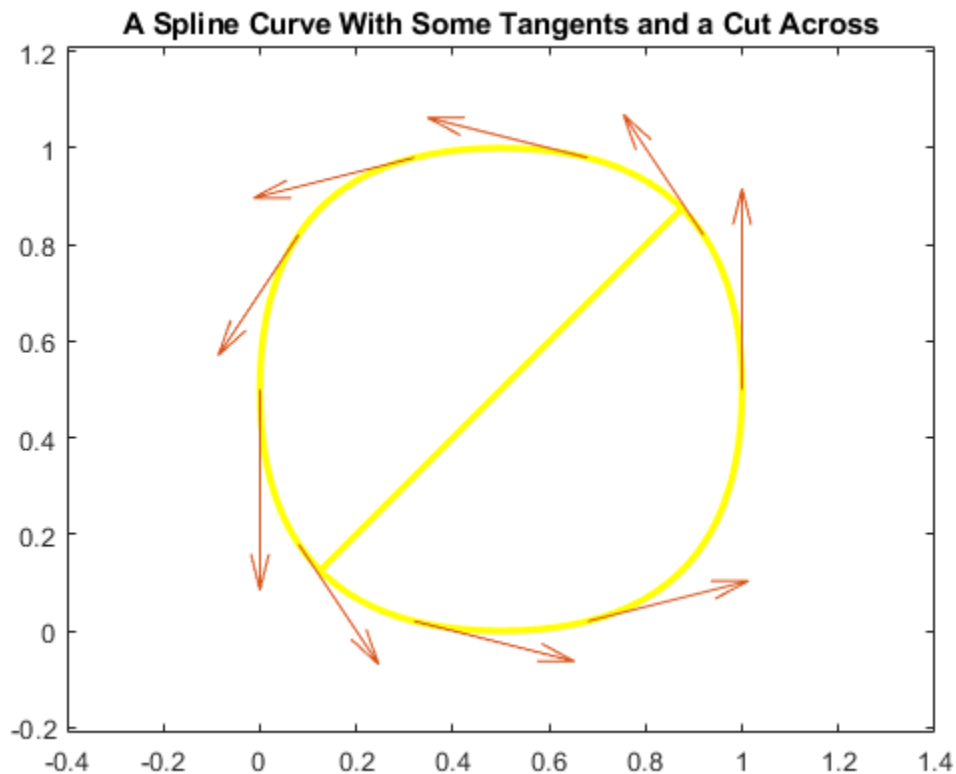

SPCRV: The Control Polygon and the Corresponding Spline Curve

Spline curves are used extensively in the generation of illustrations in which nothing more than a smooth curve of a certain roughly imagined shape is required. For this, Curve Fitting Toolbox contains a special command, spcrv, which can be used independently of the rest of the toolbox.

Given a sequence of points in the plane and, optionally, an order k, spcrv generates, by repeated midpoint knot insertion, the spline curve of order k whose control polygon is specified by the given sequence.

The figure below shows such a control polygon, and the corresponding spline curve of order 3.

```
points = [0 0; 1 0; 1 1; 0 2; -1 1; -1 0; 0 -1; 0 -2].';
values = spcrv(points,3);

plot(points(1,:),points(2,:),'k');
```

```
axis([-2 2.25 -2.1 2.2]);
hold on
plot(values(1,:),values(2,:),'r','LineWidth',1.5);
legend({'Control Polygon' 'Quadratic Spline Curve'},  'location','SE');
```
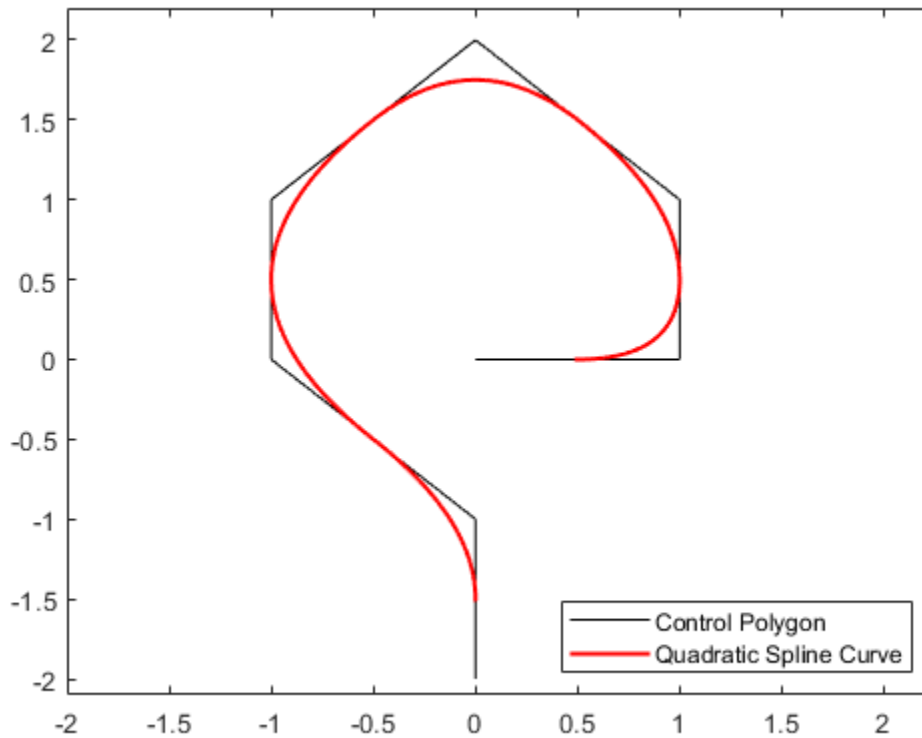

Notice that the curve touches each segment of the control polygon at its midpoint, and follows the shape outlined by the control polygon.

Raising the Order

Raising the order k will pull the curve away from the control polygon and make it smoother, but also shorter. Here, we have added the corresponding spline curve of order 4.

```
value4 = spcrv(points,4);
plot(value4(1,:),value4(2,:),'b','LineWidth',2);
legend({'Control Polygon' 'Quadratic Spline Curve' ...
        'Cubic Spline Curve'}, 'location','SE');
```
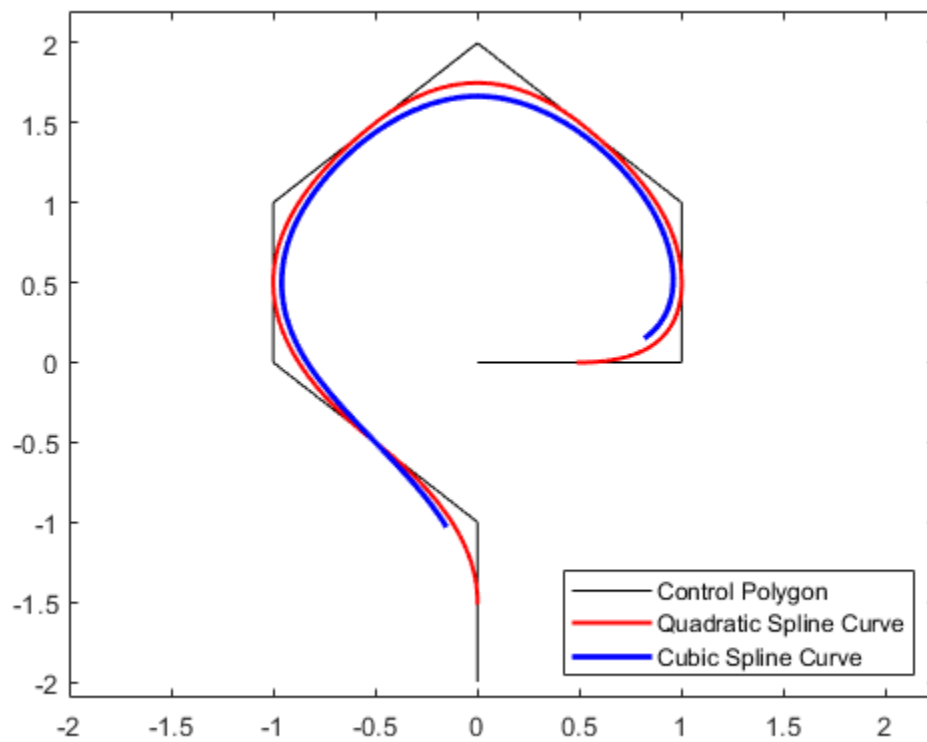
CSCVN

On the other hand, to obtain an interpolating curve, you could use the `cscvn` command, which provides a parametric `natural' cubic spline curve.

```
fnplt(cscvn(points), 'g',1.5);
legend({'Control Polygon' 'Quadratic Spline Curve' ...
        'Cubic Spline Curve' 'Interpolating Spline Curve'}, ...
        'location','SE');
```
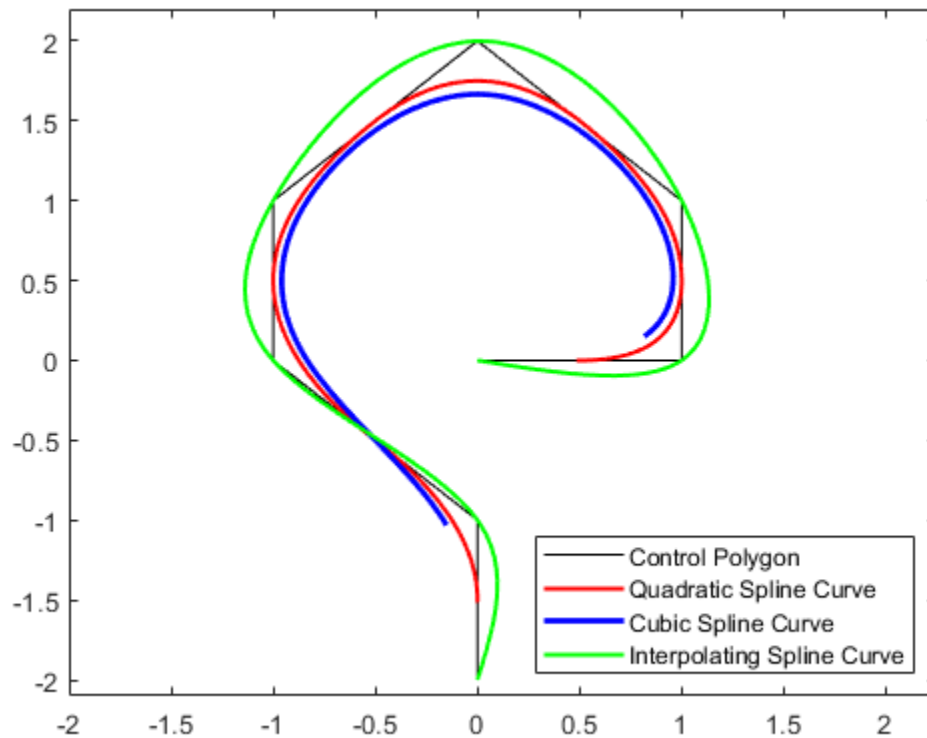
By adding the point (.95,-.05) near the second control point, (1,0), we can create an interpolating spline curve that turns faster there.

```
np = size(points, 2);
fnplt( cscvn([ points(:,1) [.95; -.05] points(:,2:np) ]), 'm',1.5);
plot(.95,-.05,'*');
legend({'Control Polygon' 'Quadratic Spline Curve' ...
        'Cubic Spline Curve' 'Interpolating Spline Curve' ...
        'Faster Turning Near (1,0)'}, ...
        'location','SE');
hold off
```
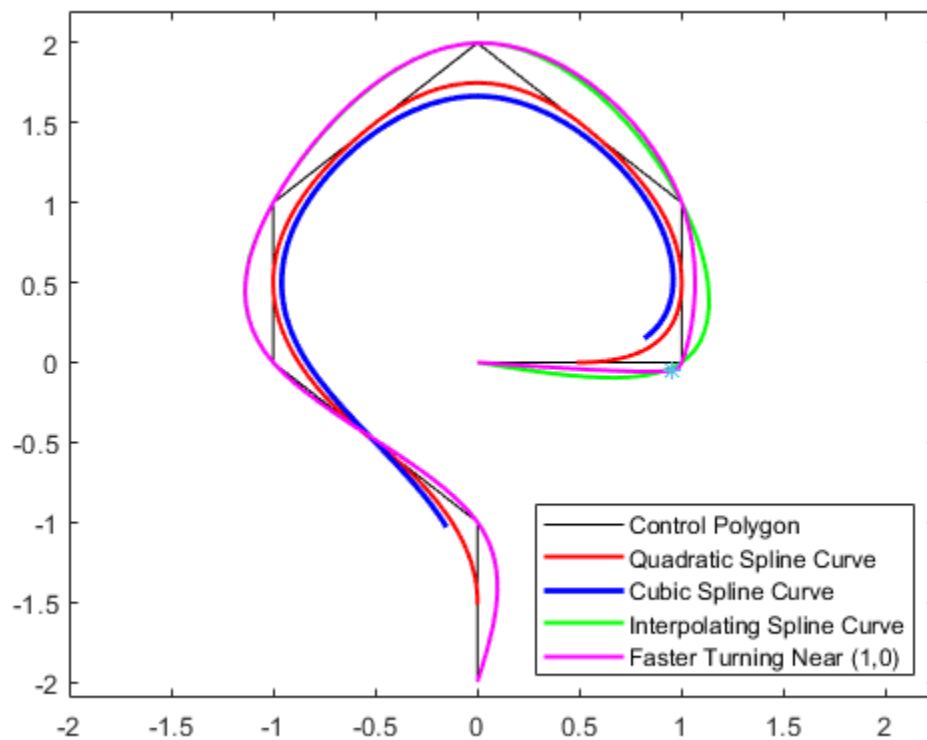
RSCVN

You can also obtain a tangent-continuous curve composed of circular arcs that passes through a given sequence of points in the plane and, optionally, is orthogonal to given normal directions at the points. The command `rscvn` provides such a curve.

For example, the following generates a circle

```
c = rscvn([-1 1 -1;0 0 0],[1 1;0 0]);
```

as its plot shows.

```
fnplt(c);
axis([-1.05 1.05 -1.05 1.05]), axis equal, axis off
```
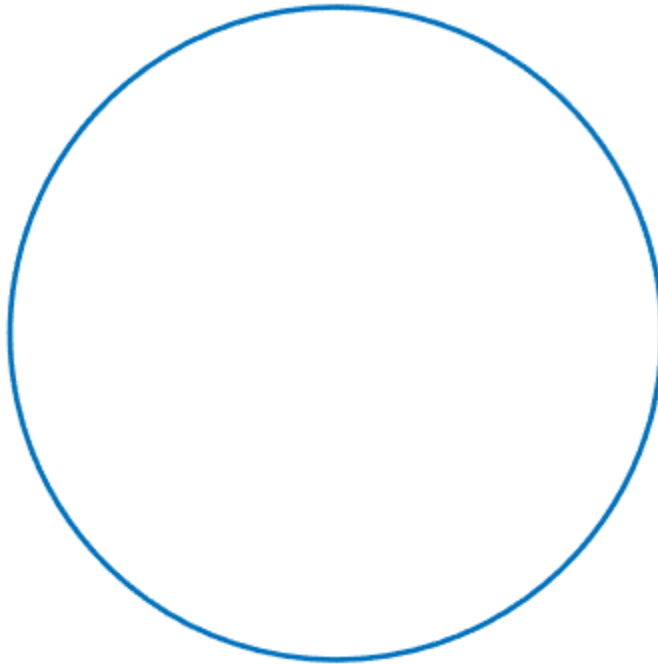
c is a quadratic rational spline consisting of just two pieces, as the following commands make clear.

```
[form, order, breaks] = fnbrk(c,'f','o','b')
```

```
form =

    'rBform'

order =

    3

breaks =

    0    2    4
```

It is easy to generate striking patterns with this tool using just a few data points. For example, here is a version of the design on the Bronze Triskele Medallion in the Ulster Museum in Belfast, supposedly done by pieces of circular arcs a long time ago.

```
pp =[zeros(1,7); 5.4, 3, 6.9, 2.75, 2.5, .5, 5];
alpha = 2*pi/3; ca = cos(alpha); sa = sin(alpha); c = [ca sa;-sa ca];
d = [0 0 .05 -.05;1 -1 .98 .98]; d = [d c*d];
yin = rscvn([pp(:,[7,1:3]),c*pp(:,3:4),pp(:,3)], d(:,[1 2 1 4 7 5 1]));
```

```
fnplt(yin), hold on, fnplt(fncmb(yin,c)), fnplt(fncmb(yin,c'))
yang = rscvn([pp(:,6),-pp(:,6),pp(:,5),c*pp(:,4)],[d(:,[2 1 1]),c(:,2)]);
fnplt(yang), fnplt(fncmb(yang,c)), fnplt(fncmb(yang,c'))
axis([-7.2 7.2 -7.2 7.2]), axis equal, axis off, hold off
```

Constructing Spline Curves in 2D and 3D

This example shows how to use the `cscvn` command from Curve Fitting Toolbox™ to construct cubic spline curves in two and three dimensions.

Selecting the Points

This example will show how to draw a smooth curve through a list of points, in the order in which they occur. First, we select some random points in the plane, and store them in a matrix, one point per *column*.

```
npts = 10;
xy = [randn(1,npts); randn(1,npts)];
plot(xy(1,:),xy(2,:),'ro','LineWidth',2);
text(xy(1,:), xy(2,:),[repmat('  ',npts,1), num2str((1:npts)')])
ax = gca;
ax.XTick = [];
ax.YTick = [];
```
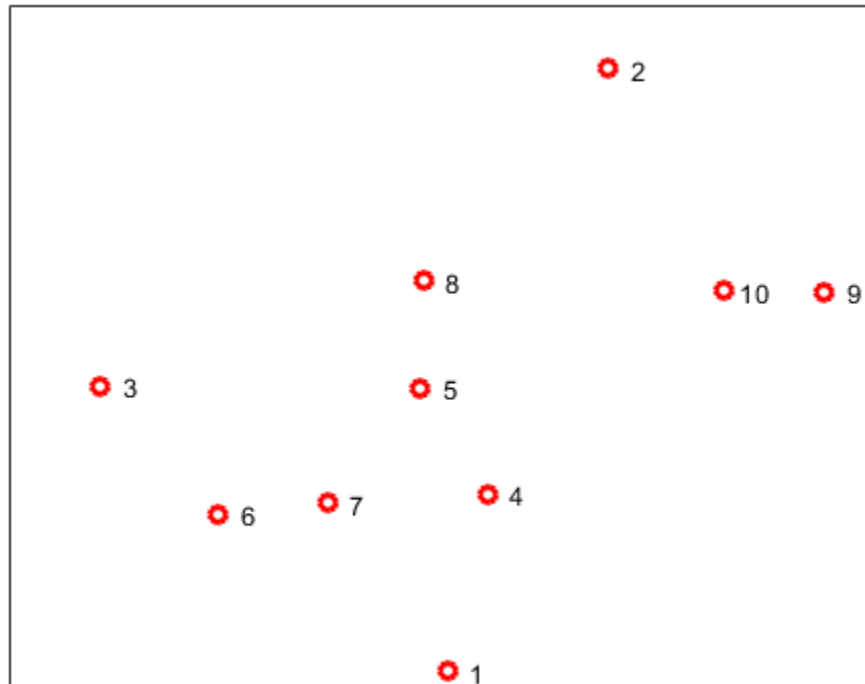

Connecting the Points

Next, construct the curve using the `cscvn` command and plot it using `fnplt`.

```
hold on
fnplt(cscvn(xy),'r',2)
hold off
```
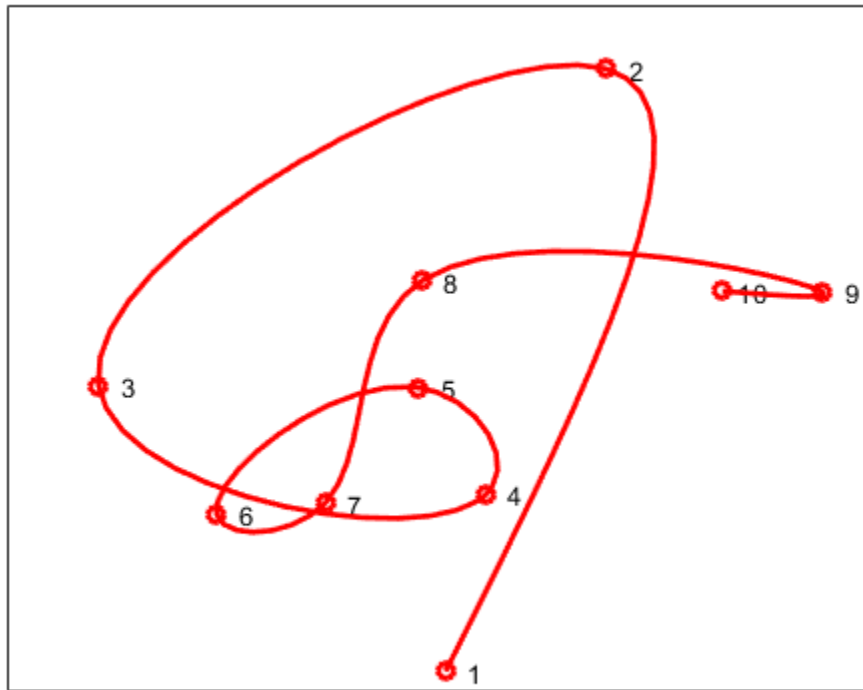
You could also use the `getcurve` command if you wanted to input the list of points interactively.

3-D Spline Curves

It's just as easy to create spline curves in three dimensions. This time, we'll do something less random. First, we generate the points.

```
npts = 13;
t = linspace(0,8*pi,npts);
z = linspace(-1,1,npts);
omz = sqrt(1-z.^2);
xyz = [cos(t).*omz; sin(t).*omz; z];
plot3(xyz(1,:),xyz(2,:),xyz(3,:),'ro','LineWidth',2);
text(xyz(1,:),xyz(2,:),xyz(3,:),[repmat('  ',npts,1), num2str((1:npts)')])
ax = gca;
ax.XTick = [];
ax.YTick = [];
ax.ZTick = [];
box on
```
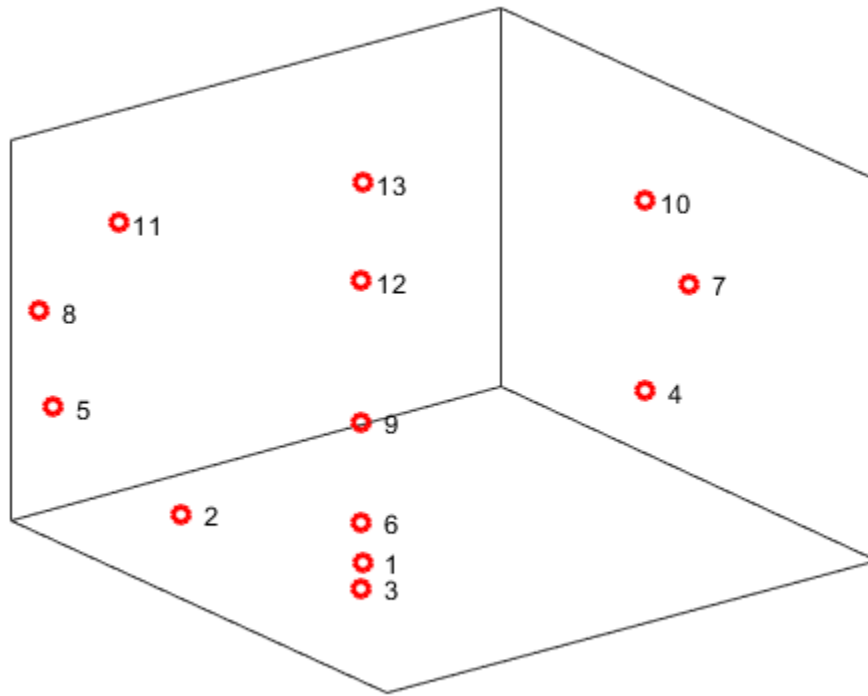
Connecting the Points

Here is the 3D spline curve through these points provided by `cscvn`. By appending the first point to the end of the list, we get a smooth *closed* curve.

```
hold on
fnplt(cscvn(xyz(:,[1:end 1])),'r',2)
hold off
```
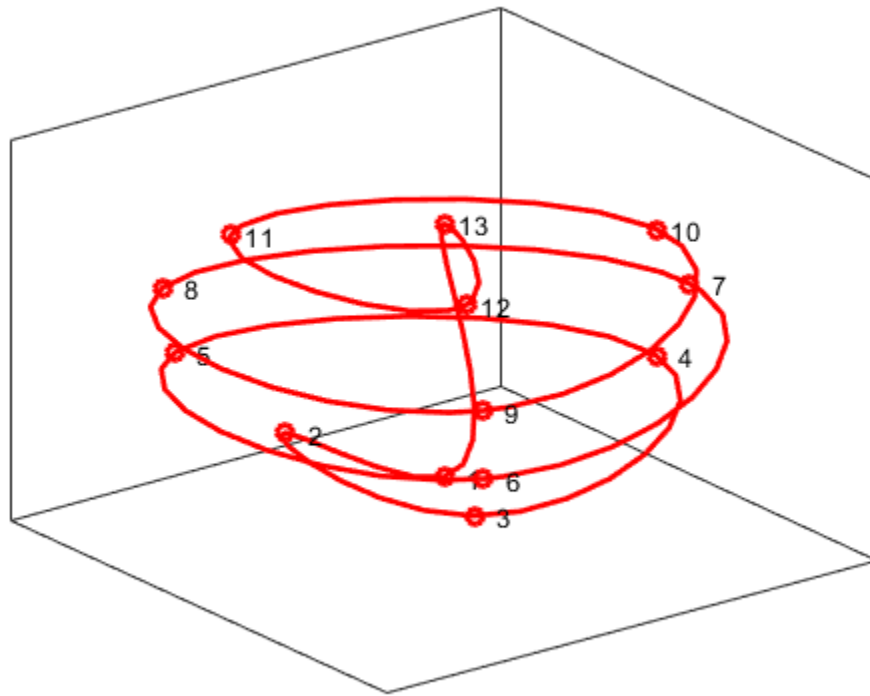
Smoothing a Histogram

This example shows how to use spline commands from Curve Fitting Toolbox™ to smooth a histogram.

Here is a histogram of some random values that might represent data that were collected on some measurement.

```
y = randn(1,5001);
hist(y);
```
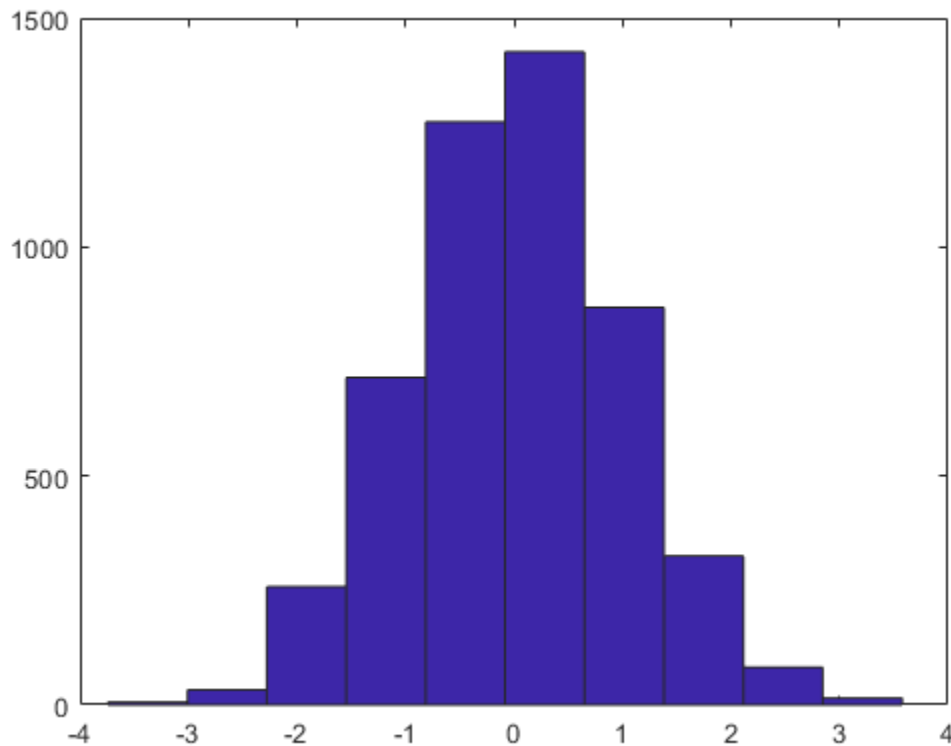

We would like to derive from this histogram a smoother approximation to the underlying distribution. We do this by constructing a spline function f whose average value over each bar interval equals the height of that bar.

If h is the height of one of these bars, and its left and right edges are at L and R, then we want the spline f to satisfy

```
integral {f(x) : L < x < R}/(R - L) = h,
```

or, with F the indefinite integral of f, i.e., DF = f,

```
F(R) - F(L) = h*(R - L).
```

```
[heights,centers] = hist(y);
hold on
```

```
ax = gca;
ax.XTickLabel = [];
n = length(centers);
w = centers(2)-centers(1);
t = linspace(centers(1)-w/2,centers(end)+w/2,n+1);
p = fix(n/2);
fill(t([p p p+1 p+1]),[0 heights([p p]),0],'w')
plot(centers([p p]),[0 heights(p)],'r:')
h = text(centers(p)-.2,heights(p)/2,'   h');
dep = -70;
tL = text(t(p),dep,'L');
tR = text(t(p+1),dep,'R');
hold off
```
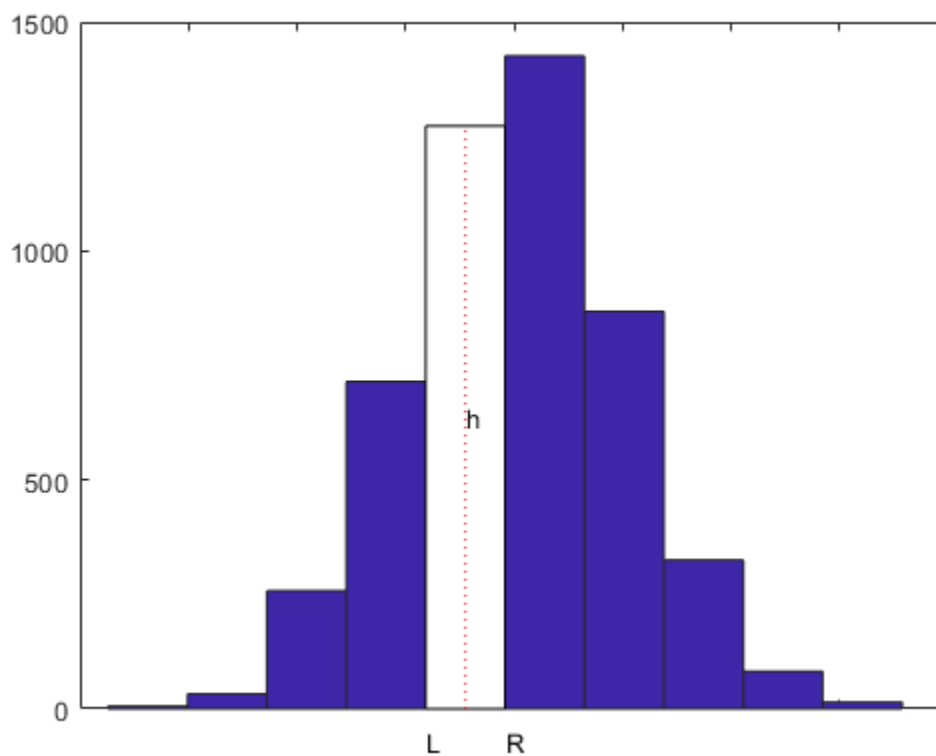

So, with `n` the number of bars, `t(i)` the left edge of the `i`-th bar, `dt(i)` its width, and `h(i)` its height, we want

`F(t(i+1)) - F(t(i)) = h(i) * dt(i), for i = 1:n,`

or, setting arbitrarily `F(t(1)) = 0`,

`F(t(i)) = sum {h(j)*dt(j) : j=1:i-1}, for i = 1:n+1.`

```
dt = diff(t);
Fvals = cumsum([0,heights.*dt]);
```

Add to this the two end conditions `DF(t(1)) = 0 = DF(t(n+1))`, and we have all the data we need to get `F` as a complete cubic spline interpolant.

```
F = spline(t, [0, Fvals, 0]);
```

The two extra zero values in the second argument indicate the zero endslope conditions.

Finally, the derivative, f = DF, of the spline F is the smoothed version of the histogram.

```
DF = fnder(F);   % computes its first derivative
h.String = 'h(i)';
tL.String = 't(i)';
tR.String = 't(i+1)';
hold on
fnplt(DF, 'r', 2)
hold off
ylims = ylim;
ylim([0,ylims(2)]);
```
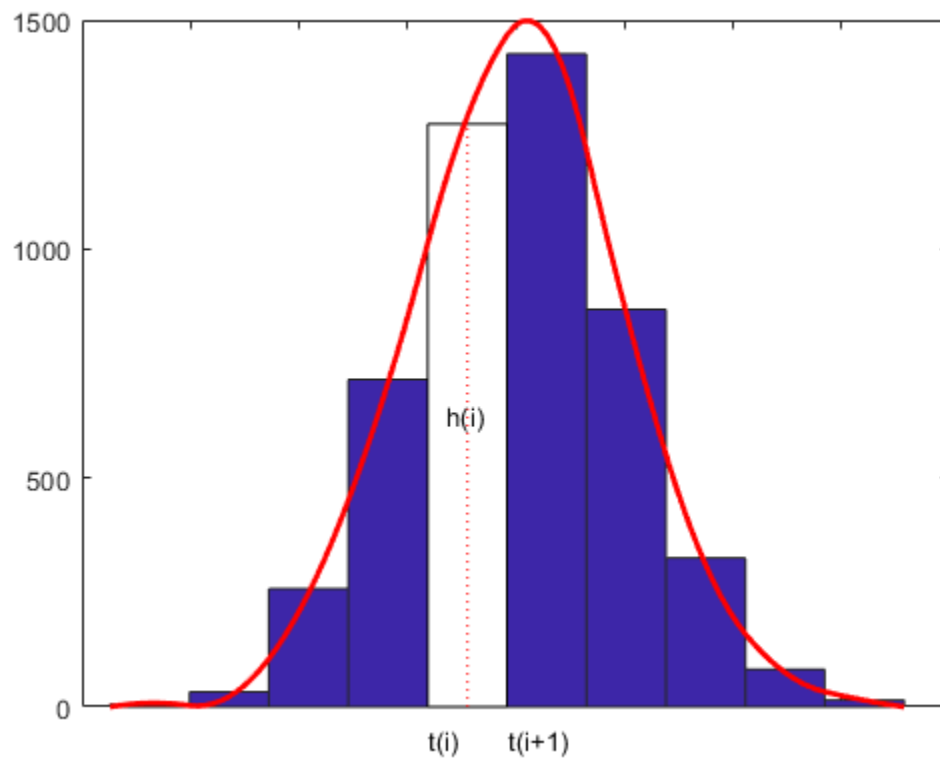
Bivariate Tensor Product Splines

This example shows how to use the spline commands in Curve Fitting Toolbox™ to fit tensor product splines to bivariate gridded data.

Introduction

Since Curve Fitting Toolbox can handle splines with *vector* coefficients, it is easy to implement interpolation or approximation to gridded data by tensor product splines. Most spline construction commands in the toolbox take advantage of this.

However, you might be interested in seeing a detailed description of how approximation to gridded data by tensor products is actually done for bivariate data. This will also come in handy when you need some tensor product construction not provided by the commands in the toolbox.

Example: Least-Squares Approximation to Gridded Data

Consider, for example, least-squares approximation to given data

z(i,j) = f(x(i),y(j)) for i = 1:I, j = 1:J.

Here are some gridded data, taken from Franke's sample function. Note that the grid is somewhat denser near the boundary, to help pin down the approximation there.

```
x = sort([(0:10)/10,.03 .07, .93 .97]);
y = sort([(0:6)/6,.03 .07, .93 .97]);
[xx,yy] = ndgrid(x,y); % note: ndgrid rather than meshgrid
z = franke(xx,yy);

mesh(x,y,z.');
xlabel('x'); ylabel('y');
view(150,50);
title('Data from the Franke Function');
```

Data from the Franke Function
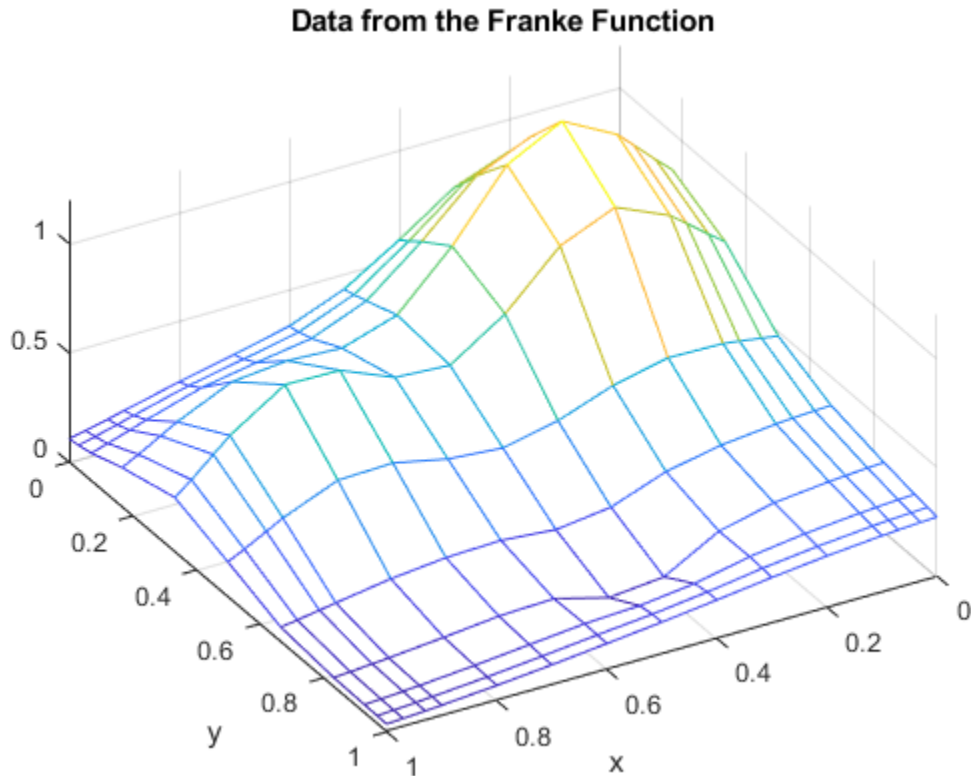
A note about NDGRID vs. MESHGRID

Note that the statements

```
[xx,yy] = ndgrid(x,y);
```

```
z = franke(xx,yy);
```

used above make certain that $z(i,j)$ is the value of the function being approximated at the grid point $(x(i),y(j))$.

However, the MATLAB® command `mesh(x,y,z)` expects $z(j,i)$ (note the reversed order of `i` and `j`) as the value at the grid point $(x(i),y(j))$. For that reason, the above plot was generated by the statement

```
mesh(x,y,z.');
```

i.e., using the transpose of the matrix `z`.

Such transposing would not have been necessary had we used `meshgrid` instead of `ndgrid`. But the resulting `z` would not have followed approximation theory standards.

Choice of Spline Space in the Y-Direction

Next, we choose a spline order `ky` and a knot sequence `knotsy` for the y-direction

```
ky = 3;
knotsy = augknt([0,.25,.5,.75,1],ky);
```

and then obtain

```
sp = spap2(knotsy,ky,y,z);
```

a spline curve whose `i`-th component is an approximation to the curve `(y,z(i,:))` for `i=1:I`.

Evaluation

In particular,

```
yy = -.1:.05:1.1;
vals = fnval(sp,yy);
```

creates the matrix `vals` whose `(i,j)`-th element can be taken as an approximation to the value `f(x(i),yy(j))` of the underlying function `f` at the grid point `(x(i),yy(j))`. This is evident when we plot `vals`.

```
mesh(x,yy,vals.');
xlabel('x'); ylabel('y');
view(150,50);
title('Simultaneous Approximation to All Curves in the Y-Direction');
```


Simultaneous Approximation to All Curves in the Y-Direction
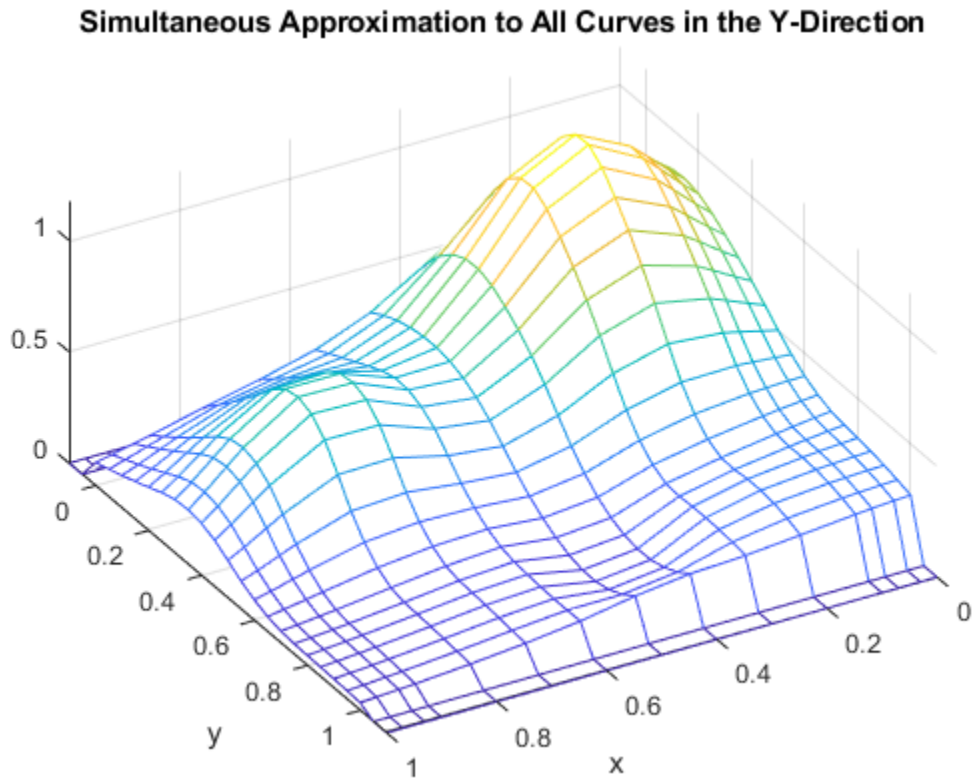
Note that, for each `x(i)`, both the first two and the last two values are zero since both the first two and the last two sites in `yy` are outside the basic interval for the spline `sp`.

Also note the "ridges" that run along the y-direction, most noticeable near the peaks of the surface. They confirm that we are plotting smooth curves in one direction only.

From Curves to a Surface; Choosing a Spline Space in the X-Direction

To get an actual surface, we now have to go one step further. Consider the coefficients `coefsy` of the spline `sp`, as obtained by

```
coefsy = fnbrk(sp,'c');
```

Abstractly, you can think of the spline `sp` as the vector-valued function

```
y |--> sum coefsy(:,r) B_{r,ky}(y)
```

```
r
```

with the i-th element, `coefsy(i,r)`, of the vector coefficient `coefsy(:,r)` corresponding to `x(i)` for `i=1:I`. This suggests approximating each curve `(x,coefsy(:,r))` by a spline, using the same order `kx` and the same appropriate knot sequence `knotsx` for every `r`.

```
kx = 4;
knotsx = augknt(0:.2:1,kx);
sp2 = spap2(knotsx,kx,x,coefsy.');
```

The use of the `spap2` command here needs, perhaps, an explanation.

Recall that `spap2(knots,k,x,fx)` treats `fx(:,j)` as the value at `x(j)`, i.e., takes each *column* of `fx` as a data value. Since we wanted to fit the value `coefsy(i,:)` at `x(i)`, for all `i`, we have to provide `spap2` with the *transpose* of `coefsy`.

Now consider the *transpose* of the coefficient matrix of the resulting spline "curve" `sp2`, obtained as

```
coefs = fnbrk(sp2,'c').';
```

`coefs` provides the *bivariate* spline approximation

```
(x,y) |--> sum sum coefs(q,r) B_{q,kx}(x) B_{r,ky}(y)
```

```
q r
```

to the original data

```
(x(i),y(j)) |--> f(x(i),y(j)) = z(i,j).
```

We use `spcol` to provide the values `B_{q,kx}(xv(i))` and `B_{r,ky}(yv(j))` needed to evaluate this spline surface at some grid points `(xv(i),yv(j))` and then plot the values.

```
xv = 0:.025:1; yv = 0:.025:1;
values = spcol(knotsx,kx,xv)*coefs*spcol(knotsy,ky,yv).';
mesh(xv,yv,values.');
xlabel('x'); ylabel('y');
view(150,50);
title('The Spline Approximant');
```

The Spline Approximant
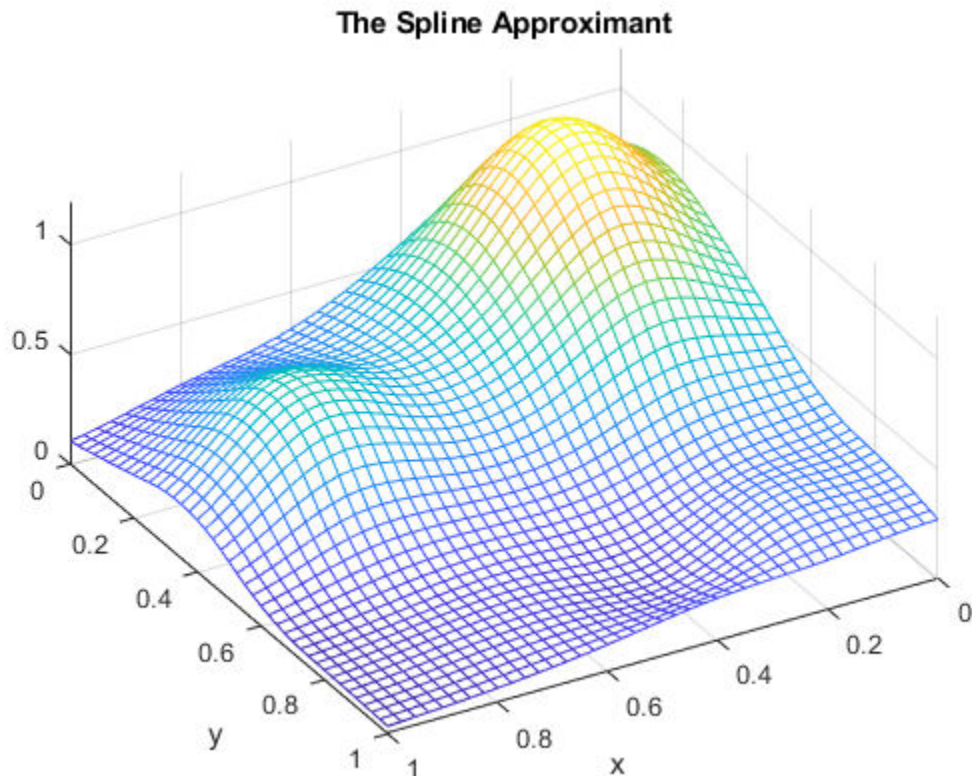
Why Does This Evaluation Work?

The statement

```
values = spcol(knotsx,kx,xv) * coefs * spcol(knotsy,ky,yv).'
```

used above makes good sense since, for example, `spcol(knotsx,kx,xv)` is the matrix whose `(i,q)`-th entry equals the value `B_{q,kx}(xv(i))` at `xv(i)` of the q-th B-spline of order `kx` for the knot sequence `knotsx`, while we want to evaluate the expression

```
sum sum coefs(q,r) B_{q,kx}(x) B_{r,ky}(y)

q r

= sum sum B_{q,kx}(x) coefs(q,r) B_{r,ky}(y)

q r
```

at `(x,y) = (xv(i),yv(j))`.

More Efficient Alternatives

Since the matrices `spcol(knotsx,kx,xv)` and `spcol(knotsy,ky,yv)` are banded, it may be more efficient for "large" `xv` and `yv` (though perhaps more memory-consuming) to make use of `fnval`.

```
value2 = fnval(spmak(knotsx,fnval(spmak(knotsy,coefs),yv).'),xv).';
```

In fact, `fnval` and `spmak` can deal directly with multivariate splines, hence the above statement can be replaced by

```
value3 = fnval( spmak({knotsx,knotsy},coefs), {xv,yv} );
```

Better yet, the construction of the approximation can be done by *one* call to `spap2`, therefore we can obtain these values directly from the given data by the statement

```
value4 = fnval( spap2({knotsx,knotsy},[kx ky],{x,y},z), {xv,yv} );
```

Check it Out

Here is a check, specifically, the *relative* difference between the values computed in these four different ways.

```
diffs = abs(values-value2) + abs(values-value3) + abs(values-value4);
max(max(diffs)) / max(max(abs(values)))
```

```
ans = 1.1206e-15
```

The four methods return the same values, up to round-off error.

Error of the Approximation

Here is a plot of the error, i.e., the difference between the given data value and the value of the spline approximation at those data sites.

```
errors = z - spcol(knotsx,kx,x)*coefs*spcol(knotsy,ky,y).';
mesh(x,y,errors.');
xlabel('x'); ylabel('y');
view(150,50);
title('Error at the Given Data Sites');
```

Error at the Given Data Sites

The *relative* error is

```
max(max(abs(errors))) / max(max(abs(z)))
```

```
ans = 0.0539
```

This is perhaps not too impressive. On the other hand, the ratio

```
(degrees of freedom used) / (number of data points)
```

is only

```
numel(coefs)/numel(z)
```

```
ans = 0.2909
```

Apparent Bias of This Approach

The approach followed here seems `biased`: We first think of the given data values `z` as describing a vector-valued function of `y`, and then we treat the matrix formed by the vector coefficients of the approximating curve as describing a vector-valued function of `x`.

What happens when we take things in the opposite order, i.e., think of `z` as describing a vector-valued function of `x`, and then treat the matrix made up from the vector coefficients of the approximating curve as describing a vector-valued function of `y`?

Perhaps surprisingly, the final approximation is the same, up to roundoff. The next section contains the numerical experiment confirming that.

Doing It the Other Way Around: Start With a Spline Space in X

First, we fit a spline curve to the data, but this time with x as the independent variable, hence it is the *rows* of z which now become the data values. Correspondingly, we must supply z.' (rather than z) to spap2, and obtain

```
spb = spap2(knotsx,kx,x,z.');
```

a spline approximation to all the curves (x,z(:,j)) for j=1:J. In particular,

```
valsb = fnval(spb,xv).';
```

creates a matrix whose (i,j)-th element can be taken as an approximation to the value f(xv(i),y(j)) of the underlying function f at the grid point (xv(i),y(j)). This is evident when we plot valsb.

```
mesh(xv,y,valsb.');
xlabel('x'); ylabel('y');
view(150,50);
title('Simultaneous Approximation to All Curves in the X-Direction');
```
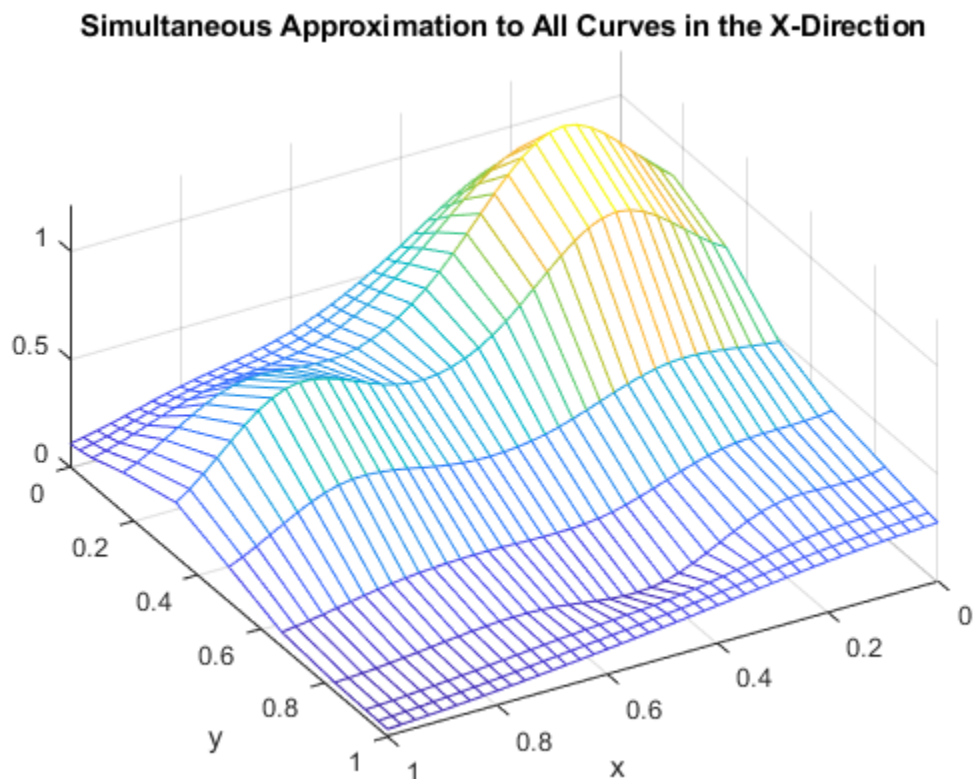

Again note the ridges, this time running along the x-direction. They confirm that, once again, we are plotting smooth curves in one direction only.

From Curves to a Surface: Using a Spline Space in the Y-Direction

Now comes the second step, to get the actual surface.

Let `coefsx` be the coefficients for `spb`, i.e.,

```
coefsx = fnbrk(spb,'c');
```

Abstractly, you can think of the spline `spb` as the vector-valued function

```
x |--> sum coefsx(r,:) B_{r,kx}(x)

r
```

with the j-th entry `coefsx(r,j)` of the vector coefficient `coefsx(r,:)` corresponding to `y(j)`, for all j. Thus, we now fit each curve (`y`,`coefsx(r,:)`) by a spline, using the same order `ky` and the same appropriate knot sequence `knotsy` for each `r`.

```
spb2 = spap2(knotsy,ky,y,coefsx.');
```

In the construction of `spb2`, we again need to transpose the coefficient matrix from `spb`, since `spap2` takes the columns of its last input argument as the data values.

For this reason, there is now no need to transpose the coefficient matrix `coefsb` of the resulting "curve".

```
coefsb = fnbrk(spb2,'c');
```

Claim: `coefsb` equals the earlier coefficient array `coefs`, up to round-off. For a proof of this, see the discussion of the tensor product construct in Curve Fitting Toolbox documentation. Here, we simply make the following check.

```
max(max(abs(coefs - coefsb)))
```

ans = 7.7716e-16

Thus, the *bivariate* spline approximation

```
(x,y) |--> sum sum coefsb(q,r) B_{q,kx}(x) B_{r,ky}(y)

q r
```

to the original data

```
(x(i),y(j)) |--> f(x(i),y(j)) = z(i,j)
```

obtained coincides with the earlier one, which generated `coefs` rather than `coefsb`.

As observed earlier, you can carry out the entire construction we just went through (in two ways), using just two statements, one for the construction of the least-squares approximant, the other for its evaluation at a rectangular mesh.

```
tsp = spap2({knotsx,knotsy},[kx,ky],{x,y},z);
valuet = fnval(tsp,{xv,yv});
```

Here, as another check, is the relative difference between the values computed earlier and those computed now:

```
max(max(abs(values-valuet))) / max(max(abs(values)))
```

ans = 3.7353e-16

Another Example: Interpolation

Since the data come from a smooth function, we should be interpolating it, i.e., using `spapi` instead of `spap2`, or, equivalently, use `spap2` with the appropriate knot sequences. For illustration, here is the same process done with `spapi`.

To recall, the (univariate) data sites were

x

x = *1×15*

| | 0 | 0.0300 | 0.0700 | 0.1000 | 0.2000 | 0.3000 | 0.4000 | 0.5000 | 0.6000 | 0.7 |

y

y = *1×11*

| | 0 | 0.0300 | 0.0700 | 0.1667 | 0.3333 | 0.5000 | 0.6667 | 0.8333 | 0.9300 | 0.9 |

We use again quadratic splines in y, hence use knots midway between data sites.

```
knotsy = augknt( [0 1 (y(2:(end-2))+y(3:(end-1)))/2 ], ky);
spi = spapi(knotsy,y,z);
coefsy = fnbrk(spi,'c');
```

Interpolation of Resulting Coefficients

We use again cubic splines in x, and use the not-a-knot condition. We therefore use all but the second and the second-to-last data points as knots.

```
knotsx = augknt(x([1,3:(end-2),end]), kx);
spi2 = spapi(knotsx,x,coefsy.');
icoefs = fnbrk(spi2,'c').';
```

Evaluation

We are now ready to evaluate the interpolant

```
ivalues = spcol(knotsx,kx,xv)*icoefs*spcol(knotsy,ky,yv).';
```

and plot the interpolant at a fine mesh.

```
mesh(xv,yv,ivalues.');
xlabel('x'); ylabel('y');
view(150,50);
title('The Spline Interpolant');
```

The Spline Interpolant
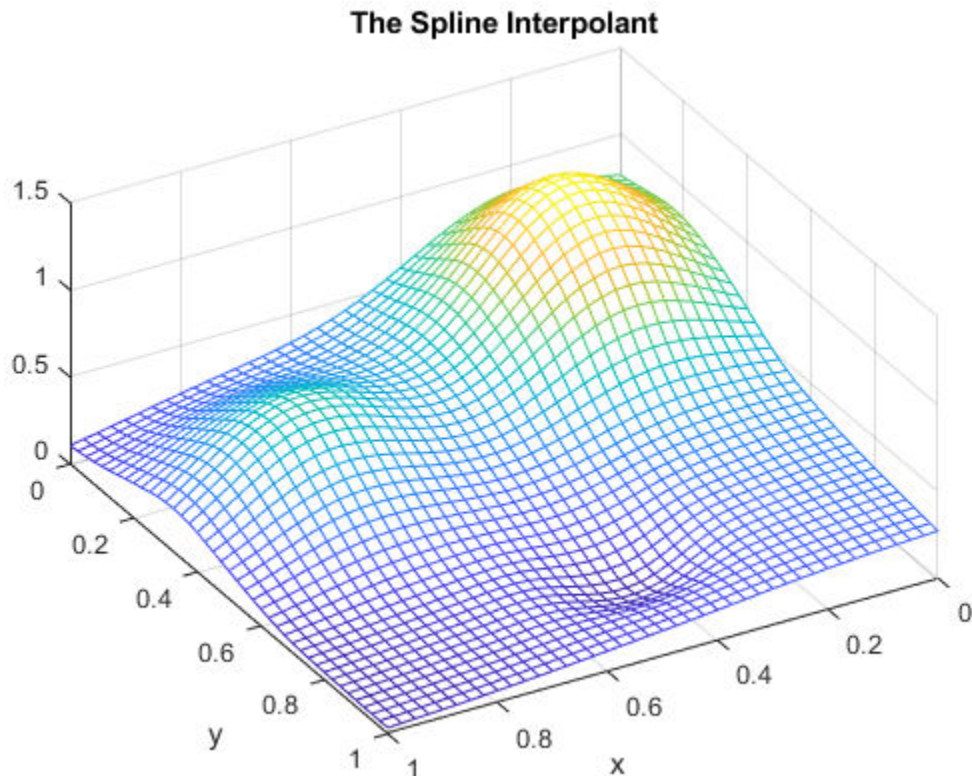
Again, the steps above can be carried out using just two statements, one for the construction of the interpolant, the other for its evaluation at a rectangular mesh.

```
tsp = spapi({knotsx,knotsy},{x,y},z);
valuet = fnval(tsp,{xv,yv});
```

For a check, we also compute the relative difference between the values computed earlier and those computed now.

```
max(max(abs(ivalues-valuet))) / max(max(abs(ivalues)))
```

```
ans = 3.6712e-16
```

Error of the Approximation

Next, we compute the error of the interpolant as an approximation to the Franke function.

```
fvalues = franke(repmat(xv.',1,length(yv)),repmat(yv,length(xv),1));
error = fvalues - ivalues;
mesh(xv,yv,error.');
xlabel('x'); ylabel('y');
view(150,50);
title('Interpolation error');
```

Interpolation error
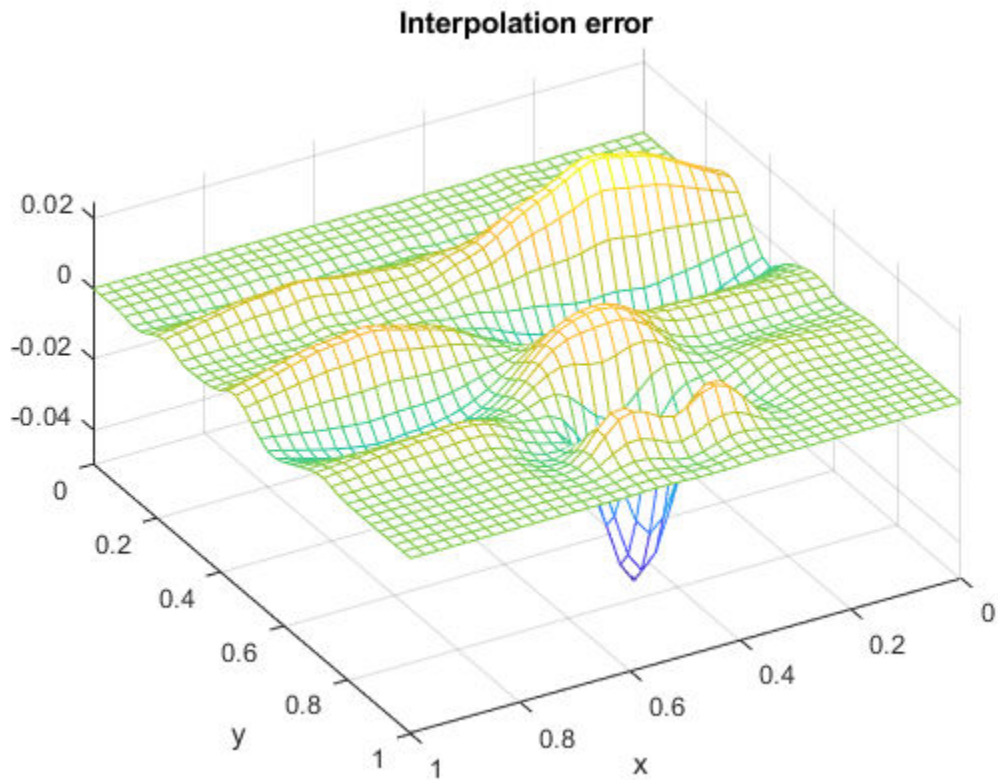
The *relative* approximation error is

```
max(max(abs(error))) / max(max(abs(fvalues)))
```

```
ans = 0.0409
```

Solving a Nonlinear ODE with a Boundary Layer by Collocation

This example shows how to use spline commands from Curve Fitting Toolbox™ solve a nonlinear ordinary differential equation (ODE).

The Problem

We consider the nonlinear singularly perturbed problem

```
epsilon D^2g(x) + (g(x))^2 = 1 on [0..1]
```

```
Dg(0) = g(1) = 0.
```

This problem is already quite difficult for `epsilon = .001`, so we choose a modest

```
epsilon = .1;
```

The Approximation Space

We seek an approximate solution by collocation from C^1 piecewise cubics with a specified break sequence `breaks`, hence want the order k to be 4.

```
breaks = (0:4)/4;
k = 4;
```

We obtain the corresponding knot sequence as

```
knots = augknt(breaks,k,2)
```

knots = *1×14*

0	0	0	0	0.2500	0.2500	0.5000	0.5000	0.7500	0.7

Whatever the choice of order and knots, the corresponding spline space has dimension

```
n = length(knots) - k
```

n = 10

Discretization

The number of degrees of freedom, 10, fits nicely with the fact that we expect to collocate at two sites per polynomial piece, for a total of 8 conditions, bringing us to 10 conditions altogether once we add the two side conditions.

We choose two Gauss sites for each interval. For the `standard' interval [-1/2 .. 1/2] of unit length, these are the two sites

```
gauss = .5773502692*[-1/2; 1/2];
```

From this, we obtain the whole collection of collocation sites by

```
ninterv = length(breaks)-1;
temp = (breaks(2:ninterv+1)+breaks(1:ninterv))/2;
temp = temp([1 1],:) + gauss*diff(breaks);
colsites = temp(:).';
```

The Numerical Problem

The numerical problem we want to solve is to find a piecewise polynomial (or *pp*) y of the given order, and with the given knots, that satisfies the nonlinear system

```
Dy(0) = 0
```

```
(y(x))^2 + epsilon D^2y(x) = 1 for x in colsites
```

```
y(1) = 0
```

Linearization

If y is our current approximation to the solution, then the linear problem for the better (?) solution z by Newton's method reads

```
Dz(0) = 0
```

```
w_0(x)z(x) + epsilon D^2z(x) = b(x) for x in colsites
```

```
z(1) = 0
```

with $w\_0(x) := 2y(x)$ and $b(x) := (y(x))^2 + 1$.

In fact, by choosing $w\_0(1) := 1, w\_1(0) := 1$, and

```
w_2(x) := epsilon, w_1(x) := 0 for x in colsites
```

and choosing all other values of w_0, w_1, w_2, and b not yet specified to be zero, we can give our system the uniform shape

```
w_0(x)z(x) + w_1(x)Dz(x) + w_2(x)D^2z(x) = b(x) for x in sites
```

where

```
sites = [0,colsites,1];
```

Linear System to be Solved

This system converts into one for the B-spline coefficients of its solution z. For this, we need the zeroth, first, and second derivative at every x in `sites` and for every relevant B-spline. These values are supplied by the `spcol` command.

Here is the essential part of the documentation for `spcol`:

```
SPCOL B-spline collocation matrix.
```

```
COLLOC = SPCOL(KNOTS,K,TAU) is the matrix
```

```
[ D^m(i)B_j(TAU(i)) : i=1:length(TAU), j=1:length(KNOTS)-K ],
```

```
with D^m(i)B_j the m(i)-fold derivative of B_j,
```

```
B_j the j-th B-spline of order K for the knot sequence KNOTS,
```

```
TAU a sequence of sites,
```

both KNOTS and TAU are assumed to be nondecreasing, and

m(i) is the integer #{ j<i : TAU(j) = TAU(i) }, i.e., the

'cumulative' multiplicity of TAU(i) in TAU.

We use `spcol` to supply the matrix

```
colmat = spcol(knots,k, brk2knt(sites,3));
```

with `brk2knt` used here to triple each entry of `sites`, and thus we get in `colmat`, for each x in `sites`, the value and the first and second derivatives at x of all the relevant B-splines.

From this, we get the collocation matrix by combining the row triple associated with x using the weights `w_0(x)`, `w_1(x)`, `w_2(x)` to get the row corresponding to x of the matrix of our linear system.

Need Initial Guess for Y

We also need a current approximation y from our spline space. Initially, we get it by interpolating some reasonable initial guess from our pp space at `sites`. For that guess, we use the parabola

x^2 - 1

which does satisfy the end conditions and lies in our spline space. We obtain its B-form by interpolation at `sites`. We select the relevant interpolation matrix from the full matrix `colmat`. Here it is, in several cautious steps:

```
intmat = colmat([2 1+(1:(n-2))*3,1+(n-1)*3],:);
coefs = intmat\[0 colsites.*colsites-1 0].';
y = spmak(knots,coefs.');
```

We plot the result, to be sure -- it should be exactly x^2-1.

```
fnplt(y,'g');
legend('Initial Guess (x^2-1)','location','NW');
axis([-0.01 1.01 -1.01 0.01]);
hold on
```
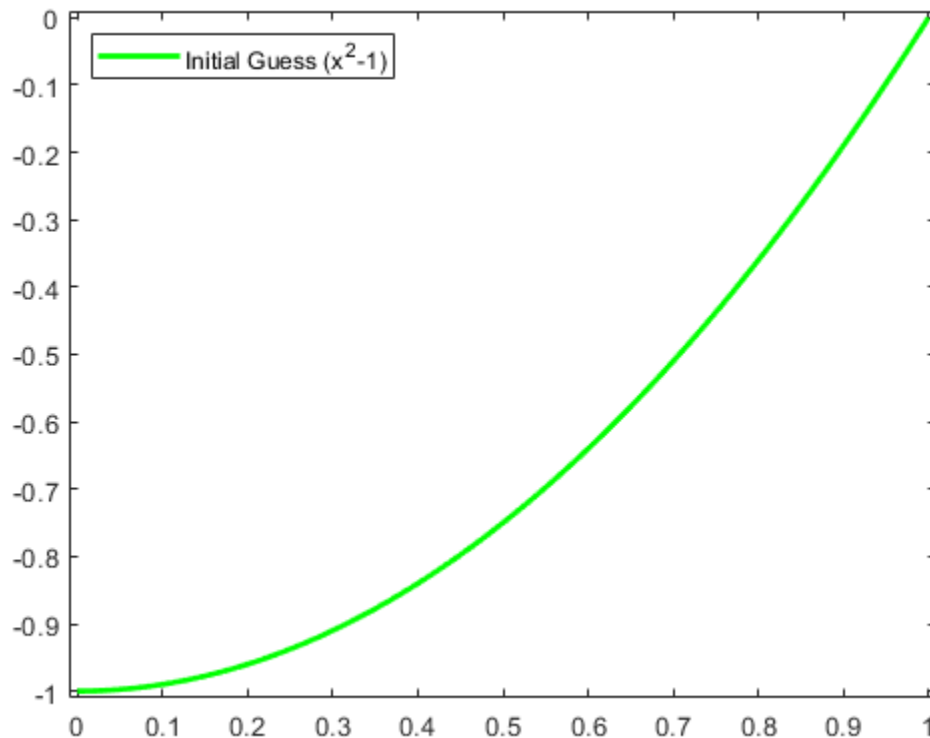
Iteration

We can now complete the construction and solution of the linear system for the improved approximate solution z from our current guess y. In fact, with the initial guess y available, we now set up an iteration, to be terminated when the change z-y is less than a specified tolerance.

```
tolerance = 6.e-9;
```

The max-norms of the change z-y at each iteration are shown as output below, and the figure shows each of the iterates.

```matlab
while 1
   vtau = fnval(y,colsites);
   weights = [0 1 0;
             [2*vtau.' zeros(n-2,1) repmat(epsilon,n-2,1)];
             1 0 0];
   colloc = zeros(n,n);
   for j = 1:n
      colloc(j,:) = weights(j,:)*colmat(3*(j-1)+(1:3),:);
   end
   coefs = colloc\[0 vtau.*vtau+1 0].';
   z = spmak(knots,coefs.');
   fnplt(z,'k');
   maxdif = max(max(abs(z.coefs-y.coefs)));
   fprintf('maxdif = %g\n',maxdif)
   if (maxdif<tolerance), break, end
```

```
    % now reiterate
    y = z;
end
```

maxdif = 0.206695

maxdif = 0.01207

maxdif = 3.95151e-05

maxdif = 4.43216e-10

```
legend({'Initial Guess (x^2-1)' 'Iterates'},'location','NW');
```
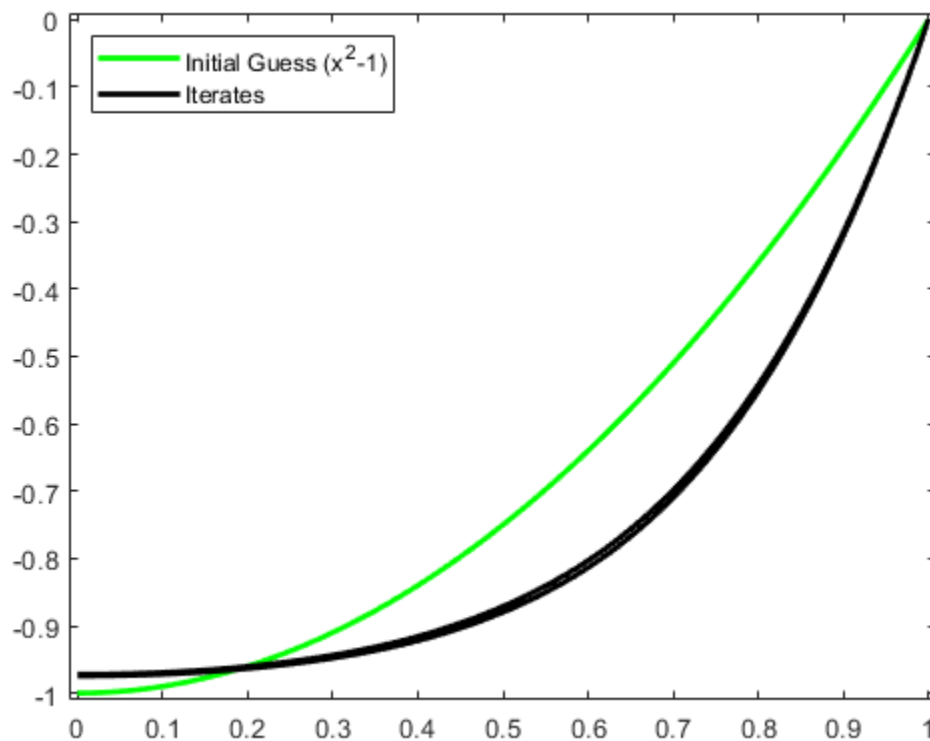

That looks like quadratic convergence, as expected from a Newton iteration.

Getting Ready for a Smaller Epsilon

If we now decrease `epsilon`, we create more of a boundary layer near the right endpoint, and this calls for a nonuniform mesh. We use `newknt` to construct an appropriate (finer) mesh from the current approximation.

```
knots = newknt(z, ninterv+1);
breaks = knt2brk(knots);
knots = augknt(breaks,4,2);
n = length(knots)-k;
```

Collocation Sites for New Breaks

Next, we get the collocation sites corresponding to the new `breaks`

```
ninterv = length(breaks)-1;
temp = ((breaks(2:ninterv+1)+breaks(1:ninterv))/2);
temp = temp([1 1], :) + gauss*diff(breaks);
colsites = temp(:).';
sites = [0,colsites,1];
```

and then the new collocation matrix.

```
colmat = spcol(knots,k, brk2knt(sites,3));
```

Initial Guess

We obtain the initial guess y as the interpolant from the current spline space to the computed solution z. We plot the resulting interpolant to be sure -- it should be close to our current solution.

```
intmat = colmat([2 1+(1:(n-2))*3,1+(n-1)*3],:);
y = spmak(knots,[0 fnval(z,colsites) 0]/intmat.');
fnplt(y,'c');
ax = gca;
h = ax.Children;
legend(h([6 5 1]),{'Initial Guess (x^2-1)' 'Iterates' ...
                   'New Initial Guess for New Value of epsilon'}, ...
                   'location','NW');
```
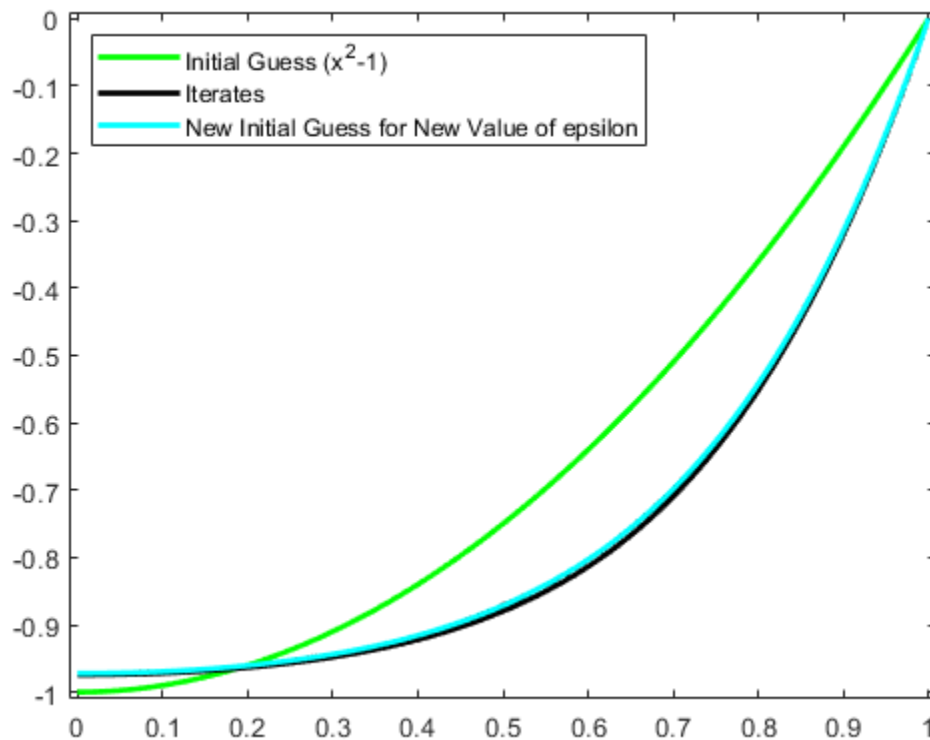
Iteration with Smaller Epsilon

Now we divide `epsilon` by 3 and repeat the above iteration. Convergence is again quadratic.

```
epsilon = epsilon/3;
while 1
   vtau = fnval(y,colsites);
   weights = [0 1 0;
              [2*vtau.' zeros(n-2,1) repmat(epsilon,n-2,1)];
              1 0 0];
   colloc = zeros(n,n);
   for j = 1:n
       colloc(j,:) = weights(j,:)*colmat(3*(j-1)+(1:3),:);
   end
   coefs = colloc\[0 vtau.*vtau+1 0].';
   z = spmak(knots,coefs.');
   fnplt(z,'b');
   maxdif = max(max(abs(z.coefs-y.coefs)));
   fprintf('maxdif = %g\n',maxdif)
   if (maxdif<tolerance), break, end

   % now reiterate
   y = z;
end
```

```
maxdif = 0.237937

maxdif = 0.0184488

maxdif = 0.000120467

maxdif = 4.78116e-09
```

```
ax = gca;
h = ax.Children;
legend(h([10 9 5 4]), ...
       {'Initial Guess (x^2-1) for epsilon = .1' 'Iterates' ...
        sprintf('Initial Guess for epsilon = %.3f',epsilon) ...
        'Iterates'}, 'location','NW');
```
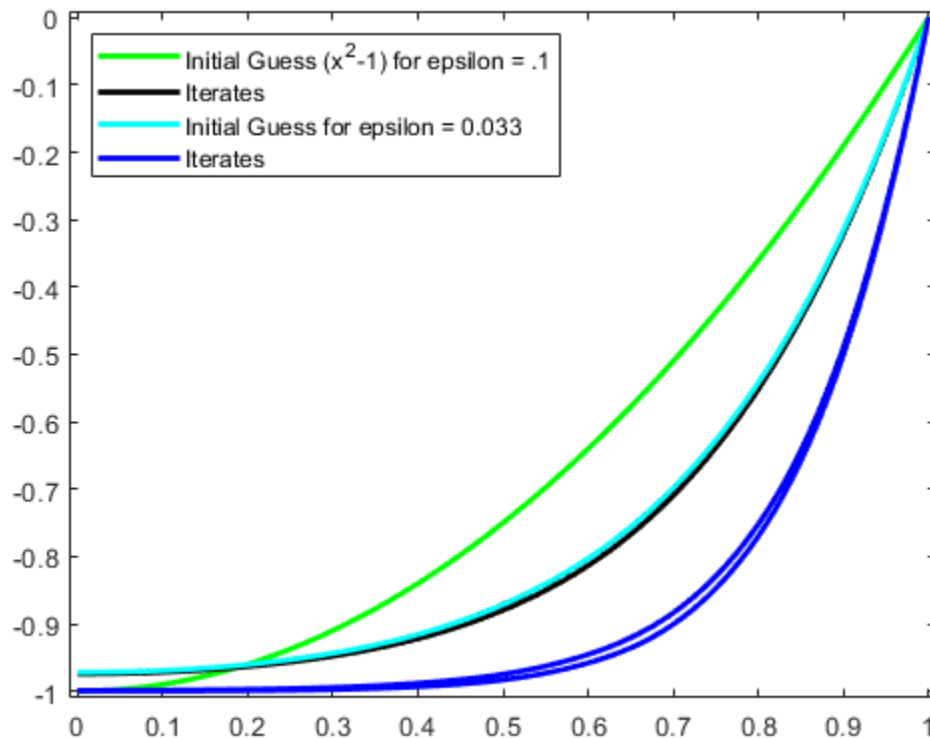
Very Small Epsilon

For a much smaller `epsilon`, we merely repeat these calculations, dividing `epsilon` by 3 each time.

```
for ee = 1:4
    knots = newknt(z, ninterv+1);
    breaks = knt2brk(knots);
    knots = augknt(breaks,4,2);
    n = length(knots)-k;

    ninterv = length(breaks)-1;
    temp = ((breaks(2:ninterv+1)+breaks(1:ninterv))/2);
    temp = temp([1 1], :) + gauss*diff(breaks);
    colsites = temp(:).';
    sites = [0,colsites,1];

    colmat = spcol(knots,k, brk2knt(sites,3));

    intmat = colmat([2 1+(1:(n-2))*3,1+(n-1)*3],:);
    y = spmak(knots,[0 fnval(z,colsites) 0]/intmat.');
    fnplt(y,'c')

    epsilon = epsilon/3;
    while 1
        vtau = fnval(y,colsites);
        weights = [0 1 0;
                   [2*vtau.' zeros(n-2,1) repmat(epsilon,n-2,1)];
                   1 0 0];
```

```
        colloc = zeros(n,n);
        for j = 1:n
         colloc(j,:) = weights(j,:)*colmat(3*(j-1)+(1:3),:);
        end
        coefs = colloc\[0 vtau.*vtau+1 0].';
        z = spmak(knots,coefs.');
        fnplt(z,'b');
        maxdif = max(max(abs(z.coefs-y.coefs)));
        if (maxdif<tolerance), break, end

        % now reiterate
        y = z;
    end
end
ax = gca;
h = ax.Children;
legend(h([30 29 25 24]), ...
        {'Initial Guess (x^2-1) for epsilon = .1' 'Iterates' ...
         'Initial Guesses for epsilon = .1/3^j, j=1:5' ...
         'Iterates'},'location','NW');
```
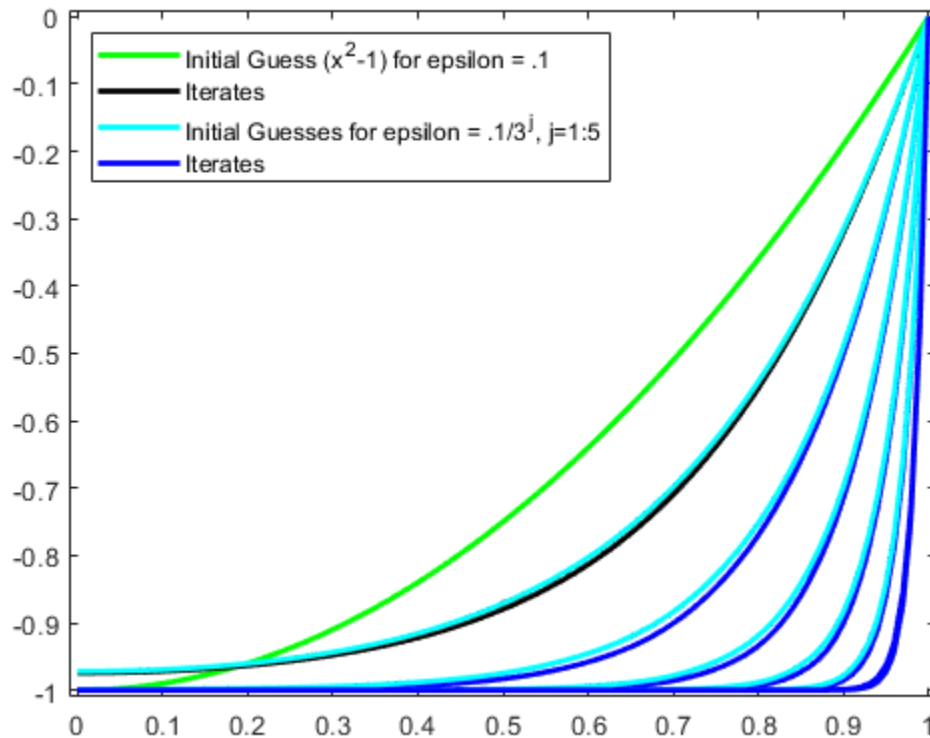

Plot the Breaks Used for Smallest Epsilon

Here is the final distribution of `breaks`, showing `newknt` to have worked well in this case.

```
breaks = fnbrk(fn2fm(z,'pp'),'b');
bb = repmat(breaks,3,1);
cc = repmat([0;-1;NaN],1,length(breaks));
```

```
plot(bb(:),cc(:),'r');
hold off
ax = gca;
h = ax.Children;
legend(h([31 30 26 25 1]), ...
        {'Initial Guess (x^2-1) for epsilon = .1' 'Iterates' ...
         'Initial Guesses for epsilon = .1/3^j, j=1:5' ...
         'Iterates' 'Breaks for epsilon = .1/3^5'},'location','NW');
```
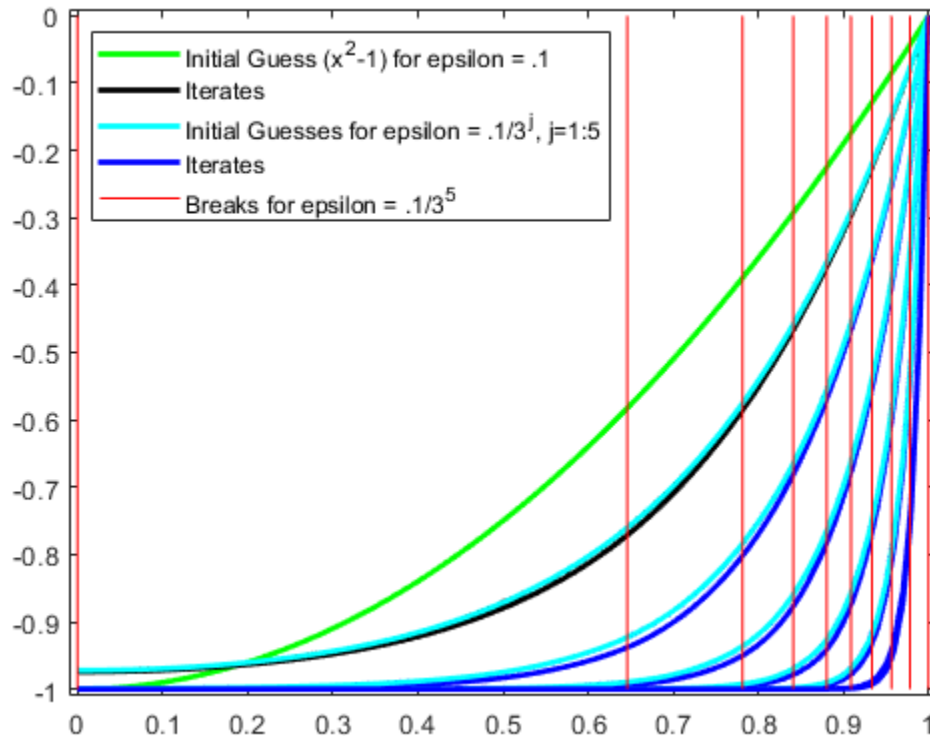

Plot Residual for Smallest Epsilon

Recall that we are solving the ODE

epsilon D^2g(x) + (g(x))^2 = 1 on [0..1]

As a check, we compute and plot the residual for the computed solution for the smallest epsilon. This, too, looks satisfactory.

```
xx = linspace(0,1,201);
plot(xx, 1 - epsilon*fnval(fnder(z,2),xx) - (fnval(z,xx)).^2)
title('Residual for the Computed Solution for Smallest epsilon');
```
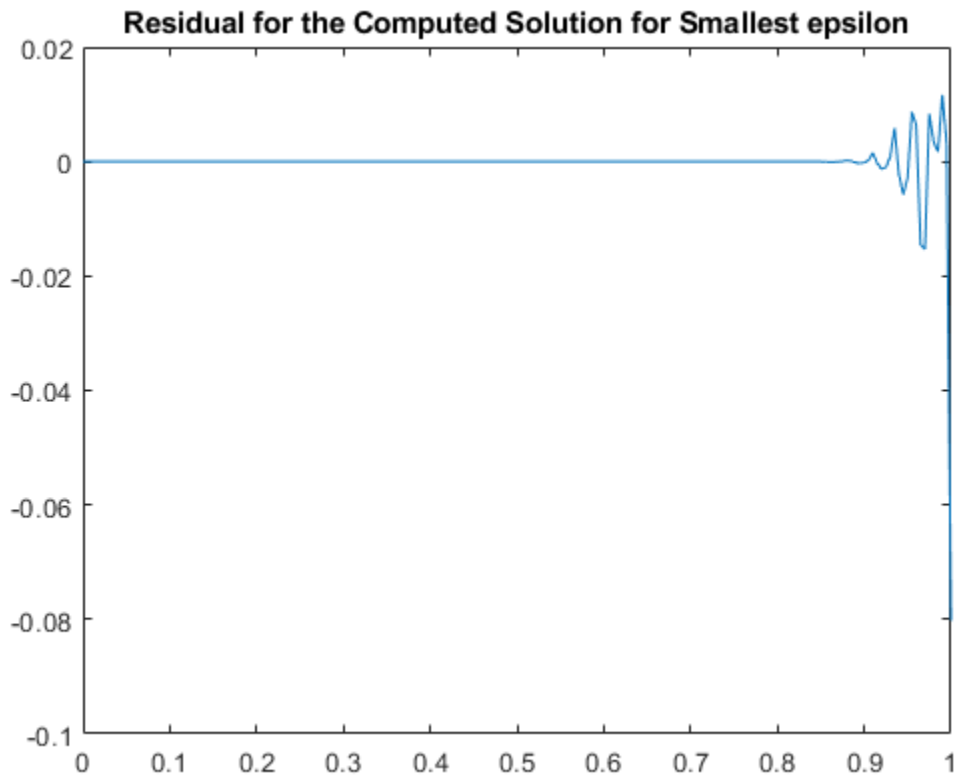
Construction of a Chebyshev Spline

This example shows how to use commands from Curve Fitting Toolbox™ to construct a Chebyshev spline.

Chebyshev (a.k.a. Equioscillating) Spline Defined

By definition, for given knot sequence t of length n+k, C = C_{t,k} is the unique element of S_{t,k} of max-norm 1 that maximally oscillates on the interval [t_k .. t_{n+1}] and is positive near t_{n+1}. This means that there is a unique strictly increasing tau of length n so that the function C in S_{k,t} given by

C(tau(i)) = (-1)^{n-i},

for all i, has max-norm 1 on [t_k .. t_{n+1}]. This implies that

tau(1) = t_k,

tau(n) = t_{n+1},

and that

t_i < tau(i) < t_{k+i},

for all i. In fact,

t_{i+1} <= tau(i) <= t_{i+k-1},

for all i. This brings up the point that the knot sequence t is assumed to make such an inequality possible, which turns out to be equivalent to having all the elements of S_{k,t} continuous.

```
t = augknt([0 1 1.1 3 5 5.5 7 7.1 7.2 8], 4 );
[tau,C] = chbpnt(t,4);
xx = sort([linspace(0,8,201),tau]);
plot(xx,fnval(C,xx),'LineWidth',2);
hold on
breaks = knt2brk(t);
bbb = repmat(breaks,3,1);
sss = repmat([1;-1;NaN],1,length(breaks));
plot(bbb(:), sss(:),'r');
hold off
ylim([-2 2]);
title('The Chebyshev Spline for a Particular Knot Sequence');
legend({'Chebyshev Spline' 'Knots'});
```
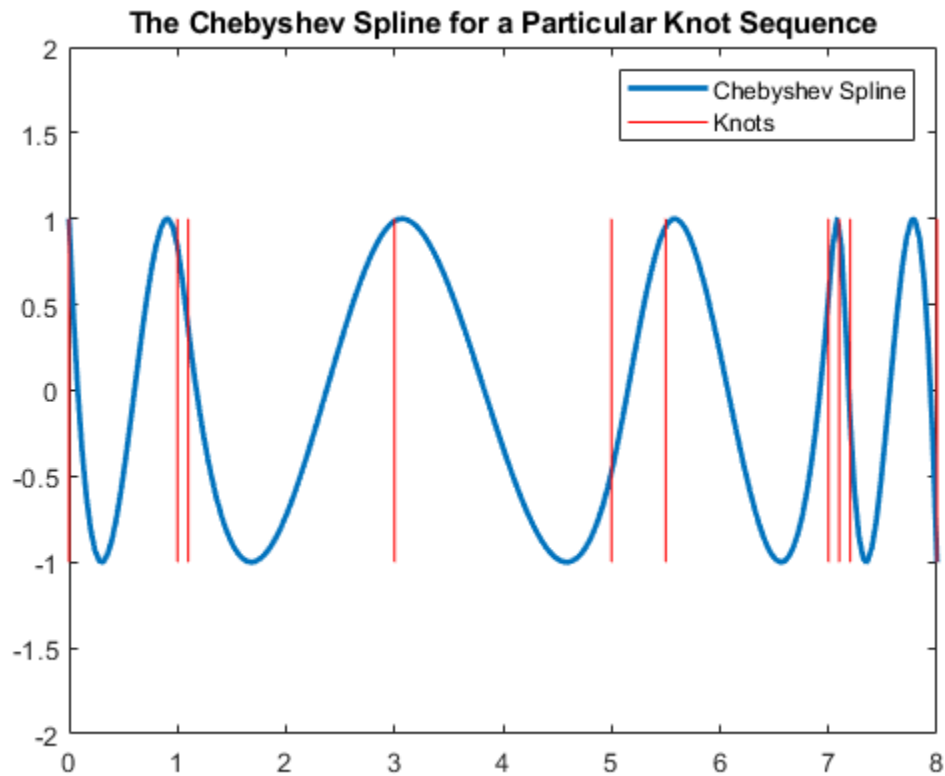
The Chebyshev Spline for a Particular Knot Sequence

In short, the Chebyshev spline `C` looks just like the Chebyshev polynomial. It performs similar functions. For example, its extrema `tau` are particularly good sites to interpolate at from `S_{k,t}` since the norm of the resulting projector is about as small as can be.

```
hold on
plot(tau,zeros(size(tau)),'k+');
hold off
legend({'Chebyshev Spline' 'Knots' 'Extrema'});
```

The Chebyshev Spline for a Particular Knot Sequence
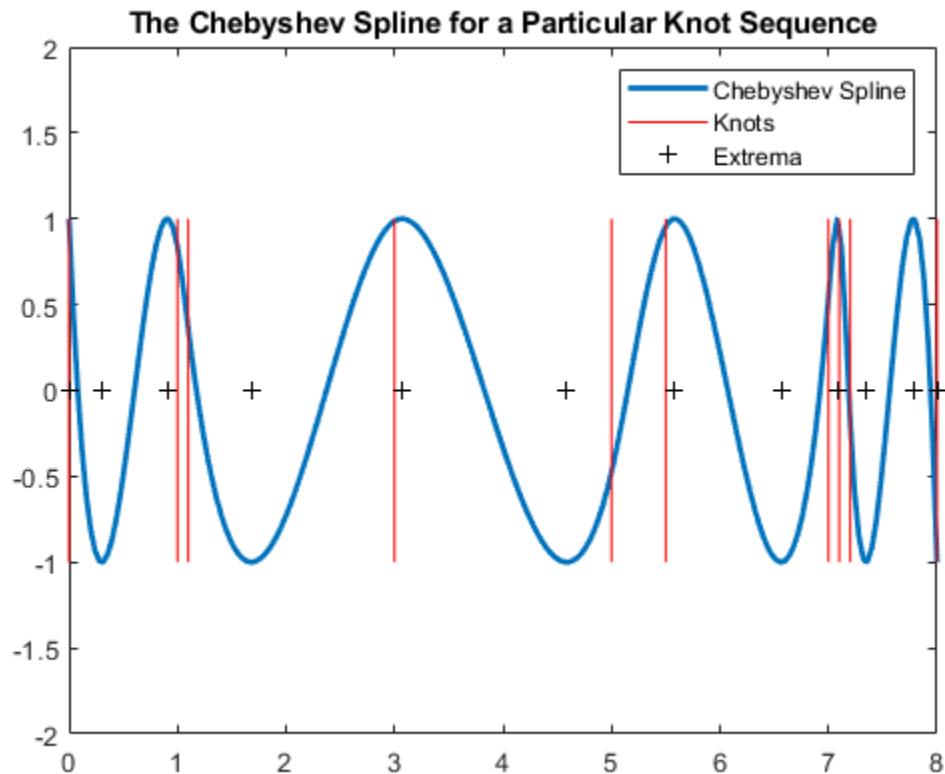
Choice of Spline Space

In this example, we try to construct C for a given spline space.

We deal with cubic splines with simple interior knots, specified by

```
k = 4;
breaks = [0 1 1.1 3 5 5.5 7 7.1 7.2 8];
t = augknt(breaks, k)
```

```
t = 1×16
```

0	0	0	0	1.0000	1.1000	3.0000	5.0000	5.5000	7.0

thus getting a spline space of dimension

```
n = length(t)-k
```

```
n = 12
```

Initial Guess

As our initial guess for the tau, we use the knot averages

```
tau(i) = (t_{i+1} + ... + t_{i+k-1})/(k-1)
```

recommended as good interpolation site choices, and plot the resulting first approximation to C.

```
tau = aveknt(t,k)
```

tau = *1×12*

```
       0    0.3333    0.7000    1.7000    3.0333    4.5000    5.8333    6.5333    7.1000    7.4
```

```
b = (-ones(1,n)).^(n-1:-1:0);
c = spapi(t,tau,b);
plot(breaks([1 end]),[1 1],'k', breaks([1 end]),[-1 -1],'k');
hold on
fnplt(c,'r',1);
hold off
ylim([-2 2]);
title('First Approximation to an Equioscillating Spline');
```
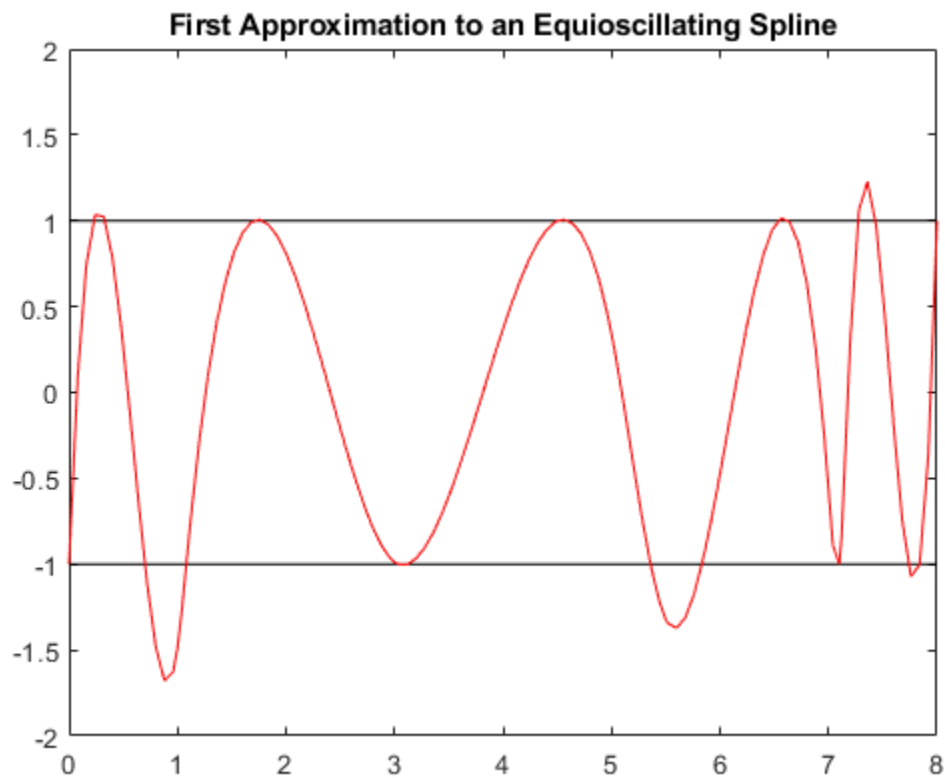

Iteration

For the complete leveling, we use the Remez algorithm. This means that we construct a new `tau` as the extrema of our current approximation, `c`, to `C` and try again.

Finding these extrema is itself an iterative process, namely, for finding the zeros of the derivative `Dc` of our current approximation `c`.

```
Dc = fnder(c);
```

We take the zeros of the control polygon of `Dc` as our first guess for the zeros of `Dc`. This control polygon has the vertices (`tstar(i),coefs(i)`), where `coefs` are the coefficients of `Dc`, and `tstar` the knot averages.

```
[knots,coefs,np,kp] = fnbrk(Dc, 'knots', 'coefs', 'n', 'order');
tstar = aveknt(knots,kp);
```

Since the control polygon is piecewise linear, its zeros are easy to compute. Here are those zeros.

```
npp = 1:np-1;
guess = tstar(npp) - coefs(npp).*(diff(tstar)./diff(coefs));
fnplt(Dc,'r');
hold on
plot(tstar,coefs,'k.:');
plot(guess,zeros(1,np-1),'o');
hold off
title('First Derivative of the Approximation');
legend({'Dc' 'Control Polygon' 'Zeros of Control Polygon'});
```
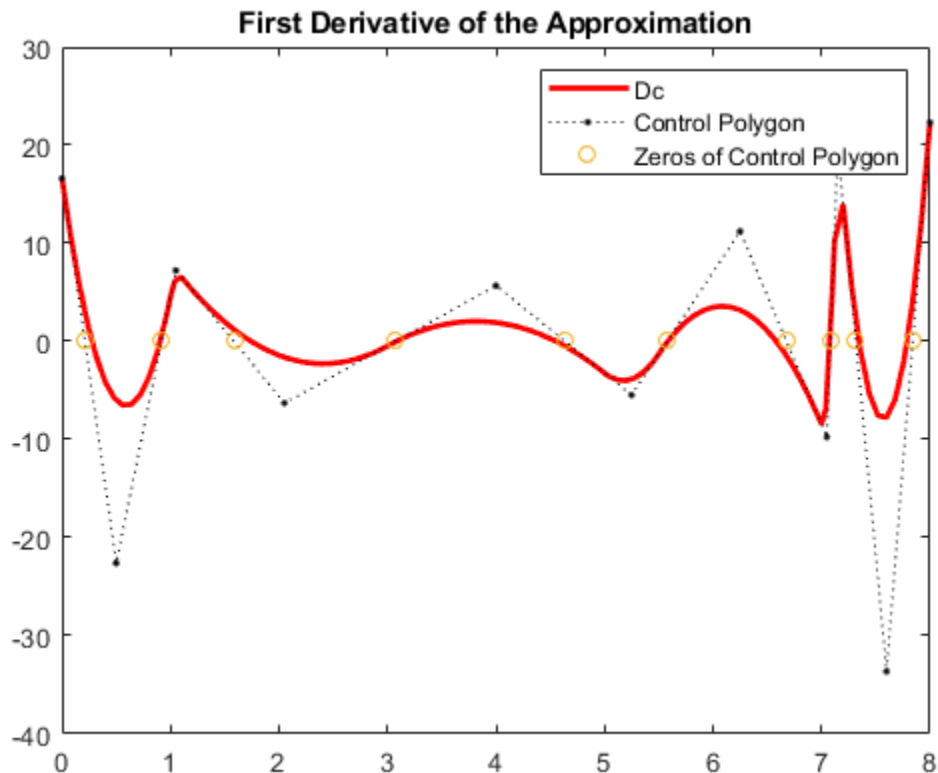

This provides a very good first guess for the actual zeros of `Dc`.

Now we evaluate `Dc` at both these sets of sites.

```
sites = [guess; tau(2:n-1)];
values = fnval(Dc,sites);
```

Then we use two steps of the secant method, getting iterates `sites(3,:)` and `sites(4,:)`, with `values(3,:)` and `values(4,:)` the corresponding values of `Dc`.

```
sites(3:4,:) = 0;
values(3:4,:) = 0;
for j = 2:3
    rows = [j,j-1];
    Dcd = diff(values(rows,:));
    Dcd(Dcd==0) = 1; % guard against division by zero
    sites(j+1,:) = sites(j,:)-values(j,:).*(diff(sites(rows,:))./Dcd);
    values(j+1,:) = fnval(Dc,sites(j+1,:));
end
```

We take the last iterate as the computed zeros of Dc, i.e., the extrema of our current approximation, c. This is our new guess for tau.

```
tau = [tau(1) sites(4,:) tau(n)]
```

tau = *1×12*

```
         0    0.2759    0.9082    1.7437    3.0779    4.5532    5.5823    6.5843    7.0809    7.
```

```
plot(breaks([1 end]),[1 1],'k', breaks([1 end]),[-1 -1],'k');
hold on
fnplt(c,'r',1);
plot(guess,zeros(1,np-1),'o');
plot(tau(2:n-1),zeros(1,n-2),'x');
hold off
title('First Approximation to an Equioscillating Spline');
ax = gca;
h = ax.Children;
legend(h([3 1 2]),{'Approximation' 'Extrema' ...
                  'Zeros of First Derivative''s Control Polygon'});
axis([0 8 -2 2]);
```

First Approximation to an Equioscillating Spline
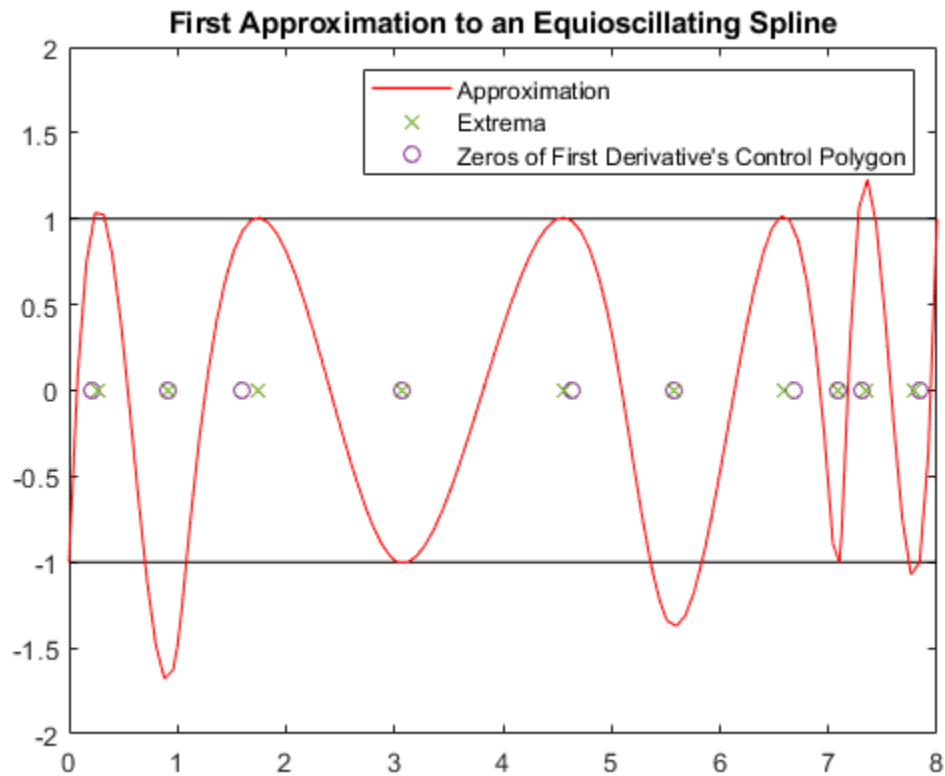
End of First Iteration Step

We compute the resulting new approximation to the Chebyshev spline using the new guess for `tau`.

```
cnew = spapi(t,tau,b);
```

The new approximation is more nearly an equioscillating spline.

```
plot(breaks([1 end]),[1 1],'k', breaks([1 end]),[-1 -1],'k');
hold on
fnplt(c,'r',1);
fnplt(cnew, 'k', 1);
hold off
ax = gca;
h = ax.Children;
legend(h([2 1]),{'First Approximation' 'Updated Approximation'});
axis([0 8 -2 2]);
```
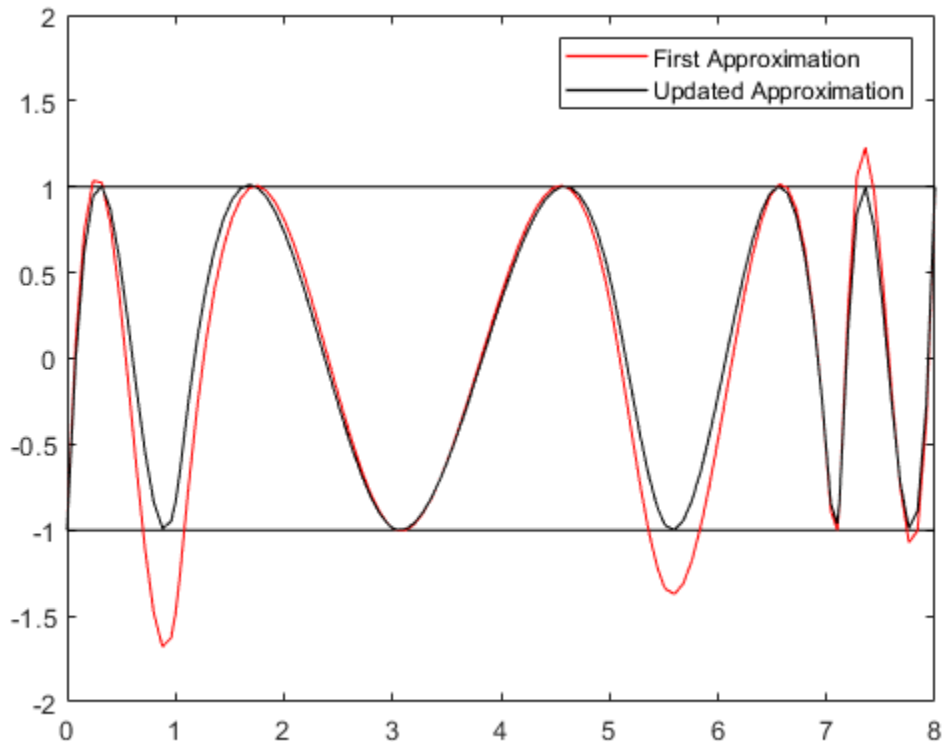
If this is not close enough, simply try again, starting from this new `tau`. For this particular example, the next iteration already provides the Chebyshev spline to graphic accuracy.

Use of Chebyshev-Demko Points

The Chebyshev spline for a given spline space $S\_\{k,t\}$, along with its extrema, are available as optional outputs from the `chbpnt` command in the toolbox. These extrema were proposed as good interpolation sites by Steven Demko, hence are now called the Chebyshev-Demko sites. This section shows an example of their use.

If you have decided to approximate the square-root function on the interval `[0 .. 1]` by cubic splines with knot sequence

```
k = 4;
n = 10;
t = augknt(((0:n)/n).^8,k);
```

then a good approximation to the square-root function from that specific spline space is given by

```
tau = chbpnt(t,k);
sp = spapi(t,tau,sqrt(tau));
```

as is evidenced by the near equioscillation of the error.

```
xx = linspace(0,1,301);
plot(xx, fnval(sp,xx)-sqrt(xx));
title({'Error in Interpolant to Square Root','at Chebyshev-Demko Sites.'});
```

Error in Interpolant to Square Root at Chebyshev-Demko Sites.
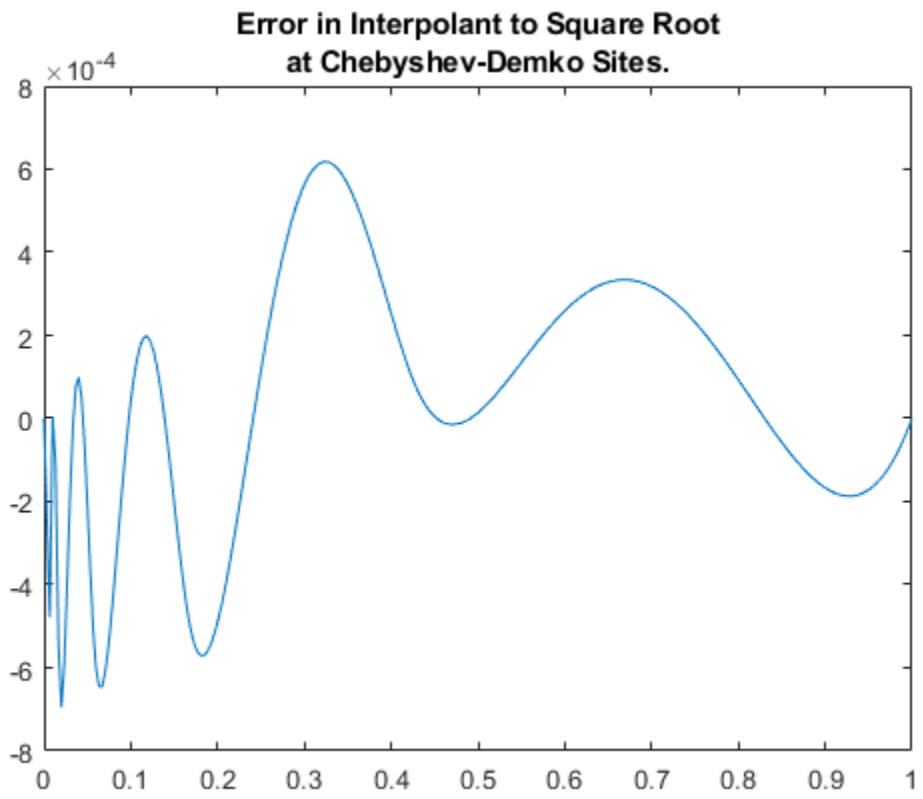
Functions

Curve Fitting

Fit curves and surfaces to data

Description

The **Curve Fitting** app provides a flexible interface where you can interactively fit curves and surfaces to data and view plots.

You can:

- Create, plot, and compare multiple fits.
- Use linear or nonlinear regression, interpolation, smoothing, and custom equations.
- View goodness-of-fit statistics, display confidence intervals and residuals, remove outliers and assess fits with validation data.
- Automatically generate code to fit and plot curves and surfaces, or export fits to the workspace for further analysis.
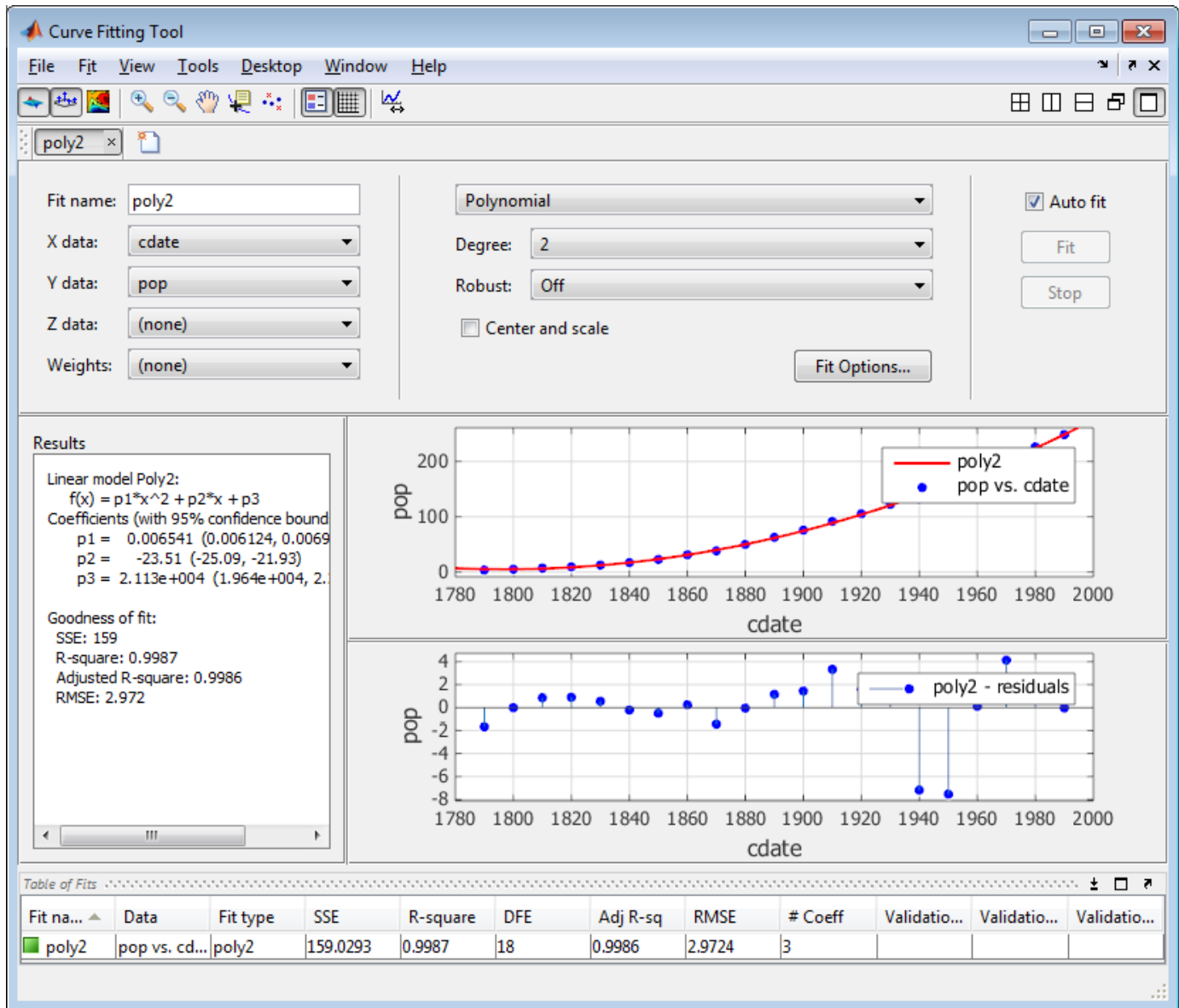
Open the Curve Fitting App

- MATLAB Toolstrip: On the **Apps** tab, under **Math, Statistics and Optimization**, click the app icon.
- MATLAB command prompt: Enter `cftool`.

Examples

- "Interactive Curve and Surface Fitting" on page 2-2
- "Data Selection" on page 2-8

- "Compare Fits in Curve Fitting App" on page 2-16
- "Generating MATLAB Code and Exporting Fits" on page 2-15

Programmatic Use

`cftool` opens Curve Fitting app or brings focus to the app if it is already open.

`cftool( x, y )` creates a curve fit to x input and y output. x and y must be numeric, have two or more elements, and have the same number of elements. `cftool` opens Curve Fitting app if necessary.

`cftool( x, y, z )` creates a surface fit to x and y inputs and z output. x, y, and z must be numeric, have two or more elements, and have compatible sizes. Sizes are compatible if x, y, and z all have the same number of elements or x and y are vectors, z is a 2D matrix, `length(x ) = n`, and `length(y) = m` where `[m,n] = size(z)`. `cftool` opens Curve Fitting app if necessary.

`cftool( x, y, [], w )` creates a curve fit with weights w. w must be numeric and have the same number of elements as x and y.

`cftool( x, y, z, w )` creates a surface fit with weights w. w must be numeric and have the same number of elements as z.

`cftool( filename )` loads the Curve Fitting session in `filename` into Curve Fitting app. The `filename` must have the extension `.sfit`.

See Also

Functions
`fit`

Topics
"Interactive Curve and Surface Fitting" on page 2-2
"Data Selection" on page 2-8
"Compare Fits in Curve Fitting App" on page 2-16
"Generating MATLAB Code and Exporting Fits" on page 2-15

Introduced before R2006a

aptknt

Acceptable knot sequence

Syntax

```
knots = aptknt(tau,k)
[knots,k] = aptknt(tau,k)
```

Description

`knots = aptknt(tau,k)` returns a knot sequence suitable for interpolation at the data sites `tau` by splines of order k with that knot sequence, provided `tau` has at least k entries, is nondecreasing, and satisfies `tau(i)<tau(i+k-1)` for all `i`. In that case, there is exactly one spline of order k with knot sequence `knots` that matches given values at those sites. This is so because the sequence `knots` returned satisfies the Schoenberg-Whitney conditions

```
knots(i) < tau(i) < knots(i+k),        i=1:length(tau)
```

with equality only at the extreme knots, each of which occurs with exact multiplicity k.

If `tau` has fewer than k entries, then k is reduced to the value `length(tau)`. An error results if `tau` fails to be nondecreasing and/or `tau(i)` equals `tau(i+k-1)` for some `i`.

`[knots,k] = aptknt(tau,k)` also returns the actual k used (which equals the smaller of the input k and `length(tau)`).

Examples

If `tau` is equally spaced, e.g., equal to `linspace(a,b,n)` for some n>=4, and y is a sequence of the same size as `tau`, then `sp = spapi(aptknt(tau,4),tau,y)` gives the cubic spline interpolant with the not-a-knot end condition. This is the same cubic spline as produced by the command `spline(tau,y)`, but in B-form rather than ppform.

Cautionary Note

If `tau` is very nonuniform, then use of the resulting knot sequence for interpolation to data at the sites `tau` may lead to unsatisfactory results.

Algorithms

The `(k-1)`-point averages `sum(tau(i+1:i+k-1))/(k-1)` of the sequence `tau`, as supplied by `aveknt(tau,k)`, are augmented by a k-fold `tau(1)` and a k-fold `tau(end)`. In other words, the command gives the same result as `augknt([tau(1),aveknt(tau,k),tau(end)],k)`, provided `tau` has at least k entries and k is greater than 1.

See Also
augknt | aveknt | newknt | optknt

argnames

Input argument names of `cfit`, `sfit`, or `fittype` object

Syntax

```
args = argnames(fun)
```

Description

`args = argnames(fun)` returns the input argument (variable and coefficient) names of the `cfit`, `sfit`, or `fittype` object `fun` as an n-by-1 cell array of character vectors `args`, where `n = numargs(fun)`.

Examples

```
f = fittype('a*x^2+b*exp(n*x)');
nargs = numargs(f)
nargs =
     4
args = argnames(f)
args =
    'a'
    'b'
    'n'
    'x'
```

See Also

`fittype` | `formula` | `numargs`

Introduced in R2006b

augknt

Augment knot sequence

Syntax

```
augknt(knots,k)
augknt(knots,k,mults)
[augknot,addl] = augknt(...)
```

Description

`augknt(knots,k)` returns a nondecreasing and augmented knot sequence that has the first and last knot with exact multiplicity k. (This may actually shorten the knot sequence.))

`augknt(knots,k,mults)` makes sure that the augmented knot sequence returned will, in addition, contain each interior knot `mults` times. If `mults` has exactly as many entries as there are interior knots, then the *j*th one will appear `mults(j)` times. Otherwise, the uniform multiplicity `mults(1)` is used. If `knots` is strictly increasing, this ensures that the splines of order k with knot sequence `augknot` satisfy `k-mults(j)` smoothness conditions across `knots(j+1)`, `j=1:length(knots)-2`.

`[augknot,addl] = augknt(...)` also returns the number `addl` of knots added on the left. (This number may be negative.)

Examples

If you want to construct a cubic spline on the interval [`a..b`], with two continuous derivatives, and with the interior break sequence `xi`, then `augknt([a,b,xi],4)` is the knot sequence you should use.

If you want to use Hermite cubics instead, i.e., a cubic spline with only one continuous derivative, then the appropriate knot sequence is `augknt([a,xi,b],4,2)`.

`augknt([1 2 3 3 3],2)` returns the vector [1 1 2 3 3], as does `augknt([3 2 3 1 3],2)`. In either case, `addl` would be 1.

aveknt

Provide knot averages

Syntax

```
tstar = aveknt(t,k)
```

Description

`tstar = aveknt(t,k)` returns the averages of successive `k-1` knots, i.e., the sites

$$t_i^* := (t_{i+1} + \cdots + t_{i+k-1})/(k-1), \quad i = 1:n$$

which are recommended as good interpolation site choices when interpolating from splines of order `k` with knot sequence $t = (t_i)_{i=1}^{n+k}$.

Examples

`aveknt([1 2 3 3 3],3)` returns the vector `[2.5000 3.0000]`, while `aveknt([1 2 3],3)` returns the empty vector.

With k and the strictly increasing sequence `breaks` given, the statements

```
t = augknt(breaks,k); x = aveknt(t);
sp = spapi(t,x,sin(x));
```

provide a spline interpolant to the sine function on the interval `[breaks(1)..breaks(end)]`.

For `sp` the B-form of a scalar-valued univariate spline function, and with `tstar` and `a` computed as

```
tstar = aveknt(fnbrk(sp,'knots'),fnbrk(sp,'order'));
a = fnbrk(sp,'coefs');
```

the points (tstar(*i*), a(*i*)) constitute the *control points* of the spline, i.e., the vertices of the spline's *control polygon*.

See Also
`aptknt` | `chbpnt` | `optknt`

bkbrk

Part(s) of almost block-diagonal matrix

Syntax

```
[nb,rows,ncols,last,blocks] = bkbrk(blokmat)
bkbrk(blokmat)
```

Description

`[nb,rows,ncols,last,blocks] = bkbrk(blokmat)` returns the details of the almost block-diagonal matrix contained in `blokmat`, with `rows` and `last` `nb`-vectors, and `blocks` a matrix of size `[sum(rows),ncols]`.

This utility program is not likely to be of interest to the casual user. It is used in `slvblk` to decode the information, provided by `spcol`, about a spline collocation matrix in an almost block diagonal form especially suited for splines. But `bkbrk` can also decode the almost block-diagonal form used in [1].

`bkbrk(blokmat)` returns nothing, but the details are printed out. This is of use when trying to understand what went wrong with such a matrix.

References

[1] C. de Boor and R. Weiss. "SOLVEBLOK: A package for solving almost block diagonal linear systems." *ACM Trans. Mathem. Software 6* (1980), 80–87.

See Also
`slvblk` | `spcol`

brk2knt

Convert breaks with multiplicities into knots

Syntax

```
knots = brk2knt(breaks,mults)
```

Description

`knots = brk2knt(breaks,mults)` returns the sequence `knots` that is the sequence `breaks` but with `breaks(i)` occurring `mults(i)` times, all `i`. In particular, `breaks(i)` will not appear unless `mults(i)>0`. If, as one would expect, `breaks` is a strictly increasing sequence, then `knots` contains each `breaks(i)` exactly `mults(i)` times.

If `mults` does not have exactly as many entries as does `breaks`, then all `mults(i)` are set equal to `mults(1)`.

Examples

The statements

```
t = [1 1 2 2 2 3 4 5 5];
[xi,m] = knt2brk(t);
tt = brk2knt(xi,m)
```

give `[1 2 3 4 5]` for `xi`, `[2 3 1 1 2]` for `m`, and, finally, `t` for `tt`.

See Also
augknt

bspligui

Experiment with B-spline as function of its knots

Syntax

```
bspligui
```

Description

`bspligui` starts a graphical user interface (GUI) for exploring how a B-spline depends on its knots. As you add, move, or delete knots, you see the B-spline and its first three derivatives change accordingly.

You observe the following basic facts about the B-spline with knot sequence $t_0 \leq \cdots \leq t_k$:

- The B-spline is positive on the open interval $(t_0..t_k)$. It is zero at the end knots, t_0 and t_k, unless they are knots of multiplicity k. The B-spline is also zero outside the closed interval $[t_0..t_k]$, but that part of the B-spline is not shown in the GUI.

- Even at its maximum, the B-spline is never bigger than 1. It reaches the value 1 inside the interval $(t_0..t_k)$ only at a knot of multiplicity at least k–1. On the other hand, that maximum cannot be arbitrarily small; it seems smallest when there are no interior knots.

- The B-spline is piecewise polynomial of order k, i.e., its polynomial pieces all are of degree $<k$. For $k = 1$:4, you can even observe that all its nonzero polynomial pieces are of exact degree k – 1, by looking at the first three derivatives of the B-spline. This means that the degree goes up/down by 1 every time you add/delete a knot.

- Each knot t_j is a break for the B-spline, but it is permissible for several knots to coincide. Therefore, the number of nontrivial polynomial pieces is maximally k (when all the knots are different) and minimally 1 (when there are no "interior" knots), and any number between 1 and k is possible.

- The smoothness of the B-spline across a break depends on the multiplicity of the corresponding knot. If the break occurs in the knot sequence m times, then the $(k$–$m)$th derivative of the B-spline has a jump across that break, while all derivatives of order lower than $(k$–$m)$ are continuous across that break. Thus, by varying the multiplicity of a knot, you can control the smoothness of the B-spline across that knot.

- As one knot approaches another, the highest derivative that is continuous across both develops a jump and the higher derivatives become unbounded. But nothing dramatic happens in any of the lower-order derivatives.

- The B-spline is *bell-shaped* in the following sense: if the first derivative is not identically zero, then it has exactly one sign change in the interval $(t_0..t_k)$, hence the B-spline itself is *unimodal*, meaning that it has exactly one maximum. Further, if the second derivative is not identically zero, then it has exactly two sign changes in that interval. Finally, if the third derivative is not identically zero, then it has exactly three sign changes in that interval. This illustrates the fact that, for $j = 0$:k – 1, if the jth derivative is not identically zero, then it has exactly j sign changes in the interval $(t_0..t_k)$; it is this property that is meant by the term "bell-shaped". For this claim to be strictly true, one has to be careful with the meaning of "sign change" in case there are knots with multiplicities. For example, the $(k$–1)st derivative is piecewise constant, hence it cannot have k–1 sign changes in the straightforward sense unless there are k polynomial pieces, i.e., unless all the knots are simple.

See Also
bspline | chbpnt | spcol

bspline

Plot B-spline and its polynomial pieces

Syntax

```
bspline(t)
bspline(t,window)
pp = bspline(t)
```

Description

bspline(t) plots the B-spline with knot sequence t, as well as the polynomial pieces of which it is composed. For more information about spline fitting, see About Splines in Curve Fitting Toolbox on page 1-8.

bspline(t,window) plots the B-spline with knot sequence t, as well as the polynomial pieces of which it is composed, in the subplot window specified by window.

pp = bspline(t) plots nothing and returns the ppform of the B-spline.

Examples

Plot a B-spline of Order 4

This example shows how to create a B-spline of order 4 using the bspline function.

Create a Knot Sequence and Plot the B-spline

This figure shows a B-spline of order 4 and the four cubic polynomials that composes it.
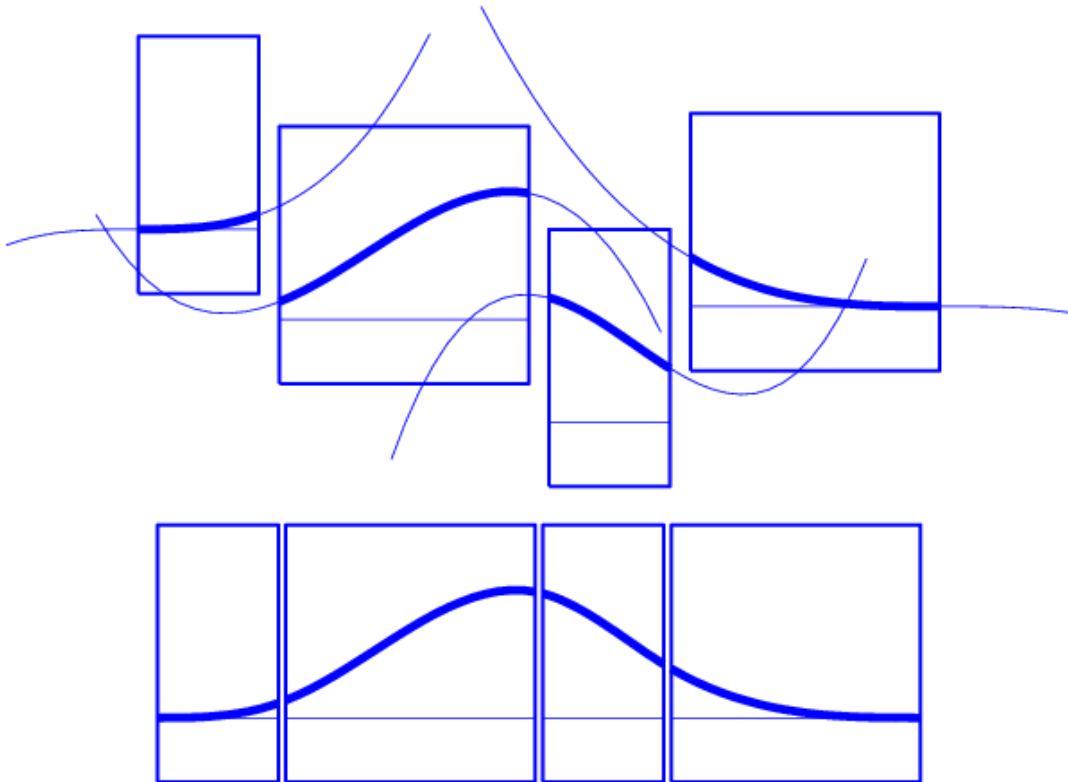
To replicate this figure in MATLAB ®, first create a knot sequence. A knot sequence must be non-decreasing.

```
t = [0 1.5 2.3 4 5];
```

As you have defined five knots, the B-spline will be of order 4.

Then plot the B-spline with knot sequence t, as well as its polynomial pieces, by using the bspline function.

```
bspline(t)
```
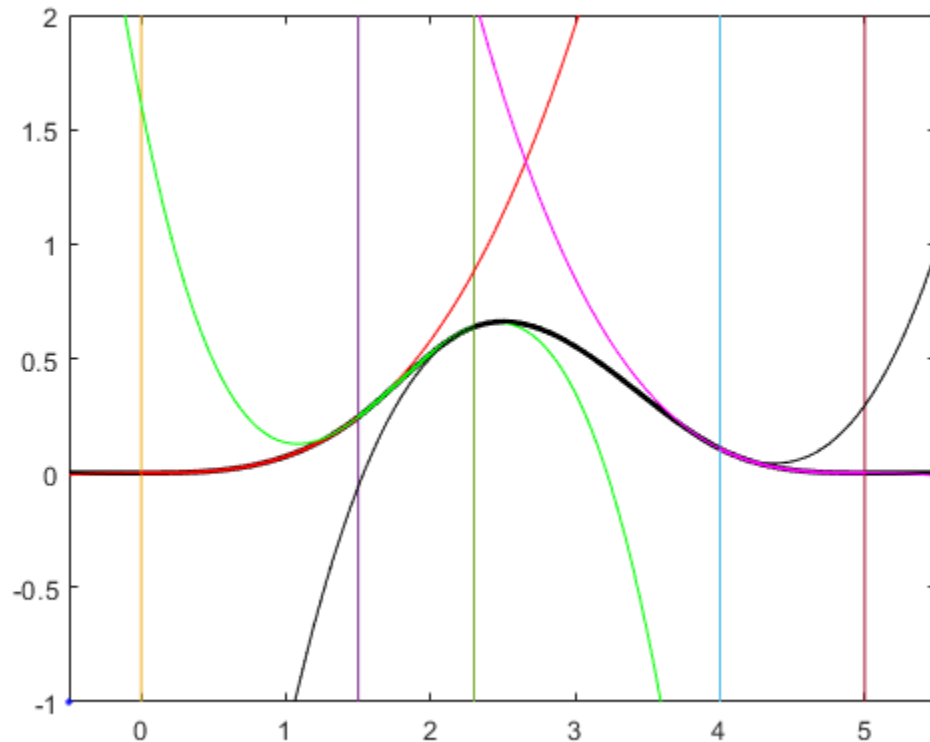
This B-spline consists of 4 polynomial pieces: the red, green, purple, and black curves in the plot, each of degree 3.

The vertical lines mark the knots t that you have previously defined.

The B-spline with knots t(i)≤···≤ t(i+k) is positive on the interval (t(i)..t(i+k)) and is zero outside that interval. It is piecewise-polynomial of order k with breaks at the sites t(i),...,t(i+k). These knots may coincide, and the precise *multiplicity* governs the smoothness with which the two polynomial pieces join there.

Plot Another B-spline in a New Subplot

Create a second knot sequence.

```
t2=[2 3 4 5];
```

Plot the B-spline with knot sequence t2 in the same figure but in a different subplot.

```
bspline(t,1)
bspline(t2 ,2)
```
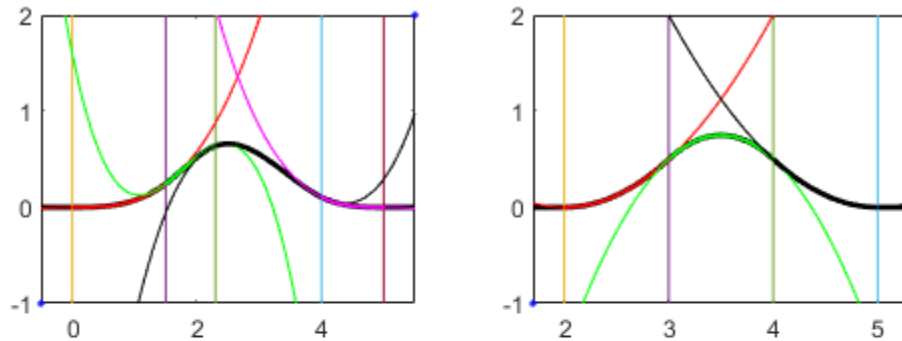
Copyright 2019 The MathWorks, Inc.

Input Arguments

t — Knot sequence of spline
vector (default)

Non-decreasing sequence of the knots of the B-spline, specified as a vector.

Data Types: `single` | `double`

window — Subplot window
scalar (default)

Index of the subplot of a 2x2 window, specified as a scalar. This value must be less than 5.

Data Types: `single` | `double`

Output Arguments

pp — Spline in ppform
spline structure

Spline in ppform, returned as a structure with these fields. For more information on ppform, see The ppform on page 10-8

Form — Form of spline
pp

Form of the spline, returned as pp. pp indicates that the spline is given in piecewise polynomial form.

Breaks — Knot locations of spline
vector | cell array

Knot positions of the spline, returned as a vector or as a cell array of vectors for multivariate data. Vectors contain strictly increasing elements that represent the start and end of each of the intervals over which the polynomial pieces are defined.

Coefs — Coefficients of polynomials
matrix | array

Coefficients of polynomials for each piece, returned as a matrix or as an array for multivariate data.

Pieces — Number of polynomial pieces
scalar | vector

Number of polynomial pieces describing the spline, returned as a scalar or as a vector of numbers of pieces in each variable for multivariate data.

Order — Order of polynomials
scalar | vector

Order of the polynomial function describing each polynomial piece of the spline, returned as a scalar or as a vector containing the order in each variable for multivariate data.

Dim — Dimensionality
scalar

Dimensionality of the target function, returned as a scalar.

See Also
bspligui

Topics
"Introducing Spline Fitting" on page 8-2
"Types of Splines: ppform and B-form" on page 10-2

Introduced in R2006b

category

Category of fit of `cfit`, `sfit`, or `fittype` object

Syntax

```
cname = category(fun)
```

Description

`cname = category(fun)` returns the fit category `cname` of the `cfit`, `sfit`, or `fittype` object `fun`, where `cname` is one of `'custom'`, `'interpolant'`, `'library'`, or `'spline'`.

Examples

```
f1 = fittype('a*x^2+b*exp(n*x)');
category(f1)
ans =
custom

f2 = fittype('pchipinterp');
category(f2)
ans =
interpolant

f3 = fittype('fourier4');
category(f3)
ans =
library

f4 = fittype('smoothingspline');
category(f4)
ans =
spline
```

See Also
`fittype` | `type`

Topics
"List of Library Models for Curve and Surface Fitting" on page 4-10

Introduced in R2006b

cfit

Constructor for `cfit` object

Syntax

```
cfun = cfit(ffun, coeff1,...,coeffn)
```

Description

`cfun = cfit(ffun, coeff1,...,coeffn)` constructs the `cfit` object `cfun` using the model type specified by the `fittype` object `ffun` and the coefficient values `coeff1`, `coeff2`, etc.

Note `cfit` is called by the `fit` function when fitting `fittype` objects to data. To create a `cfit` object that is the result of a regression, use `fit`.

You should only call `cfit` directly if you want to assign values to coefficients and problem parameters of a `fittype` object *without* performing a fit.

Examples

Construct a `cfit` object from a `fittype` object

Create a `fittype` object and call the `cfit` function.

```
f = fittype('a*x^2+b*exp(n*x)')
c = cfit(f,1,10.3,-1e2)

f =
     General model:
       f(a,b,n,x) = a*x^2+b*exp(n*x)
c =
     General model:
       c(x) = a*x^2+b*exp(n*x)
     Coefficients:
       a =             1
       b =          10.3
       n =          -100
```

Input Arguments

`ffun` — Model type
`fittype` (default)

Model type the `cfit` function uses to construct the `cfit` object, specified as a `fittype` constructed with the `fittype` function.

Example: `fittype('poly2')`

coeff1,...,coeffn — Coefficient values
scalar (default)

Coefficient values of the `cfit` object, specified as scalars.

Data Types: `single` | `double`

Output Arguments

cfun — `cfit` object
`cfit`

Function output, returned as a `cfit` object.

See Also
`feval` | `fit` | `fittype`

Topics
"Evaluate a Curve Fit" on page 7-16
"Fit Postprocessing"

Introduced before R2006a

chbpnt

Good data sites, Chebyshev-Demko points

Syntax

```
tau = chbpnt(t,k)
chbpnt(t,k,tol)
[tau,sp] = chbpnt(...)
```

Description

`tau = chbpnt(t,k)` are the extreme sites of the Chebyshev spline of order `k` with knot sequence `t`. These are particularly good sites at which to interpolate data by splines of order `k` with knot sequence `t` because the resulting interpolant is often quite close to the best uniform approximation from that spline space to the function whose values at `tau` are being interpolated.

`chbpnt(t,k,tol)` also specifies the tolerance `tol` to be used in the iterative process that constructs the Chebyshev spline. This process is terminated when the relative difference between the absolutely largest and the absolutely smallest local extremum of the spline is smaller than `tol`. The default value for `tol` is `.001`.

`[tau,sp] = chbpnt(...)` also returns, in `sp`, the Chebyshev spline.

Examples

`chbpnt([-ones(1,k),ones(1,k)],k)` provides (approximately) the extreme sites on the interval [–1 .. 1] of the Chebyshev polynomial of degree `k-1`.

If you have decided to approximate the square-root function on the interval [0 .. 1] by cubic splines, with knot sequence `t` as given by

```
k = 4; n = 10; t = augknt(((0:n)/n).^8,k);
```

then a good approximation to the square-root function from that specific spline space is given by

```
x = chbpnt(t,k); sp = spapi(t,x,sqrt(x));
```

as is evidenced by the near equi-oscillation of the error.

Algorithms

The Chebyshev spline for the given knot sequence and order is constructed iteratively, using the Remez algorithm, using as initial guess the spline that takes alternately the values 1 and −1 at the sequence `aveknt(t,k)`. The example "Constructing the Chebyshev Spline" gives a detailed discussion of one version of the process as applied to a particular example.

See Also
`aveknt`

coeffnames

Coefficient names of `cfit`, `sfit`, or `fittype` object

Syntax

```
coeffs = coeffnames(fun)
```

Description

`coeffs = coeffnames(fun)` returns the coefficient (parameter) names of the `cfit`, `sfit`, or `fittype` object `fun` as an n-by-1 cell array of character vectors `coeffs`, where `n = numcoeffs(fun)`.

Examples

```
f = fittype('a*x^2+b*exp(n*x)');
ncoeffs = numcoeffs(f)
ncoeffs =
     3
coeffs = coeffnames(f)
coeffs =
    'a'
    'b'
    'n'
```

See Also
coeffvalues | fittype | formula | numcoeffs | probnames

Introduced in R2006b

coeffvalues

Coefficient values of `cfit` or `sfit` object

Syntax

```
coeffvals= coeffvalues(fun)
```

Description

`coeffvals= coeffvalues(fun)` returns the values of the coefficients, or parameters, of the `cfit` or `sfit` object `fun` as a 1-by-n vector `coeffvals`, where `n = numcoeffs(fun)`.

Examples

Obtain the Coefficient Values of a Quadratic Polynomial Curve

Load the data in the `census.mat` file and construct a `fittype` for a quadratic polynomial curve.

```
load census
f = fittype('poly2');
```

Obtain the coefficient names and the formula for the `fittype` object `f`.

```
coefficientNames = coeffnames(f)
```

```
coefficientNames = 3x1 cell
    {'p1'}
    {'p2'}
    {'p3'}
```

```
formula(f)
```

```
ans =
'p1*x^2 + p2*x + p3'
```

Fit the curve to the data and retrieve the coefficient values.

```
c = fit(cdate,pop,f);
coefficientValues = coeffvalues(c)
```

```
coefficientValues = 1×3
10⁴ ×

    0.0000   -0.0024    2.1130
```

Input Arguments

fun — `cfit` or `sfit` object
`cfit` | `sfit`

Object of which you want to calculate the coefficients, specified as a `cfit` or `sfit` object.

Output Arguments

coeffvals — Coefficients values
scalar | vector

Values of the coefficients of the `cfit` or `sfit` object `fun`, returned as a scalar or vector.

See Also
`cfit` | `sfit` | `coeffnames` | `confint` | `predint` | `probvalues`

Topics
About Splines in Curve Fitting on page 8-2

Introduced before R2006a

confint

Confidence intervals for fit coefficients of `cfit` or `sfit` object

Syntax

```
ci = confint(fitresult)
ci = confint(fitresult,level)
```

Description

`ci = confint(fitresult)` returns 95% confidence bounds `ci` on the coefficients associated with the `cfit` or `sfit` object `fitresult`. `fitresult` must be an output from the `fit` function to contain the necessary information for `ci`. `ci` is a 2-by-n array where `n = numcoeffs(fitresult)`. The top row of `ci` contains the lower bound for each coefficient; the bottom row contains the upper bound.

`ci = confint(fitresult,level)` returns confidence bounds at the confidence level specified by `level`. `level` must be between `0` and `1`. The default value of `level` is `0.95`.

Examples

Obtain the Confidence Intervals for Fit Coefficients Using the `confint` Function

Load the data and call the `fit` function to obtain the `fitresult` information.

```
load census

fitresult = fit(cdate,pop,'poly2')

fitresult =

     Linear model Poly2:
     fitresult(x) = p1*x^2 + p2*x + p3
     Coefficients (with 95% confidence bounds):
       p1 =    0.006541  (0.006124, 0.006958)
       p2 =       -23.51  (-25.09, -21.93)
       p3 =   2.113e+04  (1.964e+04, 2.262e+04)
```

To obtain the confidence intervals, call the `confint` function on `fitresult`.

```
ci = confint(fitresult,0.95)

ci =

    0.0061242      -25.086        19641
    0.0069581      -21.934        22618
```

fit and confint display the confidence bounds in slightly different formats.

Input Arguments

fitresult — Fit information for confidence bounds calculation
cfit | sfit

Fit information for confidence bounds calculation, specified as a cfit or sfit object. fitresult must be an output from the fit function.

level — Confidence level
0.95 (default) | scalar value in the range (0,1)

Confidence level, specified as a scalar. This argument must be between 0 and 1

Data Types: single | double

Output Arguments

ci — Confidence intervals
matrix

Confidence bounds on the coefficients associated with the cfit or sfit object fitresult, returned as a matrix.

Tips

To calculate confidence bounds, confint uses R^{-1} (the inverse R factor from QR decomposition of the Jacobian), the degrees of freedom for error, and the root mean squared error. This information is automatically returned by the fit function and contained within fitresult.

If coefficients are bounded and one or more of the estimates are at their bounds, those estimates are regarded as fixed and do not have confidence bounds.

Note that you cannot calculate confidence bounds if category(fitresult) is 'spline' or 'interpolant'.

See Also
fit | predint

Topics
About Splines in Curve Fitting on page 8-2

Introduced before R2006a

csape

Cubic spline interpolation with end conditions

Syntax

```
pp = csape(x,y)
pp = csape(x,y,conds)
pp = csape(x,[e1,y,e2],conds)
pp = csape({x1,...,xn}, ___ )
```

Description

`pp = csape(x,y)` returns the cubic spline interpolation to the given data `(x,y)` in ppform form. The function applies Lagrange end conditions to each end of the data, and matches the spline endslopes to the slope of the cubic polynomial that fits the last four data points at each end. Data values at the same site are averaged.

`pp = csape(x,y,conds)` uses the end conditions specified by `conds`.

`pp = csape(x,[e1,y,e2],conds)` uses the end conditions specified by `conds` and the values `e1` and `e2` for the left and right end condition values, respectively.

`pp = csape({x1,...,xn}, ___ )` returns the cubic spline interpolation for gridded data using the univariate mesh inputs `x1,...,xn`. In this case, `y` is an `n+r`-dimensional array, where `r` is the dimensionality of each data value. `conds` is a cell array with `n` entries, which provides end conditions for each of the `n` variables. In some cases, you must supply end conditions for end conditions. You can use this syntax with any of the arguments in the previous syntaxes.

Examples

Use Customized End Conditions with `csape`

You can implement custom end conditions using the `csape` function. Suppose you want to enforce the following condition at the leftmost endpoint, $e =$`x(1)`

$$\lambda(s) \equiv aDs(e) + bD^2s(e) = c$$

for the given scalars a, b, and c. You can compute the cubic spine interpolation s as the sum of s_1 (the cubic spine interpolation of the given data using the default end conditions) and s_0 (the cubic spine interpolation of zero data using some nontrivial end conditions):

$$s = s_1 + \frac{c - \lambda(s_1)}{\lambda(s_0)} s_0$$

The end conditions you specify in s_0 do not have to be the final desired end conditions $\lambda(s)$.

This example uses the titanium test data, a standard data set used in data fitting. Load the data using the `titanium` function.

```
[x,y] = titanium;
```

Define the coefficients for $\lambda(s)$.

```
a = -2;
b = -1;
c = 0;
```

The end condition applies to the leftmost end of the data set.

```
e = x(1);
```

Now, calculate the cubic spline interpolation of the data set without imposing the end conditions.

```
s1 = csape(x,y);
```

To calculate s_0, use zero data of the same length as y with an additional set of nontrivial end conditions.

```
yZero = zeros(1,length(y));
```

The 1-by-2 matrix `conds` sets the end conditions by specifying the spline derivatives to fix. This example uses end conditions only on the left end of the data, so use `conds` to fix the first derivative at the left end. At the right end, fix the value of the function itself.

```
conds = [1 0];
```

To specify the values to fix the function or its derivatives to, add them as additional values to the data set for fitting - in this case, `yZero`. The first element specifies the value at the left end, while the last element specifies the value at for the right end.

At the left end, fix the first derivative of the spline to have a value of 1. At the right end, fix the function itself to be 0 (the original value of the final element of `yZero`). Concatenate these end condition values at the respective ends of `yZero` and use `csape` to find the spline that fits the data with these end condition values.

```
s0 = csape(x,[1 yZero 0],conds);
```

Calculate the fully fitted spline from that data by using the aforementioned expression for s. To do this, calculate the values for $\lambda_0 = \lambda(s_0)$ and $\lambda_1 = \lambda(s_1)$ using the first and second derivatives of the splines s_0 and s_1.

```
d1s1 = fnder(fnbrk(s1,1));
d2s1 = fnder(d1s1);
d1s0 = fnder(fnbrk(s0,1));
d2s0 = fnder(d1s0);
```

Calculate the derivatives of the first polynomial piece of the spline, as the end conditions apply to the left end of the data only.

```
lam1 = a*fnval(d1s1, e) + b*fnval(d2s1,e);
lam0 = a*fnval(d1s0, e) + b*fnval(d2s0,e);
```

Now use λ_1 and λ_0 to calculate the final, fully fitted spline.

```
pp = fncmb(s0,(c-lam1)/lam0,s1);
```

Plot the spline to compare the results of the default fit and the end conditions.

```
fnplt(pp,[594, 632])
hold on
fnplt(s1,'b--',[594, 632])
plot(x,y,'ro','MarkerFaceColor','r')
hold off
axis([594, 632, 0.62, 0.655])
legend 'Desired end conditions' ...
'Default end-conditions' 'Data' ...
    Location SouthEast
```
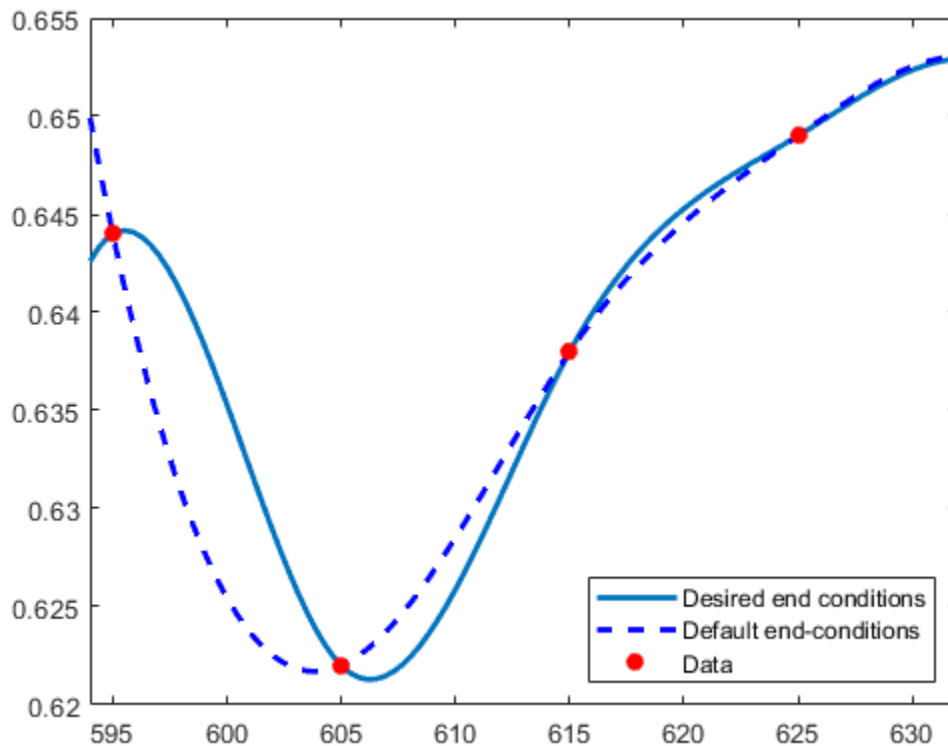

The stationary point near the first data point shows that the end conditions are implemented in the fit.

Fit Multivariate Data

Use `csape` to fit multivariate, vector-valued data. This example fits vector-valued data using different end conditions for each independent variable.

First, define the data. For this example, define the 3-dimensional vectors v over a 2-dimensional field, with clamped conditions or prescribed slopes in the x direction and periodic end conditions in the y direction.

```
x = 0:4;
y = -2:2;
```

```
s2 = 1/sqrt(2);

v = zeros( 3, 7, 5 );
v(1,:,:) = [1 0 s2 1 s2 0 -1].'*[1 0 -1 0 1];
v(2,:,:) = [1 0 s2 1 s2 0 -1].'*[0 1 0 -1 0];
v(3,:,:) = [0 1 s2 0 -s2 -1 0].'*[1 1 1 1 1];
```

v is a 3-dimensional array with v(:,i+1,j) as the vector value at coordinate x(i),y(j). Two additional entries in the x dimension specify the slope values: the data points v(:,1,j) and v(:,7,j) provide the value of the first derivative along the lines x = 0 and x = 4 for the clamped end conditions. In the y dimension, the periodic end conditions do not require any additional specification.

Now, calculate the multivariate cubic spline interpolation using csape.

```
sph = csape({x,y},v,{'clamped','periodic'});
```

To plot the result, first evaluate the spline over a suitable interval.

```
values = fnval(sph,{0:.1:4,-2:.1:2});
```

```
surf(squeeze(values(1,:,:)), ...
squeeze(values(2,:,:)), squeeze(values(3,:,:)));
```

```
axis equal
axis off
```
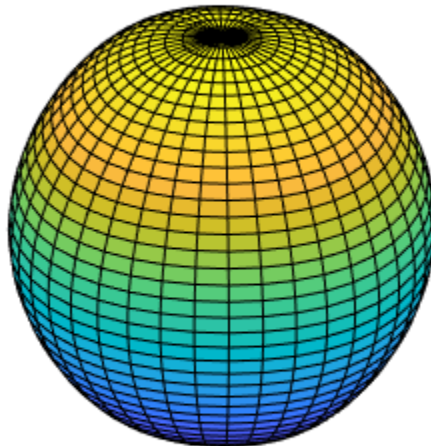
You can also evaluate and plot the spline surface using the simple command `fnplt(sph)`. Note that v is a 3-dimensional array, and `v(:,i+1,j)` is the 3-vector to match at `(x(i),y(j))`, `i=1:5`, `j=1:5`. Additionally, in accordance with `conds{1}` being `'clamped'`, `size(v,2)` is 7 (and not 5), and the first and last entry of `v(r,:,j)` specify the end slope values.

Supply End Conditions for End Conditions

In some cases, you must supply end conditions of end conditions. In this bivariate example, you reproduce the bicubic polynomial $g(x,y) = x^3y^3$ by complete bicubic interpolation. You then derive the needed data, including end condition values, directly from g to make it easier to see how the end condition values must be placed. Finally, you check the result.

```
sites = {[0 1],[0 2]}; coefs = zeros(4, 4); coefs(1,1) = 1;
g = ppmak(sites,coefs);
Dxg = fnval(fnder(g,[1 0]),sites);
Dyg = fnval(fnder(g,[0 1]),sites);
Dxyg = fnval(fnder(g,[1 1]),sites);

f = csape(sites,[Dxyg(1,1),    Dxg(1,:),     Dxyg(1,2); ...
                 Dyg(:,1), fnval(g,sites), Dyg(:,2) ; ...
                 Dxyg(2,1),    Dxg(2,:),     Dxyg(2,2)], ...
                                     {'complete','complete'});
if any(squeeze(fnbrk(f,'c'))-coefs)
    disp( 'this is wrong' )
end
```

Input Arguments

x — Data sites
vector | cell array

Data sites of data values y to fit, specified as a vector or as a cell array for multivariate data. The function creates spline *s* knots at each data site x such that `s(x(j)) = y(:,j)` for all j.

For multivariate, gridded data, specify x as a cell array that provides the data site in each variable dimension, such that `s(x1(i),x2(j),...,xn(k)) = y(:,i,j,...,k)`.

Data Types: `single` | `double`

y — Data values to fit
vector | matrix | array

Data values to fit during creation of the spline, specified as a vector, matrix, or array. You can specify the data values `y(:,j)` as scalars, matrices, or *n*-dimensional arrays. Data values given at the same data site x are averaged.

Data Types: `single` | `double`

conds — End conditions
`'clamped'` | `'complete'` | `'not-a-knot'` | `'periodic'` | `'second'` | `'variational'` | 1-by-2 matrix

End conditions for the spline, specified as `'complete'` or `'clamped'`, `'not-a-knot'`, `'periodic'`, `'second'`, `'variational'`, or as a 1-by-2 matrix. The predefined options for conds

impose identical end conditions at each end of the data. You can specify different end conditions at each end by supplying `conds` as a 1-by-2 matrix.

The available predefined end conditions are as follows.

`'complete'` or `'clamped'`	Match the endslopes to the given values `e1` and `e2`. If you do not provide values for `e1` and `e2`, this option matches the default Lagrangian end conditions.
`'not-a-knot'`	Make the second and second-last sites inactive knots. This option ignores any values you provide for `e1` and `e2`.
`'periodic'`	Match the first and second derivatives at the left end with those at the right end.
`'second'`	Match the end second derivatives to the given values `e1` and `e2`. If you do not provide values `e1` and `e2`, this option uses the default value of 0 for both. With the default values, this option is the same as `'variational'`.
`'variational'`	Set the end second derivatives equal to zero. This option ignores any values you provide for `e1` and `e2`.

To specify different end conditions at each end, supply `conds` as a 1-by-2 matrix. The elements of this matrix elements specify the order of the spline derivative fixed by the end conditions. Setting `conds(j) = i` fixes the ith derivative $D^i s$ to an end condition value.

The default end condition value is the derivative of the cubic interpolant at the left four sites when `conds(1) = 1` and is `0` otherwise. Set end condition values for the left and right sides of the data by specifying `e1` and `e2`, respectively.

You can specify the value of `conds(j)` as `0`, `1`, or `2`. If you specify a different value or do not specify `conds(j)`, then `conds(j)` is `1` and the corresponding end condition value is the default value.

The following pre-defined end conditions are available.

clamped	$Ds(e) = $ `ej`	`if conds(j) == 1`
curved	$D^2 s(e) = $ `ej`	`if conds(j) == 2`
periodic	$D^r s(a) = D^r s(b)$, $r = 1,2$	`if conds == [0 0]`
variational	$D^2 s(e) = 0$	`if conds(j) == 2 & ej == 0`

e, a, and b refer to the left or right data locations; `ej` is `e1` for the left end of the data and `e2` for the right end of the data.

You can supply the optional end condition values `e1` and `e2` whether you use predefined or user-defined options for `conds`. However, note that some predefined options for `conds` ignore any end condition values you provide.

Example: `'clamped', [1 0]`

e1 — Left end condition value
scalar

Left end condition value for the spline, specified as a scalar value. `e1` specifies the value for the ith derivative at the left end of the data, where `conds` provides i. Even if you use different end conditions

at the two ends, if you supply an end condition value at one end you must also supply one for the other end.

Note that some predefined options for `conds` ignore any end condition values you provide.

The default value for `e1` is the derivative of the cubic interpolant at the left four sites when `conds(1) = 1` and is `0` otherwise.

Data Types: `single` | `double`

e2 — Right end condition value
scalar

Right end condition value for the spline, specified as a scalar value. `e2` specifies the value for the *i*th derivative at the right end of the data, where `conds` provides *i*. Even if you use different end conditions at the two ends, if you supply an end condition value at one end you must also supply one for the other end.

Note that some predefined options for `conds` ignore any end condition values you provide.

The default value for `e2` is the derivative of the cubic interpolant at the right four sites when `conds(2) = 1` and is `0` otherwise.

Data Types: `single` | `double`

Output Arguments

pp — Spline in ppform
spline structure

Spline in ppform, returned as a structure with these fields.

Form — Form of spline
pp

Form of the spline, returned as `pp`. `pp` indicates that the spline is given in piecewise polynomial form.

Breaks — Knot locations of spline
vector | cell array

Knot positions of the spline, returned as a vector or as a cell array of vectors for multivariate data. Vectors contain strictly increasing elements that represent the start and end of each of the intervals over which the polynomial pieces are defined.

Coefs — Coefficients of polynomials
matrix | array

Coefficients of polynomials for each piece, returned as a matrix or as an array for multivariate data.

Pieces — Number of polynomial pieces
scalar | vector

Number of polynomial pieces describing the spline, returned as a scalar or as a vector of numbers of pieces in each variable for multivariate data.

Order — Order of polynomials
scalar | vector

Order of the polynomial function describing each polynomial piece of the spline, returned as a scalar or as a vector containing the order in each variable for multivariate data.

Dim — Dimensionality
scalar

Dimensionality of the target function, returned as a scalar.

Algorithms

The relevant tridiagonal linear system is constructed and solved using the sparse matrix capabilities of MATLAB.

The `csape` command calls on a much expanded version of the Fortran routine `CUBSPL` in *PGS*.

See Also
csapi | spapi | spline

Introduced before R2006a

csapi

Cubic spline interpolation

Syntax

```
pp=csapi(x,y)
values = csapi(x,y,xx)
```

Description

Note For a simpler but less flexible method to interpolate cubic splines, try the Curve Fitting app or the `fit` function and see "About Smoothing Splines" on page 6-9.

`pp=csapi(x,y)` returns the ppform of a cubic spline s with knot sequence `x` that takes the values `y(:,j)` at `x(j)` for `j=1:length(x)`. The values `y(:,j)` can be scalars, vectors, matrices, and ND-arrays. The function averages the data points with the same data site and then sorts them by their sites. With `x` the resulting sorted data sites, the spline `s` satisfies the not-a-knot end conditions, such as

$$jump_{x(2)}D_s^3 = 0 = jump_{x(end-1)}D^3(s)$$

where D^3s is the third derivative of s.

If `x` is a cell array of sequences `x1, ..., xm` of lengths `n1, ..., nm`, then `y` is an array of size `[n1,...,nm]` (or of size `[d,n1,...,nm]` if the interpolant is `d`-valued). In that case, `pp` is the ppform of an `m`-cubic spline interpolant s to such data. In particular,

$$s(x(i_1), \cdots, x(i_m)) = y(:, i_1, ..., i_m)$$

with $i_1 = 1:nl$ and $i_m = 1:nm$.

To perform operations on this interpolating cubic spline, such as evaluation, differentiation, plotting, use the pp structure. For more information, see the `fnval`, `fnder`, `fnplt` functions.

`values = csapi(x,y,xx)` returns the values of the smoothing spline evaluated at the points `xx`. This syntax is the same as `fnval(csapi(x,y),xx)`.

This command is essentially the MATLAB function `spline`, which, in turn, is a stripped-down version of the Fortran routine CUBSPL in *PGS*, except that `csapi` (and now also `spline`) accepts vector-valued data and can handle gridded data.

Examples

Interpolant to Two, Three, and Five Points

This example shows how to use the `csapi` command from Curve Fitting Toolbox™ to construct cubic spline interpolants.

Interpolant to Two Points

The command

```
values = csapi(x,y,xx)
```

returns the values at xx of the cubic spline interpolant to the given data (x,y), using the not-a-knot end condition. This interpolant is a piecewise cubic function with break sequence x, whose cubic pieces join together to form a function with two continuous derivatives. The "not-a-knot" end condition means that, at the first and last interior break, even the third derivative is continuous (up to round-off error).

Specifying only two data points results in a straight line interpolant.

```
x = [0 1];
y = [2 0];
xx = linspace(0,6,121);
plot(xx,csapi(x,y,xx),'k-',x,y,'ro')
title('Interpolant to Two Points')
```
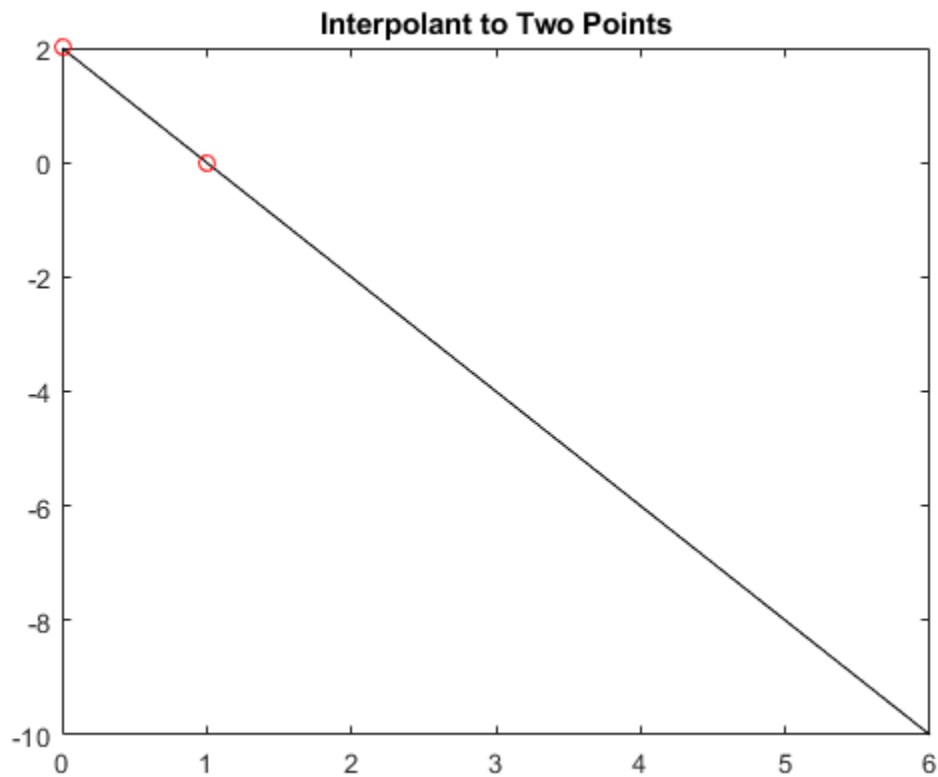

Interpolant to Three Points

If you specify three data points, the function outputs a parabola.

```
x = [2 3 5];
y = [1 0 4];
plot(xx,csapi(x,y,xx),'k-',x,y,'ro')
title('Interpolant to Three Points')
```
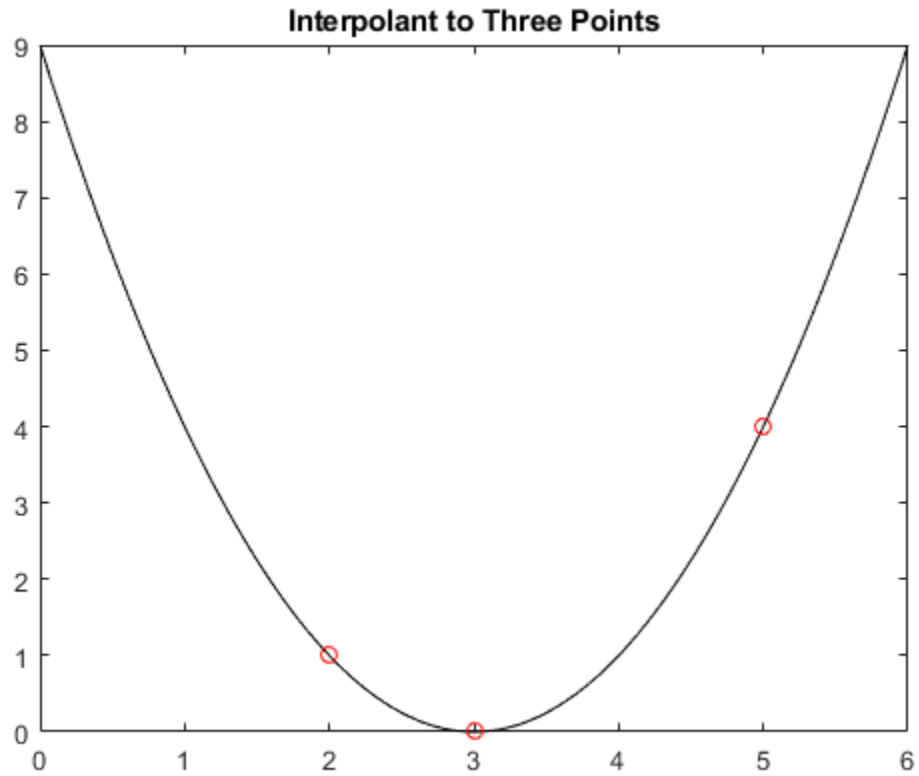
Interpolant to Five Points

More generally, if you specify four or five data points, the function outputs a cubic spline.

```
x = [1 1.5 2 4.1 5];
y = [1 -1 1 -1 1];
plot(xx,csapi(x,y,xx),'k-',x,y,'ro')
title('Cubic Spline Interpolant to Five Points')
```

Cubic Spline Interpolant to Five Points
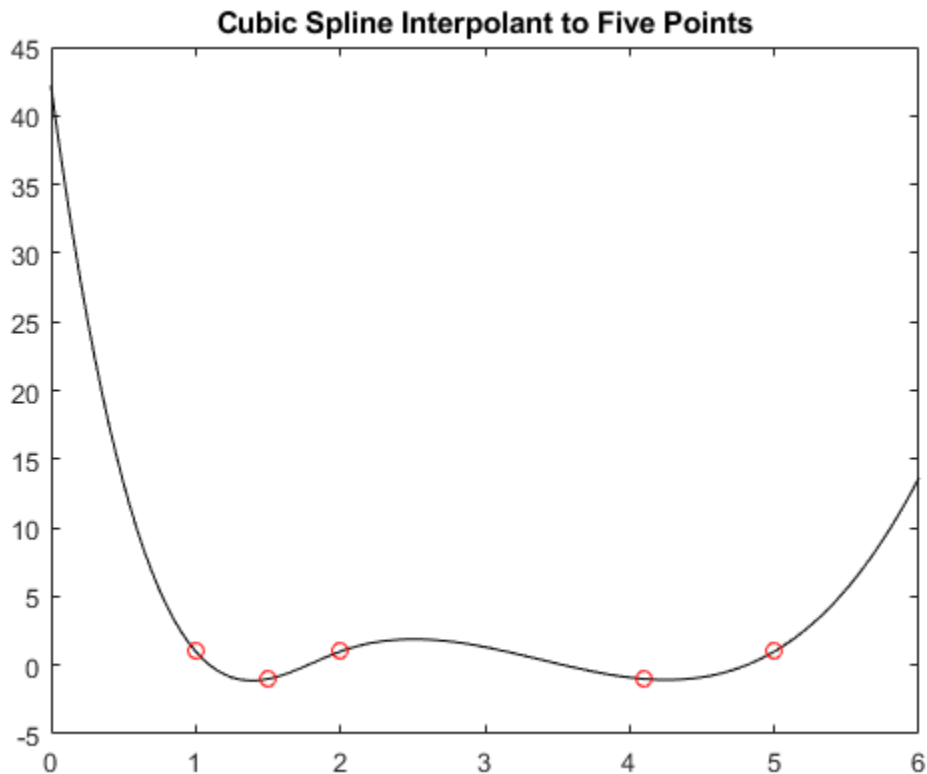
Bicubic Spline Interpolant to Mexican Hat Function

Up to rounding errors, and assuming that x is a vector with at least four entries, the statement pp = csapi(x,y) put the same spline into pp as does the statement:

```
pp = fn2fm(spapi(augknt(x([1 3:(end-2) end]),4),x,y),'pp');
```

except that the description of the spline obtained this second way will not use break at x(2) and x(n-1).

As a simple bivariate example, plot a bicubic spline interpolant to the Mexican Hat function:

```
x =.0001+[-4:.2:4];
y = -3:.2:3;
[yy,xx] = meshgrid(y,x);
r = pi*sqrt(xx.^2+yy.^2);
z = sin(r)./r;
bcs = csapi( {x,y}, z );
fnplt( bcs )
axis([-5 5 -5 5 -.5 1])
```
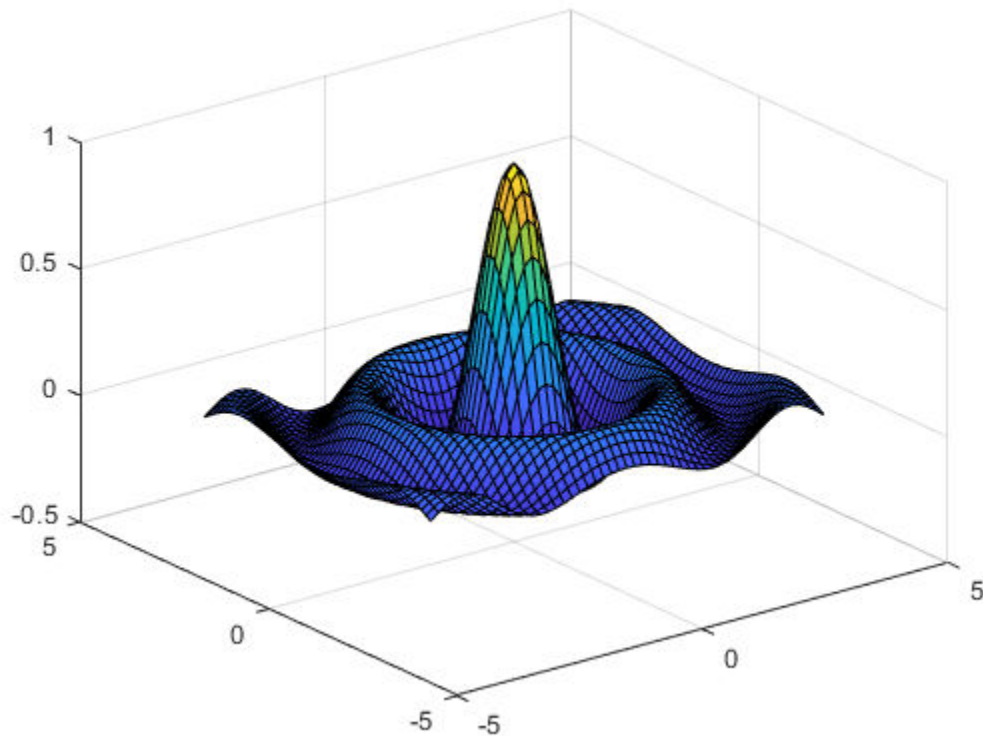
Since MATLAB® considers the entry `z(i,j)` as the value at $(x(j),y(i))$, the code reverses `x` and `y` in the call to `meshgrid`. The Curve Fitting Toolbox® instead follows the Approximation Theory standard whereas `z(i,j)` is the value at $(x(i),y(j))$.

For this reason, you have to be cautious when you are plotting values of such a bivariate spline with the aid of the MATLAB `mesh` function, as shown here:

```
xf = linspace(x(1),x(end),41);
yf = linspace(y(1),y(end),41);
mesh(xf, yf, fnval( bcs, {xf, yf}).')
```
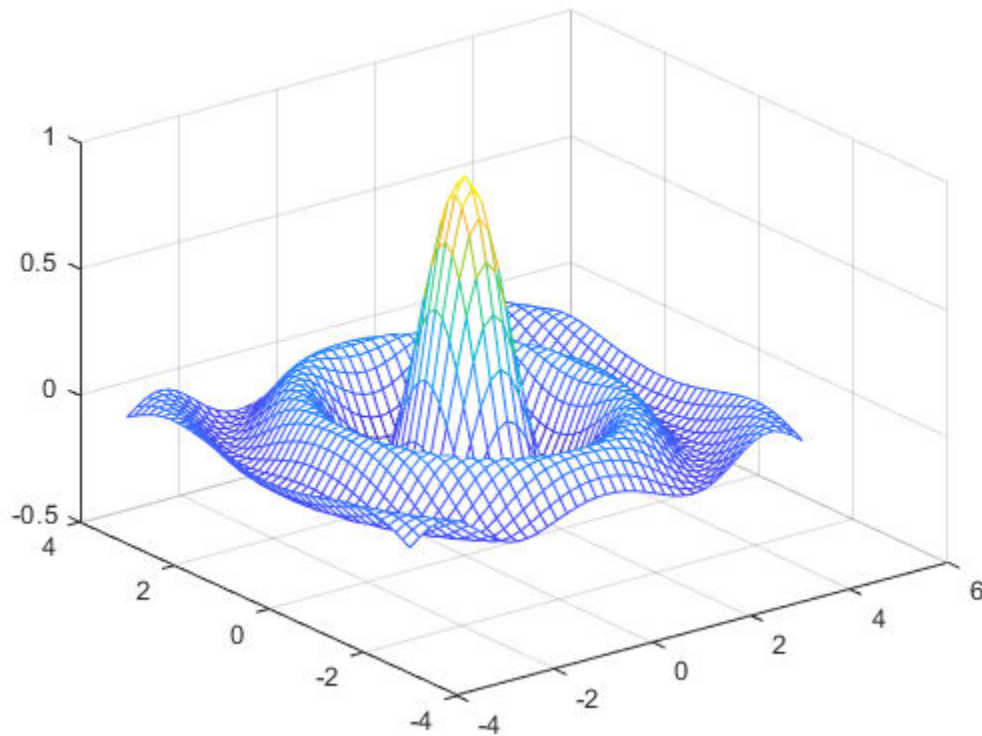
Note the use of the transpose of the matrix of values obtained from `fnval`.

Input Arguments

x — Data sites
vector | cell array

Data sites of data values y to be fit, specified as a vector or as a cell array for multivariate data. Spline *f* is created with knots at each data site x such that $f(x(j)) = y(:,j)$ for all values of j.

For multivariate, gridded data, you can specify x as a cell array that specifies the data site in each variable dimension: $f(x1(i),x2(j),...xn(k)) = y(:,i,j,...,k)$.

Data Types: `single` | `double`

y — Data values to fit
vector | matrix | array

Data values to fit during creation of the spline, specified as a vector, matrix, or array. Data values $y(:,j)$ can be scalars, matrices, or *n*-dimensional arrays. Data values given at the same data site x are averaged.

Data Types: `single` | `double`

xx — Evaluation points
vector | cell array

Evaluation points over which the spline is evaluated, specified as a vector or as a cell array of vectors for multivariate data. Spline evaluation is performed using `fnval`.

Data Types: `single` | `double`

Output Arguments

`pp` — Spline in ppform
spline structure

Spline in ppform, returned as a structure with these fields. For more information on ppform, see The ppform on page 10-8.

`Form` — Form of spline
pp

Form of the spline, returned as `pp`. `pp` indicates that the spline is given in piecewise polynomial form.

`Breaks` — Knot locations of spline
vector | cell array

Knot positions of the spline, returned as a vector or as a cell array of vectors for multivariate data. Vectors contain strictly increasing elements that represent the start and end of each of the intervals over which the polynomial pieces are defined.

`Coefs` — Coefficients of polynomials
matrix | array

Coefficients of polynomials for each piece, returned as a matrix or as an array for multivariate data.

`Pieces` — Number of polynomial pieces
scalar | vector

Number of polynomial pieces describing the spline, returned as a scalar or as a vector of numbers of pieces in each variable for multivariate data.

`Order` — Order of polynomials
scalar | vector

Order of the polynomial function describing each polynomial piece of the spline, returned as a scalar or as a vector containing the order in each variable for multivariate data.

`Dim` — Dimensionality
scalar

Dimensionality of the target function, returned as a scalar.

`values` — Evaluated spline
vector | matrix | array

Evaluated spline, returned as a vector or as a matrix or array for multivariate data. The spline is evaluated at the given evaluation points `xx`.

Algorithms

csapi is an implementation of the Fortran routine CUBSPL from *PGS*.

The algorithm constructs and solves the relevant tridiagonal linear system using the MATLAB sparse matrix capability.

The algorithm also uses the not-a-knot end condition, forcing the first and second polynomial piece of the interpolant to coincide, as well as the second-to-last and the last polynomial piece.

See Also
csape | spapi | spline

Topics
About Splines in Curve Fitting on page 8-2

Introduced in R2006b

csaps

Cubic smoothing spline

Syntax

```
pp = csaps(x,y)
pp = csaps(x,y,p)
pp = csaps(x,y,p,[],w)
values = csaps(x,y,p,xx)
values = csaps(x,y,p,xx,w)
[ ___ ] = csaps({x1,...,xm},y, ___ )
[ ___ ,P] = csaps( ___ )
```

Description

Note For a simpler but less flexible method to generate smoothing splines, try the Curve Fitting app or the `fit` function.

`pp = csaps(x,y)` returns the cubic smoothing spline interpolation to the given data (x,y) in ppform. The value of spline *f* at data site `x(j)` approximates the data value `y(:,j)` for `j = 1:length(x)`.

The smoothing spline *f* minimizes

$$p \sum_{j=1}^{n} w_j \left| y_j - f(x_j) \right|^2 \underbrace{}_{\text{error measure}} + \quad (1-p) \int \lambda(t) \left| D^2 f(t) \right|^2 dt \underbrace{}_{\text{roughness measure}}$$

Here, *n* is the number of entries of x and the integral is over the smallest interval containing all the entries of x. y_j and x_j refer to the *j*th entries of y and x, respectively. D^2f denotes the second derivative of the function *f*.

The default values for the error measure weights w_j are 1. The default value for the piecewise constant weight function λ in the roughness measure is the constant function 1. By default, `csaps` chooses a value for the smoothing parameter *p* based on the given data sites x.

To evaluate a smoothing spline outside its basic interval, you must first extrapolate it. Use the command `pp = fnxtr(pp)` to ensure that the second derivative is zero outside the interval spanned by the data sites.

`pp = csaps(x,y,p)` specifies the smoothing parameter p. You can also supply the roughness measure weights λ by providing p as a vector whose first entry is p and ith entry is the value of λ on the interval `(x(i-1),x(i))`.

`pp = csaps(x,y,p,[],w)` also specifies the weights w in the error measure.

`values = csaps(x,y,p,xx)` uses the smoothing parameter p and returns the values of the smoothing spline evaluated at the points xx. This syntax is the same as `fnval(csaps(x,y,p),xx)`.

values = csaps(x,y,p,xx,w) uses the smoothing parameter p and the error measure weights w, and returns the values of the smoothing spline evaluated at the points xx. This syntax is the same as fnval(csaps(x,y,p,[],w),xx)

[___] = csaps({x1,...,xm},y, ___) provides the ppform of an m-variate tensor-product smoothing spline to data on the rectangular grid described by {x1,...,xm}. You can use this syntax with any of the arguments in the previous syntaxes.

[___ ,P] = csaps(___) also returns the value of the smoothing parameter used in the final spline result whether or not you specify p. This syntax is useful for experimentation in which you can start with [pp,P] = csaps(x,y) and obtain a reasonable first guess for p.

Examples

Fit Splines with Different Smoothing Parameters

Fit smoothing splines using the csaps function with different values for the smoothing parameter p. Use values of p between the extremes of 0 and 1 to see how they affect the shape and closeness of the fitted spline.

Load the titanium data set.

```
[x, y] = titanium();
```

When p = 0, s0 is the least-squares straight line fit to the data. When p = 1, s1 is the variational, or natural, cubic spline interpolant.

For 0 < p < 1, sp is a smoothing spline that is a trade-off between the two extremes: smoother than the interpolant s1 and closer to the data than the straight line s0.

```
p = 0.00009;

s0 = csaps(x,y,0);
sp = csaps(x,y,p);
s1 = csaps(x,y,1);

figure
fnplt(s0);
hold on
fnplt(sp);
fnplt(s1);
plot(x,y,'ko');
hold off
title('Smoothing splines with different values for p');
legend('p = 0', ['p = ' num2str( p )], 'p = 1', 'Location', 'northwest')
```

Smoothing splines with different values for p
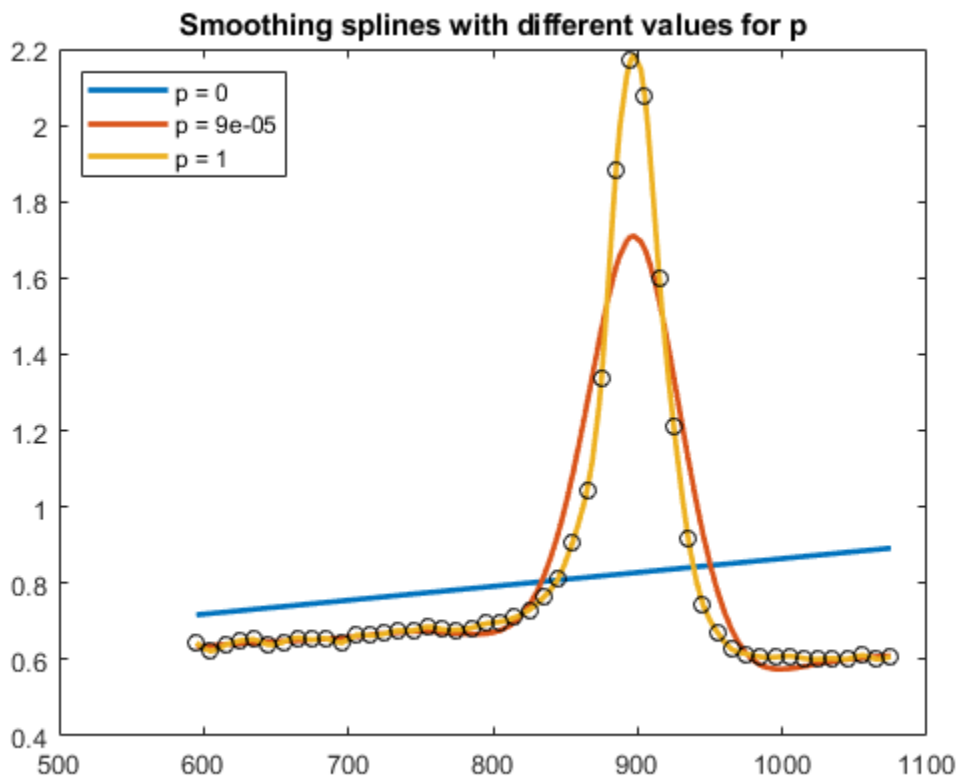
Adjust Smoothing Parameters and Weights

Adjust the smoothing parameter, error measure weights, and roughness measure weights.

Create a sine curve with noise.

```
x = linspace(0,2*pi,21); y = sin(x)+(rand(1,21)-.5)*.3;
```

Fit a smoothing spline to the data. Specify the smoothing parameter `p = 0.4` and error measure weights w that vary across the data.

```
pp = csaps(x,y,0.4,[],[ones(1,10),repmat(5,1,10), 0]);
```

The function returns a smooth fit to the noisy data that is much closer to the data in the right half because of the much larger error measure weight there. Note that the error weighting of zero for the last data point excludes this point from the fit.

Now fit a smoothing spline using the same data, smoothing parameter and error measure weights, but with adjusted roughness measure weights.

```
pp1 = csaps(x,y, [.4,ones(1,10),repmat(.2,1,10)], [], ...
             [ones(1,10), repmat(5,1,10), 0]);
```

The roughness measure weight is only 0.2 in the right half of the interval. Correspondingly, the fit is rougher but closer on the right side of the data (except for the last data point, which is ignored).

Plot both fits for comparison.

```
figure
hold on
fnplt(pp, 'b');
fnplt(pp1,'r--')
plot(x,y,'ok')
hold off
ylim([-1.5 1.5])
title(['Cubic smoothing spline, with right half treated ',...
        'differently'])
legend('Larger error weight', 'Larger error and smaller roughness weight')
```
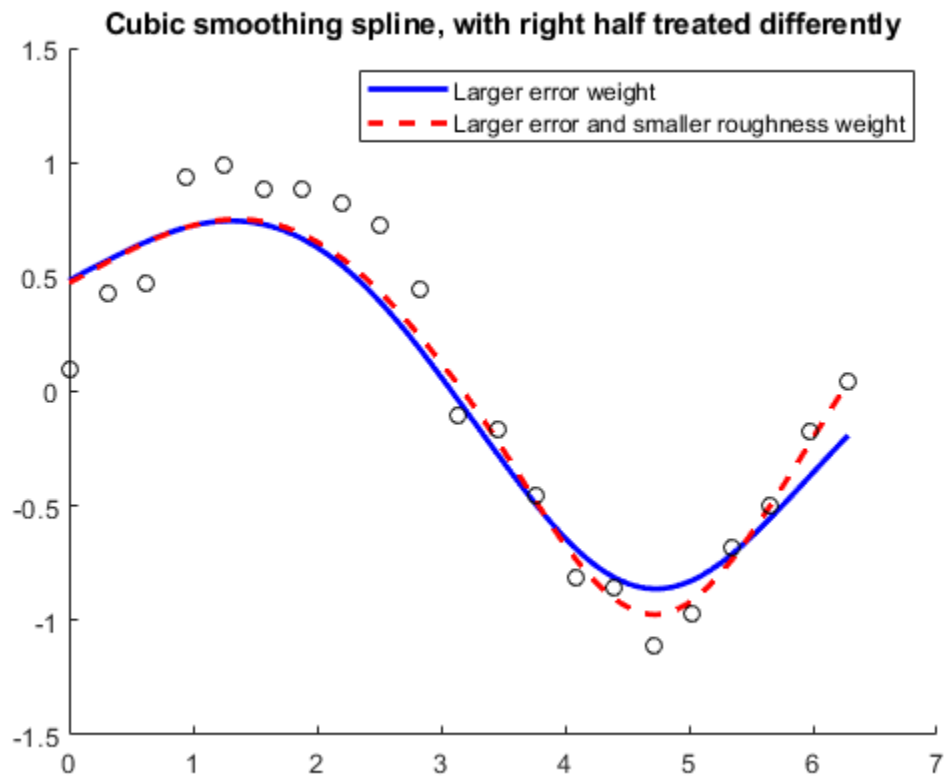

Smooth Bivariate Data

Fit a smoothing spline to bivariate data generated by the `peaks` function with added uniform noise. Use `csaps` to obtain the new, smoothed data points and the smoothing parameters `csaps` determines for the fit.

Create the grid. For this example, the grid is a 51-by-61 uniform grid.

```
x = {linspace(-2,3,51),linspace(-3,3,61)};
[xx,yy] = ndgrid(x{1},x{2});
```

Generate the noisy data using the `peaks` function and random numbers in the interval $\left[-\frac{1}{2}, \frac{1}{2}\right]$.

```
y = peaks(xx, yy);
noisy = y + (rand(size(y)) - 0.5);
figure
surf(xx,yy,noisy)
axis off
```
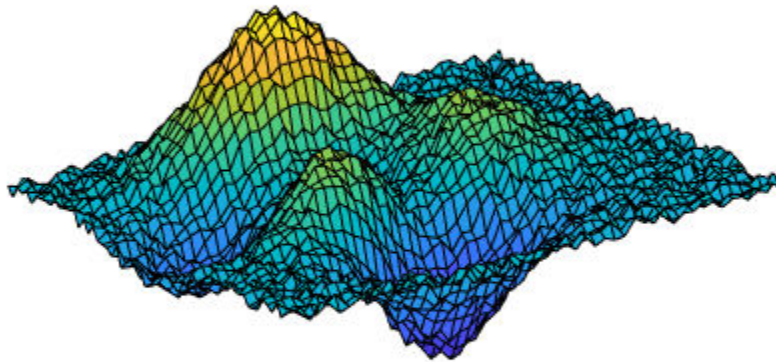

Fit the data. Use `csaps` to obtain the smoothed data values evaluated over the grid `x` and the default smoothing parameter used in the fit.

```
[sval,p] = csaps(x,noisy,[],x);
```

The plot of the fit shows that some roughness remains. Note that you must transpose the array `sval`.

```
figure
surf(x{1},x{2},sval.')
axis off
```
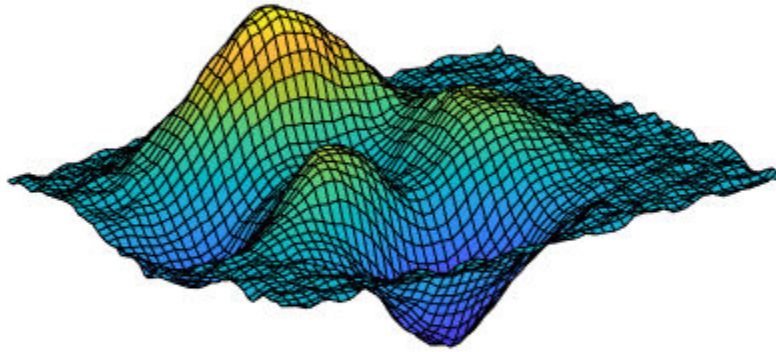
For a somewhat smoother approximation, specify a value for p that is slightly smaller than the `csaps` default value.

```
ssval = csaps(x,noisy,.996,x);
figure
surf(x{1},x{2},ssval.')
axis off
```

Input Arguments

x — Data sites
vector | cell array

Data sites of data values y to be fit, specified as a vector or as a cell array for multivariate data. Spline *f* is created with knots at each data site x such that $f(x(j)) = y(:,j)$ for all values of j.

For multivariate, gridded data, you can specify x as a cell array that specifies the data site in each variable dimension: $f(x1(i),x2(j),...xn(k)) = y(:,i,j,...,k)$.

Data Types: `single` | `double`

y — Data values to fit
vector | matrix | array

Data values to fit during creation of the spline, specified as a vector, matrix, or array. Data values $y(:,j)$ can be scalars, matrices, or *n*-dimensional arrays. Data values given at the same data site x are averaged.

Data Types: `single` | `double`

p — Smoothing parameter
scalar in the range `[0,1]` | vector | cell array | empty array

Smoothing parameter, specified as a scalar value between 0 and 1 or as a cell array of values for multivariate data. You can also specify values for the roughness measure weights λ by providing p as a vector. To provide roughness measure weights for multivariate data, use a cell array of vectors. If you provide an empty array, the function chooses a default value for p based on the data sites x and the default value of 1 for the roughness measure weight λ.

The smoothing parameter determines the relative weight to place on the contradictory demands of having f be smooth or having f be close to the data. For p = 0, f is the least-squares straight-line fit to the data. For p = 1, f is the variational, or natural, cubic spline interpolant. As p moves from 0 to 1, the smoothing spline changes from one extreme to the other.

The favorable range for p is often near $1/(1 + h^3/6)$, where h is the average spacing of the data sites. The function chooses a default value for p within this range. For uniformly spaced data, you can expect a close fit with $p = 1(1 + h^3/60)$ and some satisfactory smoothing with $p = 1/(1 + h^3/0.6)$. You can input p > 1, but this choice leads to a smoothing spline even rougher than the variational cubic spline interpolant.

If the input p is negative or empty, then the function uses the default value for p.

You can specify the roughness measure weights λ alongside the smoothing parameter by providing p as a vector. This vector must be the same size as x, with the ith entry the value of λ on the interval (x(i-1)...x(i)), for i = 2:length(x). The first entry of the input vector p is the desired value of the smoothness parameter p. By providing roughness measure weights, you can make the resulting smoothing spline smoother (with larger weight values) or closer to the data (with smaller weight values) in different parts of the interval. Roughness measure weights must be nonnegative.

If you have difficulty choosing p but have some feeling for the size of the noise in y, consider using spaps(x,y,tol) instead. This function chooses p such that the roughness measure is as small as possible, subject to the condition that the error measure does not exceed tol. In this case, the error measure usually equals the specified value for tol.

Data Types: single | double

w — Error measure weights
vector | cell array

Error measure weights w in the error measure, specified as a vector of nonnegative entries of the same size as x.

The default value for the weight vector w in the error measure is ones(size(x)).

xx — Evaluation points
vector | cell array

Evaluation points over which the spline is evaluated, specified as a vector or as a cell array of vectors for multivariate data. Spline evaluation is performed using fnval.

Data Types: single | double

Output Arguments

pp — Spline in ppform
spline structure

Spline in ppform, returned as a structure with these fields.

`Form` — Form of spline
pp

Form of the spline, returned as `pp`. `pp` indicates that the spline is given in piecewise polynomial form.

`Breaks` — Knot locations of spline
vector | cell array

Knot positions of the spline, returned as a vector or as a cell array of vectors for multivariate data. Vectors contain strictly increasing elements that represent the start and end of each of the intervals over which the polynomial pieces are defined.

`Coefs` — Coefficients of polynomials
matrix | array

Coefficients of polynomials for each piece, returned as a matrix or as an array for multivariate data.

`Pieces` — Number of polynomial pieces
scalar | vector

Number of polynomial pieces describing the spline, returned as a scalar or as a vector of numbers of pieces in each variable for multivariate data.

`Order` — Order of polynomials
scalar | vector

Order of the polynomial function describing each polynomial piece of the spline, returned as a scalar or as a vector containing the order in each variable for multivariate data.

`Dim` — Dimensionality
scalar

Dimensionality of the target function, returned as a scalar.

`values` — Evaluated spline
vector | matrix | array

Evaluated spline, returned as a vector or as a matrix or array for multivariate data. The spline is evaluated at the given evaluation points `xx`.

P — Smoothing parameter
scalar | cell array

Smoothing parameter used to calculate the spline, returned as a scalar or as a cell array of scalar values for multivariate data. `P` is between `0` and `1`.

Algorithms

`csaps` is an implementation of the Fortran routine `SMOOTH` from *PGS*.

The calculation of the smoothing spline requires solving a linear system whose coefficient matrix has the form `p*A + (1-p)*B`, with the matrices A and B depending on the data sites `x`. The default value of `p` makes `p*trace(A)` equal `(1-p)*trace(B)`.

See Also

spap2 | spaps | tpaps

Introduced before R2006a

cscvn

"Natural" or periodic interpolating cubic spline curve

Syntax

```
curve = cscvn(points)
```

Description

`curve = cscvn(points)` returns a parametric variational, or *natural,* cubic spline curve (in ppform) passing through the given sequence points (:,*j*), *j* = 1:end. The parameter value *t*(*j*) for the *j*-th point follows the Eugene Lee's [1] centripetal scheme, as accumulated square root of chord length:

$$\sum_{i<j} \sqrt{\|\text{points}(:,i+1) - \text{points}(:,i)\|_2}$$

If the first and last point coincide and there are no other repeated points) then the function constructs a periodic cubic spline curve. However, double points result in corners.

Examples

Plot Interpolating Cublic Spline Curves

This example shows how to construct and plot several different interpolating cublic spline curves using the `cscvn` function.

This code generates a sequence of points and then plots the cubic spline generated from the cscvn function. The chosen points are marked as circles:

```
points=[0 1 1 0 -1 -1 0 0; 0 0 1 2 1 0 -1 -2];
fnplt(cscvn(points)); hold on,
plot(points(1,:),points(2,:),'o'), hold off
```
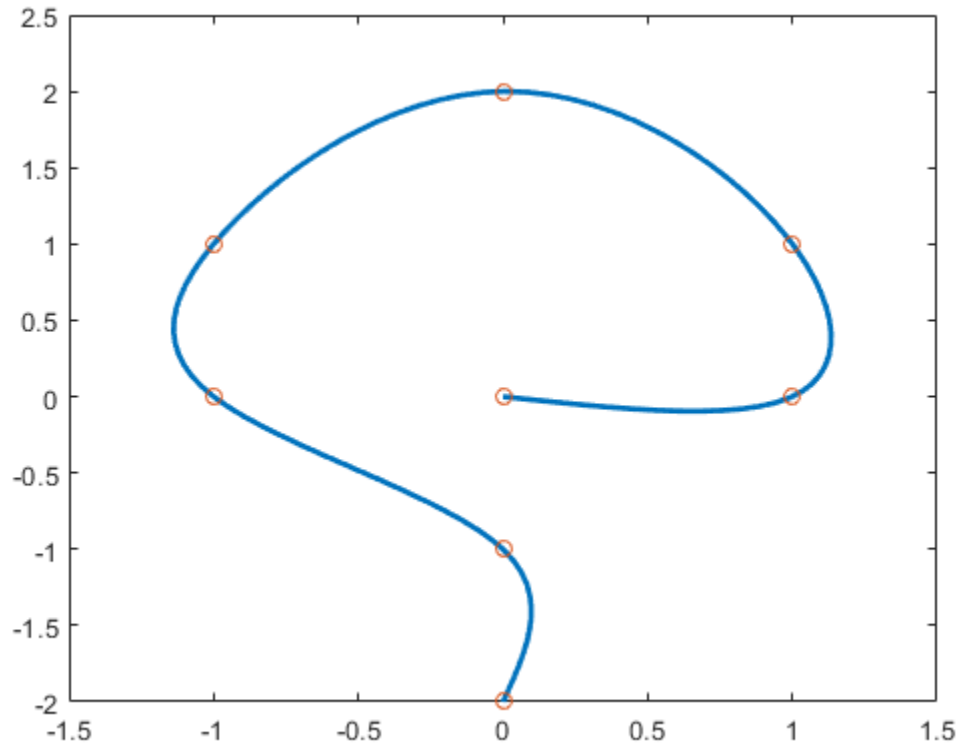
This code plots a circular curve through the four vertices of the standard diamond (because of the periodic boundary conditions enforced):

```
fnplt(cscvn( [1 0 -1   0 1;0 1 0   -1 0] ))
```
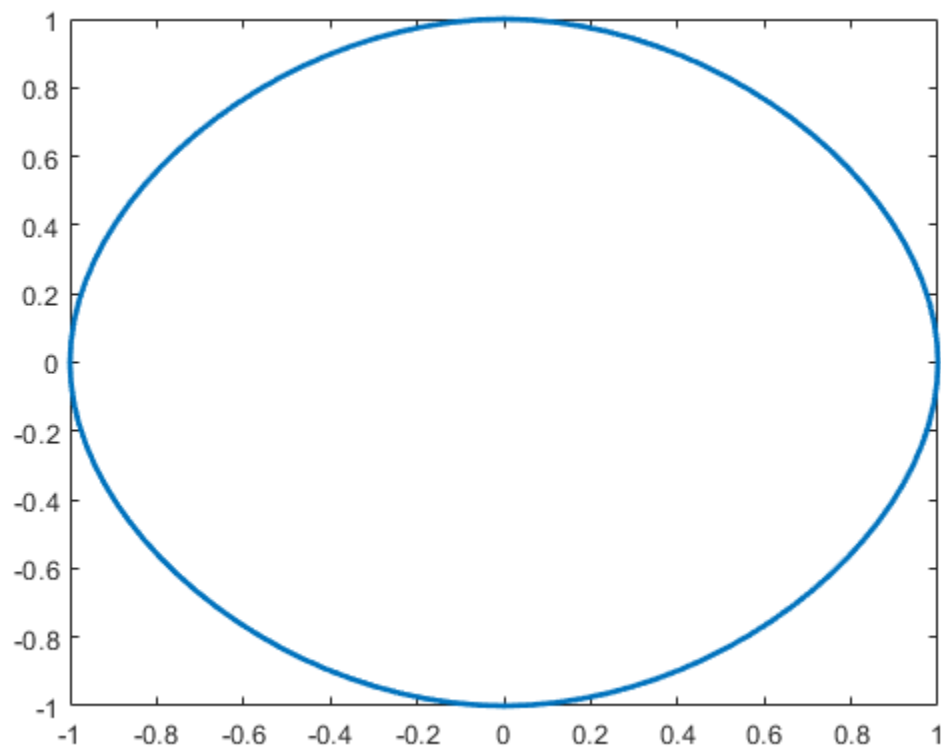
This code shows a corner at the double point as well as at the curve endpoint:

```
fnplt(cscvn( [1 0 -1 -1 0 1;0 1 0 0 -1 0] ))
```
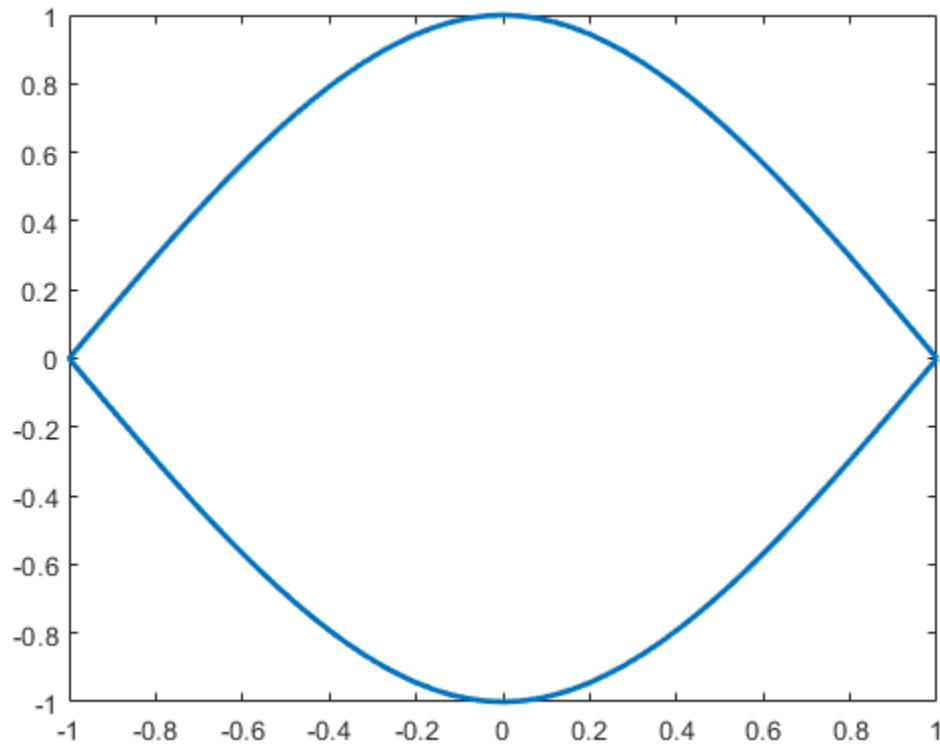
Finally, this code generates a closed curve with one double point, which results in a corner. Dedicate this to your loved ones.

```
c=fnplt(cscvn([0 .82 .92 0 0 -.92 -.82 0; .66 .9 0 ...
-.83 -.83 0 .9 .66])); fill(c(1,:),c(2,:),'r'), axis equal
```
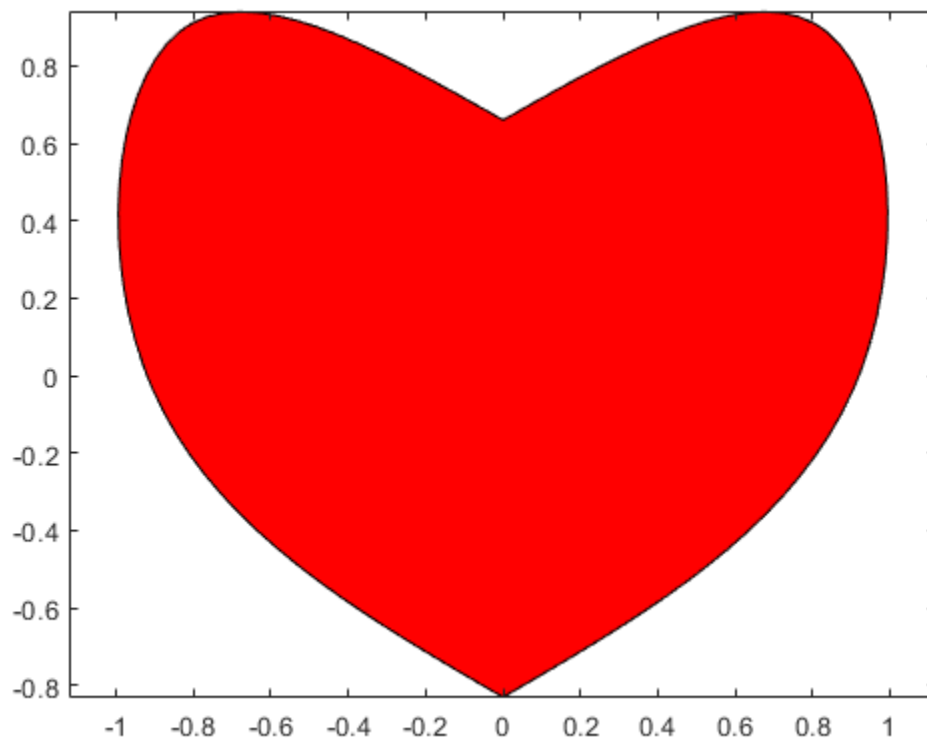
Input Arguments

`points` — Sequence of points
scalar | vector | matrix

Sequence of points at which the parametric "natural" cubic spline interpolates to, specified as a scalar, vector, or matrix.

Output Arguments

`curve` — Spline in ppform
spline structure

Spline in ppform, returned as a structure with these fields.

`Form` — Form of spline
pp

Form of the spline, returned as `pp`. `pp` indicates that the spline is given in piecewise polynomial form.

`Breaks` — Knot locations of spline
vector | cell array

Knot positions of the spline, returned as a vector or as a cell array of vectors for multivariate data. Vectors contain strictly increasing elements that represent the start and end of each of the intervals over which the polynomial pieces are defined.

`Coefs` — Coefficients of polynomials
matrix | array

Coefficients of polynomials for each piece, returned as a matrix or as an array for multivariate data.

`Pieces` — Number of polynomial pieces
scalar | vector

Number of polynomial pieces describing the spline, returned as a scalar or as a vector of numbers of pieces in each variable for multivariate data.

`Order` — Order of polynomials
scalar | vector

Order of the polynomial function describing each polynomial piece of the spline, returned as a scalar or as a vector containing the order in each variable for multivariate data.

`Dim` — Dimensionality
scalar

Dimensionality of the target function, returned as a scalar.

Algorithms

The function determines the break sequence `t` as

```
t = cumsum([0;((diff(points.').^2)*ones(d,1)).^(1/4)]).';
```

and uses `csape` (with either periodic or variational end conditions) to construct the smooth pieces between double points (if any).

References

[1] E. T. Y. Lee. "Choosing nodes in parametric curve interpolation." *Computer-Aided Design* 21 (1989), 363–370.

See Also
csape | fnplt | getcurve

Topics
About Splines in Curve Fitting on page 8-2
"The ppform" on page 10-8

Introduced in R2006b

datastats

Data statistics

Syntax

```
xds = datastats(x)
[xds,yds] = datastats(x,y)
```

Description

`xds = datastats(x)` returns statistics for the column vector `x` to the structure `xds`. Fields in `xds` are listed in the table below.

Field	Description
num	The number of data values
max	The maximum data value
min	The minimum data value
mean	The mean value of the data
median	The median value of the data
range	The range of the data
std	The standard deviation of the data

`[xds,yds] = datastats(x,y)` returns statistics for the column vectors `x` and `y` to the structures `xds` and `yds`, respectively. `xds` and `yds` contain the fields listed in the table above. `x` and `y` must be of the same size.

Examples

Compute statistics for the census data in `census.mat`:

```
load census
[xds,yds] = datastats(cdate,pop)
xds =
       num: 21
       max: 1990
       min: 1790
      mean: 1890
    median: 1890
     range: 200
       std: 62.048
yds =
       num: 21
       max: 248.7
       min: 3.9
      mean: 85.729
    median: 62.9
     range: 244.8
       std: 78.601
```

Tips

If x or y contains complex values, only the real parts are used in computing the statistics. Data containing Inf or NaN are processed using the usual MATLAB rules.

See Also

excludedata | smooth

Introduced before R2006a

dependnames

Dependent variable of `cfit`, `sfit`, or `fittype` object

Syntax

```
dep = dependnames(fun)
```

Description

`dep = dependnames(fun)` returns the (single) dependent variable name of the `cfit`, `sfit`, or `fittype` object `fun` as a 1-by-1 cell array of character vectors `dep`.

Examples

```
f1 = fittype('a*x^2+b*exp(n*x)');
dep1 = dependnames(f1)
dep1 =
    'y'

f2 = fittype('a*x^2+b*exp(n*x)','dependent','power');
dep2 = dependnames(f2)
dep2 =
    'power'
```

See Also
`fittype` | `formula` | `indepnames`

Introduced in R2006b

differentiate

Differentiate `cfit` or `sfit` object

Syntax

```
fx = differentiate(FO, X)
[fx, fxx] = differentiate(FO, X)
[fx, fy] = differentiate(FO, X, Y)
[fx, fy] = differentiate(FO, [X, Y])
[fx, fy, fxx, fxy, fyy] = differentiate(FO, ...)
```

Description

Note Use these syntaxes for `cfit` objects.

`fx = differentiate(FO, X)` differentiates the `cfit` object FO at the points specified by the vector X and returns the result in `fx`.

`[fx, fxx] = differentiate(FO, X)` differentiates the `cfit` object FO at the points specified by the vector X and returns the result in `fx` and the second derivative in `fxx`.

Note Use these syntaxes for `sfit` objects.

`[fx, fy] = differentiate(FO, X, Y)` differentiates the surface FO at the points specified by X and Y and returns the result in `fx` and `fy`.

FO is a surface fit (`sfit`) object generated by the `fit` function.

X and Y must be double-precision arrays and the same size and shape as each other.

All return arguments are the same size and shape as X and Y.

If FO represents the surface $z = f(x, y)$, then FX contains the derivatives with respect to x, that is, $\frac{df}{dx}$, and FY contains the derivatives with respect to y, that is, $\frac{df}{dy}$.

`[fx, fy] = differentiate(FO, [X, Y])`, where X and Y are column vectors, allows you to specify the evaluation points as a single argument.

`[fx, fy, fxx, fxy, fyy] = differentiate(FO, ...)` computes the first and second derivatives of the surface fit object FO.

`fxx` contains the second derivatives with respect to x, that is, $\frac{\partial^2 f}{\partial x^2}$.

`fxy` contains the mixed second derivatives, that is, $\frac{\partial^2 f}{\partial x \partial y}$.

fyy contains the second derivatives with respect to y, that is, $\frac{\partial^2 f}{\partial y^2}$.

Examples

Find the Derivatives of a Curve Using the `differentiate` Function

Create a baseline sinusoidal signal.

```
xdata = (0:.1:2*pi)';
y0 = sin(xdata);
```

Add response-dependent Gaussian noise to the signal.

```
noise = 2*y0.*randn(size(y0));
ydata = y0 + noise;
```

Fit the noisy data with a custom sinusoidal model.

```
f = fittype('a*sin(b*x)');
fit1 = fit(xdata,ydata,f,'StartPoint',[1 1]);
```

Find the derivatives of the fit at the predictors.

```
[d1,d2] = differentiate(fit1,xdata);
```

Plot the data, the fit, and the derivatives.

```
subplot(3,1,1)
plot(fit1,xdata,ydata) % cfit plot method
subplot(3,1,2)
plot(xdata,d1,'m') % double plot method
grid on
legend('1st derivative')
subplot(3,1,3)
plot(xdata,d2,'c') % double plot method
grid on
legend('2nd derivative')
```

You can also compute and plot derivatives directly with the `cfit plot` method, as follows:

```
figure
plot(fit1,xdata,ydata,{'fit','deriv1','deriv2'})
```
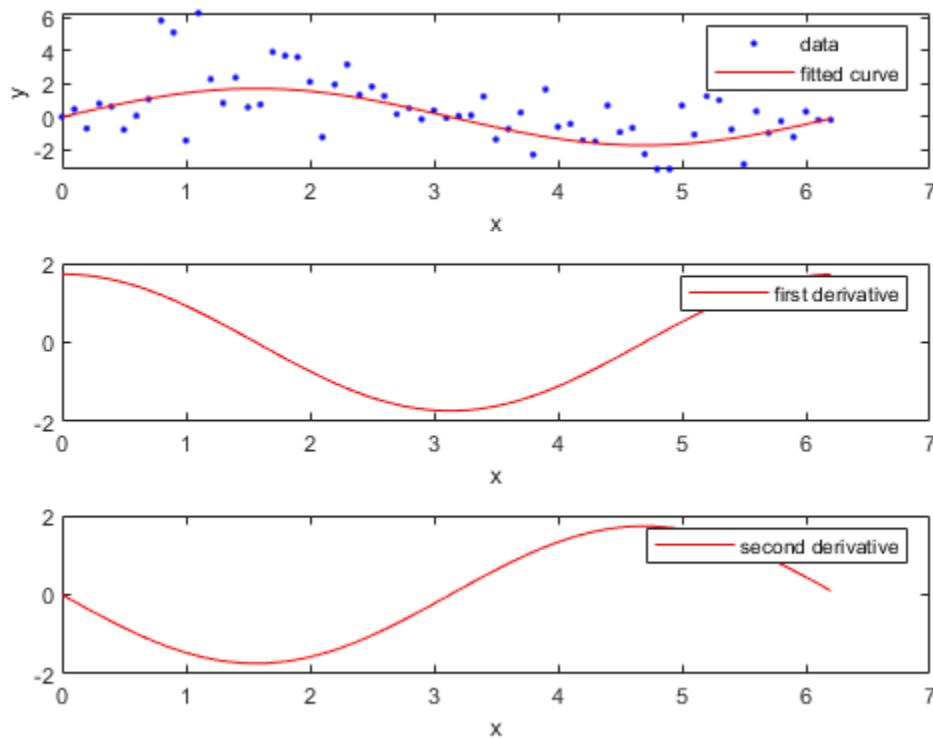
The `plot` method, however, does not return data on the derivatives, unlike the `differentiate` method.

Find the Derivatives of a Surface Using the `differentiate` Function

You can use the `differentiate` method to compute the gradients of a fit and then use the `quiver` function to plot these gradients as arrows. This example plots the gradients over the top of a contour plot.

Create the derivation points and fit the data.

```
x = [0.64;0.95;0.21;0.71;0.24;0.12;0.61;0.45;0.46;...
0.66;0.77;0.35;0.66];
y = [0.42;0.84;0.83;0.26;0.61;0.58;0.54;0.87;0.26;...
0.32;0.12;0.94;0.65];
z = [0.49;0.051;0.27;0.59;0.35;0.41;0.3;0.084;0.6;...
0.58;0.37;0.19;0.19];
fo = fit( [x, y], z, 'poly32', 'normalize', 'on' );
[xx, yy] = meshgrid( 0:0.04:1, 0:0.05:1 );
```

Compute the gradients of the fit using the `differentiate` function.

```
[fx, fy] = differentiate( fo, xx, yy );
```

Use the `quiver` function to plot the gradients.

```
plot( fo, 'Style', 'Contour' );
hold on
h = quiver( xx, yy, fx, fy, 'r', 'LineWidth', 2 );
hold off
colormap( copper )
```
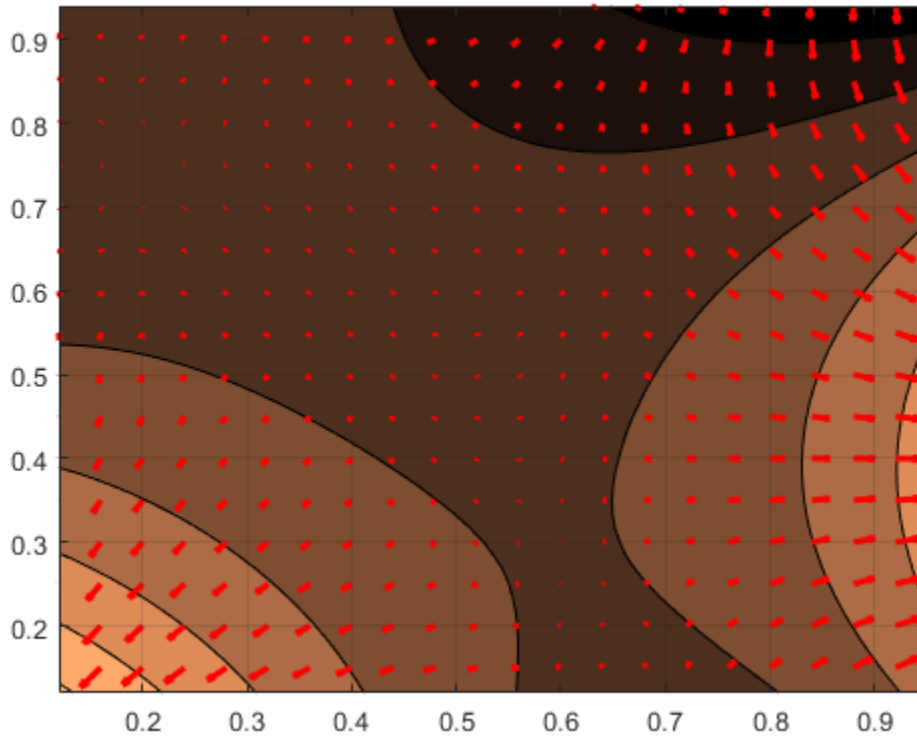

If you want to use derivatives in an optimization, you can, for example, implement an objective function for `fmincon` as follows.

```
function [z, g, H] = objectiveWithHessian( xy )

% The input xy represents a single evaluation point

z = f( xy );

if nargout > 1

[fx, fy, fxx, fxy, fyy] = differentiate( f, xy );

g = [fx, fy];

H = [fxx, fxy; fxy, fyy];

end

end
```

Input Arguments

F0 — cfit function
cfit | sfit

Function to differentiate, specified as a `cfit` object for curves or as a `sfit` object for surfaces.

X — Differentiation points
array

Points at which to differentiate the function, specified as a vector. For surfaces, this argument must have the same size and shape of **Y**.

Y — Differentiation points
array

Points at which to differentiate the function, specified as a vector. For surfaces, this argument must have the same size and shape of **X**.

Output Arguments

fx — First derivative with respect to x
vector

First derivative of the function, returned as a vector of the same size and shape of **X** and **Y**.

If **FO** is a surface, $z = f(x, y)$, then **fx** contains the derivatives with respect to x.

fxx — Second derivative with respect to x
vector

Second derivative of the function, returned as a vector of the same size and shape of **X** and **Y**.

If **FO** is a surface, $z = f(x, y)$, then **fxx** contains the second derivatives with respect to x.

fy — First derivative with respect to y
vector

First derivative of the function, returned as a vector of the same size and shape of **X** and **Y**.

If **FO** is a surface, $z = f(x, y)$, then **fy** contains the derivatives with respect to y.

fyy — Second derivative with respect to y
vector

Second derivative of the function, returned as a vector of the same size and shape of **X** and **Y**.

If **FO** is a surface, $z = f(x, y)$, then **fyy** contains the second derivatives with respect to y.

fxy — Mixed second derivative
vector

Mixed second derivative of the function, returned as a vector of the same size and shape of **X** and **Y**.

Tips

For library models with closed forms, the toolbox calculates derivatives analytically. For all other models, the toolbox calculates the first derivative using the centered difference quotient

$$\frac{df}{dx} = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x}$$

where x is the value at which the toolbox calculates the derivative, Δx is a small number (on the order of the cube root of eps), $f(x + \Delta x)$ is fun evaluated at $x + \Delta x$, and $f(x - x\Delta)$ is fun evaluated at $x - \Delta x$.

The toolbox calculates the second derivative using the expression

$$\frac{d^2 f}{dx^2} = \frac{f(x + \Delta x) + f(x - \Delta x) - 2f(x)}{(\Delta x)^2}$$

The toolbox calculates the mixed derivative for surfaces using the expression

$$\frac{\partial^2 f}{\partial x \partial y}(x, y) = \frac{f(x + \Delta x, y + \Delta y) - f(x - \Delta x, y + \Delta y) - f(x + \Delta x, y - \Delta y) + f(x - \Delta x, y - \Delta y)}{4\Delta x \Delta y}$$

See Also
fit | integrate | plot

Topics
About Splines in Curve Fitting on page 8-2

Introduced before R2006a

excludedata

Exclude data from fit

Syntax

```
tf = excludedata(x,y,'box',box)
tf = excludedata(x,y,'domain',domain)
tf = excludedata(x,y,'range',range)
tf = excludedata(x,y,'indices',indices)
```

Description

`tf = excludedata(x,y,'box',box)` returns a logical array that indicates which elements are outside the box in the xy-plane specified by `box`. The elements of `tf` equal 1 for data points outside the box and 0 for data points inside the box. To exclude data when fitting a curve using `fit`, specify `tf` as the `'Exclude'` value.

`tf = excludedata(x,y,'domain',domain)` identifies data points that have x-values outside the interval `domain`.

`tf = excludedata(x,y,'range',range)` identifies the data points with y-values outside the interval `range`.

`tf = excludedata(x,y,'indices',indices)` identifies the data points with indices equal to `indices`.

Examples

Visualize Exclusion Rules

Visualize exclusion rules using random data.

Generate random `x` and `y` data.

```
xdata = -3 + 6*rand(1,1e4);
ydata = -3 + 6*rand(1,1e4);
```

As an example, exclude data that is either inside the box `[-1 1 -1 1]` or outside the domain `[-2 2]`.

```
outliers1 = ~excludedata(xdata,ydata,'box',[-1 1 -1 1]);
outliers2 = excludedata(xdata,ydata,'domain',[-2 2]);
outliers = outliers1|outliers2;
```

Plot the data that is not excluded. The white area corresponds to regions that are excluded.

```
plot(xdata(~outliers),ydata(~outliers),'.')
axis([-3 3 -3 3])
axis square
```
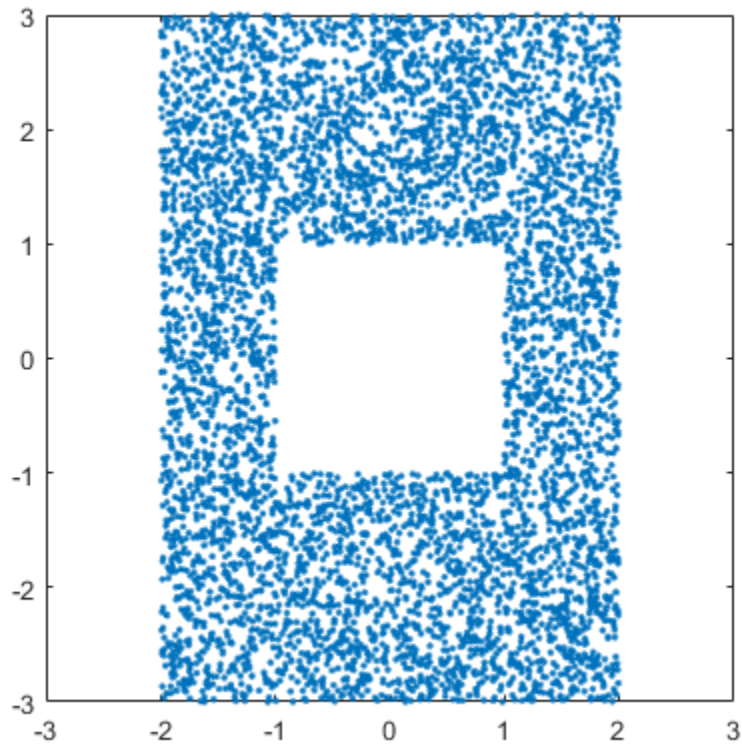
Exclude Data from Curve Fit

Load the vote counts and county names for the state of Florida from the 2000 U.S. presidential election.

```
load flvote2k
```

Use the vote counts for the two major party candidates, Bush and Gore, as predictors for the vote counts for the third-party candidate Buchanan, and plot the scatters:

```
plot(bush,buchanan,'rs')
hold on
plot(gore,buchanan,'bo')
legend('Bush data','Gore data')
```
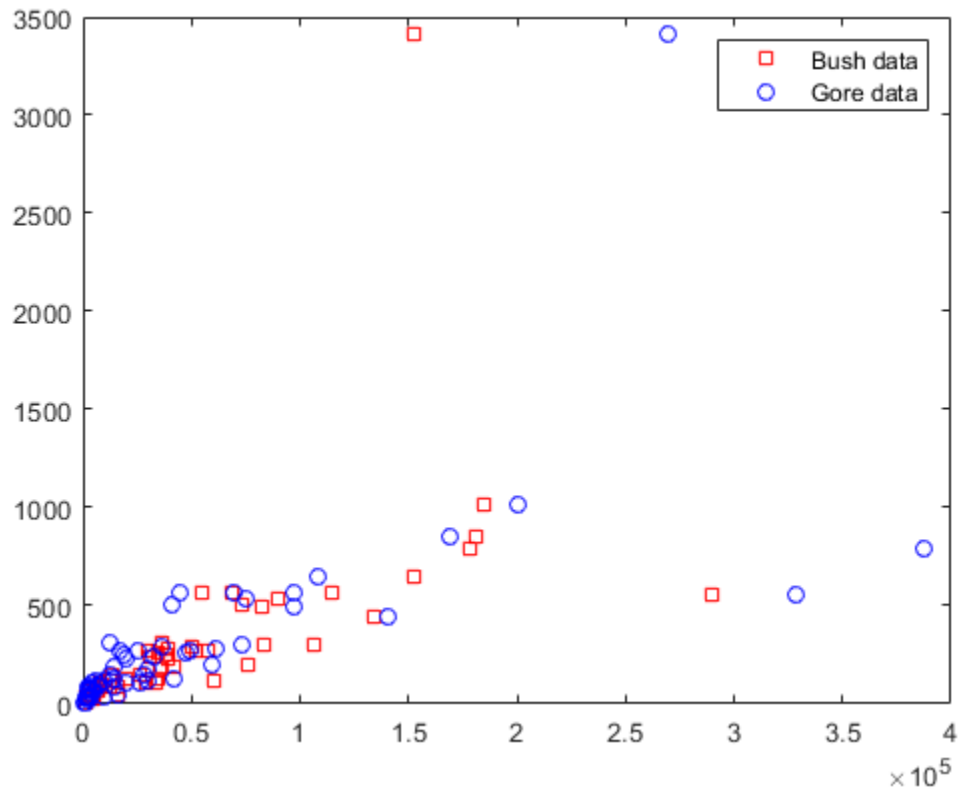
Assume a model where a fixed proportion of Bush or Gore voters choose to vote for Buchanan.

```
f = fittype({'x'})

f =
    Linear model:
    f(a,x) = a*x
```

Exclude the data from absentee voters, who did not use the controversial "butterfly" ballot.

```
nobutterfly = strcmp(counties,'Absentee Ballots');
```

Perform a bisquare weights robust fit of the model to the two data sets, excluding absentee voters.

```
bushfit = fit(bush,buchanan,f,'Exclude',nobutterfly,'Robust','on');
gorefit = fit(gore,buchanan,f,'Exclude',nobutterfly,'Robust','on');
```

Robust fits give outliers a low weight, so large residuals from a robust fit can be used to identify the outliers.

```
figure
plot(bushfit,bush,buchanan,'rs','residuals')
hold on
plot(gorefit,gore,buchanan,'bo','residuals')
```
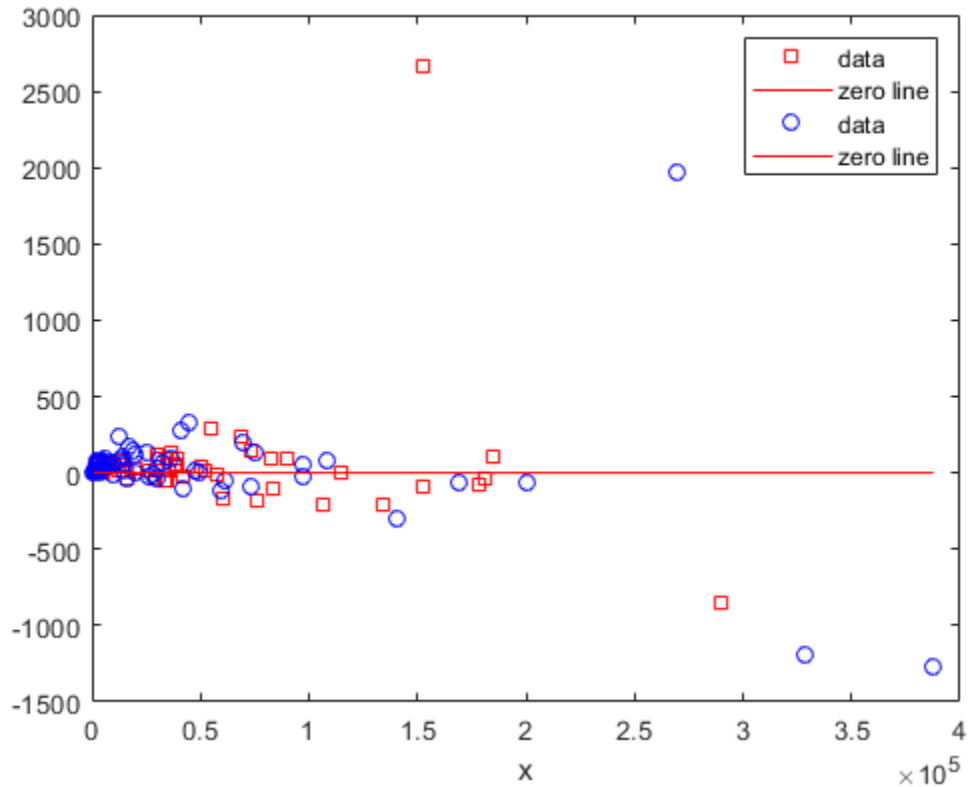
Calculate the residuals.

```
bushres = buchanan - feval(bushfit,bush);
goreres = buchanan - feval(gorefit,gore);
```

Identify large residuals as those outside the range [-500 500].

```
bushoutliers = excludedata(bush,bushres,'range',[-500 500]);
goreoutliers = excludedata(gore,goreres,'range',[-500 500]);
```

Display the counties corresponding to the outliers. Miami-Dade and Broward counties correspond to the largest predictor values. Palm Beach county, the only county in the state to use the "butterfly" ballot, corresponds to the largest residual values.

```
counties(bushoutliers)

ans = 2x1 cell
    {'Miami-Dade'}
    {'Palm Beach'}


counties(goreoutliers)

ans = 3x1 cell
    {'Broward'   }
    {'Miami-Dade'}
    {'Palm Beach'}
```

Input Arguments

x — Data sites
numeric vector

Data sites of data values, specified as a numeric vector.

y — Data values
numeric vector

Data values, specified as a numeric vector.

box — Box to find data outside of
numeric vector with four elements

Box to find data outside of, specified as a numeric vector `[xmin xmax ymin ymax]` with four elements.

Example: [-1 1 0 2]

domain — Domain to find data outside of
numeric vector with two elements

Domain to find data outside of, specified as a numeric vector `[xmin xmax]` with two elements.

Example: [-1 1]

range — Range to find data outside of
numeric vector with two elements

Domain to find data outside of, specified as a numeric vector `[ymin ymax]` with two elements.

Example: [3 4]

indices — Indices of data points to find
numeric vector

Indices of data points to find, specified as a numeric vector.

Example: [3 7 9]

See Also
`fit` | `fitoptions`

Introduced before R2006a

feval

Evaluate `cfit`, `sfit`, or `fittype` object

Syntax

```
y = feval(cfun,x)
z = feval(sfun,[x,y])
z = feval(sfun,x,y)
y = feval(ffun,coeff1,coeff2,...,x)
z = feval(ffun,coeff1,coeff2,...,x,y)
```

Description

Note You can use `feval` to evaluate fits, but you can treat fit objects as functions and call `feval` indirectly using this syntax instead:

```
y = cfun(x)         % cfit objects;
z = sfun(x,y)       % sfit objects
z = sfun([x, y])    % sfit objects
y = ffun(coef1,coef2,...,x)    % curve fittype objects;
z = ffun(coef1,coef2,...,x,y) % surface fittype objects;
```

Alternatively, you can use the `feval` method to evaluate the estimated function, either at your original data points, or at new locations. The latter is referred to as interpolation or prediction, depending on the type of model. You can also use `feval` to extrapolate the estimated function's value at new locations that are not within the range of the original data.

`y = feval(cfun,x)` evaluates the `cfit` object `cfun` at the predictor values in the column vector `x` and returns the response values in the column vector `y`.

`z = feval(sfun,[x,y])` evaluates the `sfit` object `sfun` at the predictor values in the two column matrix `[x,y]` and returns the response values in the column vector `z`.

`z = feval(sfun,x,y)` evaluates the `sfit` object `sfun` at the predictor values in the matrices `x` and `y` that must be the same size. It returns the response values in the matrix `z` that will be the same size as `x` and `y`.

`y = feval(ffun,coeff1,coeff2,...,x)` assigns the coefficients `coeff1`, `coeff2`, etc. to the `fittype` object `ffun`, evaluates it at the predictor values in the column vector `x`, and returns the response values in the column vector `y`. `ffun` cannot be a `cfit` object in this syntax. To evaluate `cfit` objects, use the first syntax.

`z = feval(ffun,coeff1,coeff2,...,x,y)` achieves a similar result for a `fittype` object for a surface.

Examples

Evaluate `fittype` and `cfit` objects using the `confint` function

Create the `fittype` and `cfit` objects, and a random matrix of predictor values.

```
f = fittype('a*x^2+b*exp(n*x)');
c = cfit(f,1,10.3,-1e2);
X = rand(2)

X =
    0.0579    0.8132
    0.3529    0.0099
```

To evaluate the `fittype` object, *f*, call the `feval` function.

```
y1 = feval(f,1,10.3,-1e2,X)

y1 =
    0.0349    0.6612
    0.1245    3.8422
```

Alternatively, you can treat fit objects as functions and call `feval` indirectly using this syntax.

```
y1 = f(1,10.3,-1e2,X)

y1 =
    0.0349    0.6612
    0.1245    3.8422
```

Now evaluate the `cfit` object, `c`.

```
y2 = feval(c,X)

y2 =
    0.0349
    0.1245
    0.6612
    3.8422
```

Alternatively, call `feval` indirectly.

```
y2 = c(X)

y2 =
    0.0349
    0.1245
    0.6612
    3.8422
```

Input Arguments

cfun — cfit function
cfit

Function to evaluate, specified as a `cfit` object.

sfun — sfit function
sfit

Function to evaluate, specified as an `sfit` object.

ffun — fittype function
fittype

Function to evaluate, specified as a fittype object.

x — Evaluation points
vector | matrix

Points at which to evaluate the function, specified as a vector or matrix.

y — Evaluation points
vector | matrix

Points at which to evaluate the function, specified as a vector matrix.

coeff1,coeff2,... — Coefficients
scalar

One or more coefficients assigned to the fittype object ffun, specified as scalars.

Output Arguments

y — Response values
column vector

Response values of the function evaluated at the predictors values in the column vector *x*, returned as a column vector.

z — Response values
matrix

Response values of the function evaluated at the predictors values in the two column matrix *[x,y]*, returned as a matrix.

See Also
cfit | fit | fittype

Topics
About Splines in Curve Fitting on page 8-2

Introduced before R2006a

fit

Fit curve or surface to data

Syntax

```
fitobject = fit(x,y,fitType)
fitobject = fit([x,y],z,fitType)
fitobject = fit(x,y,fitType,fitOptions)
fitobject = fit(x,y,fitType,Name,Value)
[fitobject,gof] = fit(x,y,fitType)
[fitobject,gof,output] = fit(x,y,fitType)
```

Description

`fitobject = fit(x,y,fitType)` creates the fit to the data in x and y with the model specified by fitType.

`fitobject = fit([x,y],z,fitType)` creates a surface fit to the data in vectors x, y, and z.

`fitobject = fit(x,y,fitType,fitOptions)` creates a fit to the data using the algorithm options specified by the `fitOptions` object.

`fitobject = fit(x,y,fitType,Name,Value)` creates a fit to the data using the library model fitType with additional options specified by one or more `Name,Value` pair arguments. Use `fitoptions` to display available property names and default values for the specific library model.

`[fitobject,gof] = fit(x,y,fitType)` returns goodness-of-fit statistics in the structure gof.

`[fitobject,gof,output] = fit(x,y,fitType)` returns fitting algorithm information in the structure output.

Examples

Fit a Quadratic Curve

Load some data, fit a quadratic curve to variables `cdate` and `pop`, and plot the fit and data.

```
load census;
f=fit(cdate,pop,'poly2')

f =
    Linear model Poly2:
    f(x) = p1*x^2 + p2*x + p3
    Coefficients (with 95% confidence bounds):
      p1 =    0.006541  (0.006124, 0.006958)
      p2 =       -23.51  (-25.09, -21.93)
      p3 =    2.113e+04  (1.964e+04, 2.262e+04)

plot(f,cdate,pop)
```
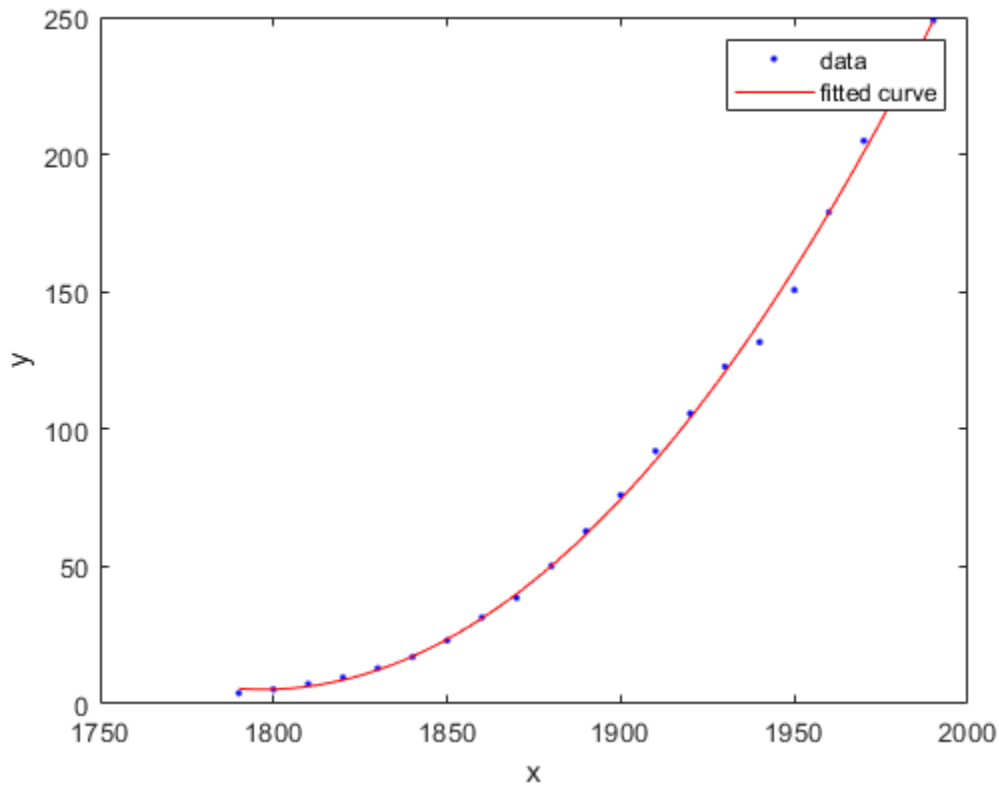
For a list of library model names, see `fitType`.

Fit a Polynomial Surface

Load some data and fit a polynomial surface of degree 2 in x and degree 3 in y. Plot the fit and data.

```
load franke
sf = fit([x, y],z,'poly23')

        Linear model Poly23:
        sf(x,y) = p00 + p10*x + p01*y + p20*x^2 + p11*x*y + p02*y^2 + p21*x^2*y
                        + p12*x*y^2 + p03*y^3
        Coefficients (with 95% confidence bounds):
          p00 =        1.118  (0.9149, 1.321)
          p10 =   -0.0002941  (-0.000502, -8.623e-05)
          p01 =        1.533  (0.7032, 2.364)
          p20 =   -1.966e-08  (-7.084e-08, 3.152e-08)
          p11 =    0.0003427  (-0.0001009, 0.0007863)
          p02 =       -6.951  (-8.421, -5.481)
          p21 =    9.563e-08  (6.276e-09, 1.85e-07)
          p12 =   -0.0004401  (-0.0007082, -0.0001721)
          p03 =        4.999  (4.082, 5.917)

plot(sf,[x,y],z)
```
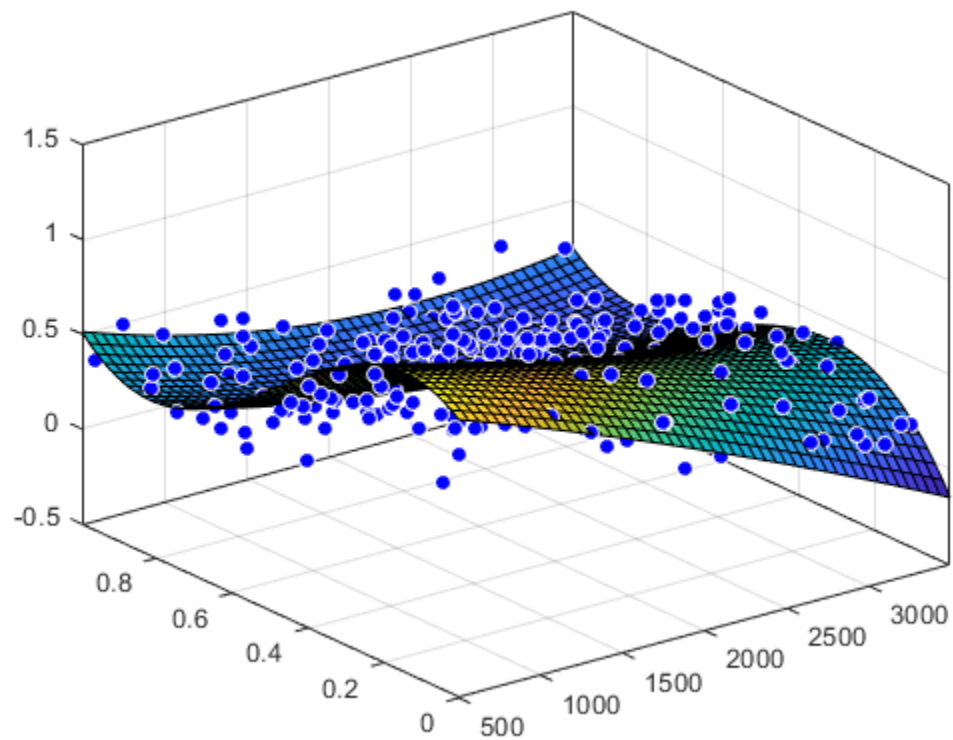
Fit a Surface Using Variables in a MATLAB Table

Load the `franke` data and convert it to a MATLAB® table.

```
load franke
T = table(x,y,z);
```

Specify the variables in the table as inputs to the `fit` function, and plot the fit.

```
f = fit([T.x, T.y],T.z,'linearinterp');
plot( f, [T.x, T.y], T.z )
```
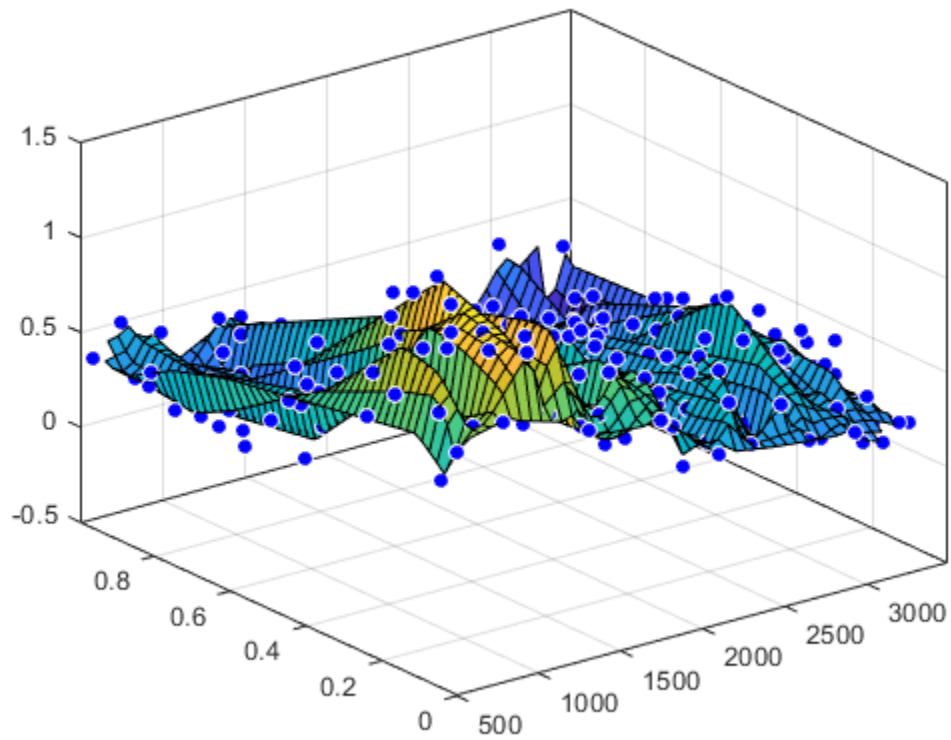
Create Fit Options and Fit Type Before Fitting

Load and plot the data, create fit options and fit type using the `fittype` and `fitoptions` functions, then create and plot the fit.

Load and plot the data in `census.mat`.

```
load census
plot(cdate,pop,'o')
```
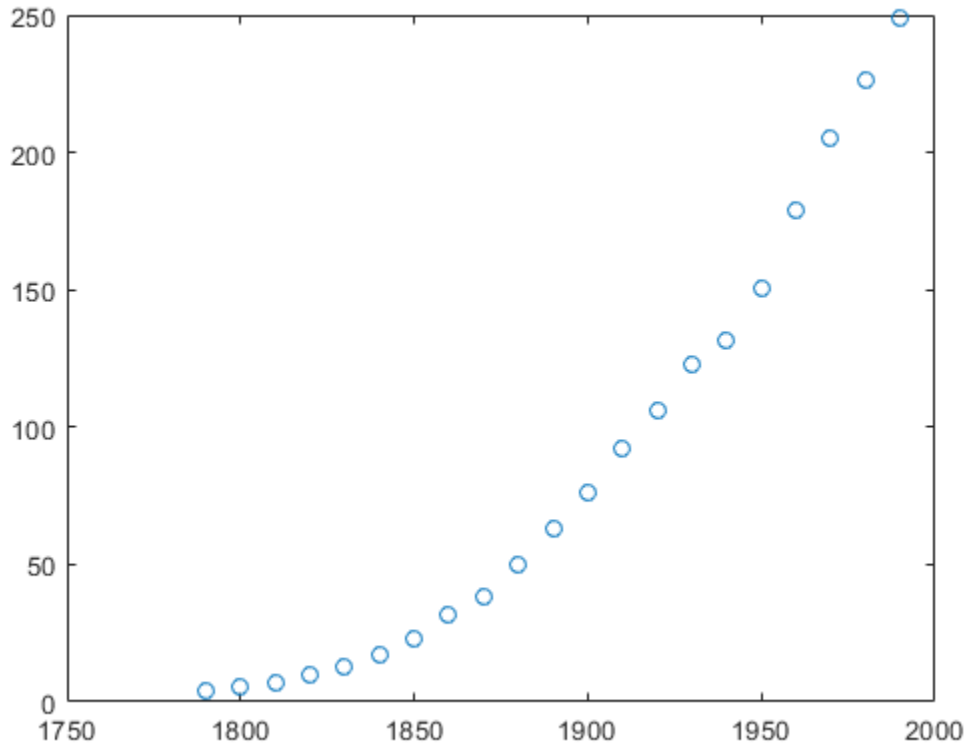
Create a fit options object and a fit type for the custom nonlinear model $y = a(x - b)^n$, where a and b are coefficients and n is a problem-dependent parameter.

```
fo = fitoptions('Method','NonlinearLeastSquares',...
                'Lower',[0,0],...
                'Upper',[Inf,max(cdate)],...
                'StartPoint',[1 1]);
ft = fittype('a*(x-b)^n','problem','n','options',fo);
```

Fit the data using the fit options and a value of $n = 2$.

```
[curve2,gof2] = fit(cdate,pop,ft,'problem',2)
```

```
curve2 =
     General model:
     curve2(x) = a*(x-b)^n
     Coefficients (with 95% confidence bounds):
       a =    0.006092  (0.005743, 0.006441)
       b =        1789  (1784, 1793)
     Problem parameters:
       n =           2

gof2 = struct with fields:
           sse: 246.1543
       rsquare: 0.9980
           dfe: 19
     adjrsquare: 0.9979
```

```
        rmse: 3.5994
```

Fit the data using the fit options and a value of $n = 3$.

```
[curve3,gof3] = fit(cdate,pop,ft,'problem',3)

curve3 =
     General model:
     curve3(x) = a*(x-b)^n
     Coefficients (with 95% confidence bounds):
       a =   1.359e-05  (1.245e-05, 1.474e-05)
       b =         1725  (1718, 1731)
     Problem parameters:
       n =              3

gof3 = struct with fields:
          sse: 232.0058
      rsquare: 0.9981
          dfe: 19
   adjrsquare: 0.9980
         rmse: 3.4944
```

Plot the fit results with the data.

```
hold on
plot(curve2,'m')
plot(curve3,'c')
legend('Data','n=2','n=3')
hold off
```
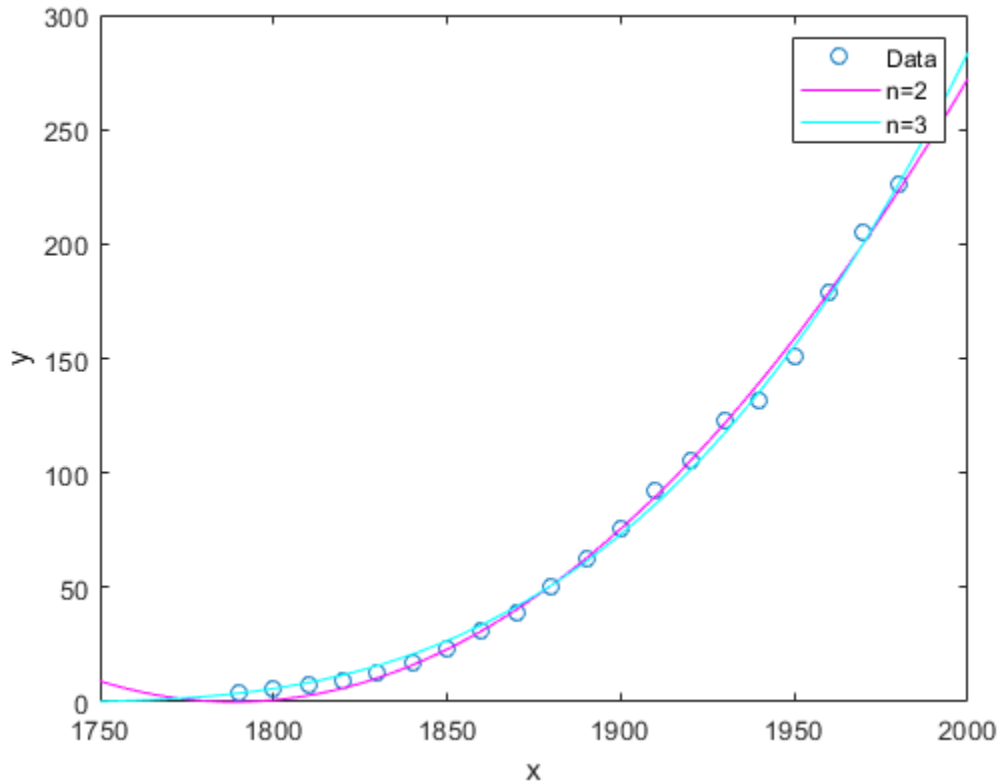
Fit a Cubic Polynomial Specifying Normalize and Robust Options

Load some data and fit and plot a cubic polynomial with center and scale (`Normalize`) and robust fitting options.

```
load census;
f=fit(cdate,pop,'poly3','Normalize','on','Robust','Bisquare')

f =
     Linear model Poly3:
     f(x) = p1*x^3 + p2*x^2 + p3*x + p4
       where x is normalized by mean 1890 and std 62.05
     Coefficients (with 95% confidence bounds):
       p1 =     -0.4619  (-1.895, 0.9707)
       p2 =       25.01  (23.79, 26.22)
       p3 =       77.03  (74.37, 79.7)
       p4 =       62.81  (61.26, 64.37)

plot(f,cdate,pop)
```

Fit a Curve Defined by a File

Define a function in a file and use it to create a fit type and fit a curve.

Define a function in a MATLAB file.

```
function y = piecewiseLine(x,a,b,c,d,k)
% PIECEWISELINE   A line made of two pieces
% that is not continuous.

y = zeros(size(x));

% This example includes a for-loop and if statement
% purely for example purposes.
for i = 1:length(x)
    if x(i) < k,
        y(i) = a + b.* x(i);
    else
        y(i) = c + d.* x(i);
    end
end
end
```

Save the file.

Define some data, create a fit type specifying the function `piecewiseLine`, create a fit using the fit type `ft`, and plot the results.

```
x = [0.81;0.91;0.13;0.91;0.63;0.098;0.28;0.55;...
0.96;0.96;0.16;0.97;0.96];
y = [0.17;0.12;0.16;0.0035;0.37;0.082;0.34;0.56;...
0.15;-0.046;0.17;-0.091;-0.071];
ft = fittype( 'piecewiseLine( x, a, b, c, d, k )' )
f = fit( x, y, ft, 'StartPoint', [1, 0, 1, 0, 0.5] )
plot( f, x, y )
```

Exclude Points from Fit

Load some data and fit a custom equation specifying points to exclude. Plot the results.

Load data and define a custom equation and some start points.

```
[x, y] = titanium;

gaussEqn = 'a*exp(-((x-b)/c)^2)+d'

gaussEqn =
'a*exp(-((x-b)/c)^2)+d'

startPoints = [1.5 900 10 0.6]

startPoints = 1×4

    1.5000  900.0000   10.0000    0.6000
```

Create two fits using the custom equation and start points, and define two different sets of excluded points, using an index vector and an expression. Use `Exclude` to remove outliers from your fit.

```
f1 = fit(x',y',gaussEqn,'Start', startPoints, 'Exclude', [1 10 25])

f1 =
    General model:
    f1(x) = a*exp(-((x-b)/c)^2)+d
    Coefficients (with 95% confidence bounds):
      a =        1.493  (1.432, 1.554)
      b =        897.4  (896.5, 898.3)
      c =         27.9  (26.55, 29.25)
      d =       0.6519  (0.6367, 0.6672)

f2 = fit(x',y',gaussEqn,'Start', startPoints, 'Exclude', x < 800)

f2 =
    General model:
    f2(x) = a*exp(-((x-b)/c)^2)+d
    Coefficients (with 95% confidence bounds):
      a =        1.494  (1.41, 1.578)
      b =        897.4  (896.2, 898.7)
      c =        28.15  (26.22, 30.09)
      d =       0.6466  (0.6169, 0.6764)
```

Plot both fits.

```
plot(f1,x,y)
title('Fit with data points 1, 10, and 25 excluded')
```
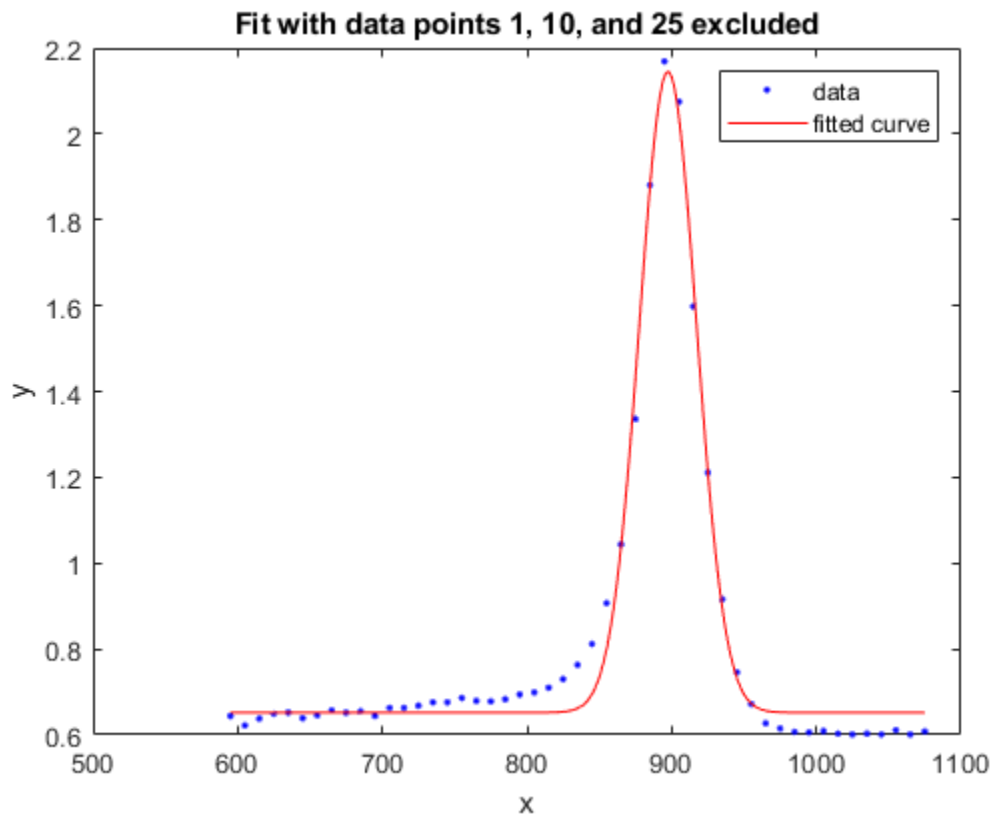


```
figure
plot(f2,x,y)
title('Fit with data points excluded such that x < 800')
```

Fit with data points excluded such that x < 800
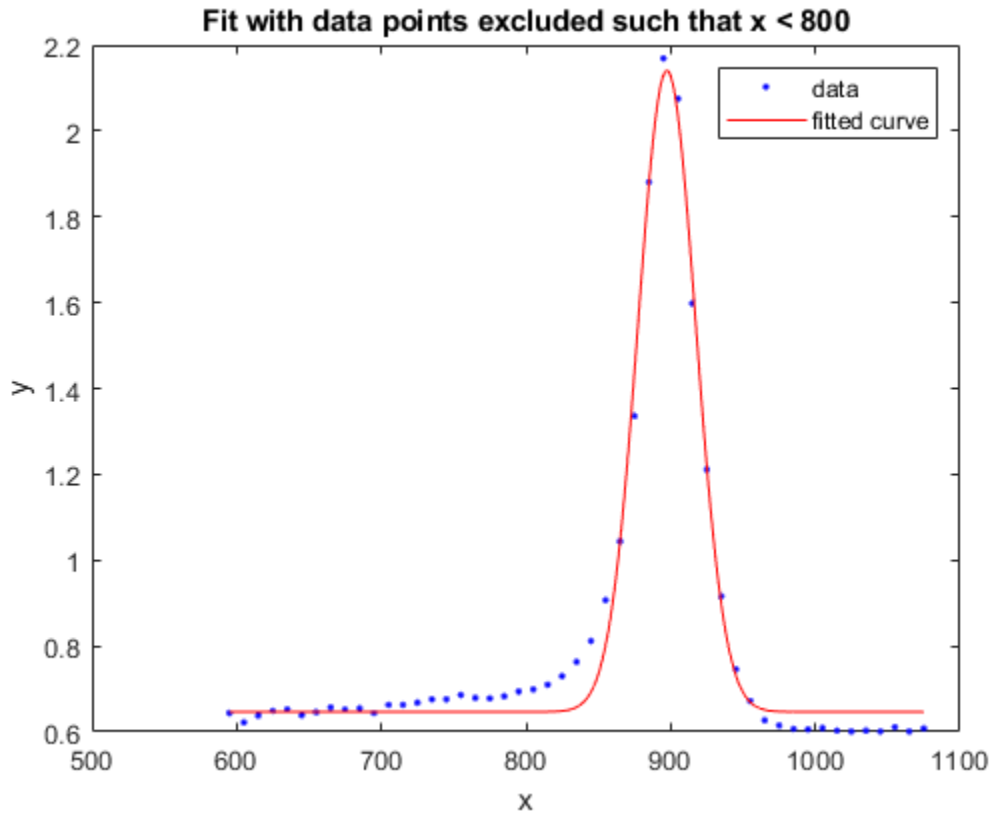
Exclude Points and Plot Fit Showing Excluded Data

You can define the excluded points as variables before supplying them as inputs to the fit function. The following steps recreate the fits in the previous example and allow you to plot the excluded points as well as the data and the fit.

Load data and define a custom equation and some start points.

```
[x, y] = titanium;
```

```
gaussEqn = 'a*exp(-((x-b)/c)^2)+d'
```

```
gaussEqn =
'a*exp(-((x-b)/c)^2)+d'
```

```
startPoints = [1.5 900 10 0.6]
```

```
startPoints = 1×4

    1.5000  900.0000   10.0000    0.6000
```

Define two sets of points to exclude, using an index vector and an expression.

```
exclude1 = [1 10 25];
exclude2 = x < 800;
```

Create two fits using the custom equation, startpoints, and the two different excluded points.

```
f1 = fit(x',y',gaussEqn,'Start', startPoints, 'Exclude', exclude1);
f2 = fit(x',y',gaussEqn,'Start', startPoints, 'Exclude', exclude2);
```

Plot both fits and highlight the excluded data.

```
plot(f1,x,y,exclude1)
title('Fit with data points 1, 10, and 25 excluded')
```
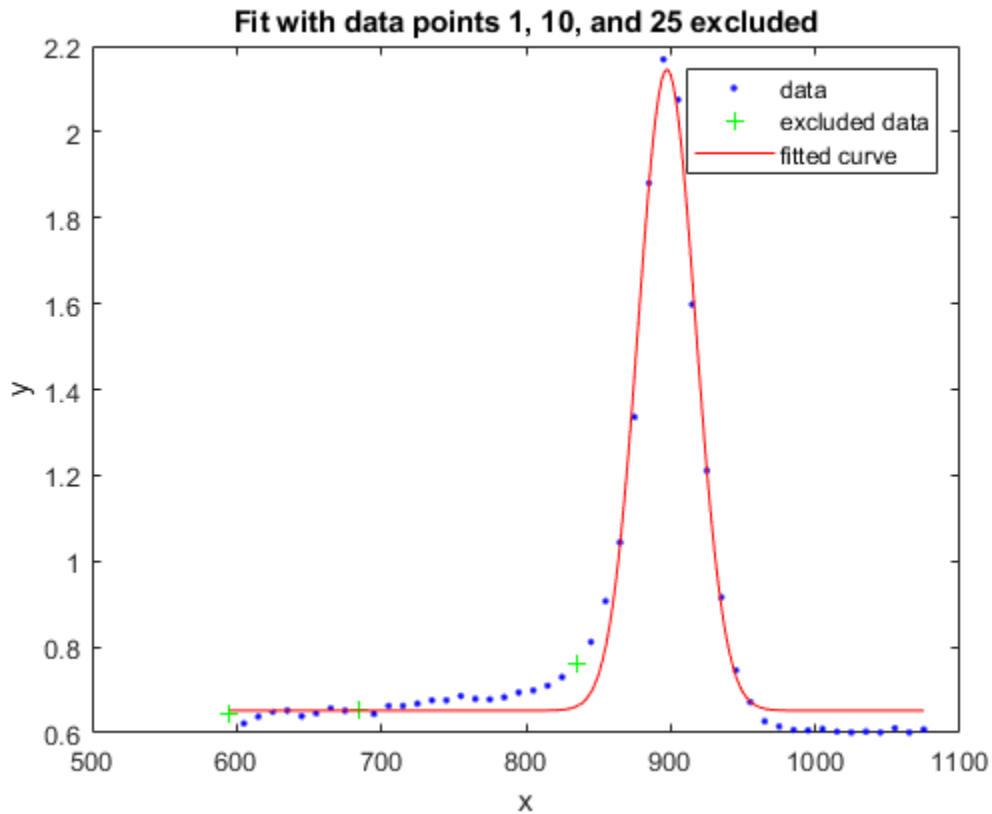


```
figure;
plot(f2,x,y,exclude2)
title('Fit with data points excluded such that x < 800')
```
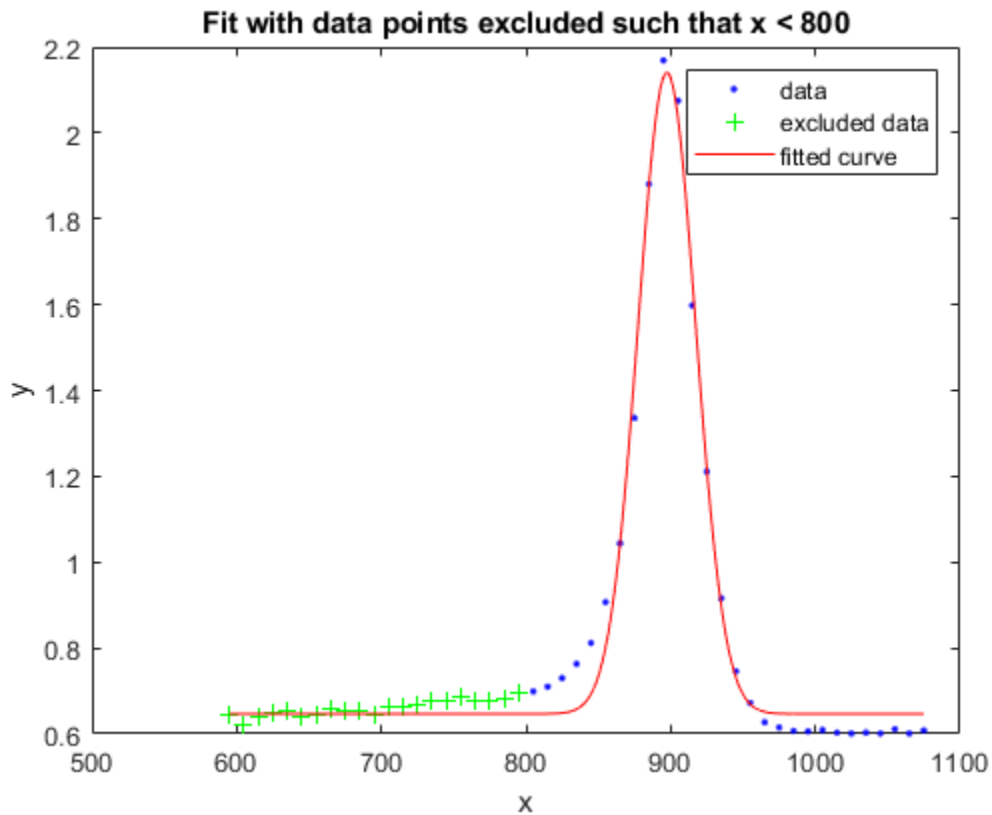
For a surface fitting example with excluded points, load some surface data and create and plot fits specifying excluded data.

```
load franke
f1 = fit([x y],z,'poly23', 'Exclude', [1 10 25]);
f2 = fit([x y],z,'poly23', 'Exclude', z > 1);

figure
plot(f1, [x y], z, 'Exclude', [1 10 25]);
title('Fit with data points 1, 10, and 25 excluded')
```

Fit with data points 1, 10, and 25 excluded
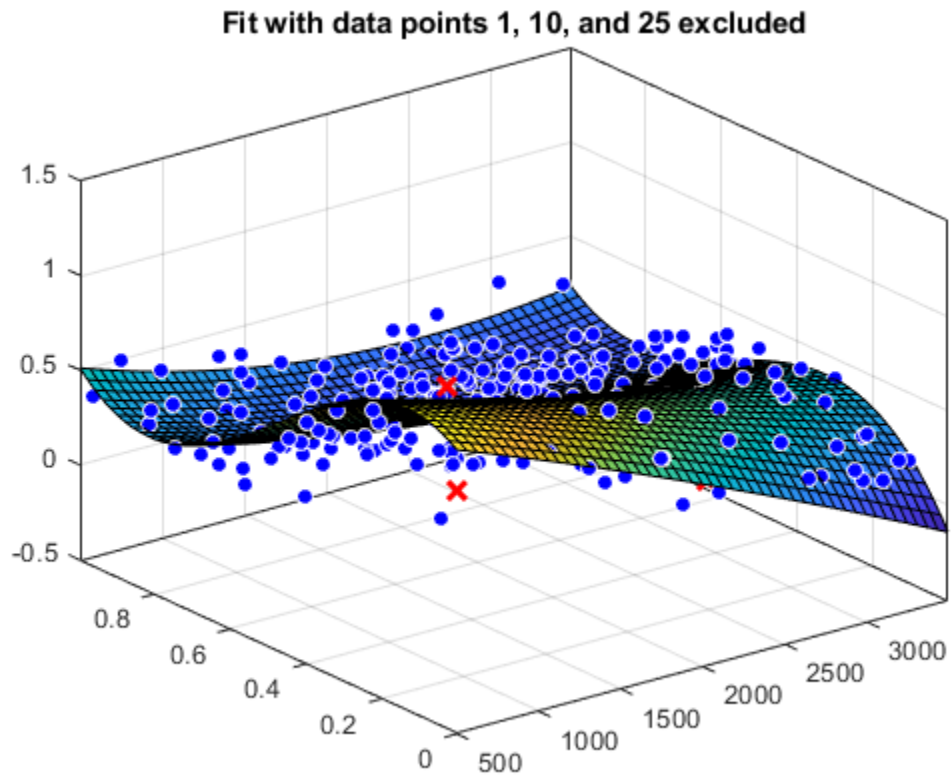

```
figure
plot(f2, [x y], z, 'Exclude', z > 1);
title('Fit with data points excluded such that z > 1')
```

Fit with data points excluded such that z > 1
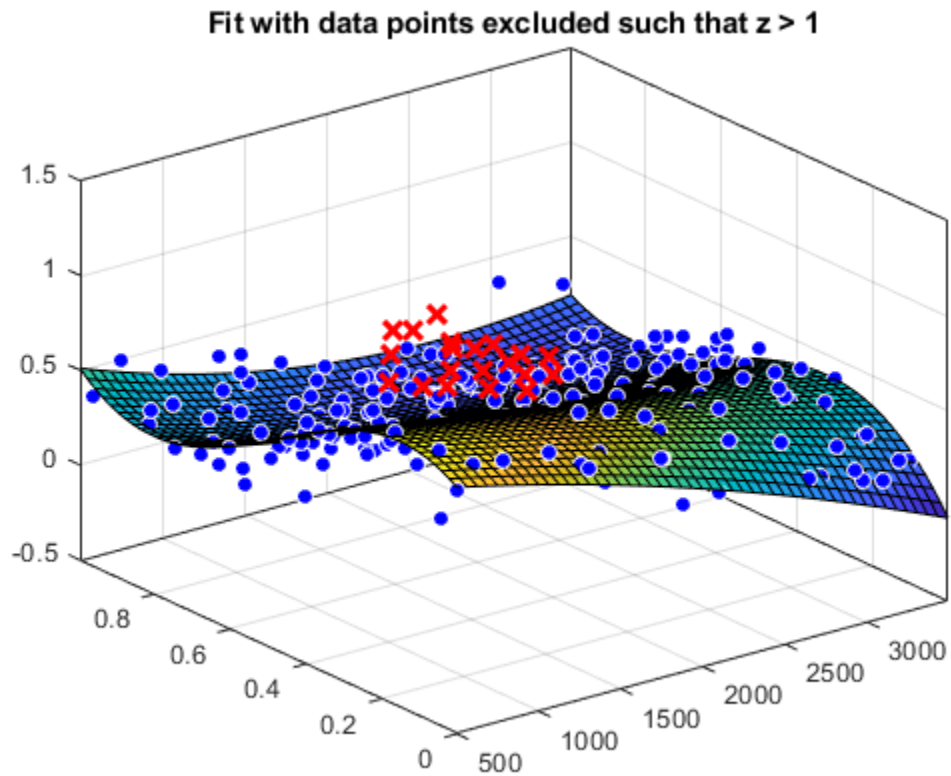
Fit a Smoothing Spline Curve and Return Goodness-of-Fit Information

Load some data and fit a smoothing spline curve through variables `month` and `pressure`, and return goodness of fit information and the output structure. Plot the fit and the residuals against the data.

```
load enso;
[curve, goodness, output] = fit(month,pressure,'smoothingspline');
plot(curve,month,pressure);
xlabel('Month');
ylabel('Pressure');
```
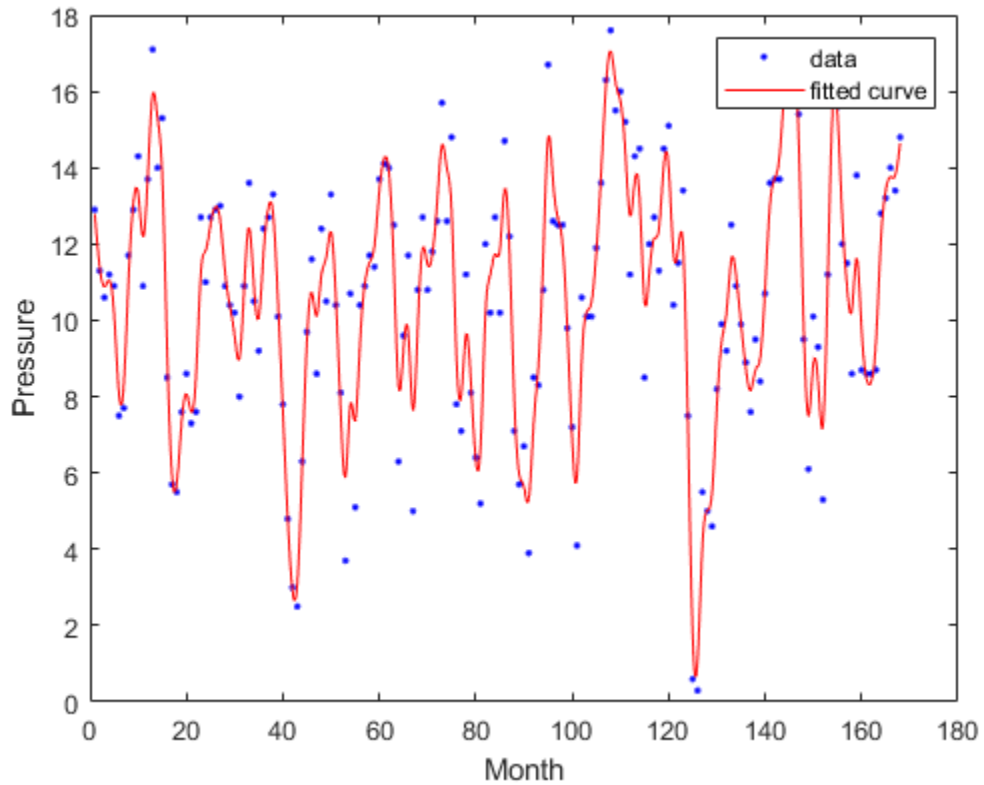
Plot the residuals against the x-data (`month`).

```
plot( curve, month, pressure, 'residuals' )
xlabel( 'Month' )
ylabel( 'Residuals' )
```
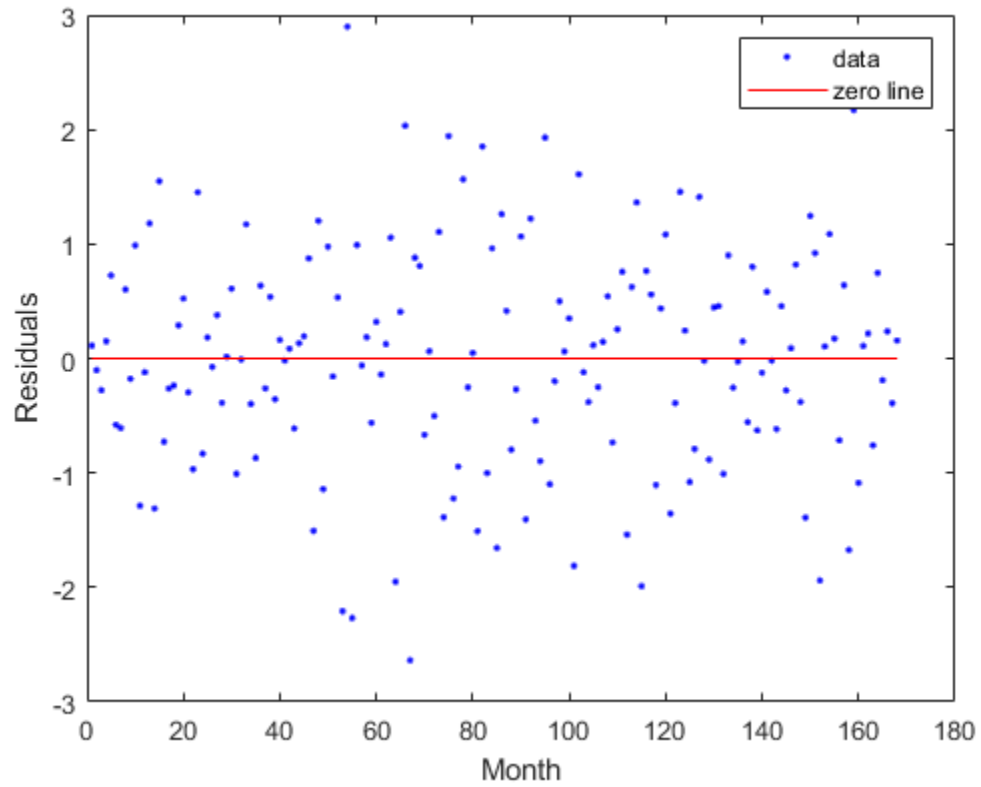
Use the data in the `output` structure to plot the residuals against the y-data (`pressure`).

```
plot( pressure, output.residuals, '.' )
xlabel( 'Pressure' )
ylabel( 'Residuals' )
```
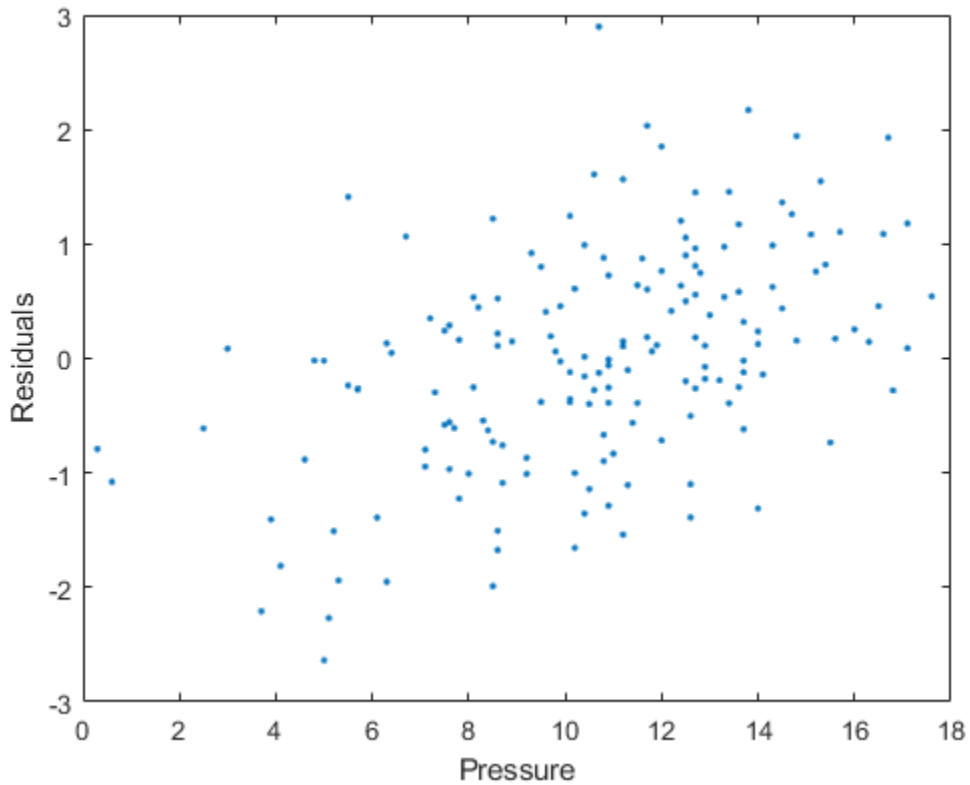
Fit a Single-Term Exponential

Generate data with an exponential trend, and then fit the data using the first equation in the curve fitting library of exponential models (a single-term exponential). Plot the results.

```
x = (0:0.2:5)';
y = 2*exp(-0.2*x) + 0.5*randn(size(x));
f = fit(x,y,'exp1');
plot(f,x,y)
```
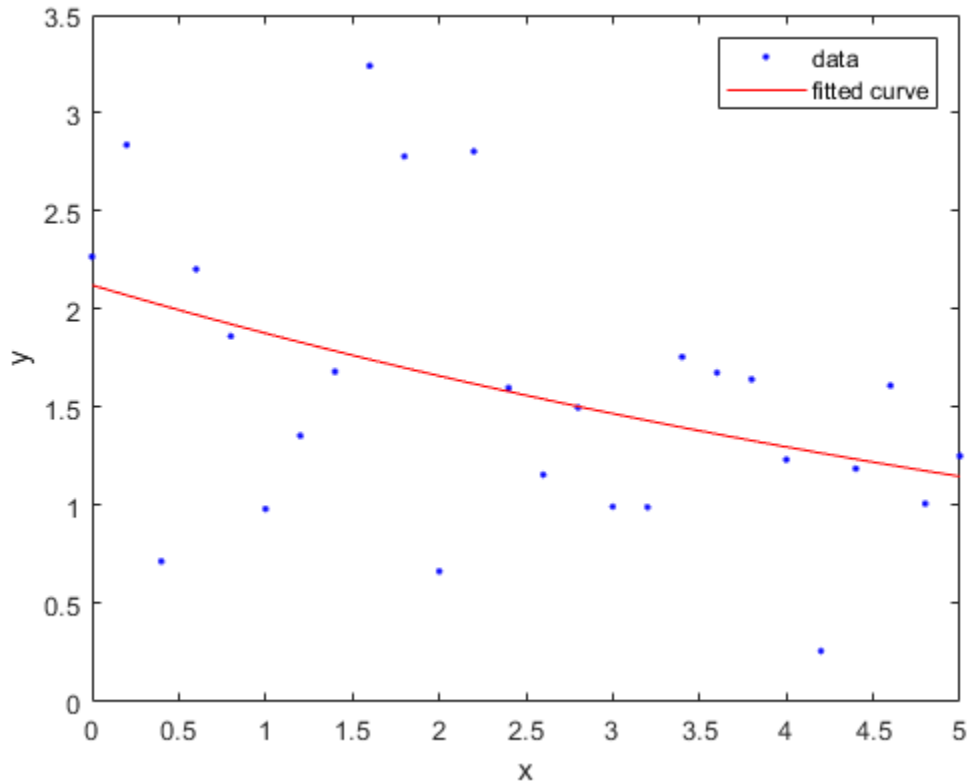
Fit a Custom Model Using an Anonymous Function

You can use anonymous functions to make it easier to pass other data into the `fit` function.

Load data and set `Emax` to `1` before defining your anonymous function:

```
data = importdata( 'OpioidHypnoticSynergy.txt' );
Propofol      = data.data(:,1);
Remifentanil  = data.data(:,2);
Algometry     = data.data(:,3);
Emax = 1;
```

Define the model equation as an anonymous function:

```
Effect = @(IC50A, IC50B, alpha, n, x, y) ...
    Emax*( x/IC50A + y/IC50B + alpha*( x/IC50A )...
    .* ( y/IC50B ) ).^n ./(( x/IC50A + y/IC50B + ...
    alpha*( x/IC50A ) .* ( y/IC50B ) ).^n  + 1);
```

Use the anonymous function `Effect` as an input to the `fit` function, and plot the results:

```
AlgometryEffect = fit( [Propofol, Remifentanil], Algometry, Effect, ...
    'StartPoint', [2, 10, 1, 0.8], ...
    'Lower', [-Inf, -Inf, -5, -Inf], ...
```

```
    'Robust', 'LAR' )
plot( AlgometryEffect, [Propofol, Remifentanil], Algometry )
```

For more examples using anonymous functions and other custom models for fitting, see the `fittype` function.

Find Coefficient Order to Set Start Points and Bounds

For the properties `Upper`, `Lower`, and `StartPoint`, you need to find the order of the entries for coefficients.

Create a fit type.

```
ft = fittype('b*x^2+c*x+a');
```

Get the coefficient names and order using the `coeffnames` function.

```
coeffnames(ft)

ans = 3x1 cell
    {'a'}
    {'b'}
    {'c'}
```

Note that this is different from the order of the coefficients in the expression used to create `ft` with `fittype`.

Load data, create a fit and set the start points.

```
load enso
fit(month,pressure,ft,'StartPoint',[1,3,5])

ans =
    General model:
    ans(x) = b*x^2+c*x+a
    Coefficients (with 95% confidence bounds):
      a =        10.94  (9.362, 12.52)
      b =    0.0001677  (-7.985e-05, 0.0004153)
      c =      -0.0224  (-0.06559, 0.02079)
```

This assigns initial values to the coefficients as follows: `a = 1`, `b = 3`, `c = 5`.

Alternatively, you can get the fit options and set start points and lower bounds, then refit using the new options.

```
options = fitoptions(ft)

options =

        Normalize: 'off'
          Exclude: []
          Weights: []
           Method: 'NonlinearLeastSquares'
           Robust: 'Off'
       StartPoint: [1x0 double]
```

```
         Lower: [1x0 double]
         Upper: [1x0 double]
     Algorithm: 'Trust-Region'
  DiffMinChange: 1.0000e-08
  DiffMaxChange: 0.1000
       Display: 'Notify'
    MaxFunEvals: 600
       MaxIter: 400
        TolFun: 1.0000e-06
          TolX: 1.0000e-06

options.StartPoint = [10 1 3];
options.Lower = [0 -Inf 0];
fit(month,pressure,ft,options)

ans =
    General model:
    ans(x) = b*x^2+c*x+a
    Coefficients (with 95% confidence bounds):
      a =        10.23  (9.448, 11.01)
      b =    4.335e-05  (-1.82e-05, 0.0001049)
      c =    5.523e-12  (fixed at bound)
```

Input Arguments

x — Data to fit
matrix

Data to fit, specified as a matrix with either one (curve fitting) or two (surface fitting) columns. You can specify variables in a MATLAB table using `tablename.varname`. Cannot contain `Inf` or `NaN`. Only the real parts of complex data are used in the fit.

Example: x

Example: [x,y]

Data Types: `double`

y — Data to fit
vector

Data to fit, specified as a column vector with the same number of rows as x. You can specify a variable in a MATLAB table using `tablename.varname`. Cannot contain `Inf` or `NaN`. Only the real parts of complex data are used in the fit.

Use `prepareCurveData` or `prepareSurfaceData` if your data is not in column vector form.

Data Types: `double`

z — Data to fit
vector

Data to fit, specified as a column vector with the same number of rows as x. You can specify a variable in a MATLAB table using `tablename.varname`. Cannot contain `Inf` or `NaN`. Only the real parts of complex data are used in the fit.

Use `prepareSurfaceData` if your data is not in column vector form. For example, if you have 3 matrices, or if your data is in grid vector form, where `length(X) = n, length(Y) = m` and `size(Z) = [m,n]`.

Data Types: `double`

fitType — Model type to fit
character vector | cell array of character vectors | anonymous function | `fittype`

Model type to fit, specified as a library model name character vector, a MATLAB expression, a cell array of linear models terms, an anonymous function, or a `fittype` constructed with the `fittype` function. You can use any of the valid first inputs to `fittype` as an input to `fit`.

For a list of library model names, see "Model Names and Equations" on page 4-11. This table shows some common examples.

Library Model Name	Description
`'poly1'`	Linear polynomial curve
`'poly11'`	Linear polynomial surface
`'poly2'`	Quadratic polynomial curve
`'linearinterp'`	Piecewise linear interpolation
`'cubicinterp'`	Piecewise cubic interpolation
`'smoothingspline'`	Smoothing spline (curve)
`'lowess'`	Local linear regression (surface)

To fit custom models, use a MATLAB expression, a cell array of linear model terms, an anonymous function, or create a `fittype` with the `fittype` function and use this as the `fitType` argument. For an example, see "Fit a Custom Model Using an Anonymous Function" on page 13-95. For examples of linear model terms, see the `fitType` function.

Example: 'poly2'

fitOptions — Algorithm options
`fitoptions`

Algorithm options constructed using the `fitoptions` function. This is an alternative to specifying name-value pair arguments for fit options.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Lower',[0,0],'Upper',[Inf,max(x)],'StartPoint',[1 1]` specifies fitting method, bounds, and start points.

Options for All Fitting Methods

Normalize — Option to center and scale data
`'off'` (default) | `'on'`

Option to center and scale the data, specified as the comma-separated pair consisting of
`'Normalize'` and `'on'` or `'off'`.

Data Types: `char`

Exclude — Points to exclude from fit
expression | index vector | logical vector | empty

Points to exclude from the fit, specified as the comma-separated pair consisting of `'Exclude'` and
one of:

- An expression describing a logical vector, e.g., `x > 10`.
- A vector of integers indexing the points you want to exclude, e.g., `[1 10 25]`.
- A logical vector for all data points where `true` represents an outlier, created by `excludedata`.

For an example, see "Exclude Points from Fit" on page 13-85.

Data Types: `logical` | `double`

Weights — Weights for fit
[ ] (default) | vector

Weights for the fit, specified as the comma-separated pair consisting of `'Weights'` and a vector the
same size as the response data `y` (curves) or `z` (surfaces).

Data Types: `double`

problem — Values to assign to problem-dependent constants
cell array | double

Values to assign to the problem-dependent constants, specified as the comma-separated pair
consisting of `'problem'` and a cell array with one element per problem dependent constant. For
details, see `fittype`.

Data Types: `cell` | `double`

Smoothing Options

SmoothingParam — Smoothing parameter
scalar value in the range (0,1)

Smoothing parameter, specified as the comma-separated pair consisting of `'SmoothingParam'` and
a scalar value between 0 and 1. The default value depends on the data set. Only available if the fit
type is `smoothingspline`.

Data Types: `double`

Span — Proportion of data points to use in local regressions
0.25 (default) | scalar value in the range (0,1)

Proportion of data points to use in local regressions, specified as the comma-separated pair
consisting of `'Span'` and a scalar value between 0 and 1. Only available if the fit type is `lowess` or
`loess`.

Data Types: `double`

Linear and Nonlinear Least-Squares Options

Robust — Robust linear least-squares fitting method
'off' (default) | LAR | Bisquare

Robust linear least-squares fitting method, specified as the comma-separated pair consisting of 'Robust' and one of these values:

- 'LAR' specifies the least absolute residual method.
- 'Bisquare' specifies the bisquare weights method.

Available when the fit type Method is LinearLeastSquares or NonlinearLeastSquares.

Data Types: char

Lower — Lower bounds on coefficients to be fitted
[ ] (default) | vector

Lower bounds on the coefficients to be fitted, specified as the comma-separated pair consisting of 'Lower' and a vector. The default value is an empty vector, indicating that the fit is unconstrained by lower bounds. If bounds are specified, the vector length must equal the number of coefficients. Find the order of the entries for coefficients in the vector value by using the coeffnames function. For an example, see "Find Coefficient Order to Set Start Points and Bounds" on page 13-96. Individual unconstrained lower bounds can be specified by -Inf.

Available when the Method is LinearLeastSquares or NonlinearLeastSquares.

Data Types: double

Upper — Upper bounds on coefficients to be fitted
[ ] (default) | vector

Upper bounds on the coefficients to be fitted, specified as the comma-separated pair consisting of 'Upper' and a vector. The default value is an empty vector, indicating that the fit is unconstrained by upper bounds. If bounds are specified, the vector length must equal the number of coefficients. Find the order of the entries for coefficients in the vector value by using the coeffnames function. For an example, see "Find Coefficient Order to Set Start Points and Bounds" on page 13-96. Individual unconstrained upper bounds can be specified by +Inf.

Available when the Method is LinearLeastSquares or NonlinearLeastSquares.

Data Types: logical

Nonlinear Least-Squares Options

StartPoint — Initial values for the coefficients
[ ] (default) | vector

Initial values for the coefficients, specified as the comma-separated pair consisting of 'StartPoint' and a vector. Find the order of the entries for coefficients in the vector value by using the coeffnames function. For an example, see "Find Coefficient Order to Set Start Points and Bounds" on page 13-96.

If no start points (the default value of an empty vector) are passed to the fit function, starting points for some library models are determined heuristically. For rational and Weibull models, and all custom nonlinear models, the toolbox selects default initial values for coefficients uniformly at random from

the interval (0,1). As a result, multiple fits using the same data and model might lead to different fitted coefficients. To avoid this, specify initial values for coefficients with a `fitoptions` object or a vector value for the `StartPoint` value.

Available when the `Method` is `NonlinearLeastSquares`.

Data Types: `double`

Algorithm — Algorithm to use for fitting procedure
'Trust-Region' (default) | 'Levenberg-Marquardt'

Algorithm to use for the fitting procedure, specified as the comma-separated pair consisting of `'Algorithm'` and either `'Levenberg-Marquardt'` or `'Trust-Region'`.

Available when the `Method` is `NonlinearLeastSquares`.

Data Types: `char`

DiffMaxChange — Maximum change in coefficients for finite difference gradients
0.1 (default)

Maximum change in coefficients for finite difference gradients, specified as the comma-separated pair consisting of `'DiffMaxChange'` and a scalar.

Available when the `Method` is `NonlinearLeastSquares`.

Data Types: `double`

DiffMinChange — Minimum change in coefficients for finite difference gradients
10^{-8} (default)

Minimum change in coefficients for finite difference gradients, specified as the comma-separated pair consisting of `'DiffMinChange'` and a scalar.

Available when the `Method` is `NonlinearLeastSquares`.

Data Types: `double`

Display — Display option in Command Window
'notify' (default) | 'final' | 'iter' | 'off'

Display option in the command window, specified as the comma-separated pair consisting of `'Display'` and one of these options:

- `'notify'` displays output only if the fit does not converge.
- `'final'` displays only the final output.
- `'iter'` displays output at each iteration.
- `'off'` displays no output.

Available when the `Method` is `NonlinearLeastSquares`.

Data Types: `char`

MaxFunEvals — Maximum number of evaluations of model allowed
600 (default)

Maximum number of evaluations of the model allowed, specified as the comma-separated pair consisting of `'MaxFunEvals'` and a scalar.

Available when the `Method` is `NonlinearLeastSquares`.

Data Types: `double`

`MaxIter` — Maximum number of iterations allowed for fit
400 (default)

Maximum number of iterations allowed for the fit, specified as the comma-separated pair consisting of `'MaxIter'` and a scalar.

Available when the `Method` is `NonlinearLeastSquares`.

Data Types: `double`

`TolFun` — Termination tolerance on model value
10^{-6} (default)

Termination tolerance on the model value, specified as the comma-separated pair consisting of `'TolFun'` and a scalar.

Available when the `Method` is `NonlinearLeastSquares`.

Data Types: `double`

`TolX` — Termination tolerance on coefficient values
10^{-6} (default)

Termination tolerance on the coefficient values, specified as the comma-separated pair consisting of `'TolX'` and a scalar.

Available when the `Method` is `NonlinearLeastSquares`.

Data Types: `double`

Output Arguments

`fitobject` — Fit result
cfit | sfit

Fit result, returned as a `cfit` (for curves) or `sfit` (for surfaces) object. See "Fit Postprocessing" for functions for plotting, evaluating, calculating confidence intervals, integrating, differentiating, or modifying your fit object.

`gof` — Goodness-of-fit statistics
gof structure

Goodness-of-fit statistics, returned as the `gof` structure including the fields in this table.

Field	Value
sse	Sum of squares due to error
rsquare	R-squared (coefficient of determination)

Field	Value
dfe	Degrees of freedom in the error
adjrsquare	Degree-of-freedom adjusted coefficient of determination
rmse	Root mean squared error (standard error)

output — Fitting algorithm information
output structure

Fitting algorithm information, returned as the output structure containing information associated with the fitting algorithm.

Fields depend on the algorithm. For example, the output structure for nonlinear least-squares algorithms includes the fields shown in this table.

Field	Value
numobs	Number of observations (response values)
numparam	Number of unknown parameters (coefficients) to fit
residuals	Vector of residuals
Jacobian	Jacobian matrix
exitflag	Describes the exit condition of the algorithm. Positive flags indicate convergence, within tolerances. Zero flags indicate that the maximum number of function evaluations or iterations was exceeded. Negative flags indicate that the algorithm did not converge to a solution.
iterations	Number of iterations
funcCount	Number of function evaluations
firstorderopt	Measure of first-order optimality (absolute maximum of gradient components)
algorithm	Fitting algorithm employed

See Also

Apps
Curve Fitting

Functions
confint | feval | fitoptions | fittype | plot | prepareCurveData | prepareSurfaceData

Topics
"Fit Postprocessing"
"List of Library Models for Curve and Surface Fitting" on page 4-10
"Custom Models" on page 5-2
"Parametric Fitting" on page 4-2

Introduced before R2006a

fitoptions

Create or modify fit options object

Syntax

```
fitOptions = fitoptions

fitOptions = fitoptions(libraryModelName)
fitOptions = fitoptions(libraryModelName,Name,Value)

fitOptions = fitoptions(fitType)

fitOptions = fitoptions(Name,Value)

newOptions = fitoptions(fitOptions,Name,Value)
newOptions = fitoptions(options1,options2)
```

Description

`fitOptions = fitoptions` creates the default fit options object `fitOptions`.

`fitOptions = fitoptions(libraryModelName)` creates the default fit options object for the library model.

`fitOptions = fitoptions(libraryModelName,Name,Value)` creates fit options for the specified library model with additional options specified by one or more `Name,Value` pair arguments.

`fitOptions = fitoptions(fitType)` gets the fit options object for the specified `fitType`. Use this syntax to work with fit options for custom models.

`fitOptions = fitoptions(Name,Value)` creates fit options with additional options specified by one or more `Name,Value` pair arguments.

`newOptions = fitoptions(fitOptions,Name,Value)` modifies the existing fit options object `fitOptions` and returns updated fit options in `newOptions` with new options specified by one or more `Name,Value` pair arguments.

`newOptions = fitoptions(options1,options2)` combines the existing fit options objects `options1` and `options2` in `newOptions`.

- If `Method` agrees, the nonempty values for the properties in `options2` override the corresponding values in `options1` in `newOptions`.
- If `Method` differs, `newOptions` contains the `options1` value for `Method` and values from `options2` for `Normalize`, `Exclude`, and `Weights`.

Examples

Modify Default Fit Options to Normalize Data

Create the default fit options object and set the option to center and scale the data before fitting.

```
options = fitoptions;
options.Normal = 'on'

options =

    Normalize: 'on'
      Exclude: [1x0 double]
      Weights: [1x0 double]
       Method: 'None'
```

Create Default Fit Options for Gaussian Fit

```
options = fitoptions('gauss2')

options =

         Normalize: 'off'
           Exclude: []
           Weights: []
            Method: 'NonlinearLeastSquares'
            Robust: 'Off'
        StartPoint: [1x0 double]
             Lower: [-Inf -Inf 0 -Inf -Inf 0]
             Upper: [1x0 double]
         Algorithm: 'Trust-Region'
      DiffMinChange: 1.0000e-08
      DiffMaxChange: 0.1000
           Display: 'Notify'
        MaxFunEvals: 600
            MaxIter: 400
             TolFun: 1.0000e-06
               TolX: 1.0000e-06
```

Set Polynomial Fit Options

Create fit options for a cubic polynomial and set center and scale and robust fitting options.

```
options = fitoptions('poly3', 'Normalize', 'on', 'Robust', 'Bisquare')

options =

    Normalize: 'on'
      Exclude: []
      Weights: []
       Method: 'LinearLeastSquares'
       Robust: 'Bisquare'
        Lower: [1x0 double]
        Upper: [1x0 double]
```

Create Fit Options for Linear Least Squares

```
options = fitoptions('Method', 'LinearLeastSquares')

options =

    Normalize: 'off'
      Exclude: []
      Weights: []
       Method: 'LinearLeastSquares'
       Robust: 'Off'
        Lower: [1x0 double]
        Upper: [1x0 double]
```

Use Identical Fit Options in Multiple Fits

Modifying the default fit options object is useful when you want to set the `Normalize`, `Exclude`, or `Weights` properties, and then fit your data using the same options with different fitting methods. For example, the following uses the same fit options to fit different library model types.

```
load census
options = fitoptions;
options.Normalize = 'on';
f1 = fit(cdate,pop,'poly3',options);
f2 = fit(cdate,pop,'exp1',options);
f3 = fit(cdate,pop,'cubicspline',options)

f3 =
    Cubic interpolating spline:
      f3(x) = piecewise polynomial computed from p
      where x is normalized by mean 1890 and std 62.05
    Coefficients:
      p = coefficient structure
```

Find and Change the Smoothing Fit Option

Find the smoothing parameter. Data-dependent fit options such as the `smooth` parameter are returned in the third output argument of the `fit` function.

```
load census
[f,gof,out] = fit(cdate,pop,'SmoothingSpline');
smoothparam = out.p

smoothparam = 0.0089
```

Modify the default smoothing parameter for a new fit.

```
options = fitoptions('Method','SmoothingSpline',...
                     'SmoothingParam',0.0098);
[f,gof,out] = fit(cdate,pop,'SmoothingSpline',options);
```

Apply Coefficient Bounds to Improve Gaussian Fit

Create a Gaussian fit, inspect the confidence intervals, and specify lower bound fit options to help the algorithm.

Create a noisy sum of two Gaussian peaks, one with a small width, and one with a large width.

```
a1 = 1; b1 = -1; c1 = 0.05;
a2 = 1; b2 = 1; c2 = 50;
x = (-10:0.02:10)';
gdata = a1*exp(-((x-b1)/c1).^2) + ...
        a2*exp(-((x-b2)/c2).^2) + ...
        0.1*(rand(size(x))-.5);
plot(x,gdata)
```
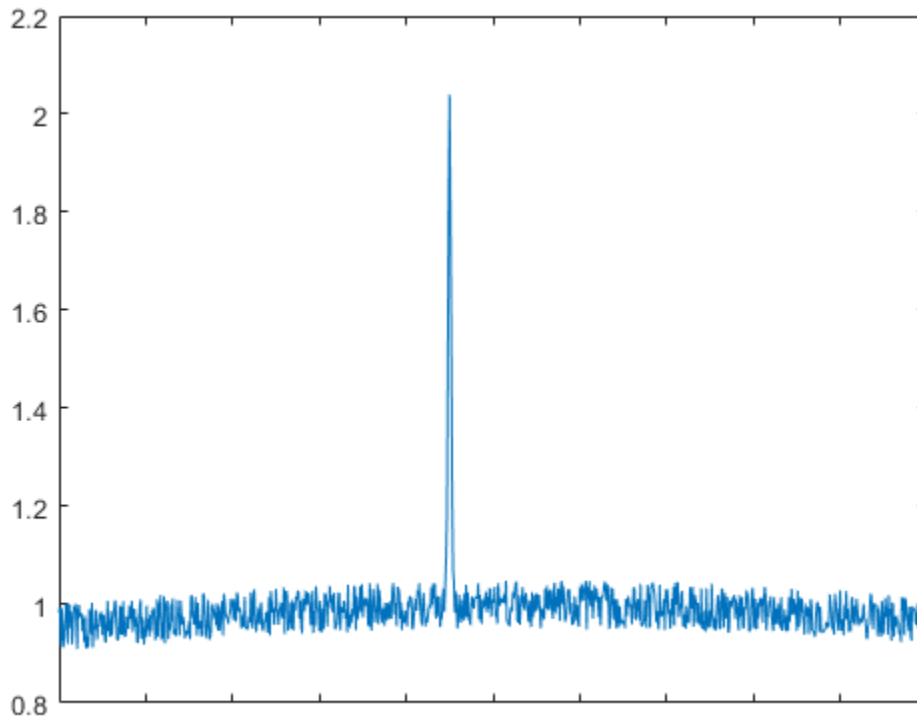

Fit the data using the two-term Gaussian library model.

```
gfit = fit(x,gdata,'gauss2')
```

```
gfit =
     General model Gauss2:
     gfit(x) =  a1*exp(-((x-b1)/c1)^2) + a2*exp(-((x-b2)/c2)^2)
     Coefficients (with 95% confidence bounds):
       a1 =      -0.145  (-1.486, 1.195)
       b1 =       9.725  (-14.71, 34.15)
       c1 =       7.117  (-15.84, 30.07)
       a2 =       14.06  (-1.958e+04, 1.961e+04)
```

```
        b2 =         607.1  (-3.194e+05, 3.207e+05)
        c2 =           376  (-9.739e+04, 9.814e+04)
```

```
plot(gfit,x,gdata)
```
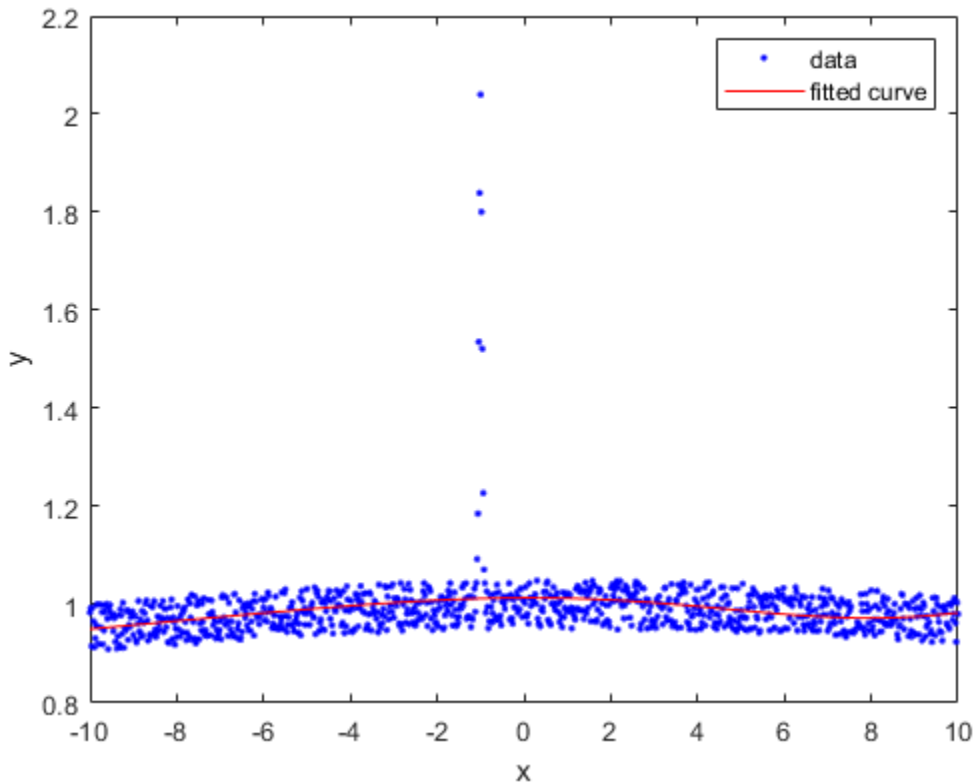

The algorithm is having difficulty, as indicated by the wide confidence intervals for several coefficients.

To help the algorithm, specify lower bounds for the nonnegative amplitudes a1 and a2 and widths c1, c2.

```
options = fitoptions('gauss2', 'Lower', [0 -Inf 0 0 -Inf 0]);
```

Alternatively, you can set properties of the fit options using the form options.Property = NewPropertyValue.

```
options = fitoptions('gauss2');
options.Lower = [0 -Inf 0 0 -Inf 0];
```

Recompute the fit using the bound constraints on the coefficients.

```
gfit = fit(x,gdata,'gauss2',options)
```

```
gfit =
     General model Gauss2:
     gfit(x) =   a1*exp(-((x-b1)/c1)^2) + a2*exp(-((x-b2)/c2)^2)
     Coefficients (with 95% confidence bounds):
       a1 =        1.005  (0.966, 1.044)
```

```
        b1 =             -1  (-1.002, -0.9988)
        c1 =         0.0491  (0.0469, 0.0513)
        a2 =         0.9985  (0.9958, 1.001)
        b2 =         0.8059  (0.3879, 1.224)
        c2 =           50.6  (46.68, 54.52)
```

```
plot(gfit,x,gdata)
```
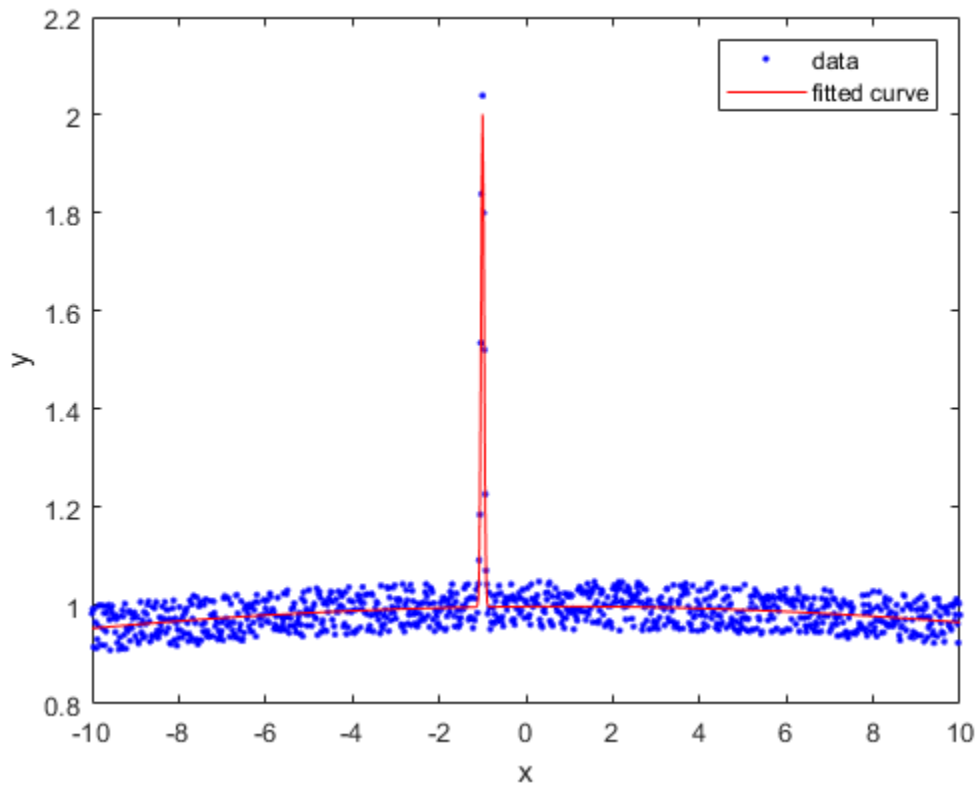

This is a much better fit. You can further improve the fit by assigning reasonable values to other properties in the fit options object.

Copy and Combine Fit Options

Create fit options and set lower bounds.

```
options = fitoptions('gauss2', 'Lower', [0 -Inf 0 0 -Inf 0])
```

```
options =

        Normalize: 'off'
          Exclude: []
          Weights: []
           Method: 'NonlinearLeastSquares'
           Robust: 'Off'
        StartPoint: [1x0 double]
```

```
           Lower: [0 -Inf 0 0 -Inf 0]
           Upper: [1x0 double]
       Algorithm: 'Trust-Region'
    DiffMinChange: 1.0000e-08
    DiffMaxChange: 0.1000
         Display: 'Notify'
      MaxFunEvals: 600
         MaxIter: 400
          TolFun: 1.0000e-06
            TolX: 1.0000e-06
```

Make a new copy of the fit options and modify the robust parameter.

```
newoptions = fitoptions(options, 'Robust','Bisquare')
```

```
newoptions =

         Normalize: 'off'
          Exclude: []
          Weights: []
           Method: 'NonlinearLeastSquares'
           Robust: 'Bisquare'
        StartPoint: [1x0 double]
            Lower: [0 -Inf 0 0 -Inf 0]
            Upper: [1x0 double]
        Algorithm: 'Trust-Region'
     DiffMinChange: 1.0000e-08
     DiffMaxChange: 0.1000
          Display: 'Notify'
       MaxFunEvals: 600
          MaxIter: 400
           TolFun: 1.0000e-06
             TolX: 1.0000e-06
```

Combine fit options.

```
options2 = fitoptions(options, newoptions)
```

```
options2 =

         Normalize: 'off'
          Exclude: []
          Weights: []
           Method: 'NonlinearLeastSquares'
           Robust: 'Bisquare'
        StartPoint: [1x0 double]
            Lower: [0 -Inf 0 0 -Inf 0]
            Upper: [1x0 double]
        Algorithm: 'Trust-Region'
     DiffMinChange: 1.0000e-08
     DiffMaxChange: 0.1000
          Display: 'Notify'
       MaxFunEvals: 600
          MaxIter: 400
           TolFun: 1.0000e-06
             TolX: 1.0000e-06
```

Change Custom Model Fit Options

Create a linear model fit type.

```
lft = fittype({'x','sin(x)','1'})

lft =
     Linear model:
     lft(a,b,c,x) = a*x + b*sin(x) + c
```

Get the fit options for the fit type `lft`.

```
fo = fitoptions(lft)

fo =

    Normalize: 'off'
      Exclude: []
      Weights: []
       Method: 'LinearLeastSquares'
       Robust: 'Off'
        Lower: [1x0 double]
        Upper: [1x0 double]
```

Set the normalize fit option.

```
fo.Normalize = 'on'

fo =

    Normalize: 'on'
      Exclude: []
      Weights: []
       Method: 'LinearLeastSquares'
       Robust: 'Off'
        Lower: [1x0 double]
        Upper: [1x0 double]
```

Input Arguments

`libraryModelName` — Library model to fit
character vector

Library model to fit, specified as a character vector. This table shows some common examples.

Library Model Name	Description
`'poly1'`	Linear polynomial curve
`'poly11'`	Linear polynomial surface
`'poly2'`	Quadratic polynomial curve
`'linearinterp'`	Piecewise linear interpolation
`'cubicinterp'`	Piecewise cubic interpolation
`'smoothingspline'`	Smoothing spline (curve)

Library Model Name	Description
`'lowess'`	Local linear regression (surface)

For a list of library model names, see "Model Names and Equations" on page 4-11.

Example: `'poly2'`

Data Types: `char`

fitType — Model type to fit
`fittype`

Model type to fit, specified as a `fittype` constructed with the `fittype` function. Use this to work with fit options for custom models.

fitOptions — Algorithm options
`fitoptions`

Algorithm options, specified as a `fitoptions` object created using the `fitoptions` function.

options1 — Algorithm options to combine
`fitoptions`

Algorithm options to combine, constructed using the `fitoptions` function.

options2 — Algorithm options to combine
`fitoptions`

Algorithm options to combine, constructed using the `fitoptions` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Method','NonlinearLeastSquares','Lower',[0,0],'Upper',` `[Inf,max(x)],'Startpoint',[1 1]` specifies fitting method, bounds, and start points.

Options for All Fitting Methods

Normalize — Option to center and scale data
`'off'` (default) | `'on'`

Option to center and scale the data, specified as the comma-separated pair consisting of `'Normalize'` and `'on'` or `'off'`.

Data Types: `char`

Exclude — Points to exclude from fit
expression | index vector | logical vector | empty

Points to exclude from the fit, specified as the comma-separated pair consisting of `'Exclude'` and one of:

- An expression describing a logical vector, e.g., `x > 10`.

- A vector of integers indexing the points you want to exclude, e.g., [1 10 25].

- A logical vector for all data points where `true` represents an outlier, created by `excludedata`.

For examples, see `fit`.

Weights — Weights for fit
[ ] (default) | vector

Weights for the fit, specified as the comma-separated pair consisting of `'Weights'` and a vector the same size as number of data points.

Data Types: `double`

Method — Fitting method
`'None'` (default) | character vector

Fitting method, specified as the comma-separated pair consisting of `'Method'` and one of the fitting methods in this table.

Fitting Method	Description
`'NearestInterpolant'`	Nearest neighbor interpolation
`'LinearInterpolant'`	Linear interpolation
`'PchipInterpolant'`	Piecewise cubic Hermite interpolation (curves only)
`'CubicSplineInterpolant'`	Cubic spline interpolation
`'BiharmonicInterpolant'`	Biharmonic surface interpolation
`'SmoothingSpline'`	Smoothing spline
`'LowessFit'`	Lowess smoothing (surfaces only)
`'LinearLeastSquares'`	Linear least squares
`'NonlinearLeastSquares'`	Nonlinear least squares

Data Types: `char`

Smoothing Options

SmoothingParam — Smoothing parameter
scalar value in the range (0,1)

Smoothing parameter, specified as the comma-separated pair consisting of `'SmoothingParam'` and a scalar value between 0 and 1. The default value depends on the data set. Only available if the `Method` is `SmoothingSpline`.

Data Types: `double`

Span — Proportion of data points to use in local regressions
0.25 (default) | scalar value in the range (0,1)

Proportion of data points to use in local regressions, specified as the comma-separated pair consisting of `'Span'` and a scalar value between 0 and 1. Only available if the `Method` is `LowessFit`.

Data Types: `double`

Linear and Nonlinear Least-Squares Options

Robust — Robust linear least-squares fitting method
'off' (default) | 'LAR' | 'Bisquare'

Robust linear least-squares fitting method, specified as the comma-separated pair consisting of 'Robust' and one of these values:

- 'LAR' specifies the least absolute residual method.
- 'Bisquare' specifies the bisquare weights method.

Available when the Method is LinearLeastSquares or NonlinearLeastSquares.

Data Types: char

Lower — Lower bounds on coefficients to be fitted
[ ] (default) | vector

Lower bounds on the coefficients to be fitted, specified as the comma-separated pair consisting of 'Lower' and a vector. The default value is an empty vector, indicating that the fit is unconstrained by lower bounds. If bounds are specified, the vector length must equal the number of coefficients. Find the order of the entries for coefficients in the vector value by using the coeffnames function. For an example, see fit. Individual unconstrained lower bounds can be specified by -Inf.

Available when the Method is LinearLeastSquares or NonlinearLeastSquares.

Data Types: double

Upper — Upper bounds on coefficients to be fitted
[ ] (default) | vector

Upper bounds on the coefficients to be fitted, specified as the comma-separated pair consisting of 'Upper' and a vector. The default value is an empty vector, indicating that the fit is unconstrained by upper bounds. If bounds are specified, the vector length must equal the number of coefficients. Find the order of the entries for coefficients in the vector value by using the coeffnames function. For an example, see fit. Individual unconstrained upper bounds can be specified by +Inf.

Available when the Method is LinearLeastSquares or NonlinearLeastSquares.

Data Types: logical

Nonlinear Least-Squares Options

StartPoint — Initial values for coefficients
[ ] (default) | vector

Initial values for the coefficients, specified as the comma-separated pair consisting of 'StartPoint' and a vector. Find the order of the entries for coefficients in the vector value by using the coeffnames function. For an example, see fit.

If no start points (the default value of an empty vector) are passed to the fit function, starting points for some library models are determined heuristically. For rational and Weibull models, and all custom nonlinear models, the toolbox selects default initial values for coefficients uniformly at random from the interval (0,1). As a result, multiple fits using the same data and model might lead to different fitted coefficients. To avoid this, specify initial values for coefficients with a vector value for the StartPoint property.

Available when the Method is NonlinearLeastSquares.

Data Types: double

Algorithm — Algorithm to use for fitting procedure
'Levenberg-Marquardt' (default) | 'Trust-Region'

Algorithm to use for the fitting procedure, specified as the comma-separated pair consisting of 'Algorithm' and either 'Levenberg-Marquardt' or 'Trust-Region'.

Available when the Method is NonlinearLeastSquares.

Data Types: char

DiffMaxChange — Maximum change in coefficients for finite difference gradients
0.1 (default)

Maximum change in coefficients for finite difference gradients, specified as the comma-separated pair consisting of 'DiffMaxChange' and a scalar.

Available when the Method is NonlinearLeastSquares.

Data Types: double

DiffMinChange — Minimum change in coefficients for finite difference gradients
10^{-8} (default)

Minimum change in coefficients for finite difference gradients, specified as the comma-separated pair consisting of 'DiffMinChange' and a scalar.

Available when the Method is NonlinearLeastSquares.

Data Types: double

Display — Display option in the Command Window
'notify' (default) | 'final' | 'iter' | 'off'

Display option in the command window, specified as the comma-separated pair consisting of 'Display' and one of these options:

- 'notify' displays output only if the fit does not converge.
- 'final' displays only the final output.
- 'iter' displays output at each iteration.
- 'off' displays no output.

Available when the Method is NonlinearLeastSquares.

Data Types: char

MaxFunEvals — Maximum number of evaluations of model allowed
600 (default)

Maximum number of evaluations of the model allowed, specified as the comma-separated pair consisting of 'MaxFunEvals' and a scalar.

Available when the Method is NonlinearLeastSquares.

Data Types: `double`

`MaxIter` — Maximum number of iterations allowed for fit
400 (default)

Maximum number of iterations allowed for the fit, specified as the comma-separated pair consisting of `'MaxIter'` and a scalar.

Available when the `Method` is `NonlinearLeastSquares`.

Data Types: `double`

`TolFun` — Termination tolerance on model value
10^{-6} (default)

Termination tolerance on the model value, specified as the comma-separated pair consisting of `'TolFun'` and a scalar.

Available when the `Method` is `NonlinearLeastSquares`.

Data Types: `double`

`TolX` — Termination tolerance on coefficient values
10^{-6}. (default)

Termination tolerance on the coefficient values, specified as the comma-separated pair consisting of `'TolX'` and a scalar.

Available when the `Method` is `NonlinearLeastSquares`.

Data Types: `double`

Output Arguments

`fitOptions` — Algorithm options
`fitoptions`

Algorithm options, returned as a `fitoptions` object.

`newOptions` — New algorithm options
`fitoptions`

New algorithm options, returned as a `fitoptions` object.

See Also

Apps
Curve Fitting

Functions
`fit` | `fittype` | `get` | `set` | `setoptions`

Topics
"Specifying Fit Options and Optimized Starting Points" on page 4-5
"Fit Postprocessing"

"List of Library Models for Curve and Surface Fitting" on page 4-10

Introduced before R2006a

fittype

Fit type for curve and surface fitting

Syntax

```
aFittype = fittype(libraryModelName)

aFittype = fittype(expression)
aFittype = fittype(expression,Name,Value)

aFittype = fittype(linearModelTerms)
aFittype = fittype(linearModelTerms,Name,Value)

aFittype = fittype(anonymousFunction)
aFittype = fittype(anonymousFunction,Name,Value)
```

Description

`aFittype = fittype(libraryModelName)` creates the `fittype` object `aFittype` for the model specified by `libraryModelName`.

`aFittype = fittype(expression)` creates a fit type for the model specified by the MATLAB expression.

`aFittype = fittype(expression,Name,Value)` constructs the fit type with additional options specified by one or more `Name,Value` pair arguments.

`aFittype = fittype(linearModelTerms)` creates a fit type for a custom linear model with terms specified by the cell array of character vector expressions in `linearModelTerms`.

`aFittype = fittype(linearModelTerms,Name,Value)` constructs the fit type with additional options specified by one or more `Name,Value` pair arguments.

`aFittype = fittype(anonymousFunction)` creates a fit type for the model specified by `anonymousFunction`.

`aFittype = fittype(anonymousFunction,Name,Value)` constructs the fit type with additional options specified by one or more `Name,Value` pair arguments.

Examples

Create Fit Types for Library Models

Construct fit types by specifying library model names.

Construct a `fittype` object for the cubic polynomial library model.

```
f = fittype('poly3')
```

```
f =
     Linear model Poly3:
     f(p1,p2,p3,p4,x) = p1*x^3 + p2*x^2 + p3*x + p4
```

Construct a fit type for the library model `rat33` (a rational model of the third degree for both the numerator and denominator).

```
f = fittype('rat33')
```

```
f =
     General model Rat33:
     f(p1,p2,p3,p4,q1,q2,q3,x) = (p1*x^3 + p2*x^2 + p3*x + p4) /
                 (x^3 + q1*x^2 + q2*x + q3)
```

For a list of library model names, see `libraryModelName`.

Create Custom Linear Model

To use a linear fitting algorithm, specify a cell array of terms.

Identify the linear model terms you need to input to `fittype`: `a*x + b*sin(x) + c`. The model is linear in `a`, `b` and `c`. It has three terms `x`, `sin(x)` and `1` (because `c=c*1`). To specify this model you use this cell array of terms: `LinearModelTerms = {'x','sin(x)','1'}`.

Use the cell array of linear model terms as the input to `fittype`.

```
ft = fittype({'x','sin(x)','1'})
```

```
ft =
     Linear model:
     ft(a,b,c,x) = a*x + b*sin(x) + c
```

Create a linear model fit type for `a*cos(x) + b`.

```
ft2 = fittype({'cos(x)','1'})
```

```
ft2 =
     Linear model:
     ft2(a,b,x) = a*cos(x) + b
```

Create the fit type again and specify coefficient names.

```
ft3 = fittype({'cos(x)','1'},'coefficients',{'a1','a2'})
```

```
ft3 =
     Linear model:
     ft3(a1,a2,x) = a1*cos(x) + a2
```

Create Custom Nonlinear Models and Specify Problem Parameters and Independent Variables

Construct fit types for custom nonlinear models, designating problem-dependent parameters and independent variables.

Construct a fit type for a custom nonlinear model, designating n as a problem-dependent parameter and u as the independent variable.

```
g = fittype('a*u+b*exp(n*u)',...
             'problem','n',...
             'independent','u')

g =
    General model:
    g(a,b,n,u) = a*u+b*exp(n*u)
```

Construct a fit type for a custom nonlinear model, designating time as the independent variable.

```
g = fittype('a*time^2+b*time+c','independent','time','dependent','height')

g =
    General model:
    g(a,b,c,time) = a*time^2+b*time+c
```

Construct a fit type for a logarithmic fit to some data, use the fit type to create a fit, and plot the fit.

```
x = linspace(1,100);
y = 5 + 7*log(x);
myfittype = fittype('a + b*log(x)',...
    'dependent',{'y'},'independent',{'x'},...
    'coefficients',{'a','b'})

myfittype =
    General model:
    myfittype(a,b,x) = a + b*log(x)

myfit = fit(x',y',myfittype)

Warning: Start point not provided, choosing random start point.

myfit =
    General model:
    myfit(x) = a + b*log(x)
    Coefficients (with 95% confidence bounds):
      a =             5  (5, 5)
      b =             7  (7, 7)

plot(myfit,x,y)
```
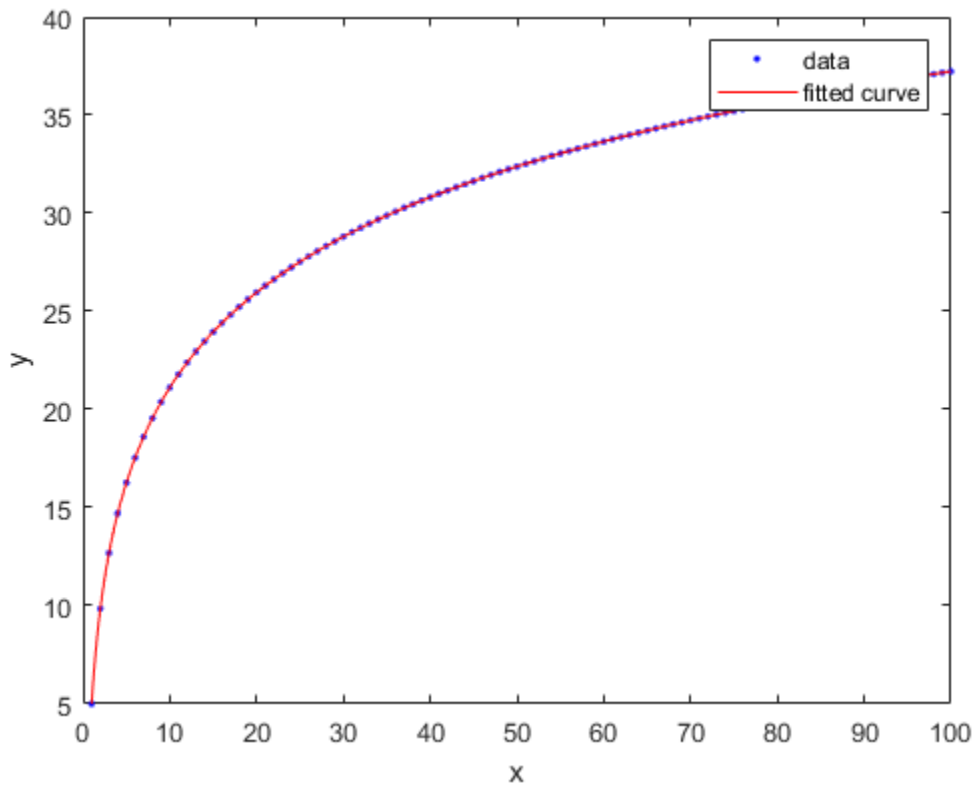
You can specify any MATLAB command and therefore any `.m` file.

Fit a Curve Defined by a File

Define a function in a file and use it to create a fit type and fit a curve.

Define a function in a MATLAB file.

```matlab
function y = piecewiseLine(x,a,b,c,d,k)
% PIECEWISELINE   A line made of two pieces
% that is not continuous.

y = zeros(size(x));

% This example includes a for-loop and if statement
% purely for example purposes.
for i = 1:length(x)
    if x(i) < k,
        y(i) = a + b.* x(i);
    else
        y(i) = c + d.* x(i);
    end
end
end
```

Save the file.

Define some data, create a fit type specifying the function `piecewiseLine`, create a fit using the fit type `ft`, and plot the results.

```
x = [0.81;0.91;0.13;0.91;0.63;0.098;0.28;0.55;...
    0.96;0.96;0.16;0.97;0.96];
y = [0.17;0.12;0.16;0.0035;0.37;0.082;0.34;0.56;...
    0.15;-0.046;0.17;-0.091;-0.071];
ft = fittype( 'piecewiseLine( x, a, b, c, d, k )' )
f = fit( x, y, ft, 'StartPoint', [1, 0, 1, 0, 0.5] )
plot( f, x, y )
```

Create Custom Linear Model

To use a linear fitting algorithm, specify a cell array of terms.

Identify the linear model terms you need to input to `fittype`: `a*x + b*sin(x) + c`. The model is linear in `a`, `b` and `c`. It has three terms `x`, `sin(x)` and `1` (because `c=c*1`). To specify this model you use this cell array of terms: `LinearModelTerms = {'x','sin(x)','1'}`.

Use the cell array of linear model terms as the input to `fittype`.

```
ft = fittype({'x','sin(x)','1'})

ft =
    Linear model:
    ft(a,b,c,x) = a*x + b*sin(x) + c
```

Create a linear model fit type for `a*cos(x) + b`.

```
ft2 = fittype({'cos(x)','1'})

ft2 =
    Linear model:
    ft2(a,b,x) = a*cos(x) + b
```

Create the fit type again and specify coefficient names.

```
ft3 = fittype({'cos(x)','1'},'coefficients',{'a1','a2'})

ft3 =
    Linear model:
    ft3(a1,a2,x) = a1*cos(x) + a2
```

Create Fit Types Using Anonymous Functions

Create a fit type using an anonymous function.

```
g = fittype( @(a, b, c, x) a*x.^2+b*x+c )
```

Create a fit type using an anonymous function and specify independent and dependent parameters.

```
g = fittype( @(a, b, c, d, x, y) a*x.^2+b*x+c*exp(...
 -(y-d).^2 ), 'independent', {'x', 'y'},...
     'dependent', 'z' );
```

Create a fit type for a surface using an anonymous function and specify independent and dependent parameters, and problem parameters that you will specify later when you call `fit`.

```
g = fittype( @(a,b,c,d,x,y) a*x.^2+b*x+c*exp( -(y-d).^2 ), ...
        'problem', {'c','d'}, 'independent', {'x', 'y'}, ...
        'dependent', 'z' );
```

Use an Anonymous Function to Pass in Workspace Data to the Fit

Use an anonymous function to pass workspace data into the `fittype` and `fit` functions.

Create and plot an S-shaped curve. In later steps, you stretch and move this curve to fit to some data.

```
% Breakpoints.
xs = (0:0.1:1).';
% Height of curve at breakpoints.
ys = [0; 0; 0.04; 0.1; 0.2; 0.5; 0.8; 0.9; 0.96; 1; 1];
% Plot S-shaped curve.
xi = linspace( 0, 1, 241 );
plot( xi, interp1( xs, ys, xi, 'pchip' ), 'LineWidth', 2 )
hold on
plot( xs, ys, 'o', 'MarkerFaceColor', 'r' )
hold off
title S-curve
```

Create a fit type using an anonymous function, taking the values from the workspace for the curve breakpoints (`xs`) and the height of the curve at the breakpoints (`ys`). Coefficients are `b` (base) and `h` (height).

```
ft = fittype( @(b, h, x) interp1( xs, b+h*ys, x, 'pchip' ) )
```

Plot the `fittype` specifying example coefficients of base `b=1.1` and height `h=-0.8`.

```
plot( xi, ft( 1.1, -0.8, xi ), 'LineWidth', 2 )
title 'Fittype with b=1.1 and h=-0.8'
```

Load and fit some data, using the fit type `ft` created using workspace values.

```
% Load some data
xdata = [0.012;0.054;0.13;0.16;0.31;0.34;0.47;0.53;0.53;...
   0.57;0.78;0.79;0.93];
ydata = [0.78;0.87;1;1.1;0.96;0.88;0.56;0.5;0.5;0.5;0.63;...
   0.62;0.39];
% Fit the curve to the data
f = fit( xdata, ydata, ft, 'Start', [0, 1] )
% Plot fit
plot( f, xdata, ydata )
title 'Fitted S-curve'
```

Use Anonymous Functions to Work with Problem Parameters and Workspace Variables

This example shows the differences between using anonymous functions with problem parameters and workspace variable values.

Load data, create a fit type for a curve using an anonymous function with problem parameters, and call `fit` specifying the problem parameters.

```matlab
% Load some data.
xdata = [0.098;0.13;0.16;0.28;0.55;0.63;0.81;0.91;0.91;...
    0.96;0.96;0.96;0.97];
ydata = [0.52;0.53;0.53;0.48;0.33;0.36;0.39;0.28;0.28;...
    0.21;0.21;0.21;0.2];

% Create a fittype that has a problem parameter.
g = fittype( @(a,b,c,x) a*x.^2+b*x+c, 'problem', 'c' )

% Examine coefficients. Observe c is not a coefficient.
coeffnames( g )

% Examine arguments. Observe that c is an argument.
argnames( g )

% Call fit and specify the value of c.
f1 = fit( xdata, ydata, g, 'problem', 0, 'StartPoint', [1, 2] )

% Note: Specify start points in the calls to fit to
% avoid warning messages about random start points
% and to ensure repeatability of results.

% Call fit again and specify a different value of c,
% to get a new fit.
f2 = fit( xdata, ydata, g, 'problem', 1, 'start', [1, 2] )

% Plot results. Observe the specified c constants
% do not make a good fit.
plot( f1, xdata, ydata )
hold on
plot( f2, 'b' )
hold off
```

Modify the previous example to create the same fits using workspace values for variables, instead of using problem parameters. Using the same data, create a fit type for a curve using an anonymous function with a workspace value for variable `c`:

```matlab
% Remove c from the argument list.
try
    g = fittype( @(a,b,x) a*x.^2+b*x+c )
catch e
    disp( e.message )
end
% Observe error because now c is undefined.
% Define c and create fittype:
c = 0;
g1 = fittype( @(a,b,x) a*x.^2+b*x+c )

% Call fit (now no need to specify problem parameter).
f1 = fit( xdata, ydata, g1, 'StartPoint', [1, 2] )
```

```
% Note that this f1 is the same as the f1 above.
% To change the value of c, recreate the fittype.
c = 1;
g2 = fittype( @(a,b,x) a*x.^2+b*x+c ) % uses c = 1
f2 = fit( xdata, ydata, g2, 'StartPoint', [1, 2] )
% Note that this f2 is the same as the f2 above.
% Plot results
plot( f1, xdata, ydata )
hold on
plot( f2, 'b' )
hold off
```

Input Arguments

`libraryModelName` — Library model to fit
character vector

Library model to fit, specified as a character vector. This table shows some common examples.

Library Model Name	Description
`'poly1'`	Linear polynomial curve
`'poly11'`	Linear polynomial surface
`'poly2'`	Quadratic polynomial curve
`'linearinterp'`	Piecewise linear interpolation
`'cubicinterp'`	Piecewise cubic interpolation
`'smoothingspline'`	Smoothing spline (curve)
`'lowess'`	Local linear regression (surface)

For a list of library model names, see "Model Names and Equations" on page 4-11.

Example: `'poly2'`

Data Types: `char`

`expression` — Model to fit
character vector

Model to fit, specified as a character vector. You can specify any MATLAB command and therefore any `.m` file. See "Fit a Curve Defined by a File" on page 13-121.

Data Types: `char`

`linearModelTerms` — Model to fit
cell array of character vectors

Model to fit, specified as a cell array of character vectors. Specify the model terms by the expressions in the character vectors. Do not include coefficients in the expressions for the terms. See "Linear Model Terms" on page 13-128.

Data Types: `cell`

`anonymousFunction` — Model to fit
anonymous function

Model to fit, specified as an anonymous function. For details, see "Input Order for Anonymous Functions" on page 13-127.

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'coefficients',{'a1','a2'}

coefficients — Coefficient names
character vector | cell array of character vectors

Coefficient names, specified as the comma-separated pair consisting of 'coefficients' and a character vector, or a cell array of character vectors for multiple names. You can use multicharacter symbol names. You cannot use these names: i, j, pi, inf, nan, eps.

Data Types: char | cell

dependent — Dependent (response) variable name
y (default) | character vector

Dependent (response) variable name, specified as the comma-separated pair consisting of 'dependent' and a character vector. If you do not specify the dependent variable, the function assumes y is the dependent variable.

Data Types: char

independent — Independent (response) variable names
x (default) | character vector | cell array of character vectors

Independent (response) variable names, specified as the comma-separated pair consisting of 'independent' and a character vector or cell array of character vectors. If you do not specify the independent variable, the function assumes x is the independent variable.

Data Types: char

options — Fit options
fitoptions

Fit options, specified as the comma-separated pair consisting of 'options' and the name of a fitoptions object.

problem — Problem-dependent (fixed) parameter names
cell array

Problem-dependent (fixed) parameter names, specified as the comma-separated pair consisting of 'problem' and a character vector, or cell array of character vectors with one element per problem dependent constant.

Data Types: char | cell

Output Arguments

aFittype — Model to fit
`fittype` object

Model to fit, returned as a `fittype`. A `fittype` encapsulates information describing a model. To create a fit, you need data, a `fittype`, and (optionally) `fitoptions` and an exclusion rule. You can use a `fittype` as an input to the `fit` function.

More About

Dependent and Independent Variables

How do I decide which variables are dependent and independent?

To determine dependent and independent variables and coefficients, consider this equation:

$y = f(x) = a + (b*x) + (c*x^2)$.

- y is the dependent variable.
- x is the independent variable.
- a, b, and c are the coefficients.

The `'independent'` variable is what you control. The `'dependent'` variable is what you measure, i.e., it depends on the independent variable. The `'coefficients'` are the parameters that the fitting algorithm estimates.

For example, if you have census data, then the year is the independent variable because it does not depend on anything. Population is the dependent variable, because its value depends on the year in which the census is taken. If a parameter like growth rate is part of the model, so the fitting algorithm estimates it, then the parameter is one of the `'coefficients'`.

The `fittype` function determines input arguments by searching the fit type expression input for variable names. `fittype` assumes x is the independent variable, y is the dependent variable, and all other variables are coefficients of the model. x is used if no variable exists.

Input Order for Anonymous Functions

If the fit type expression input is an anonymous function, then the order of inputs must be correct. The input order enables the `fittype` function to determine which inputs are coefficients to estimate, problem-dependent parameters, and independent variables.

The order of the input arguments to the anonymous function must be:

`fcn = @(coefficients,problemparameters,x,y) expression`

You need at least one coefficient. The problem parameters and y are optional. The last arguments, x and y, represent the independent variables: just x for curves, but x and y for surfaces. If you don't want to use x and/or y to name the independent variables, then specify different names using the `'independent'` argument name-value pair. However, whatever name or names you choose, these arguments must be the last arguments to the anonymous function.

Anonymous functions make it easier to pass other data into the `fittype` and `fit` functions.

1 Create a fit type using an anonymous function and a variable value (`c`) from the workspace.

```
c = 1;
 g = fittype( @(a, b, x) a*x.^2+b*x+c )
```

2 The `fittype` function can use the variable values in your workspace when you create the fit type. To pass in new data from the workspace, recreate the fit type, e.g.,

```
c = 5 % Change value of c.
g = fittype( @(a, b, x) a*x.^2+b*x+c )
```

3 Here, the value of `c` is fixed when you create the fit type. To specify the value of `c` at the time you call `fit`, you can use problem parameters. For example, make a fit with `c = 2` and then a new fit with `c = 3`.

```
g = fittype( @(a,b,x,c) a*x.^2+b*x+c, 'problem', 'c' )
f1 = fit( xdata, ydata, g, 'problem', 2 )
f2 = fit( xdata, ydata, g, 'problem', 3 )
```

Linear Model Terms

How do I define linear model terms?

To use a linear fitting algorithm, specify `linearModelTerms` as a cell array of terms.

```
afittype = fittype({expr1,...,exprn})
```

Specify the model terms by the expressions in the character vectors `expr2,...,exprn`. Do not include coefficients in the expressions for the terms. If there is a constant term, use `'1'` as the corresponding expression in the cell array.

To specify a linear model of the following form:

```
 coeff1 * term1 + coeff2 * term2 + coeff3 * term3 + ...
```

where no coefficient appears within any of `term1`, `term2`, etc., use a cell array where each term, without coefficients, is specified in a cell of `expr`, as follows:

```
LinearModelTerms = {'term1', 'term2', 'term3', ... }
```

For example, the model

```
a*x + b*sin(x) + c
```

is linear in `a`, `b`, and `c`. It has three terms `x`, `sin(x)` and `1` (because c=c*1) and therefore `expr` is:

```
LinearModelTerms = {'x','sin(x)','1'}
```

In the Curve Fitting app, see the `Linear Fitting` model type.

Algorithms

If the fit type expression input is a character vector or anonymous function, then the toolbox uses a nonlinear fitting algorithm to fit the model to data.

If the fit type expression input is a cell array of terms, then the toolbox uses a linear fitting algorithm to fit the model to data.

See Also

Apps
Curve Fitting

Functions
`fit` | `fitoptions`

Topics
"Custom Linear Fitting" on page 5-6
"Parametric Fitting" on page 4-2

Introduced before R2006a

fn2fm

Convert to specified form

Syntax

```
g = fn2fm(f,form)
sp = fn2fm(f,'B-',sconds)
fn2fm(f)
```

Description

`g = fn2fm(f,form)` describes the same function as is described by `f`, but in the form specified by the character vector `form`. Choices for `form` are `'B-'`, `'pp'`, `'BB'`, `'rB'`, `'rp'`, for the B-form, the ppform, the BBform, and the two rational spline forms, respectively.

The B-form describes a function as a weighted sum of the B-splines of a given order k for a given knot sequence, and the BBform (or, Bernstein-Bézier form) is the special case when each knot in that sequence appears with maximal multiplicity, k. The ppform describes a function in terms of its local polynomial coefficients. The B-form is good for constructing and/or shaping a function, while the ppform is cheaper to evaluate.

Conversion from a polynomial form to the corresponding rational form is possible only if the function in the polynomial form is vector-valued, in which case its last component is designated as the denominator. Converting from a rational form to the corresponding polynomial form simply reverses this process by reinterpreting the denominator of the function in the rational form as an additional component of the piecewise polynomial function.

Conversion to or from the stform is not possible at present.

If `form` is `'B-'` (and `f` is in ppform), then the actual smoothness of the function in `f` across each of its interior breaks has to be guessed. This is done by looking, for each interior break, for the first derivative whose jump across that break is not *small* compared to the size of that derivative nearby. The default tolerance used in this is `1.e-12`.

`sp = fn2fm(f,'B-',sconds)` permits you to supply, as the input argument `sconds`, a tolerance (strictly between 0 and 1) to be used in the conversion from ppform to B-form.

Alternatively, you can input `sconds` as a vector with integer entries, with at least as many entries as the ppform in `f` has *interior* breaks. In that case, `sconds(i)` specifies the number of smoothness conditions to be used across the `i`th *interior* break. If the function in `f` is a tensor product, then `sconds`, if given, must be a cell array.

`fn2fm(f)` converts a possibly old version of a form into its present version.

Examples

`sp = fn2fm(spline(x,y),'B-')` gives the interpolating cubic spline provided by the MATLAB command `spline`, but in B-form rather than in ppform.

```
p0 = ppmak([0 1],[3 0 0]);
p1 = fn2fm(fn2fm(fnrfn(p0,[.4 .6]),'B-'),'pp');
```

gives `p1` identical to `p0` (up to round-off in the coefficients) since the spline has no discontinuity in any derivative across the additional breaks introduced by `fnrfn`, hence conversion to B-form ignores these additional breaks, and conversion to ppform does not retain any knot multiplicities (like the knot multiplicities introduced, by conversion to B-form, at the endpoints of the spline's basic interval).

Cautionary Note

When going from B-form to ppform, any jump discontinuity at the first and last knot, `t(1)` or `t(end)`, will be lost since the ppform considers *f* to be defined outside its basic interval by extension of the first, respectively, the last polynomial piece. For example, while `sp=spmak([0 1],1)` gives the characteristic function of the interval [0..1], `pp=fn2fm(spmak([0 1],1),'pp')` is the constant polynomial, $x \mapsto 1$.

Algorithms

For a multivariate (tensor-product) function, univariate algorithms are applied in each variable.

For the conversion from B-form (or BBform) to ppform, the utility command `sprpp` is used to convert the B-form of all polynomial pieces to their local power form, using repeated knot insertion at the left endpoint.

The conversion from B-form to BBform is accomplished by inserting each knot enough times to increase its multiplicity to the order of the spline.

The conversion from ppform to B-form makes use of the dual functionals discussed in . on page 10-2 Without further information, such a conversion has to ascertain the actual smoothness across each interior break of the function in `f`.

See Also
ppmak | rsmak | spmak | stmak

fnbrk

Name and part(s) of form

Syntax

```
[out1,...,outn] = fnbrk(f,part1,...,partm)
fnbrk(f,interval)
fnbrk(pp,j)
fnbrk(f)
```

Description

`[out1,...,outn] = fnbrk(f,part1,...,partm)` returns the part(s) of the form in `f` specified by `part1,...,partn` (assuming that n<=m). These are the parts used when the form was put together, in `spmak` or `ppmak` or `rpmak` or `rsmak` or `stmak`, but also other parts derived from these.

You only need to specify the beginning character(s) of the relevant character vector.

Regardless of what particular form `f` is in, `parti` can be one of the following.

`'form'`	The particular form used
`'variables'`	The dimension of the function's domain
`'dimension'`	The dimension of the function's target
`'coefficients'`	The coefficients in that particular form
`'interval'`	The basic interval of that form

Depending on the form in `f`, additional parts may be asked for.

If `f` is in B-form (or BBform or rBform), then additional choices for `parti` are

`'knots'`	The knot sequence
`'coefficients'`	The B-spline coefficients
`'number'`	The number of coefficients
`'order'`	The polynomial order of the spline

If `f` is in ppform (or rpform), then additional choices for `parti` are

`'breaks'`	The break sequence
`'coefficients'`	The local polynomial coefficients
`'pieces'`	The number of polynomial pieces
`'order'`	The polynomial order of the spline
`'guide'`	The local polynomial coefficients, but in the form needed for PPVALU in PGS

If the function in `f` is multivariate, then the corresponding multivariate parts are returned. This means, e.g., that knots, breaks, and the basic interval, are cell arrays, the coefficient array is, in general, higher than two-dimensional, and order, number and pieces are vectors.

If `f` is in stform, then additional choices for `parti` are

`'centers'`	The centers
`'coefficients'`	The coefficients
`'number'`	Number of coefficients or terms
`'type'`	The particular type

`fnbrk(f,interval)` with `interval` a 1-by-2 matrix `[a b]` with `a<b` does not return a particular part. Rather, it returns a description of the univariate function described by `f` and in the same form but with the basic interval changed, to the interval given. If, instead, `interval` is `[ ]`, `f` is returned unchanged. This is of particular help when the function in `f` is *m*-variate, in which case `interval` must be a cell array with *m* entries, with the *i*th entry specifying the desired interval in the *i*th dimension. If that *i*th entry is `[ ]`, the basic interval in the *i*th dimension is unchanged.

`fnbrk(pp,j)`, with `pp` the ppform of a univariate function and `j` a positive integer, does not return a particular part, but returns the ppform of the `j`th polynomial piece of the function in `pp`. If `pp` is the ppform of an *m*-variate function, then `j` must be a cell array of length *m*. In that case, each entry of `j` must be a positive integer or else an interval, to single out a particular polynomial piece or else to specify the basic interval in that dimension.

`fnbrk(f)` returns nothing, but a description of the various parts of the form is printed at the command line instead.

Examples

If `p1` and `p2` contain the B-form of two splines of the same order, with the same knot sequence, and the same target dimension, then

```
p1plusp2 = spmak(fnbrk(p1,'k'),fnbrk(p1,'c')+fnbrk(p2,'c'));
```

provides the (pointwise) sum of those two functions.

If `pp` contains the ppform of a bivariate spline with at least four polynomial pieces in the first variable, then `ppp=fnbrk(pp,{4,[-1 1]})` gives the spline that agrees with the spline in `pp` on the rectangle `[b4 .. b5]` x `[-1 .. 1]`, where `b4`, `b5` are the fourth and fifth entry in the break sequence for the first variable.

See Also
ppmak | rpmak | rsmak | spmak | stmak

fnchg

Change part(s) of form

Syntax

```
f = fnchg(f,part,value)
```

Description

`f = fnchg(f,part,value)` returns the given function description `f` but with the specified `part` changed to the specified `value`.

The character vector `part` can be (the beginning character(s) of) :

`'dimension'`	The dimension of the function's target
`'interval'`	The basic interval of that form

The specified `value` for `part` is not checked for consistency with the rest of the description in `f` in case the character vector `part` terminates with the letter `z`.

Examples

`fndir(f,directions)` returns a vector-valued function even when the function described by `f` is ND-valued. You can correct this by using `fnchg` as follows:

```
fdir = fnchg( fndir(f,directions),...
          'dim',[fnbrk(f,'dim'),size(directions,2)] );
```

See Also
fnbrk

fncmb

Arithmetic with function(s)

Syntax

```
fn = fncmb(function,operation)
f = fncmb(function,function)
fncmb(function,matrix,function)
fncmb(function,matrix,function,matrix)
f = fncmb(function,op,function)
```

Description

The intent is to make it easy to carry out the standard linear operations of scaling and adding within a spline space without having to deal explicitly with the relevant parts of the function(s) involved.

`fn = fncmb(function,operation)` returns (a description of) the function obtained by applying to the values of the function in `function` the operation specified by `operation`. The nature of the operation depends on whether `operation` is a *scalar,* a *vector,* a *matrix,* or a *character vector,* as follows.

Scalar	Multiply the function by that scalar.
Vector	Add that vector to the function's values; this requires the function to be vector-valued.
Matrix	Apply that matrix to the function's coefficients.
Character array	Apply the function specified by that character vector to the function's coefficients.

The remaining options only work for *univariate* functions. See Limitations for more information.

`f = fncmb(function,function)` returns (a description of) the pointwise sum of the two functions. The two functions must be of the same form. This particular case of just two input arguments is not included in the above table since it only works for univariate functions.

`fncmb(function,matrix,function)` is the same as `fncmb(fncmb(function,matrix),function)`.

`fncmb(function,matrix,function,matrix)` is the same as `fncmb((fncmb(function,matrix),fncmb(function,matrix)))`.

`f = fncmb(function,op,function)` returns the ppform of the spline obtained by the pointwise combining of the two functions, as specified by the character vector `op`. The argument `op` can be one of the character vectors `'+'`, `'-'`, `'*'`. If the second function is to be a constant, it is sufficient simply to supply here that constant.

Examples

`fncmb(fn,3.5)` multiplies (the coefficients of) the function in `fn` by 3.5.

`fncmb(f,3,g,-4)` returns the linear combination, with weights 3 and –4, of the function in `f` and the function in `g`.

`fncmb(f,3,g)` adds 3 times the function in `f` to the function in `g`.

If the function *f* in `f` happens to be scalar-valued, then `f3=fncmb(f,[1;2;3])`contains the description of the function whose value at *x* is the 3-vector (*f*(*x*), 2*f*(*x*), 3*f*(*x*)). Note that, by the convention throughout this toolbox, the subsequent statement fnval(*f3*, *x*) returns a 1-*column*-matrix.

If `f` describes a surface in R^3, i.e., the function in `f` is 3-vector-valued bivariate, then `f2 = fncmb(f, [1 0 0;0 0 1])` describes the projection of that surface to the (*x*, *z*)-plane.

The following commands produce the picture of a ... spirochete?

```
c = rsmak('circle');
fnplt(fncmb(c,diag([1.5,1]))); axis equal, hold on
sc = fncmb(c,.4);
fnplt(fncmb(sc,-[.2;-.5]))
fnplt(fncmb(sc,-[.2;-.5]))
hold off, axis off
```

If `t` is a knot sequence of length `n+k` and `a` is a matrix with `n` columns, then `fncmb(spmak(t,eye(n)),a)` is the same as `spmak(t,a)`.

`fncmb(spmak([0:4],1),'+',ppmak([-1 5],[1 -1]))` is the piecewise-polynomial with breaks `-1:5` that, on the interval [0 .. 4], agrees with the function $x| \to B(x|0,1,2,3,4) + x$ (but has no active break at 0 or 1, hence differs from this function outside the interval [0 .. 4]).

`fncmb(spmak([0:4],1),'-',0)` has the same effect as `fn2fm(spmak([0:4],1),'pp')`.

Assuming that `sp` describes the B-form of a spline of order `<k`, the output of

```
 fn2fm(fncmb(sp,'+',ppmak(fnbrk(sp,'interv'),zeros(1,k))),'B-')
```

describes the B-form of the same spline, but with its order raised to `k`.

Limitations

`fncmb` only works for *univariate* functions, except for the case `fncmb(function,operation)`, i.e., when there is just one function in the input.

Further, if two functions are involved, then they must be of the same type. This means that they must either both be in B-form or both be in ppform, and, moreover, have the same knots or breaks, the same order, and the same target. The only exception to this is the command of the form `fncmb(function,op,function)`.

Algorithms

The coefficients are extracted (via `fnbrk`) and operated on by the specified matrix or operation (and, possibly, added), then recombined with the rest of the function description (via `ppmak`, `spmak`,`rpmak`,`rsmak`,`stmak`). To be sure, when the function is rational, the matrix is only applied to the coefficients of the numerator. Again, if we are to translate the function values by a given vector and the function is in ppform, then only the coefficients corresponding to constant terms are so translated.

If there are two functions input, then they must be of the same type (see Limitations, below) *except* for the following.

`fncmb(f1,op,f2)` returns the ppform of the function

$$x| \rightarrow f1(x) \ op \ f2(x)$$

with `op` one of `'+'`, `'-'`, `'*'`, and `f1`, `f2` of arbitrary polynomial form. If, in addition, `f2` is a scalar or vector, it is taken to be the function that is constantly equal to that scalar or vector.

fnder

Differentiate function

Syntax

```
fprime = fnder(f,dorder)
fnder(f)
```

Description

`fprime = fnder(f,dorder)` returns the `dorder`-th derivative of the function in `f`. The default value of `dorder` is 1. For negative `dorder`, the particular |`dorder`|-th indefinite integral is returned that vanishes |`dorder`|-fold at the left endpoint of the basic interval.

The output is of the same form as the input, they are either both ppforms, or both B-forms, or both stforms.

If the function in `f` is *m*-variate, then `dorder` must be given, and must be of length *m*.

Also:

- If `f` is in ppform, or in B-form with its last knot of sufficiently high multiplicity, then, up to rounding errors, `f` and `fnder(fnint(f))` are the same.

- If `f` is in ppform and `fa` is the value of the function in `f` at the left end of its basic interval, then, up to rounding errors, `f` and `fnint(fnder(f),fa)` are the same, unless the function described by `f` has jump discontinuities.

- If `f` contains the B-form of *f*, and t_1 is its leftmost knot, then, up to rounding errors, `fnint(fnder(f))` contains the B-form of $f - f(t_1)$. However, its leftmost knot will have lost one multiplicity (if it had multiplicity > 1 to begin with). Also, its rightmost knot will have full multiplicity even if the rightmost knot for the B-form of *f* in `f` doesn't. To verify this, create a spline, `sp = spmak([0 0 1], 1)`. This spline is, on its basic interval [0..1], the straight line that is 1 at 0 and 0 at 1. Now integrate its derivative: `spdi = fnint(fnder(sp))`. The spline in `spdi` has the same basic interval, but, on that interval, it agrees with the straight line that is 0 at 0 and –1 at 1.

`fnder(f)` is the same as `fnder(f,1)`.

Examples

B-splines with Simple Knots and Their Derivatives

This example shows how to calculate the first and second order derivative functions of three B-splines of order 2, 3, and 4. Then it plots the splines, and their derivatives, and compares the results.

```
% Create the knots sequences
t1 = [0 .8 2];
t2 = [3 4.4 5  6];
t3 = [7  7.9  9.2 10 11];
```

```
tt = [t1 t2 t3];

% Accessory variables and commands for plotting purposes
cl = ['g','r','b','k','k'];
v = 5.4; d1 = 2.5; d2 = 0; s1 = 1; s2 = .5;
ext = tt([1 end])+[-.5 .5];
plot(ext([1 2]),[v v],cl(5))
hold on
plot(ext([1 2]),[d1 d1],cl(5))
plot(ext([1 2]),[d2 d2],cl(5))
ts = [tt;tt;NaN(size(tt))];
ty = repmat(.2*[-1;0;NaN],size(tt));
plot(ts(:),ty(:)+v,cl(5))
plot(ts(:),ty(:)+d1,cl(5))
plot(ts(:),ty(:)+d2,cl(5))

% Spline 1 (linear)
b1 = spmak(t1,1);
p1 = [t1;0 1 0];
% Calculate the first and second derivative of spline 1
db1 = fnder(b1);
p11 = fnplt(db1,'j');
p12 = fnplt(fnder(db1));
lw = 2;
plot(p1(1,:),p1(2,:)+v,cl(2),'LineWidth',lw)
plot(p11(1,:),s1*p11(2,:)+d1,cl(2),'LineWidth',lw)
plot(p12(1,:),s2*p12(2,:)+d2,cl(2),'LineWidth',lw)

% Spline 2 (quadratic)
b1 = spmak(t2,1);
p1 = fnplt(b1);
% Calculate the first and second derivative of spline 2
db1 = fnder(b1);
p11 = [t2;fnval(db1,t2)];
p12 = fnplt(fnder(db1),'j');
plot(p1(1,:),p1(2,:)+v,cl(3),'LineWidth',lw)
plot(p11(1,:),s1*p11(2,:)+d1,cl(3),'LineWidth',lw)
plot(p12(1,:),s2*p12(2,:)+d2,cl(3),'LineWidth',lw)

% Spline 3 (cubic)
b1 = spmak(t3,1);
p1 = fnplt(b1);
% Calculate the first and second derivative of spline 3
db1 = fnder(b1);
p11 = fnplt(db1);
p12=[t3;fnval(fnder(db1),t3)];
plot(p1(1,:),p1(2,:)+v,cl(4),'LineWidth',lw)
plot(p11(1,:),s1*p11(2,:)+d1,cl(4),'LineWidth',lw)
plot(p12(1,:),s2*p12(2,:)+d2,cl(4),'LineWidth',lw)

% Formatting the plot
tey = v+1.5;
text(t1(2)-.5,tey,'linear','FontSize',12,'Color',cl(2))
text(t2(2)-.8,tey,'quadratic','FontSize',12,'Color',cl(3))
text(t3(3)-.5,tey,'cubic','FontSize',12,'Color',cl(4))
text(-2,v,'B','FontSize',12)
text(-2,d1,'DB','FontSize',12)
text(-2,d2,'D^2B')
```

```
axis([-1 12 -2 7.5])
title({'B-splines with Simple Knots and Their Derivatives'})
axis off
hold off
```


B-splines with Simple Knots and Their Derivatives
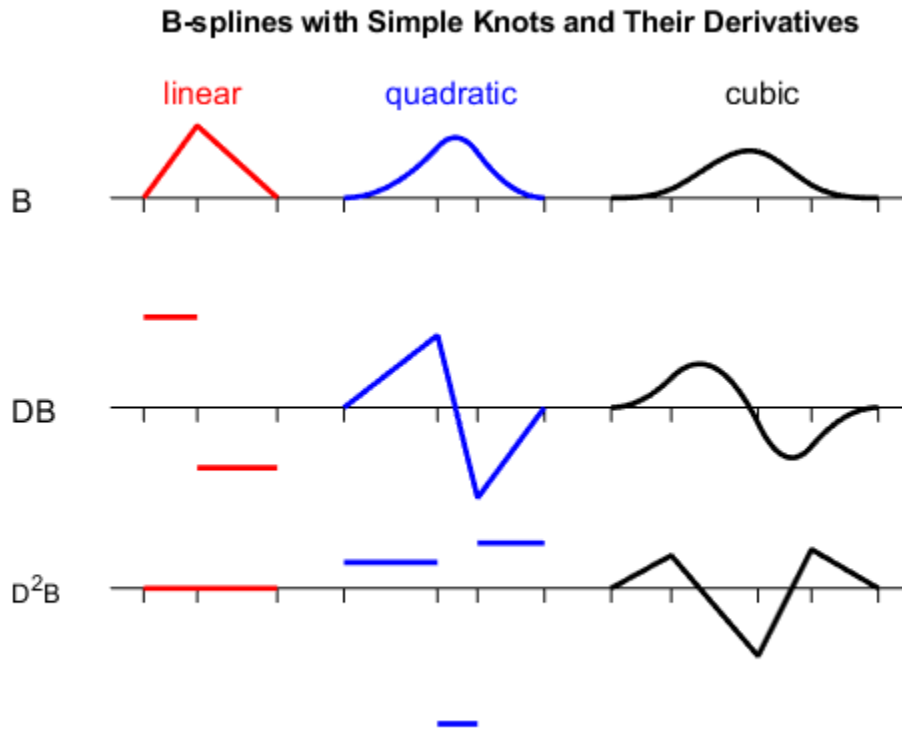
Input Arguments

f — Spline function
spline structure

Spline in either ppform, B-form or stform, specified as a structure with these fields:

Form — Form of spline
pp | B- | tp00

Form of the spline, returned as pp, B-, or tp00. pp indicates that the spline is given in piecewise polynomial form, B- indicates that the spline is given in B-form, and tp00 indicates that the spline is given in stform.

Knots — Knot locations of spline
vector | cell array

Knot positions of the spline, returned as a vector or as a cell array of vectors for multivariate data. Vectors contain strictly increasing elements that represent the start and end of each of the intervals over which the polynomial pieces are defined.

Coefs — Coefficients of polynomials
matrix | array

Coefficients of polynomials for each piece, returned as a matrix or as an array for multivariate data.

Number — Number of polynomial pieces
scalar | vector

Number of polynomial pieces describing the spline, returned as a scalar or as a vector of numbers of pieces in each variable for multivariate data.

Order — Order of polynomials
scalar | vector

Order of the polynomial function describing each polynomial piece of the spline, returned as a scalar or as a vector containing the order in each variable for multivariate data.

Dim — Dimensionality
scalar

Dimensionality of the target function, returned as a scalar.

dorder — Derivative order
scalar | vector

Order of the derivative of the function f, specified as a scalar or vector for multivariate functions.

Data Types: single | double

Output Arguments

fprime — Derivative function
spline structure

Derivative function of the f spline in either ppform, B-form, or stform, returned as a structure with these fields:

Form — Form of spline
pp | B- | tp00

Form of the spline, returned as pp, B-, or tp00. The form of the derivative function is the same as the form of the f function. pp indicates that the spline is given in piecewise polynomial form, B- indicates that the spline is given in B-form, and tp00 indicates that the spline is given in stform.

Knots — Knot locations of spline
vector | cell array

Knot positions of the spline, returned as a vector or as a cell array of vectors for multivariate data. Vectors contain strictly increasing elements that represent the start and end of each of the intervals over which the polynomial pieces are defined.

Coefs — Coefficients of polynomials
matrix | array

Coefficients of polynomials for each piece, returned as a matrix or as an array for multivariate data.

Number — Number of polynomial pieces
scalar | vector

Number of polynomial pieces describing the spline, returned as a scalar or as a vector of numbers of pieces in each variable for multivariate data.

Order — Order of polynomials
scalar | vector

Order of the polynomial function describing each polynomial piece of the spline, returned as a scalar or as a vector containing the order in each variable for multivariate data.

Dim — Dimensionality
scalar

Dimensionality of the target function, returned as a scalar.

Limitations

- The `fnder` function does not work with rational splines. To work with rational splines, use the `fntlr` function instead.
- The `fnder` function works for stforms only in a limited way: if the type is `tp00`, then `dorder` can be `[1,0]` or `[0,1]`.

Algorithms

For differentiation of either polynomial form, the `fnder` function finds the derivatives in the piecewise-polynomial sense. The function differentiates each polynomial piece separately, and ignores jump discontinuities between polynomial pieces during differentiation.

For the B-form, the function uses the [*PGS*; (X.10)] formulas for differentiation.

For the stform, differentiation relies on knowing a formula for the relevant derivative of the basis function of the particular type.

See Also
`fndir` | `fnint` | `fnplt` | `fnval`

Topics
About Splines in Curve Fitting on page 8-2

Introduced before R2006a

fndir

Directional derivative of function

Syntax

```
df = fndir(f,y)
```

Description

`df = fndir(f,y)` is the ppform of the directional derivative, of the function *f* in `f`, in the direction of the (column-)vector `y`. This means that `df` describes the function
$D_y f(x) := \lim_{t \to 0} (f(x + ty) - f(x))/t$.

If `y` is a matrix, with `n` columns, and *f* is `d`-valued, then the function in `df` is `prod(d)*n`-valued. Its value at *x*, reshaped to be of size `[d,n]`, has in its *j*th "column" the directional derivative of *f* at *x* in the direction of the *j*th column of `y`. If you prefer `df` to reflect explicitly the actual size of *f*, use instead

```
df = fnchg( fndir(f,y), 'dim',[fnbrk(f,'dim'),size(y,2)] );
```

Since `fndir` relies on the ppform of the function in `f`, it does not work for rational functions nor for functions in stform.

Examples

For example, if `f` describes an `m`-variate `d`-vector-valued function and `x` is some point in its domain, then, e.g., with this particular ppform `f` that describes a scalar-valued bilinear polynomial,

```
f = ppmak({0:1,0:1},[1 0;0 1]); x = [0;0];
[d,m] = fnbrk(f,'dim','var');
jacobian = reshape(fnval(fndir(f,eye(m)),x),d,m)
```

is the Jacobian of that function at that point (which, for this particular *scalar*-valued function, is its gradient, and it is zero at the origin).

As a related example, the next statements plot the gradients of (a good approximation to) the Franke function at a regular mesh:

```
xx = linspace(-.1,1.1,13); yy = linspace(0,1,11);
[x,y] = ndgrid(xx,yy); z = franke(x,y);
pp2dir = fndir(csapi({xx,yy},z),eye(2));
grads = reshape(fnval(pp2dir,[x(:) y(:)].'),...
    [2,length(xx),length(yy)]);
quiver(x,y,squeeze(grads(1,:,:)),squeeze(grads(2,:,:)))
```

Here is the resulting plot.
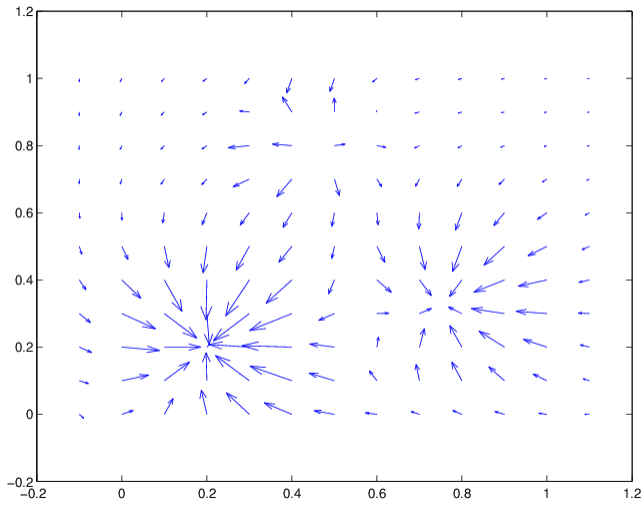
Algorithms

The function in f is converted to ppform, and the directional derivative of its polynomial pieces is computed formally and in one vector operation, and put together again to form the ppform of the directional derivative of the function in f.

See Also
fnchg | fnder | fnint | franke

fnint

Integrate function

Syntax

```
intgrf = fnint(f,value)
fnint(f)
```

Description

`intgrf = fnint(f,value)` is the description of an indefinite integral of the *univariate* function whose description is contained in `f`. The integral is normalized to have the specified `value` at the left endpoint of the function's basic interval, with the default value being zero.

The output is of the same type as the input, i.e., they are both ppforms or both B-forms. `fnint` does not work for rational splines nor for functions in stform.

`fnint(f)` is the same as `fnint(f,0)`.

Indefinite integration of a *multivariate* function, in coordinate directions only, is available via `fnder(f,dorder)` with `dorder` having nonpositive entries.

Examples

The statement `diff(fnval(fnint(f),[a b]))` provides the definite integral over the interval [a .. b] of the function described by `f`.

If `f` is in ppform, or in B-form with its last knot of sufficiently high multiplicity, then, up to rounding errors, `f` and `fnder(fnint(f))` are the same.

If `f` is in ppform and `fa` is the value of the function in `f` at the left end of its basic interval, then, up to rounding errors, `f` and `fnint(fnder(f),fa)` are the same, unless the function described by `f` has jump discontinuities.

If `f` contains the B-form of *f*, and t_1 is its leftmost knot, then, up to rounding errors, `fnint(fnder(f))` contains the B-form of *f* – *f*(t_1). However, its leftmost knot will have lost one multiplicity (if it had multiplicity > 1 to begin with). Also, its rightmost knot will have full multiplicity even if the rightmost knot for the B-form of *f* in `f` doesn't.

Here is an illustration of this last fact. The spline in `sp = spmak([0 0 1], 1)` is, on its basic interval [0..1], the straight line that is 1 at 0 and 0 at 1. Now integrate its derivative: `spdi = fnint(fnder(sp))`. As you can check, the spline in `spdi` has the same basic interval, but, on that interval, it agrees with the straight line that is 0 at 0 and -1 at 1.

See the examples "Intro to B-form" and "Intro to ppform" for examples.

Algorithms

For the B-form, the formula [*PGS*; (X.22)] for integration is used.

See Also

fnder | fnplt | fnval

fnjmp

Jumps, i.e., f(x+)-f(x-)

Syntax

```
jumps = fnjmp(f,x)
```

Description

`jumps = fnjmp(f,x)` is like `fnval(f,x)` except that it returns the jump $f(x+) – f(x–)$ across `x` (rather than the value at `x`) of the function f described by `f` and that it only works for univariate functions.

This is a function for spline specialists.

Examples

`fnjmp(ppmak(1:4,1:3),1:4)` returns the vector `[0,1,1,0]` since the `pp` function here is 1 on [1 .. 2], 2 on [2 .. 3], and 3 on [3 .. 4], hence has zero jump at 1 and 4 and a jump of 1 across both 2 and 3.

If `x` is `cos([4:-1:0]*pi/4)`, then `fnjmp(fnder(spmak(x,1),3),x)` returns the vector `[12 -24 24 -24 12]` (up to round-off). This is consistent with the fact that the spline in question is a so called perfect cubic B-spline, i.e., has an absolutely constant third derivative (on its basic interval). The modified command

```
fnjmp(fnder(fn2fm(spmak(x,1),'pp'),3),x)
```

returns instead the vector `[0 -24 24 -24 0]`, consistent with the fact that, in contrast to the B-form, a spline in ppform does not have a discontinuity in any of its derivatives at the endpoints of its basic interval. Note that `fnjmp(fnder(spmak(x,1),3),-x)` returns the vector `[12,0,0,0,12]` since `-x`, though theoretically equal to `x`, differs from `x` by round-off, hence the third derivative of the B-spline provided by `spmak(x,1)` does not have a jump across `-x(2)`,`-x(3)`, and `-x(4)`.

fnmin

Minimum of function in given interval

Syntax

```
fnmin(f)
fnmin(f,interv)
[minval,minsite] = fnmin(f,...)
```

Description

`fnmin(f)` returns the minimum value of the *scalar-valued univariate* spline in `f` on its basic interval.

`fnmin(f,interv)` returns the minimum value of the *scalar-valued univariate* spline in `f` on the interval [a..b] specified by `interv`.

`[minval,minsite] = fnmin(f,...)` also returns a location, `minsite`, at which the function in `f` takes that minimum value, `minval`.

Examples

Compute the Maximum and Minimum Values of a Spline

This example shows how to calculate the maximum and minimum values of a spline in `f` using the `fnmin` function.

Calculate the Maximum Value

Construct and plot a spline `f` with 21 knots and 15 random coefficients.

```
f = spmak(1:21,rand(1,15)-.5);
fnplt(f)
```

Compute the maximum value of `f` as the negative of the minimum of `-f,` then plot it as a horizontal line at the height of the computed maximum.

```
maxval = -fnmin(fncmb(f,-1));
hold on, plot(fnbrk(f,'interv'),maxval([1 1])), hold off
```
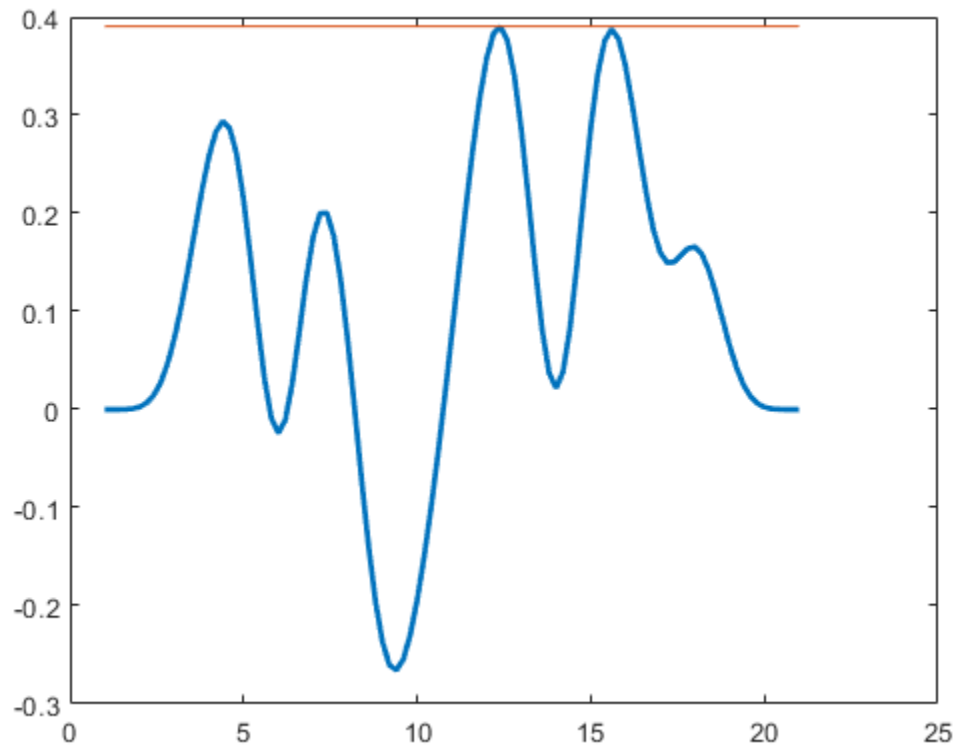
Calculate the Minimum Value

Construct and plot a spline using the `spmak` function.

```
f2 = spmak(1:5,-1);
fnplt(f2)
```
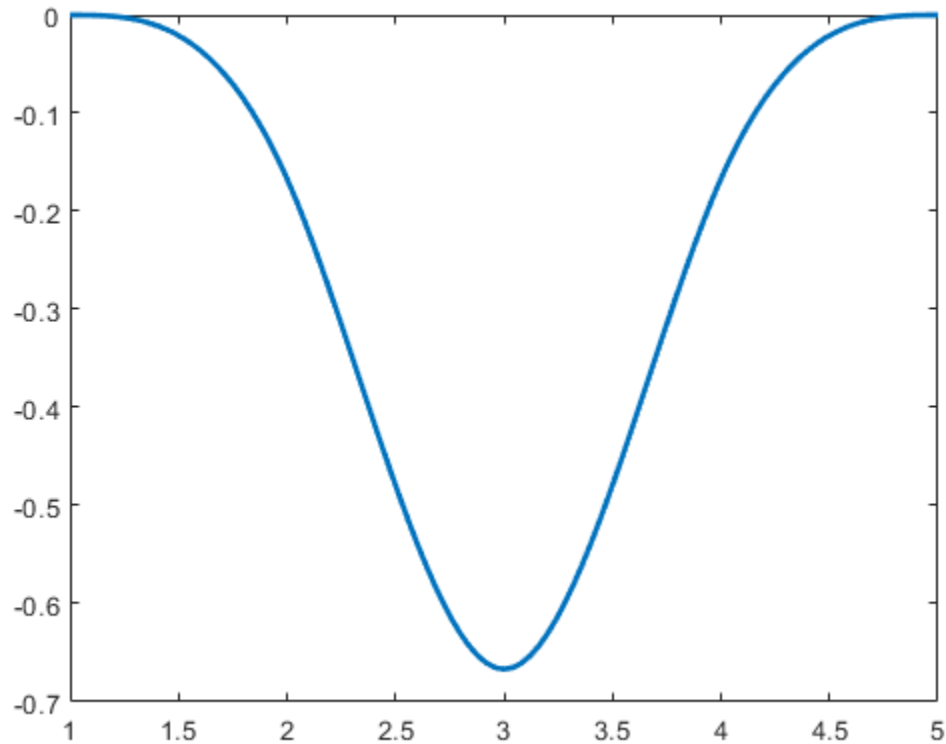
Compute the minimum value of `f2` and the site at which the spline takes on this minimum value.

```
[y,x] = fnmin(spmak(1:5,-1))

y = -0.6667

x = 3
```

Input Arguments

`f` — Spline structure
structure

Structure of a spline with the fields:

`form` — Polynomial spline representation
char

Form of the spline, returned as char.

`knots` — Knot sequence
vector

Non-decreasing sequence of the knots of the spline, returned as a vector.

`coefs` — Coefficients of spline
scalar | vector | matrix

Coefficients of the spline, returned as a scalar, vector, matrix.

`number` — Spline number
scalar

Number of pieces of the spline, returned as a scalar.

`order` — Spline order
scalar

Order of the spline, returned as a scalar.

`dim` — Dimension of coefficients
scalar

Dimension of the coefficients of the spline, returned as a scalar.

`interv` — Searching interval
vector (default)

Range of values where the function computes the minimum value of `f`, specified as a vector.

Data Types: `single` | `double`

Output Arguments

`minval` — Minimum value of spline
scalar

Minimum value of the scalar-valued univariate spline in `f`, returned as a scalar.

`minsite` — Site of minimum value
scalar

Site at which the spline in `f` takes on the minimum value, `minval`, returned as a scalar.

Algorithms

The `fnmin` algorithm first changes the basic interval of the function to the given interval, if any. On the interval, `fnmin` then finds all local extrema of the function as left and right limits at a jump and as zeros of the function's first derivative. It then evaluates the function at these extrema and at the endpoints of the interval, and determines the minimum over all these values.

See Also

fnval on page 13-160 | fnzeros on page 13-170

Topics
About Splines in Curve Fitting on page 8-2

Introduced in R2006b

fnplt

Plot function

Syntax

```
fnplt(f)
fnplt(f,symbol,interv,linewidth,jumps)
points = fnplt(f,...)
[points, t] = fnplt(f,...)
```

Description

`fnplt(f)` plots the function in `f` on its basic interval.

If *f* is univariate, then:

- If *f* is scalar-valued, `fnplt` plots the graph of *f*.
- If *f* is 2-vector-valued, `fnplt` plots the planar curve.
- If *f* is *d*-vector-valued with *d* > 2, `fnplt` plots the space curve given by the first three components of *f*.

If *f* is bivariate, then:

- If *f* is scalar-valued, `fnplt` plots the graph of *f*, using `surf`.
- If *f* is 2-vector-valued, `fnplt` plots the image in the plane of a regular grid in its domain.
- If *f* is *d*-vector-valued with *d* > 2, `fnplt` plots the parametric surface given by the first three components of its values, using `surf`.

If *f* is a function of more than two variables, then `fnplt` plots the bivariate function, obtained by choosing the midpoint of the basic interval in each of the variables other than the first two.

Note The basic interval for *f* in B-form is the interval containing *all* the knots. This means that *f* is sure to vanish at the endpoints of the basic interval unless the first and the last knot are both of full multiplicity *k*, with *k* the order of the spline *f*. Failure to have such full multiplicity is particularly annoying when *f* is a spline curve, since the plot of that curve as produced by `fnplt` is then bound to start and finish at the origin, regardless of what the curve might otherwise do.

Further, since B-splines are zero outside their support, any function in B-form is zero outside the basic interval of its form. This is very much in contrast to a function in ppform whose values outside the basic interval of the form are given by the extension of its leftmost, respectively rightmost, polynomial piece.

`fnplt(f,symbol,interv,linewidth,jumps)` permits you to modify the plotting by the specification of additional input arguments. You can place these arguments in whatever order you like, from the following list:

- A *character vector* that specifies a plotting symbol, such as `'-.'` or `'*'`; the default is `'-'`.

- A *scalar* to specify the linewidth; the default value is 1.

- A *character vector* that starts with the letter `'j'` to indicate that any jump in the *univariate* function being plotted appears as a jump. The default is to fill in any jump by a (near-)vertical line.

- A *vector* of the form `[a,b]`, to indicate the interval over which to plot the `univariate` function in `f`. If the function in `f` is *m*-variate, then this optional argument must be a cell array whose ith entry specifies the interval over which the *i*th argument is to vary. In effect, for this `arg`, the command `fnplt(f,arg,...)` has the same effect as the command `fnplt(fnbrk(f,arg),...)`. The default is the basic interval of `f`.

- An empty matrix or character vector, to indicate use of default(s). This option is useful when your particular choice depends on some other variables.

`points = fnplt(f,...)` plots nothing and returns the two-dimensional points or three-dimensional points it would have plotted instead.

`[points, t] = fnplt(f,...)` also returns, for a vector-valued `f`, the corresponding vector `t` of parameter values.

Examples

Plot a Spline Using the fnplt Function

This simple example shows how to plot a spline using the `fnplt` function.

Create a vector of data sites.

```
x=linspace(0,2*pi,21);
```

Generate a spline with the data sites `x` previously created.

```
f = spapi(4,x,sin(x))
```

```
f = struct with fields:
      form: 'B-'
     knots: [1x25 double]
     coefs: [1x21 double]
    number: 21
     order: 4
       dim: 1
```

Finally plot the spline using the `fnplt` function.

```
fnplt(f,'r',3,[1 3])
```
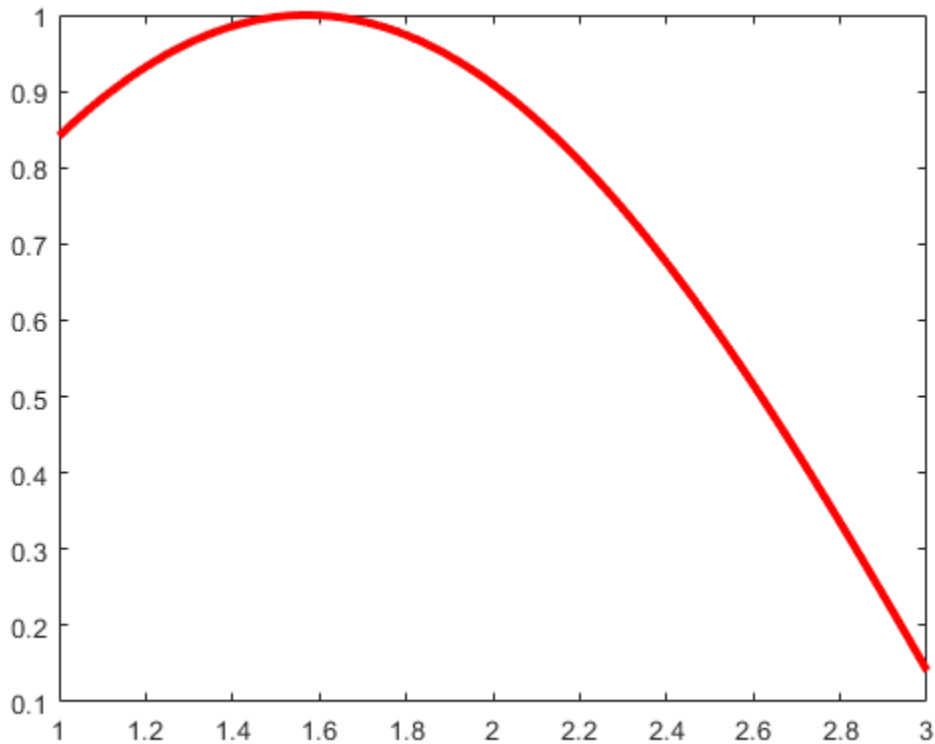
Input Arguments

`f` — Function to plot
scalar | vector | ND-array | spline structure

Function you want to plot, specified as a scalar, vector, ND-array, or a spline in either ppform, B-form or stform.

`symbol` — Plotting symbol
character vector

Symbol used to plot the function, specified as a character vector.

`interv` — Plotting interval
vector | cell array

Interval over which to plot the `univariate` function in `f`, specified as a vector. If the function in `f` is *m*-variate, then this parameter must be a cell array whose i-th entry specifies the interval over which the *i*-th argument is to vary.

Data Types: `single` | `double`

`linewidth` — Plotting line width
scalar

Width of the plotting line, specified as a scalar.

Data Types: single | double

jumps — Jump plotting specification
character vector

Specify how to plot a jump in the *univariate* function, specified as a character vector. The default is to fill in any jump by a (near-)vertical line.

Output Arguments

`points` — Plotting function points
vector | matrix

Two dimensional or three dimensional points of the function that would have been plotted, returned as a vector or matrix.

`t` — Parameter values
vector | matrix

Corresponding parameter values of function `f`, returned as a vector or matrix.

Algorithms

The `fnplt` functions generates a vector `x` of evaluation points by the union of:

1 101 equally spaced sites filling out the plotting interval
2 Any breakpoints in the plotting interval.

Then `fnplt` evaluates the univariate function f described by `f` at these `x` evaluation points. If f is real-valued, it plots the points $(x, f(x))$. If f is vector-valued, it plots the first two or three components of $f(x)$.

The bivariate function f described by `f` is evaluated on a 51-by-51 uniform grid if f is scalar-valued or d-vector-valued with $d > 2$ and the result plotted by `surf`. In the contrary case, f is evaluated along the meshlines of a 11-by-11 grid, and the resulting planar curves are plotted.

See Also
fnder | fnint | fnval

Topics
About Splines in Curve Fitting on page 8-2

Introduced before R2006a

fnrfn

Refine partition of form

Syntax

```
g = fnrfn(f,addpts)
```

Description

`g = fnrfn(f,addpts)` describes the same function as does `f`, but uses more terms to do it. This is of use when the sum of two or more functions of different forms is wanted or when the number of degrees of freedom in the form is to be increased to make fine local changes possible. The precise action depends on the form in `f`.

If the form in `f` is a B-form or BBform, then the entries of `addpts` are inserted into the existing knot sequence, subject to the following restriction: The multiplicity of no knot exceed the order of the spline. The equivalent B-form with this refined knot sequence for the function given by `f` is returned.

If the form in `f` is a ppform, then the entries of `addpts` are inserted into the existing break sequence, subject to the following restriction: The break sequence be strictly increasing. The equivalent ppform with this refined break sequence for the function in `f` is returned.

`fnrfn` does not work for functions in stform.

If the function in `f` is m-variate, then `addpts` must be a cell array, `{addpts1,..., addptsm}`, and the refinement is carried out in each of the variables. If the *i*th entry in this cell array is empty, then the knot or break sequence in the *i*th variable is unchanged.

Examples

Construct a spline in B-form, plot it, then apply two midpoint refinements, and also plot the control polygon of the resulting refined spline, expecting it to be quite close to the spline itself:

```
k = 4; sp = spapi( k, [1,1:10,10], [cos(1),sin(1:10),cos(10)] );
      fnplt(sp), hold on
      sp3 = fnrfn(fnrfn(sp));
      plot( aveknt( fnbrk(sp3,'knots'),k), fnbrk(sp3,'coefs'), 'r')
      hold off
```

A third refinement would have made the two curves indistinguishable.

Use `fnrfn` to add two B-splines of the same order:

```
B1 = spmak([0:4],1); B2 = spmak([2:6],1);
      B1r = fnrfn(B1,fnbrk(B2,'knots'));
      B2r = fnrfn(B2,fnbrk(B1,'knots'));
      B1pB2 = spmak(fnbrk(B1r,'knots'),fnbrk(B1r,'c')+fnbrk(B2r,'c'));
      fnplt(B1,'r'),hold on, fnplt(B2,'b'), fnplt(B1pB2,'y',2)
      hold off
```

Algorithms

The standard *knot insertion* algorithm is used for the calculation of the B-form coefficients for the refined knot sequence, while Horner's method is used for the calculation of the local polynomial coefficients at the additional breaks in the refined break sequence.

See Also
fncmb | ppmak | spmak

fntlr

Taylor coefficients

Syntax

```
taylor = fntlr(f,dorder,x)
```

Description

`taylor = fntlr(f,dorder,x)` returns the nonnormalized Taylor coefficients, up to the given order `dorder` and at the given `x`, of the function described in `f` .

For a univariate function and a scalar `x`, this is the vector

$$T(f, \text{dorder}, x) := [f(x); Df(x); ...; D^{\text{dorder}-1}f(x)]$$

If, more generally, the function in `f` is `d`-valued with `d>1` or even `prod(d)>1` and/or is `m`-variate for some `m>1`, then `dorder` is expected to be an `m`-vector of positive integers, `x` is expected to be a matrix with `m` rows, and, in that case, the output is of size `[prod(d)*prod(dorder),size(x,2)]`, with its `j`-th column containing

$$T(f, \text{dorder}, x(:, j))(i1, ..., im) = D_1^{i1-1}...D_m^{im-1}f(x(:, j))$$

for `i1=1:dorder(1)`, ..., `im=1:dorder(m)`. Here, $D_i f$ is the partial derivative of f with respect to its ith argument.

Examples

If `f` contains a univariate function and `x` is a scalar or a 1-row matrix, then `fntlr(f,3,x)` produces the same output as the statements

```
df = fnder(f); [fnval(f,x); fnval(df,x); fnval(fnder(df),x)];
```

As a more complicated example, look at the Taylor vectors of order 3 at 21 equally spaced points for the rational spline whose graph is the unit circle:

```
ci = rsmak('circle'); in = fnbrk(ci,'interv');
t = linspace(in(1),in(2),21); t(end)=[];
v = fntlr(ci,3,t);
```

We plot `ci` along with the points `v(1:2,:)`, to verify that these are, indeed, points on the unit circle.

```
fnplt(ci), hold on, plot(v(1,:),v(2,:),'o')
```

Next, to verify that `v(3:4,j)` is a vector tangent to the circle at the point `v(1:2,j)`, we use the MATLAB `quiver` command to add the corresponding arrows to our plot:

```
quiver(v(1,:),v(2,:),v(3,:),v(4,:))
```

Finally, what about `v(5:6,:)`? These are second derivatives, and we add the corresponding arrows by the following `quiver` command, thus finishing "First and Second Derivative of a Rational Spline Giving a Circle" on page 13-159.

```
quiver(v(1,:),v(2,:),v(5,:),v(6,:)), axis equal, hold off
```
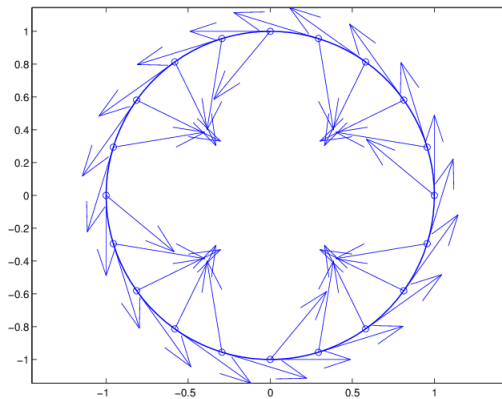

First and Second Derivative of a Rational Spline Giving a Circle

Now, our curve being a circle, you might have expected the 2nd derivative arrows to point straight to the center of that circle, and that would have been indeed the case if the function in `ci` had been using arclength as its independent variable. Since the parameter used is not arclength, we use the formula, given in "Example: B-form Spline Approximation to a Circle" on page 10-18, to compute the curvature of the curve given by `ci` at these selected points. For ease of comparison, we switch over to the variables used there and then simply use the commands from there.

```
dspt = v(3:4,:); ddspt = v(5:6,:);
kappa = abs(dspt(1,:).*ddspt(2,:)-dspt(2,:).*ddspt(1,:))./...
   (sum(dspt.^2)).^(3/2);
max(abs(kappa-1))
ans = 2.2204e-016
```

The numerical answer is reassuring: at all the points tested, the curvature is 1 to within roundoff.

See Also
fnder | fndir

fnval

Evaluate spline function

Syntax

```
v = fnval(f,x)
fnval(x,f)
fnval(...,'l')
```

Description

`v = fnval(f,x)` provides the value $f(x)$ at the points in `x` of the spline function f whose description is contained in `f`.

If `f` is scalar-valued and univariate, the output `v` is obtained by replacing each entry of `x` by the value of f at that entry. This is the intent in all other cases, except that, for a `d`-valued `m`-variate function, `d`-vectors replaces `m`-vectors.

For a univariate f :

*   If f is scalar-valued, then v is of the same size as `x`.
*   If f is `[d1,...,dr]`-valued, and `x` has size `[n1,...,ns]`, then `v` has size `[d1,...,dr, n1,...,ns]`, with `v(:,...,:, j1,...,js)` the value of f at `x(j1,...,js)`, – except that:

    *   `n1` is ignored if it is `1` and `s` is `2`, i.e., if `x` is a row vector;
    *   MATLAB ignores any trailing singleton dimensions of `x`.

For an `m`-variate f with `m>1`, with f `[d1,...,dr]`-valued, `x` might be either an array, or else a cell array `{x1,...,xm}`.

*   If `x` is an array, of size `[n1,...,ns]`, then `n1` must equal `m`, and `v` has size `[d1,...,dr, n2,...,ns]`, with `v(:,...,:, j2,...,js)` the value of f at `x(:,j2,...,js)`, – except that:

    *   `d1, ..., dr` is ignored in case f is scalar-valued, i.e., both `r` and `n1` are `1`;
    *   MATLAB ignores any trailing singleton dimensions of `x`.

*   If `x` is a cell array, then it must be of the form `{x1,...,xm}`, with `xj` a vector, of length `nj`, and, in that case, `v` has size `[d1,...,dr, n1,...,nm]`, with `v(:,...,:, j1,...,jm)` the value of f at `(x1(j1), ..., xm(jm))`, – except that `d1, ..., dr` is ignored in case f is scalar-valued, i.e., both `r` and `n1` are `1`.

If f has a jump discontinuity at `x`, then the value $f(x +)$, i.e., the limit from the right, is returned, except when `x` equals the right end of the basic interval of the form; for such `x`, the value $f(x–)$, i.e., the limit from the left, is returned.

`fnval(x,f)` is the same as `fnval(f,x)`.

`fnval(...,'l')` treats f as continuous from the left. This means that if f has a jump discontinuity at `x`, then the value $f(x–)$, i.e., the limit from the left, is returned, except when `x` equals the left end of the basic interval; for such `x`, the value $f(x +)$ is returned.

If the function is *multivariate*, then the above statements concerning continuity from the left and right apply coordinate wise.

Examples

Evaluate Functions at Specified Points

This example shows how to interpolate some data and plot and evaluate the resulting functions.

Define some data.

```
x = [0.074 0.31 0.38 0.53 0.57 0.58 0.59 0.61 0.61 0.65 0.71 0.81 0.97];
y = [0.91 0.96 0.77 0.5 0.5 0.51 0.51 0.53 0.53 0.57 0.62 0.61 0.31];
```

Interpolate the data and plot the resulting function, f.

```
f = csapi( x, y )

f = struct with fields:
      form: 'pp'
    breaks: [1x12 double]
     coefs: [11x4 double]
    pieces: 11
     order: 4
       dim: 1


fnplt( f )
```
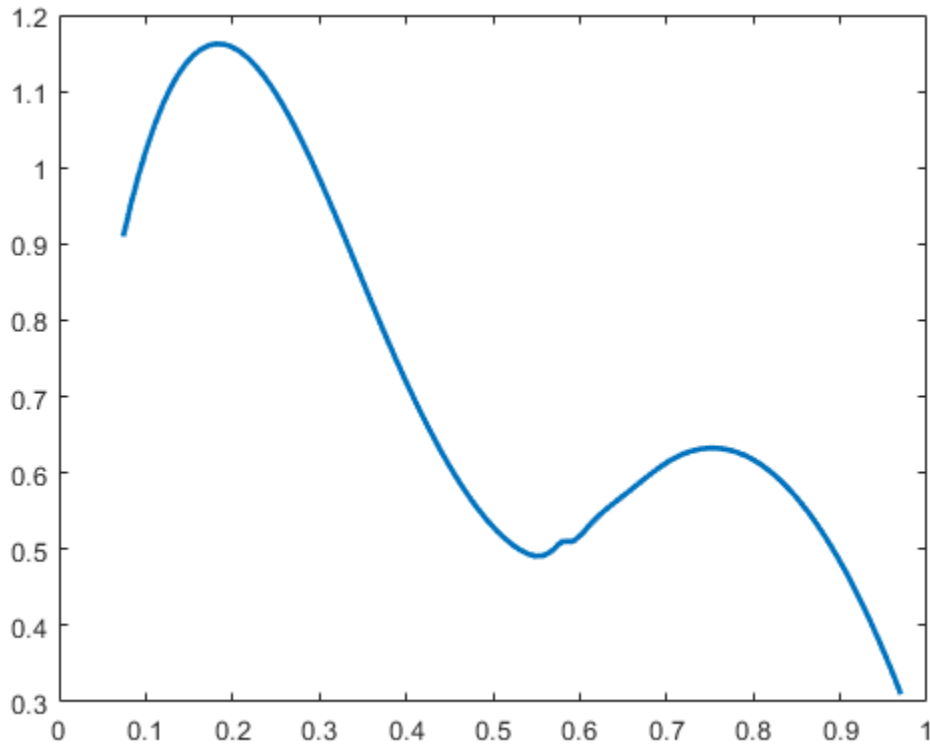
Find the value of the function f at x = 0.5.

```
fnval( f, 0.5 )
```

```
ans = 0.5294
```

Find the value of the function f at 0, 0.1, ..., 1.

```
fnval( f, 0:0.1:1 )
```

```
ans = 1×11
```

```
    0.3652    1.0220    1.1579    0.9859    0.7192    0.5294    0.5171    0.6134    0.6172    0.4
```

Create a function f2 that represents a surface.

```
x = 0.0001+(-4:0.2:4);
y = -3:0.2:3;
[yy, xx] = meshgrid( y, x );
r = pi*sqrt( xx.^2+yy.^2 );
z = sin( r )./r;
f2 = csapi( {x,y}, z );
```

Plot the function f2.

```
fnplt( f2 )
axis( [-5, 5, -5, 5, -0.5, 1] );
```
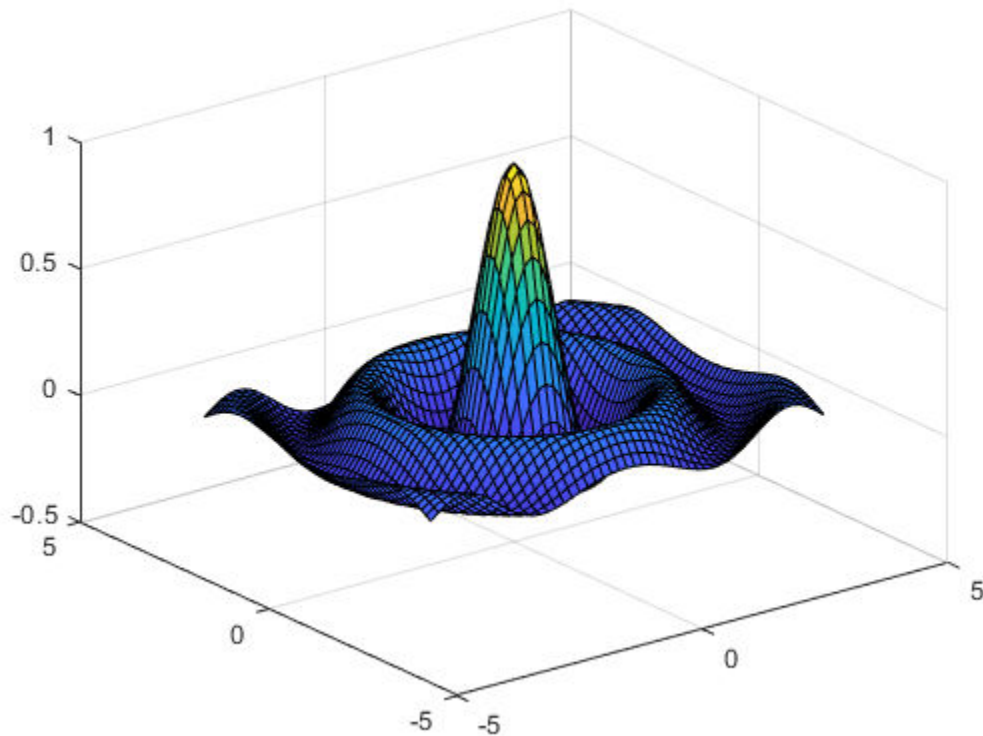
Find the value of the function f2 at x = -2 and y = 3.

```
fnval( f2, [-2; 3] )
```

```
ans = -0.0835
```

Input Arguments

f — Spline function
object

Spline function that you want to evaluate, specified as an object.

x — Evaluation points
vector | matrix | cell

Points at which you want to evaluate the spline function f, specified as a vector, matrix or cell array.

Output Arguments

v — Values at specified points
scalar | vector | matrix | cell

Value $f(x)$ at the points in x of the spline function f, returned as a scalar, vector, matrix or cell array.

Algorithms

For each entry of x, the function determines the relevant break-interval or knot-interval and assembles the relevant information. Depending on whether f is in ppform or in B-form, nested multiplication or the B-spline recurrence (see, e.g., [*PGS*; X.(3)]) are then used vector-fashion for the simultaneous evaluation at all entries of x. Evaluation of a multivariate polynomial spline function takes full advantage of the tensor product structure.

Evaluation of a rational spline follows up evaluation of the corresponding vector-valued spline by division of all but its last component by its last component.

Evaluation of a function in stform makes essential use of stcol, and tries to keep the matrices involved to reasonable size.

See Also

fnbrk | ppmak | rsmak | spmak | stmak

Topics
About Splines in Curve Fitting on page 8-2
"The ppform" on page 10-8
"The B-form" on page 10-13

Introduced in R2006b

fnxtr

Extrapolate spline

Syntax

```
g = fnxtr(f,order)
g = fnxtr(f)
```

Description

`g = fnxtr(f,order)` returns a spline of order `order` that extrapolates the spline `f`. `g` equals `f` on its basic interval, but `g` is a polynomial of the given order outside that interval. `g` satisfies at least `order` smoothness conditions at the ends of the basic interval of `f`, that is, at the new breaks. It is most useful to use a positive value of `order` that is smaller than the order of `f`.

`g = fnxtr(f)` uses an extrapolation order equal to 2. It is equivalent to `fnxtr(f,2)`.

Examples

Extrapolate Cubic Smoothing Spline

Create a cubic smoothing spline on the unit interval.

```
x = rand(1,21);
spline = csaps(x,x.^3);
```

Create an extrapolating spline of order 2.

```
order = 2;
extrSpline = fnxtr(spline,order);
```

Plot the original spline together with the extrapolating spline.

```
fnplt(spline,[-.5 1.4])
hold on
fnplt(extrSpline,[-.5 1.4])
legend('Cubic smoothing spline','Extrapolating spline')
hold off
```
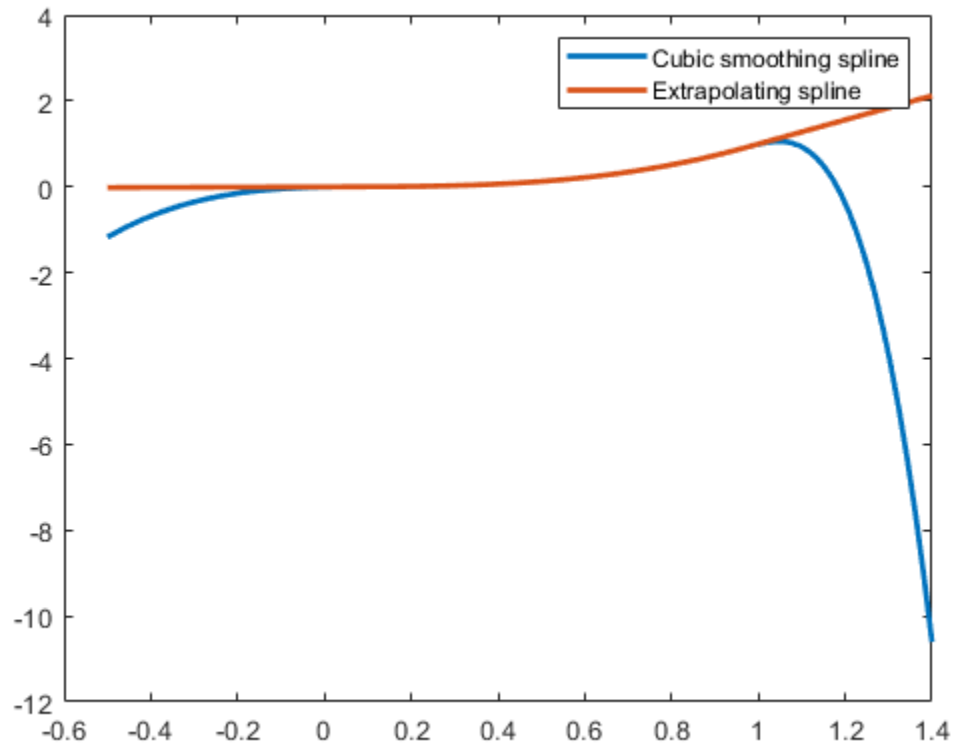
Extrapolate Bivariate B-Spline

Create and plot a bivariate B-spline.

```
spline = spmak({0:3,0:4},1);
fnplt(spline)
```
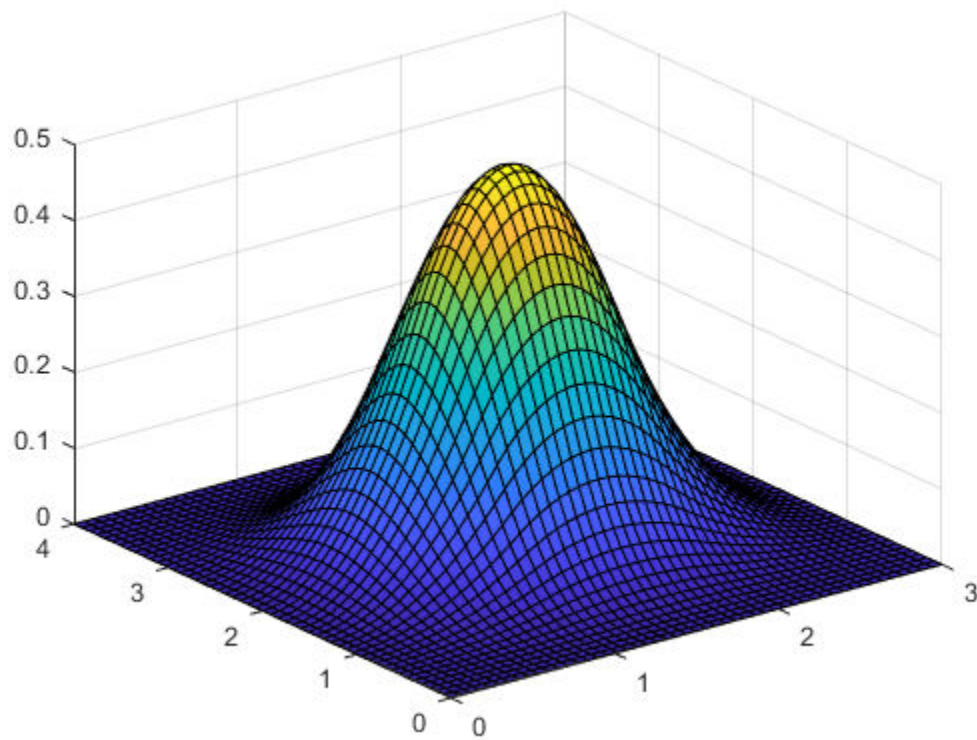
Create an extrapolating spline. To extrapolate in the first variable only, specify a negative integer as the extrapolation order in the second variable.

```
order = [3,-1];
extrSpline = fnxtr(spline,order);
fnplt(extrSpline)
```
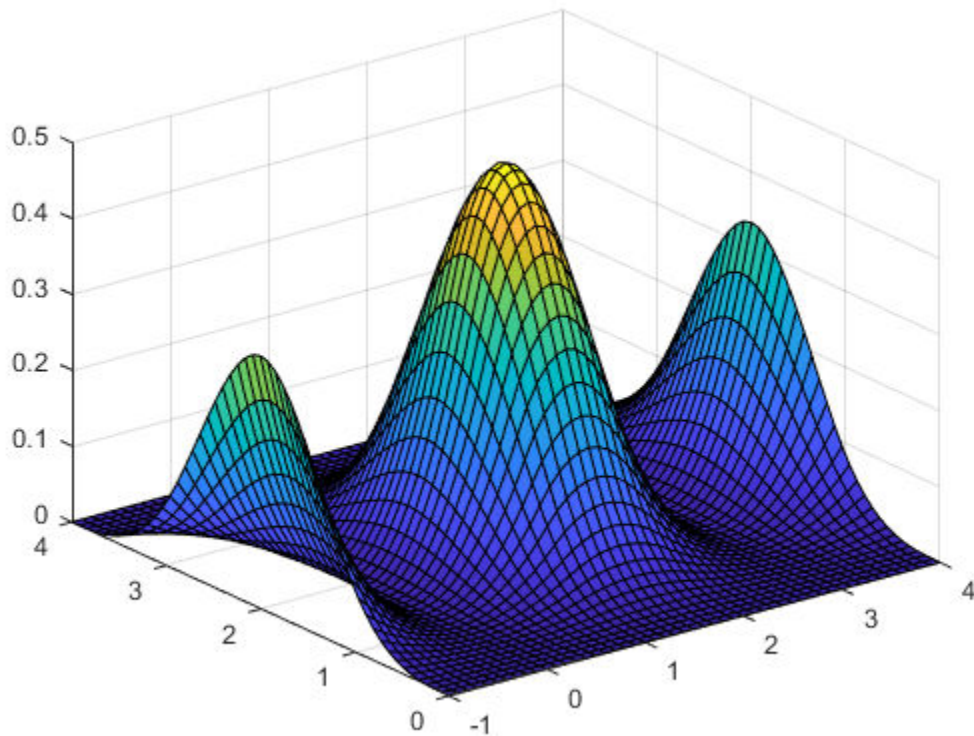
Input Arguments

f — Spline to extrapolate
structure

Spline to extrapolate, specified as a structure. f must be a spline in B-form, BBform, or ppform.

Data Types: struct

order — Order of extrapolating spline
integer | vector of integers

Order of extrapolating spline, specified as an integer or a vector of integers.

If order is zero, then the extrapolating spline describes the same spline as fn2fm(f,'B-'), but is in ppform and has a larger basic interval.

If order is at least as large as the order of f, then the extrapolating spline is the same spline as gn2fm(f,'pp'), but uses two more pieces and has a larger basic interval.

If f is *m*-variate, then order can be a vector with *m* elements, where order(i) is the order used to extrapolate in the i-th variable. To exclude the i-th variable from being used in the extrapolation, specify order(i) as a negative integer.

Example: 1

Output Arguments

pp — Spline in ppform
spline structure

Spline in ppform, returned as a structure with these fields.

Form — Form of spline
pp

Form of the spline, returned as pp. pp indicates that the spline is given in piecewise polynomial form.

Breaks — Knot locations of spline
vector | cell array

Knot positions of the spline, returned as a vector or as a cell array of vectors for multivariate data. Vectors contain strictly increasing elements that represent the start and end of each of the intervals over which the polynomial pieces are defined.

Coefs — Coefficients of polynomials
matrix | array

Coefficients of polynomials for each piece, returned as a matrix or as an array for multivariate data.

Pieces — Number of polynomial pieces
scalar | vector

Number of polynomial pieces describing the spline, returned as a scalar or as a vector of numbers of pieces in each variable for multivariate data.

Order — Order of polynomials
scalar | vector

Order of the polynomial function describing each polynomial piece of the spline, returned as a scalar or as a vector containing the order in each variable for multivariate data.

Dim — Dimensionality
scalar

Dimensionality of the target function, returned as a scalar.

See Also
fn2fm | ppmak | spmak

Introduced in R2006a

fnzeros

Roots of spline

Syntax

```
x = fnzeros(spline)
x = fnzeros(spline,I)
```

Description

`x = fnzeros(spline)` finds the zeros of a univariate spline in its basic interval. A spline zero is either a point or a closed interval over which the spline is zero.

`x = fnzeros(spline,I)` finds the zeros in the interval `I`.

Examples

Find Roots of Spline

Create and plot a piecewise linear spline.

```
spline = spmak(augknt(1:7,2),[1,0,1,-1,0,0,1]);
figure
fnplt(spline)
```
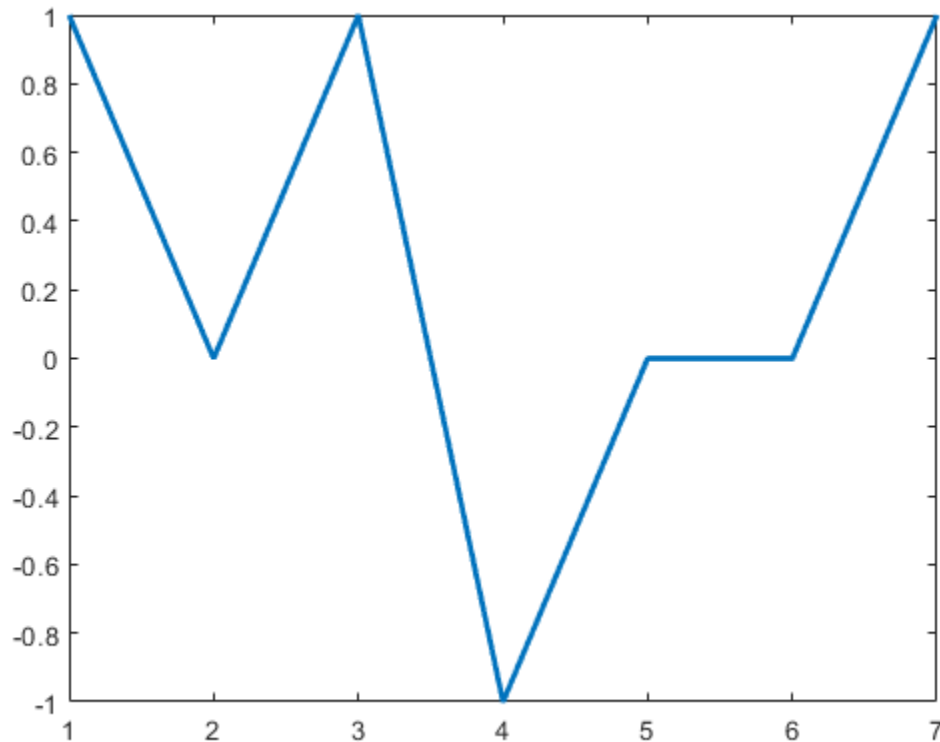
Find the roots of the spline. The spline has each of the three kinds of zeros: touch zero, cross zero, and zero over an interval.

```
x = fnzeros(spline)
```

x = 2×3

```
    2.0000    3.5000    5.0000
    2.0000    3.5000    6.0000
```

Plot the roots of the spline using right-pointing triangles at the left endpoints and left-pointing triangles at the right endpoints.

```
nz = size(x,2);
hold on
plot(x(1,:),zeros(1,nz),'r>',x(2,:),zeros(1,nz),'r<','MarkerSize',10)
hold off
```
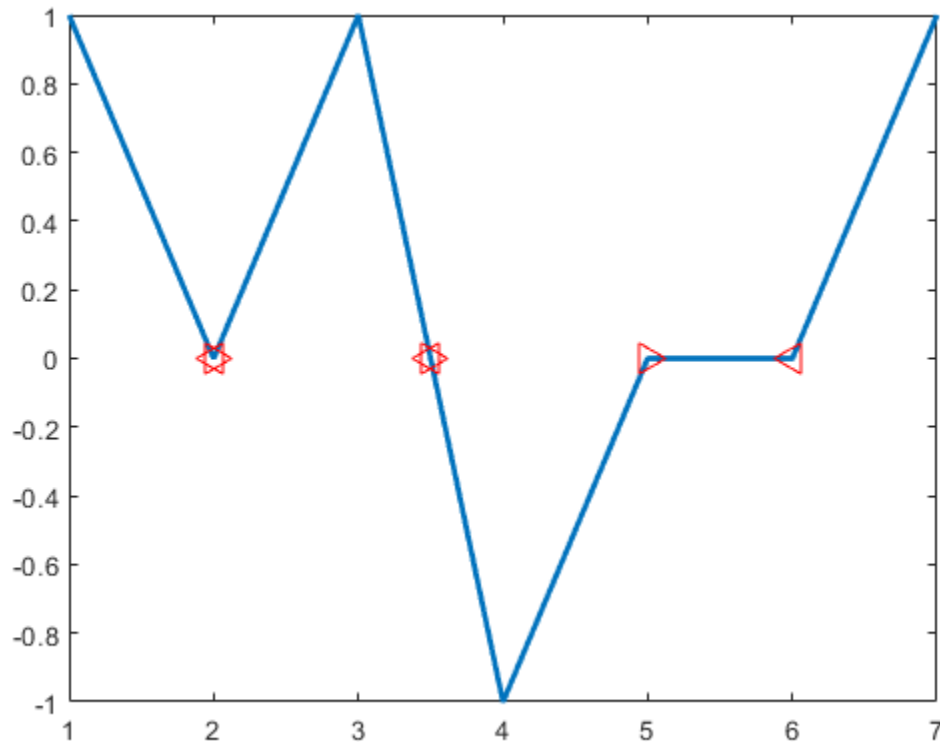
Find Local Extrema of Spline

Create a spline with many local extrema. Locate the local extrema of the spline by finding the roots of its first derivative.

```
spline = spmak(1:31,rand(1,25)-0.5);
x = fnzeros(fnder(spline));
```

Plot the spline and its local extrema.

```
figure
fnplt(spline)
hold on
x = unique(x(:));
y = fnval(spline,x);
plot(x,y,'ro')
xlim([-Inf Inf])
hold off
```
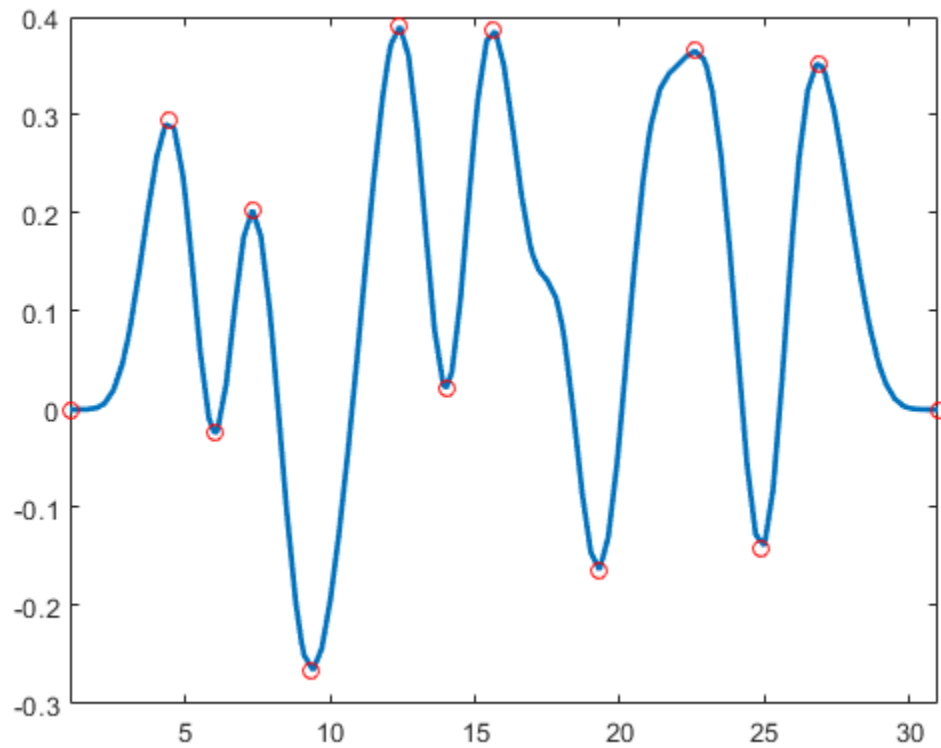
Find Roots of Discontinuous Spline

Create and plot a discontinuous piecewise linear spline.

```
spline = spmak([0 0 1 1 2 2],[-1 1 -1 1]);
figure
fnplt(spline);
```
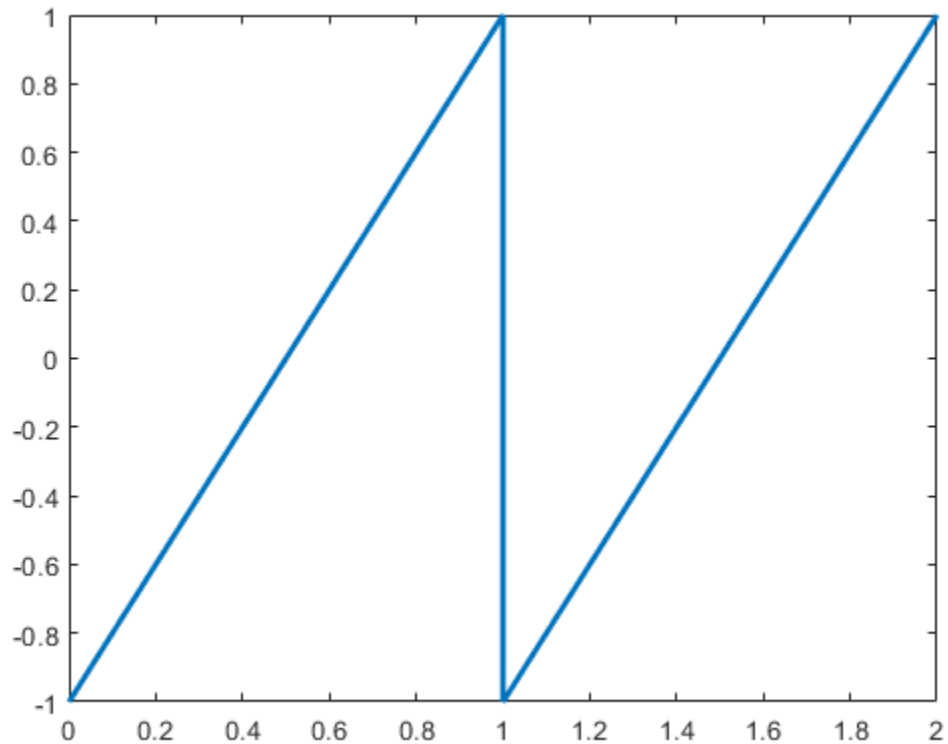
Find the roots of the spline. The roots include the jump through zero at x = 1.

```
x = fnzeros(spline)
```

x = *2×3*

```
    0.5000    1.0000    1.5000
    0.5000    1.0000    1.5000
```

Input Arguments

spline — Spline
structure

Spline, specified as a structure.

Data Types: `struct`

I — Interval to find zeros in
numeric vector with two elements

Interval to find zeros in, specified as a numeric vector with two elements.

Output Arguments

x — Location of roots
matrix with two rows

Locations of roots of spline, returned as a matrix with two rows. The elements of the first row are the left endpoints of the intervals and the elements of the second row are the right endpoints. Each column contains the left and right endpoint of a single interval. The roots are in increasing order.

There are three different types of intervals:

- If the endpoints are different, then the spline is zero on the entire interval. In this case, the largest possible interval is given, regardless of knots that may be in the interior of the interval.
- If the endpoints are the same and coincident with a knot, then the spline has a zero at that point. The spline could cross zero, touch zero, or be discontinuous at this point.
- If the endpoints are the same and not coincident with a knot, then the spline has a zero crossing at this point.

Algorithms

To find the roots of a spline, `fnzeros` first converts the spline to B-form. The function then performs some preprocessing to handle discontinuities and then uses the algorithm of [1].

References

[1] Mørken, Knut, and Martin Reimers. "An unconditionally convergent method for computing zeros of splines and polynomials." *Mathematics of Computation* 76, no. 258 (2007): 845-865.

See Also
`fnmin | fnval`

Introduced before R2006a

formula

Formula of `cfit`, `sfit`, or `fittype` object

Syntax

```
formula(fun)
```

Description

`formula(fun)` returns the formula of the `cfit`, `sfit`, or `fittype` object `fun` as a character array.

Examples

```
f = fittype('weibull');
formula(f)
ans =
a*b*x^(b-1)*exp(-a*x^b)

g = fittype('cubicspline');
formula(g)
ans =
piecewise polynomial
```

See Also

coeffnames | coeffvalues | fittype | numcoeffs | probnames

Introduced in R2006b

franke

Franke's bivariate test function

Syntax

```
z = franke(x,y)
```

Description

`z = franke(x,y)` returns the value `z(i)` of Franke's function at the site (`x(i)`,`y(i)`), `i=1:numel(x)`, with `z` of the same size as `x` and `y` (which must be of the same size).

Franke's function is the following weighted sum of four exponentials:

$$\frac{3}{4}e^{-\left((9x-2)^2+(9y-2)^2\right)/4} + \frac{3}{4}e^{-\left((9x+1)^2/49-(9y+1)/10\right)}$$

$$+\frac{1}{2}e^{-\left((9x-7)^2+(9y-3)^2\right)/4} - \frac{1}{5}e^{-\left((9x-4)^2+(9y-7)^2\right)}$$

Examples

The following commands provide a plot of Franke's function:

```
pts = (0:50)/50; [x,y] = ndgrid(pts,pts); z = franke(x,y);
surf(x,y,z), view(145,-2), set(gca,'Fontsize',16)
```
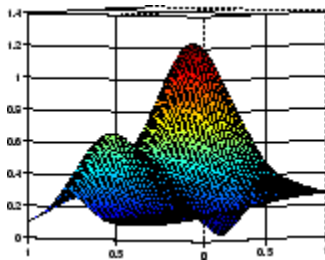

References

[1] Richard Franke. "A critical comparison of some methods for interpolation of scattered data." *Naval Postgraduate School Tech.Rep.* NPS-53-79-003, March 1979.

get

Get fit options structure property names and values

Syntax

```
get(options)
s = get(options)
value = get(options,field)
```

Description

get(options) displays all property names and values of the fit options structure options.

s = get(options) returns a copy of the fit options structure options as the structure s.

value = get(options,field) returns the value of the property *field* of the fit options structure options. *field* can be a cell array of character vectors, in which case value is also a cell array.

Examples

Obtain Property Values of a fit options Structure Using get

Create a fitoptions structure and call the get function to obtain its property values.

```
options = fitoptions('fourier1');
methodOptions = get(options,'Method')
maxIterOptions = get(options,'MaxIter')
set(options,'Maxiter',1e3);
maxIterOptions = get(options,'MaxIter')
```

```
methodOptions =

    'NonlinearLeastSquares'

maxIterOptions =

   400

maxIterOptions =

        1000
```

Property values can also be referenced and assigned using the dot notation. For example:

```
options.MaxIter
```

```
ans =
        1000
```

```
options.MaxIter = 500;
options.MaxIter
```

```
ans =
    500
```

Input Arguments

options — fitoptions object
fitoptions (default)

fitoptions object of which you want to get the property names and values, specified as a fitoptions object.

field — Property value
character vector (default)

Property value of the fitoptions object options that you want to get, specified as a character vector, or a cell array of character vectors.

Data Types: single | double

Output Arguments

s — fitoptions object
fitoptions

Copy of the fitoptions object, returned as a fitoption object.

value — Property value
scalar | vector | matrix | cell

Property value of the fitoptions object s, returned as a scalar, a vector, a matrix, or a cell.

See Also
fitoptions | set

Introduced before R2006a

getcurve

Interactive creation of cubic spline curve

Syntax

```
[xy,spcv] = getcurve
```

Description

`[xy,spcv] = getcurve` displays a gridded window and asks you for input. As you click on points in the gridded window, the broken line connecting these points is displayed. To indicate that you are done, click outside the gridded window. Then a cubic spline curve, `spcv`, through the point sequence, `xy`, is computed (via `cscvn`) and drawn. The point sequence and, optionally, the spline curve are output.

If you want a closed curve, place the last point *close* to the initial point.

If you would like the curve to have a corner at some point, click on that point twice (or more times) in succession.

You can't use `getcurve` over an existing figure, but you can use these functions to do the same thing: MATLAB function `ginput` , and `cscvn` in Curve Fitting Toolbox.

Examples

Draw a Spline Over an Image

You can't use `getcurve` over an existing figure, but you can use these functions to do the same thing. The following example code allows you to click an existing image to draw a spline through the points.

Draw the default image.

```
image
```

The function `ginput` collects mouse click points until you press **Enter**.

```
[x, y] = ginput
```

Click on the axis to select points. Press **Enter** when you have finished selecting points.

Fit and plot a spline through the points using the `cscvn` function.

```
spcv = cscvn( [x, y].' )
hold on
fnplt( spcv )
hold off
```

See Also

cscvn

indepnames

Independent variable of `cfit`, `sfit`, or `fittype` object

Syntax

```
indep = indepnames(fun)
```

Description

`indep = indepnames(fun)` returns the independent variable name or names (`indep`) of the `cfit`, `sfit`, or `fittype` object `fun`. For curves `indep` is a 1-by-1 cell array of character vectors, and for surfaces `indep` is a 2-by-1 cell array of character vectors.

Examples

```
f1 = fittype('a*x^2+b*exp(n*x)');
indep1 = indepnames(f1)
indep1 =
     'x'

f2 = fittype('a*x^2+b*exp(n*x)','independent','n');
indep2 = indepnames(f2)
indep2 =
     'n'
```

See Also
dependnames | fittype | formula

Introduced in R2006b

integrate

Integrate `cfit` object

Syntax

```
int = integrate(fun,x,x0)
```

Description

`int = integrate(fun,x,x0)` integrates the `cfit` object `fun` at the points specified by the vector `x`, starting from `x0`, and returns the result in `int`.

Examples

Find the Integral of a Fit Using the `integrate` Function

Create a baseline sinusoidal signal.

```
xdata = (0:.1:2*pi)';
y0 = sin(xdata);
```

Add response-dependent Gaussian noise to the signal.

```
noise = 2*y0.*randn(size(y0));
ydata = y0 + noise;
```

Fit the noisy data with a custom sinusoidal model.

```
f = fittype('a*sin(b*x)');
fit1 = fit(xdata,ydata,f,'StartPoint',[1 1]);
```

Find the integral of the fit at the predictors.

```
int = integrate(fit1,xdata,0);
```

Plot the data, the fit, and the integral.

```
subplot(2,1,1)
plot(fit1,xdata,ydata) % cfit plot method
subplot(2,1,2)
plot(xdata,int,'m') % double plot method
grid on
legend('integral')
```
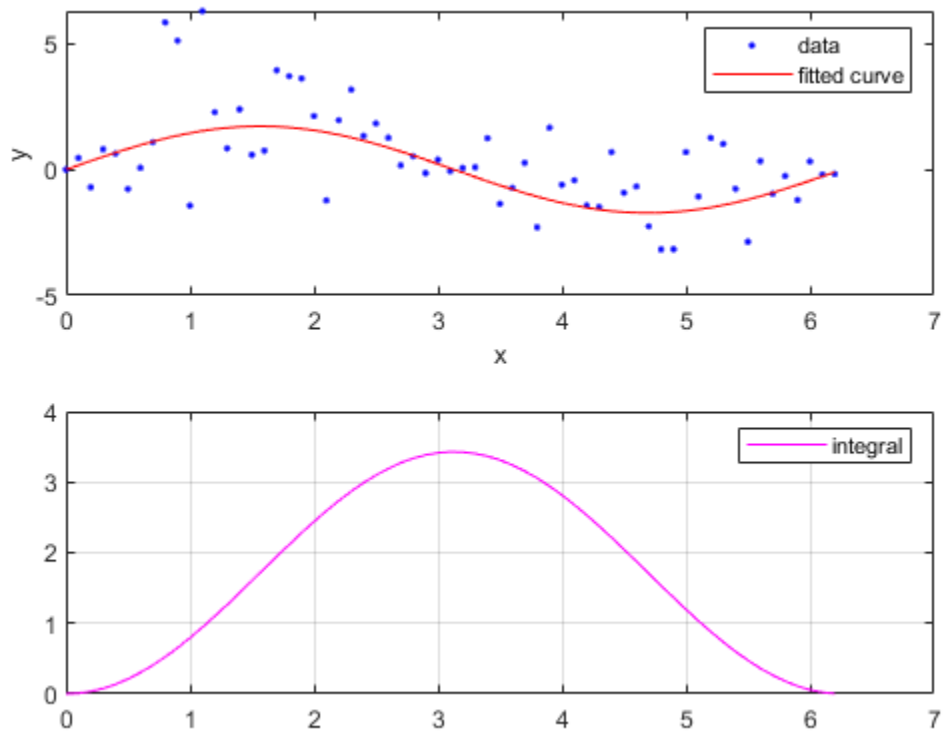
You can also compute integrals and plot them directly with the `cfit plot` method:

```
figure
plot(fit1,xdata,ydata,{'fit','integral'})
```
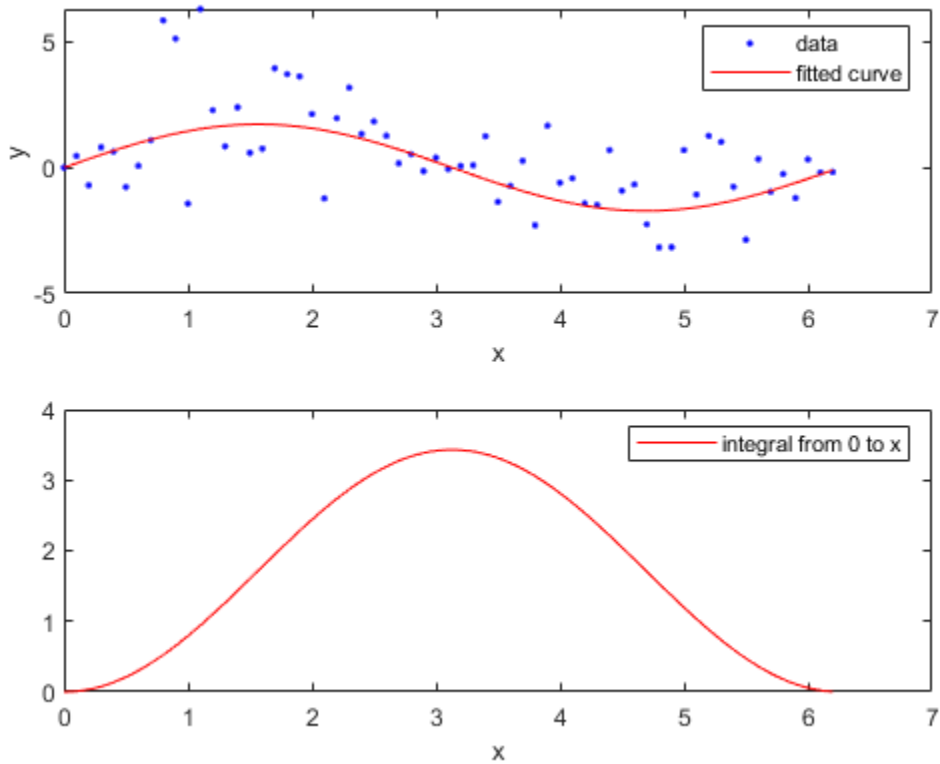
The `plot` method, however, does not return data on the integral.

Input Arguments

fun — `cfit` function
cfit

Function to integrate, specified as a `cfit` object.

x — Integration points
vector

Points at which to integrate the function, specified as a vector.

x0 — First point of integration
scalar

First point of integration, specified as a scalar.

Output Arguments

int — Integration results
vector

Result of the integration, returned as a vector of the same size of **x**.

See Also
differentiate | fit | plot

Topics
About Splines in Curve Fitting on page 8-2

Introduced before R2006a

islinear

Determine if `cfit`, `sfit`, or `fittype` object is linear

Syntax

```
flag = islinear(fun)
```

Description

`flag = islinear(fun)` returns a `flag` of `1` if the `cfit`, `sfit`, or `fittype` object `fun` represents a linear model, and a `flag` of `0` if it does not.

Note `islinear` assumes that all custom models specified by the `fittype` function using the syntax `ftype = fittype('expr')` are nonlinear models. To create a linear model with `fittype` that will be recognized as linear by `islinear` (and, importantly, by the algorithms of `fit`), use the syntax `ftype = fittype({'expr1','expr2',...,'exprn'})`.

Examples

```
f = fittype('a*x+b')
f =
     General model:
       f(a,b,x) = a*x+b
g = fittype({'x','1'})
g =
     Linear model:
       g(a,b,x) = a*x + b
h = fittype('poly1')
h =
     Linear model Poly1:
       h(p1,p2,x) = p1*x + p2

islinear(f)
ans =
     0
islinear(g)
ans =
     1
islinear(h)
ans =
     1
```

See Also
`fittype`

Introduced in R2006b

knt2brk, knt2mlt

Convert knots to breaks and their multiplicities

Syntax

```
knt2brk(knots)
[breaks,mults] = knt2brk(knots)
m = knt2mlt(t)
[m,sortedt] = knt2mlt(t)
```

Description

The commands extract the distinct elements from a sequence, as well as their multiplicities in that sequence, with *multiplicity* taken in two slightly different senses.

`knt2brk(knots)` returns the distinct elements in `knots`, and in increasing order, hence is the same as `unique(knots)`.

`[breaks,mults] = knt2brk(knots)` additionally provides, in `mults`, the multiplicity with which each distinct element occurs in `knots`. Explicitly, `breaks` and `mults` are of the same length, and `knt2brk` is complementary to `brk2knt` in that, for any knot sequence `knots`, the two commands `[xi,mlts] = knt2brk(knots); knots1 = brk2knt(xi,mlts);` give `knots1` equal to `knots`.

`m = knt2mlt(t)` returns a vector of the same length as `t`, with `m(i)` counting, in the vector `sort(t)`, the number of entries before its *i*th entry that are equal to that entry. This kind of multiplicity vector is needed in `spapi` or `spcol` where such multiplicity is taken to specify which particular derivatives are to be matched at the sites in `t`. Precisely, if `t` is nondecreasing and `z` is a vector of the same length, then `sp = spapi(knots, t, z)` attempts to construct a spline *s* (with knot sequence `knots`) for which $D^{m(i)}s(t(i))$ equals $z(i)$, all *i*.

`[m,sortedt] = knt2mlt(t)` also returns the output from `sort(t)`.

Neither `knt2brk` nor `knt2mlt` is likely to be used by the casual user of this toolbox.

Examples

`[xi,mlts]=knt2brk([1 2 3 3 1 3])` returns `[1 2 3]` for `xi` and `[2 1 3]` for `mlts`.

`[m,t]=knt2mlt([1 2 3 3 1 3])` returns `[0 1 0 0 1 2]` for `m` and `[1 1 2 3 3 3]` for `t`.

See Also
`brk2knt` | `spapi` | `spcol`

newknt

New break distribution

Syntax

```
newknots = newknt(f,newl)
newknt(f)
[...,distfn] = newknt(...)
```

Description

`newknots = newknt(f,newl)` returns the knot sequence whose interior knots cut the basic interval of `f` into `newl` pieces, in such a way that a certain piecewise linear monotone function related to the high derivative of `f` is equidistributed.

The intent is to choose a knot sequence suitable to the fine approximation of a function *g* whose rough approximation in `f` is assumed to contain enough information about *g* to make this feasible.

`newknt(f)` uses for `newl` its default value, namely the number of polynomial pieces in `f`.

`[...,distfn] = newknt(...)` also returns, in `distfn`, the ppform of that piecewise linear monotone function being equidistributed.

Examples

If the error in the least-squares approximation `sp` to some data `x,y` by a spline of order `k` seems uneven, you might try for a more equitable distribution of knots by using

```
spap2(newknt(sp),k,x,y);
```

For another example, see "Solving a Nonlinear ODE with a Boundary Layer by Collocation" on page 12-145.

Algorithms

This is the Fortran routine `NEWNOT` in *PGS*. With *k* the order of the piecewise-polynomial function *f* in `pp`, the function $|D^k f|$ is approximated by a piecewise constant function obtained by local, discrete, differentiation of the variation of $D^{k-1}f$. The new break sequence is chosen to subdivide the basic interval of the piecewise-polynomial *f* in such a way that

$$\int_{\text{newknots}(i)}^{\text{newknots}(i+1)} \left| D^k f \right|^{1/k} = \text{const}, \ i = k : k + newl - 1$$

numargs

Number of input arguments of `cfit`, `sfit`, or `fittype` object

Syntax

```
nargs = numargs(fun)
```

Description

`nargs = numargs(fun)` returns the number of input arguments `nargs` of the `cfit`, `sfit`, or `fittype` object `fun`.

Examples

```
f = fittype('a*x^2+b*exp(n*x)');
nargs = numargs(f)
nargs =
     4
args = argnames(f)
args =
    'a'
    'b'
    'n'
    'x'
```

See Also
`argnames` | `fittype` | `formula`

Introduced in R2006b

numcoeffs

Number of coefficients of `cfit`, `sfit`, or `fittype` object

Syntax

```
ncoeffs = numcoeffs(fun)
```

Description

`ncoeffs = numcoeffs(fun)` returns the number of coefficients `ncoeffs` of the `cfit`, `sfit`, or `fittype` object `fun`.

Examples

```
f = fittype('a*x^2+b*exp(n*x)');
ncoeffs = numcoeffs(f)
ncoeffs =
     3
coeffs = coeffnames(f)
coeffs =
    'a'
    'b'
    'n'
```

See Also
`coeffnames` | `fittype` | `formula`

Introduced in R2006b

optknt

Knot distribution "optimal" for interpolation

Syntax

```
knots = optknt(tau,k,maxiter)
optknt(tau,k)
```

Description

`knots = optknt(tau,k,maxiter)` provides the knot sequence `t` that is *best* for interpolation from $S_{k,t}$ at the site sequence `tau`, with `10` the default for the optional input `maxiter` that bounds the number of iterations to be used in this effort. Here, *best* or *optimal* is used in the sense of Micchelli/Rivlin/Winograd and Gaffney/Powell, and this means the following: For any *recovery scheme R* that provides an interpolant *Rg* that matches a given *g* at the sites `tau(1), ..., tau(n)`, we may determine the smallest constant $const_R$ for which $\|g - Rg\| \leq const_R \|D^k g\|$ for all smooth functions *g*.

Here, $\|f\| := \sup_{tau(1) < x < tau(n)} |f(x)|$. Then we may look for the optimal recovery scheme as the scheme *R* for which $const_R$ is as small as possible. Micchelli/Rivlin/Winograd have shown this to be interpolation from $S_{k,t}$, with `t` uniquely determined by the following conditions:

1. `t(1) = ... = t(k) = tau(1);`
2. `t(n+1) = ... = t(n+k) = tau(n);`
3. Any absolutely constant function *h* with sign changes at the sites `t(k+1), ..., t(n)` and nowhere else satisfies

$$\int_{tau(1)}^{tau(n)} f(x)h(x)dx = 0 \text{ for all } f \in S_{k,t}$$

Gaffney/Powell called this interpolation scheme *optimal* since it provides the *center* function in the band formed by all interpolants to the given data that, in addition, have their *k*th derivative between *M* and –*M* (for large *M*).

`optknt(tau,k)` is the same as `optknt(tau,k,10)`.

Examples

See the last part of the example "Spline Interpolation" for an illustration. For the following highly nonuniform knot sequence

```
t = [0, .0012+[0, 1, 2+[0,.1], 4]*1e-5, .002, 1];
```

the command `optknt(t,3)` will fail, while the command `optknt(t,3,20)`, using a high value for the optional parameter `maxiter`, will succeed.

Algorithms

This is the Fortran routine `SPLOPT` in *PGS*. It is based on an algorithm described in [1], for the construction of that sign function *h* mentioned above. It is essentially Newton's method for the

solution of the resulting nonlinear system of equations, with `aveknt(tau,k)` providing the first guess for `t(k+1)`, …,`t(n)`, and some damping used to maintain the Schoenberg-Whitney conditions.

References

[1] C. de Boor. "Computational aspects of optimal recovery." In *Optimal Estimation in Approximation Theory*, C.A. Micchelli & T.J. Rivlin eds., Plenum Publ., New York, 1977, 69-91.

[2] P.W. Gaffney & M.J.D. Powell. "Optimal interpolation." In *Numerical Analysis*, G.A. Watson ed., *Lecture Notes in Mathematics, No. 506*, Springer-Verlag, 1976, 90-99.

[3] C.A. Micchelli, T.J. Rivlin & S. Winograd. "The optimal recovery of smooth functions." *Numer. Math*. **80**, (1974), 903-906.

See Also
`aptknt` | `aveknt` | `newknt`

plot

Plot `cfit` or `sfit` object

Syntax

```
plot(sfit)
plot(sfit, [x, y], z)
H = plot(sfit, ..., Name,Value)
H = plot(sfit, ...)
plot(cfit)
plot(cfit,x,y)
plot(cfit,x,y,DataLineSpec)
plot(cfit,FitLineSpec,x,y,DataLineSpec)
plot(cfit,x,y,outliers)
plot(cfit,x,y,outliers,OutlierLineSpec)
plot(...,ptype,...)
plot(...,ptype,level)
H = plot(...)
```

Description

Note These syntaxes are available for surfaces, or `sfit` objects.

`plot(sfit)` plots the `sfit` object over the range of the current axes, if any, or otherwise over the range stored in the fit.

`plot(sfit, [x, y], z)` plots z versus x and y and plots `sfit` over the range of x and y.

`H = plot(sfit, ..., Name,Value)` selects which way to plot the surface fit object `sfit`.

`H = plot(sfit, ...)` returns a vector of handles of the plotted objects.

Note These syntaxes are available for curves, or `cfit` objects.

`plot(cfit)` plots the `cfit` object over the domain of the current axes, if any. If there are no current axes, and `fun` is an output from the `fit` function, the plot is over the domain of the fitted data.

`plot(cfit,x,y)` plots `cfit` together with the predictor data x and the response data y.

`plot(cfit,x,y,DataLineSpec)` plots the predictor and response data using the color, marker symbol, and line style specified by the `DataLineSpec` formatting character vector.

`plot(cfit,FitLineSpec,x,y,DataLineSpec)` plots `fun` using the color, marker symbol, and line style specified by the `FitLineSpec` formatting character vector, and plots x and y using the color, marker symbol, and line style specified by the `DataLineSpec` formatting character vector.

`plot(cfit,x,y,outliers)` plots data indicated by `outliers` in a different color. `outliers` can be an expression describing a logical vector, e.g., x > 10, a vector of integers indexing the points

you want to exclude, e.g., [1 10 25], or a logical array where `true` represents an outlier. You can create the array with `excludedata`.

`plot(cfit,x,y,outliers,OutlierLineSpec)` plots `outliers` using the color, marker symbol, and line style specified by the `OutlierLineSpec`.

`plot(...,ptype,...)` uses the plot type specified by `ptype`.

`plot(...,ptype,level)` plots prediction intervals with a confidence level specified by `level`.

Note This syntax is available for both curves and surfaces.

Plot types can be single or multiple, with multiple plot types specified as a cell array of character vectors. With a single plot type, `plot` draws to the current axes and can be used with commands like `hold` and `subplot`. With multiple plot types, `plot` creates subplots for each plot type.

`H = plot(...)` returns a vector of handles to the plotted objects.

Examples

Plot data, outliers, and the results of the fits using the `plot` function

This example shows how to plot the data, the outliers, and the results of three fit objects with different colors and line styles.

Create a baseline sinusoidal signal.

```
xdata = (0:0.1:2*pi)';
y0 = sin(xdata);
```

Add noise to the signal with non-constant variance.

```
% Response-dependent Gaussian noise
gnoise = y0.*randn(size(y0));

% Salt-and-pepper noise
spnoise = zeros(size(y0));
p = randperm(length(y0));
sppoints = p(1:round(length(p)/5));
spnoise(sppoints) = 5*sign(y0(sppoints));

ydata = y0 + gnoise + spnoise;
```

Fit the noisy data with a baseline sinusoidal model.

```
f = fittype('a*sin(b*x)');
fit1 = fit(xdata,ydata,f,'StartPoint',[1 1]);
```

Identify "outliers" as points at a distance greater than 1.5 standard deviations from the baseline model, and refit the data with the outliers excluded.

```
fdata = feval(fit1,xdata);
I = abs(fdata - ydata) > 1.5*std(ydata);
outliers = excludedata(xdata,ydata,'indices',I);
```

```
fit2 = fit(xdata,ydata,f,'StartPoint',[1 1],...
           'Exclude',outliers);
```

Compare the effect of excluding the outliers with the effect of giving them lower bisquare weight in a robust fit.

```
fit3 = fit(xdata,ydata,f,'StartPoint',[1 1],'Robust','on');
```

Plot the data, the outliers, and the results of the fits.

```
plot(fit1,'r-',xdata,ydata,'k.',outliers,'m*')
hold on
plot(fit2,'c--')
plot(fit3,'b:')
xlim([0 2*pi])
```


Plot the residuals for the two fits considering outliers.

```
figure
plot(fit2,xdata,ydata,'co','residuals')
hold on
plot(fit3,xdata,ydata,'bx','residuals')
hold off
```
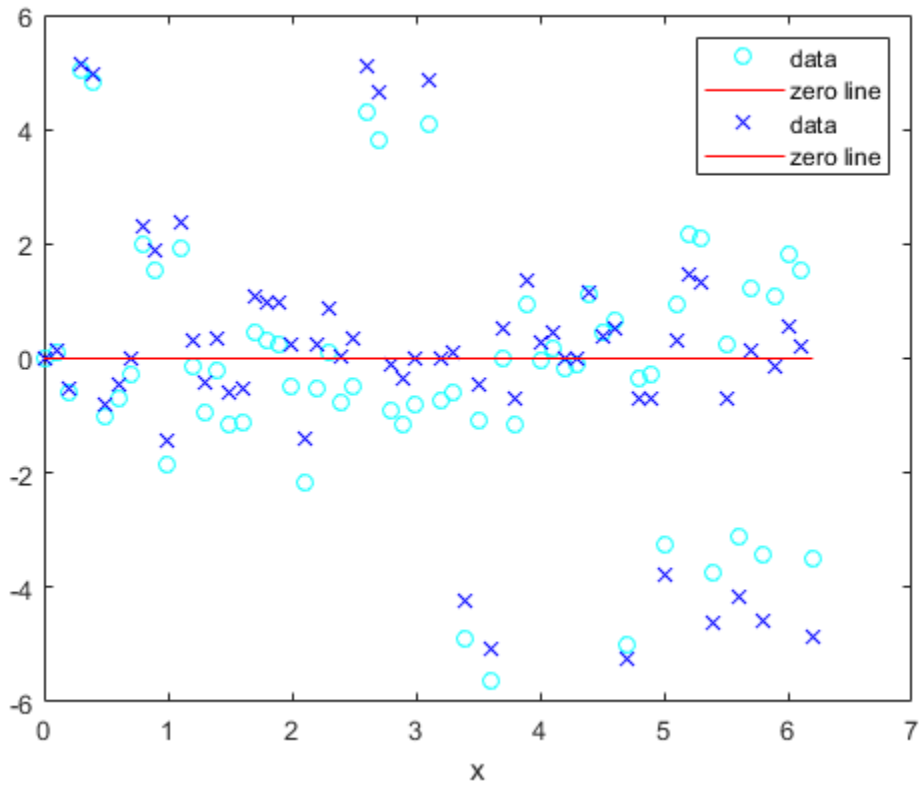
Load data and fit a Gaussian, excluding some data with an expression, then plot the fit, data and the excluded points.

```
[x, y] = titanium;
f1 = fit(x',y','gauss2', 'Exclude', x<800);
plot(f1,x,y,x<800)
```
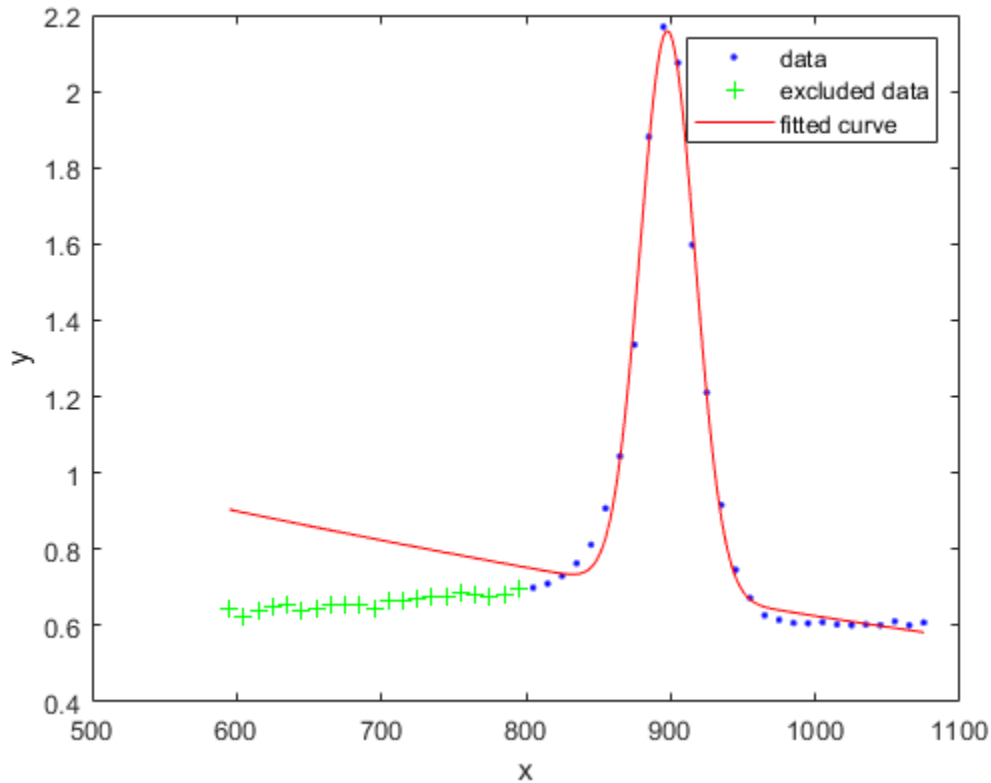
Input Arguments

sfit — Surface fit object
sfit

Fit object to plot, specified as an `sfit` object.

x — Data to plot
matrix

Data to plot, specified as a matrix with either one (curve fitting) or two (surface fitting) columns.

y — Data to plot
matrix

Data to plot, specified as a matrix with either one (curve fitting) or two (surface fitting) columns.

z — Data to plot
matrix

Data to plot, specified as a matrix with either one (curve fitting) or two (surface fitting) columns.

cfit — Curve fit object
cfit

Fit object to plot, specified as a `cfit` object.

outliers — Outlier data
expression | index vector | logical vector

- An expression describing a logical vector, e.g., `x > 10`.
- A vector of integers indexing the points you want to exclude, e.g., `[1 10 25]`.
- A logical vector for all data points where `true` represents an outlier, created by `excludedata`.

For an example, see "Exclude Points from Fit" on page 13-85.

Data Types: `logical` | `double`

DataLineSpec — Line style, marker, and color for *x* and *y*
character vector | string

Line style, marker, and color used to plot the predictor data **x** and response data **y**, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `'--or'` is a red dashed line with circle markers

Line Style	Description
-	Solid line (default)
--	Dashed line
:	Dotted line
-.	Dash-dot line

Marker	Description
o	Circle
+	Plus sign
*	Asterisk
.	Point
x	Cross
s	Square
d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
p	Pentagram
h	Hexagram

Color	Description
y	yellow

Color	Description
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

FitLineSpec — Line style, marker, and color for the function
character vector | string

Line style, marker, and color used to plot the **cfit** function, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `'--or'` is a red dashed line with circle markers

Line Style	Description
-	Solid line (default)
--	Dashed line
:	Dotted line
-.	Dash-dot line

Marker	Description
o	Circle
+	Plus sign
*	Asterisk
.	Point
x	Cross
s	Square
d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
p	Pentagram
h	Hexagram

Color	Description
y	yellow

Color	Description
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

OutlierLineSpec — Line style, marker, and color for the outliers
character vector | string

Line style, marker, and color used to plot the **outliers**, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `'--or'` is a red dashed line with circle markers

Line Style	Description
-	Solid line (default)
--	Dashed line
:	Dotted line
-.	Dash-dot line

Marker	Description
o	Circle
+	Plus sign
*	Asterisk
.	Point
x	Cross
s	Square
d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
p	Pentagram
h	Hexagram

Color	Description
y	yellow

Color	Description
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

ptype — Plot type
`'fit'` | `'predfunc'` | `'predobs'` | `'residuals'` | `'stresiduals'` | `'deriv1'` | `'deriv2'` | `'integral'`

Plot type, specified as one of these supported types:

- `'fit'` — Data and fit (default)
- `'predfunc'` — Data and fit with prediction bounds for the fit
- `'predobs'` — Data and fit with prediction bounds for new observations
- `'residuals'` — Residuals
- `'stresiduals'` — Standardized residuals (residuals divided by their standard deviation)
- `'deriv1'` — First derivative of the fit
- `'deriv2'` — Second derivative of the fit
- `'integral'` — Integral of the fit

level — Confidence level
`0.95` (default) | positive scalar in the range (0,1)

Level of confidence of the prediction intervals, specified as a scalar between `0` and `1`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Exclude',[1 10 25],'Level',0.95`

Exclude — Plot excluded data in different color
expression | index vector | logical vector | empty

Excluded data points to plot in a different color, specified as the comma-separated pair consisting of `'Exclude'` and one of:

- An expression describing a logical vector, e.g., `x > 10`.
- A vector of integers indexing the points you want to exclude, e.g., `[1 10 25]`.
- A logical vector for all data points where `true` represents an outlier, created by `excludedata`.

For an example, see "Exclude Points from Fit" on page 13-85.

Data Types: `logical` | `double`

Style — Option to plot the surface fit object `sfit`
`'Surface'` (default) | `'PredFunc'` | `'PredObs'` | `'Residuals'` | `'Contour'`

Way to plot the surface fit object `sfit`, specified as the comma-separated pair consisting of `'Style'` and one of:

- `'Surface'` Plot the fit object as a surface (default)
- `'PredFunc'` Surface with prediction bounds for function
- `'PredObs'` Surface with prediction bounds for new observation
- `'Residuals'` Plot the residuals (fit is the plane Z=0)
- `'Contour'` Make a contour plot of the surface

Level — Confidence level
scalar

Confidence level used in the plot, specified as the comma-separated pair consisting of `'Level'` and a positive scalar less than 1. The default value is `0.95`, for 95% confidence. This option only applies to the `'PredFunc'` and `'PredObs'` plot styles.

XLim — *x*-axis limit
scalar | vector

Limits of the *x*-axis used for the plot, specified as the comma-separated pair consisting of `'XLim'` and a scalar or vector. By default the axes limits are taken from the data, XY. If no data is given, then the limits are taken from the surface fit object, `sfit`.

YLim — *y*-axis limit
scalar | vector

Limits of the *y*-axis used for the plot, specified as the comma-separated pair consisting of `'YLim'` and a scalar or vector. By default the axes limits are taken from the data, XY. If no data is given, then the limits are taken from the surface fit object, `sfit`.

Parent — Handle of axes
Axes object

Handle of the axes, specified as the comma-separated pair consisting of `'Parent'` and a value.

Output Arguments

H — Vector of handles
object array

Vector of handles to the plotted objects, returned as an object array.

See Also

Apps
Curve Fitting

Functions
differentiate | excludedata | fit | integrate

Topics
About Splines in Curve Fitting on page 8-2

Introduced before R2006a

ppmak

Put together spline in ppform

Syntax

```
ppmak(breaks,coefs)
ppmak
ppmak(breaks,coefs,d)
ppmak(breaks,coefs,sizec)
```

Description

The command `ppmak(...)` puts together a spline in ppform from minimal information, with the rest inferred from that information. `fnbrk` provides any or all of the parts of the completed description. In this way, the actual data structure used for the storage of the ppform is easily modified without any effect on the various `fn...` commands that use this construct. However, the casual user is not likely to use `ppmak` explicitly, relying instead on the various spline construction commands in the toolbox to construct particular splines.

`ppmak(breaks,coefs)` returns the ppform of the spline specified by the break information in `breaks` and the coefficient information in `coefs`. How that information is interpreted depends on whether the function is univariate or multivariate, as indicated by `breaks` being a sequence or a cell array.

If `breaks` is a sequence, it must be nondecreasing, with its first entry different from its last. Then the function is assumed to be univariate, and the various parts of its ppform are determined as follows:

1 The number `l` of polynomial pieces is computed as `length(breaks)-1`, and the basic interval is, correspondingly, the interval `[breaks(1) .. breaks(l+1)]`.

2 The dimension `d` of the function's target is taken to be the number of rows in `coefs`. In other words, each column of `coefs` is taken to be one coefficient. More explicitly, `coefs(:,i*k+j)` is assumed to contain the `j`th coefficient of the `(i+1)`st polynomial piece (with the first coefficient the highest and the `k`th coefficient the lowest, or constant, coefficient). Thus, with `kl` the number of columns of `coefs`, the order `k` of the piecewise-polynomial is computed as `fix(kl/l)`.

After that, the entries of `coefs` are reordered, by the command

```
 coefs = reshape(permute(reshape(coefs,[d,k,l]),[1 3 2]),[d*l,k])
```

in order to conform with the internal interpretation of the coefficient array in the ppform for a univariate spline. This only applies when you use the syntax `ppmak(breaks,coefs)` where `breaks` is a sequence (row vector), not when it is a cell-array. The permutation is not made when you use the three-argument forms of `ppmak`. For the three-argument forms only a reshape is done, not a permute.

If `breaks` is a cell array, of length `m`, then the function is assumed to be `m`-variate (tensor product), and the various parts of its ppform are determined from the input as follows:

1 The `m`-vector `l` has `length(breaks{i})-1` as its `i`th entry and, correspondingly, the `m`-cell array of its basic intervals has the interval `[breaks{i}(1) .. breaks{i}(end)]` as its `i`th entry.

2 The dimension d of the function's target and the m-vector k of (coordinate-wise polynomial) orders of its pieces are obtained directly from the size of coefs, as follows.

    a If coefs is an m-dimensional array, then the function is taken to be scalar-valued, i.e., d is 1, and the m-vector k is computed as size(coefs)./l. After that, coefs is reshaped by the command coefs = reshape(coefs,[1,size(coefs)]).

    b If coefs is an (r+m)-dimensional array, with sizec = size(c) say, then d is set to sizec(1:r), and the vector k is computed as sizec(r+(1:m))./l. After that, coefs is reshaped by the command coefs = reshape(coefs,[prod(d),sizec(r+(1:m))]).

Then, coefs is interpreted as an equivalent array of size [d,l(1),k(1),l(2),k(2),...,l(m),k(m)], with its (:,i(1),r(1),i(2),r(2),...,i(m),r(m))th entry the coefficient of

$$\prod_{\mu=1}^{m} (x(\mu) - \text{breaks}|\mu](i(\mu)))^{(k(\mu) - r(\mu))}$$

in the local polynomial representation of the function on the (hyper)rectangle with sides

$$[\text{breaks}|\mu](i(\mu)) .. \text{breaks}|\mu](i(\mu) + 1)], \quad \mu = 1{:}m$$

This is, in fact, the internal interpretation of the coefficient array in the ppform of a multivariate spline.

ppmak prompts you for breaks and coefs.

ppmak(breaks,coefs,d) with d a positive integer, also puts together the ppform of a spline from the information supplied, but expects the function to be univariate. In that case, coefs is taken to be of size [d*l,k], with l obtained as length(breaks)-1, and this determines the order, k, of the spline. With this, coefs(i*d+j,:) is taken to be the jth components of the coefficient vector for the (i+1)st polynomial piece.

ppmak(breaks,coefs,sizec) with sizec a row vector of positive integers, also puts together the ppform of a spline from the information supplied, but interprets coefs to be of size sizec (and returns an error when prod(size(coefs)) differs from prod(sizec)). This option is important only in the rare case that the input argument coefs is an array with one or more trailing singleton dimensions. For, MATLAB suppresses trailing singleton dimensions, hence, without this explicit specification of the intended size of coefs, ppmak would interpret coefs incorrectly.

Examples

The two splines

```
p1 = ppmak([1 3 4],[1 2 5 6;3 4 7 8]);
p2 = ppmak([1 3 4],[1 2;3 4;5 6;7 8],2);
```

have exactly the same ppform (2-vector-valued, of order 2). But the second command provides the coefficients in the arrangement used internally.

ppmak([0:2],[1:6]) constructs a piecewise-polynomial function with basic interval [0..2] and consisting of two pieces of order 3, with the sole interior break 1. The resulting function is scalar, i.e., the dimension d of its target is 1. The function happens to be continuous at that break since the first piece is $x| \to x^2 + 2x + 3$, while the second piece is $x| \to 4(x-1)^2 + 5(x-1) + 6$.

When the function is univariate and the dimension d is not explicitly specified, then it is taken to be the row number of `coefs`. The column number should be an integer multiple of the number l of pieces specified by `breaks`. For example, the statement `ppmak([0:2],[1:3;4:6])` leads to an error, since the break sequence `[0:2]` indicates two polynomial pieces, hence an even number of columns are expected in the coefficient matrix. The modified statement `ppmak([0:1],[1:3;4:6])` specifies the parabolic curve $x|{\rightarrow}(1,4)x^2 + (2,5)x + (3,6)$. In particular, the dimension d of its target is 2. The differently modified statement `ppmak([0:2],[1:4;5:8])` also specifies a planar curve (i.e., d is 2), but this one is piecewise linear; its first polynomial piece is $x|{\rightarrow}(1,5)x + (2,6)$.

Explicit specification of the dimension d leads, in the univariate case, to a different interpretation of the entries of `coefs`. Now the column number indicates the polynomial order of the pieces, and the row number should equal d times the number of pieces. Thus, the statement `ppmak([0:2], [1:4;5:8],2)` is in error, while the statement `ppmak([0:2],[1:4;5:8],1)` specifies a scalar piecewise cubic whose first piece is $x|{\rightarrow}x^3 + 2x^2 + 3x + 4$.

If you wanted to make up the constant polynomial, with basic interval [0..1] say, whose value is the matrix eye(2), then you would have to use the full optional third argument, i.e., use the command

```
pp = ppmak(0:1,eye(2),[2,2,1,1]);
```

Finally, if you want to construct a 2-vector-valued bivariate polynomial on the rectangle [–1 .. 1] x [0 .. 1], linear in the first variable and constant in the second, say

```
coefs = zeros(2,2,1); coefs(:,:,1) = [1 0; 0 1];
```

then the straightforward

```
pp = ppmak({[-1 1],[0 1]},coefs);
```

will fail, producing a scalar-valued function of order 2 in each variable, as will

```
pp = ppmak({[-1 1],[0 1]},coefs,size(coefs));
```

while the following command will succeed:

```
pp = ppmak({[-1 1],[0 1]},coefs,[2 2 1]);
```

See the example "Intro to ppform" for other examples.

See Also
fnbrk

predint

Prediction intervals for `cfit` or `sfit` object

Syntax

```
ci = predint(fitresult,x)
ci = predint(fitresult,x,level)
ci = predint(fitresult,x,level,intopt,simopt)
[ci,y] = predint(...)
```

Description

`ci = predint(fitresult,x)` returns upper and lower 95% prediction bounds for response values associated with the `cfit` object `fitresult` at the new predictor values specified by the vector `x`. `fitresult` must be an output from the `fit` function to contain the necessary information for `ci`. `ci` is an n-by-2 array where `n = length(x)`. The left column of `ci` contains the lower bound for each coefficient; the right column contains the upper bound.

`ci = predint(fitresult,x,level)` returns prediction bounds with a confidence level specified by `level`. `level` must be between `0` and `1`. The default value of `level` is `0.95`.

`ci = predint(fitresult,x,level,intopt,simopt)` specifies the type of bounds to compute.

Observation bounds are wider than functional bounds because they measure the uncertainty of predicting the fitted curve plus the random variation in the new observation. Non-simultaneous bounds are for individual elements of `x`; simultaneous bounds are for all elements of `x`.

`[ci,y] = predint(...)` returns the response values `y` predicted by `fitresult` at the predictors in `x`.

Note `predint` cannot compute prediction intervals for non-parametric regression methods such as `Interpolant`, `Lowess`, and `Spline`.

Examples

Compute Prediction Intervals

Compute and plot observation and functional prediction intervals for a fit to noisy data.

Generate noisy data with an exponential trend.

```
x = (0:0.2:5)';
y = 2*exp(-0.2*x) + 0.5*randn(size(x));
```

Fit a curve to the data using a single-term exponential.

```
fitresult = fit(x,y,'exp1');
```

Compute 95% observation and functional prediction intervals, both simultaneous and nonsimultaneous. Non-simultaneous bounds are for individual elements of x; simultaneous bounds are for all elements of x.

```
p11 = predint(fitresult,x,0.95,'observation','off');
p12 = predint(fitresult,x,0.95,'observation','on');
p21 = predint(fitresult,x,0.95,'functional','off');
p22 = predint(fitresult,x,0.95,'functional','on');
```

Plot the data, fit, and prediction intervals. Observation bounds are wider than functional bounds because they measure the uncertainty of predicting the fitted curve plus the random variation in the new observation.

```
subplot(2,2,1)
plot(fitresult,x,y), hold on, plot(x,p11,'m--'), xlim([0 5]), ylim([-1 5])
title('Nonsimultaneous Observation Bounds','FontSize',9)
legend off

subplot(2,2,2)
plot(fitresult,x,y), hold on, plot(x,p12,'m--'), xlim([0 5]), ylim([-1 5])
title('Simultaneous Observation Bounds','FontSize',9)
legend off

subplot(2,2,3)
plot(fitresult,x,y), hold on, plot(x,p21,'m--'), xlim([0 5]), ylim([-1 5])
title('Nonsimultaneous Functional Bounds','FontSize',9)
legend off

subplot(2,2,4)
plot(fitresult,x,y), hold on, plot(x,p22,'m--'), xlim([0 5]), ylim([-1 5])
title('Simultaneous Functional Bounds','FontSize',9)
legend({'Data','Fitted curve', 'Prediction intervals'},...
       'FontSize',8,'Location','northeast')
```

Nonsimultaneous Observation Bounds · Simultaneous Observation Bounds · Nonsimultaneous Functional Bounds · Simultaneous Functional Bounds · Data · Fitted curve · Prediction intervals

Input Arguments

fitresult — cfit or sfit function
cfit | sfit

Function of which you want to find the prediction intervals, specified as a `cfit` or `sfit` object.

`fitresult` must be an output from the `fit` function to contain the necessary information for `ci`.

x — Predictor values
vector

Predictor values used to compute the upper and lower prediction bounds of `fitresult`, specified as a vector.

level — Confidence level
0.95 (default) | positive scalar in the range (0,1)

Confidence level of the prediction bounds, specified as a positive scalar between `0` and `1`.

intopt — Type of bounds
'observation' (default) | 'functional'

Type of bounds, specified as either:

- `'observation'` — Bounds for a new observation (default)
- `'functional'` — Bounds for the fitted curve

simopt — Type of bounds
`'off'` (default) | `'on'`

Type of bounds, specified as either:

- `'off'` — Non-simultaneous bounds (default)
- `'on'` — Simultaneous bounds

Output Arguments

ci — Prediction bounds
array

Upper and lower prediction bounds, returned as an array of size n-by-2, where `n = length(x)`.

y — Response values
vector

Response values predicted by `fitresult` at the predictors in x, returned as a vector.

See Also
`confint` | `fit` | `plot`

Topics
About Splines in Curve Fitting on page 8-2

Introduced in R2013a

prepareCurveData

Prepare data inputs for curve fitting

Syntax

```
[XOut,YOut] = prepareCurveData(XIn,YIn)
[XOut,YOut,WOut] = prepareCurveData(XIn,YIn,WIn)
```

Description

`[XOut,YOut] = prepareCurveData(XIn,YIn)` transforms data, if necessary, for curve fitting with the `fit` function. The `prepareCurveData` function transforms data as follows:

- Return data as columns regardless of the input shapes. Error if the number of elements do not match. Warn if the number of elements match, but the sizes differ.
- Convert complex to real (remove imaginary parts) and warn of this conversion.
- Remove `NaN` or `Inf` from data and warn of this removal.
- Convert nondouble to double and warn of this conversion.

Specify `XIn` as empty if you want to fit curves to *y* data against the index. If `XIn` is empty, then `XOut` is a vector of indices into `YOut`. The `fit` function can use the vector `XOut` for the *x* data when there is only *y* data.

`[XOut,YOut,WOut] = prepareCurveData(XIn,YIn,WIn)` transforms data including weights (`WIn`) for curve fitting with the `fit` function.

When you generate code from the Curve Fitting app, the generated code includes a call to `prepareCurveData` (or `prepareSurfaceData` for surface fits). You can call the generated file from the command line with your original data or new data as input arguments, to recreate your fits and plots. If you call the generated file with new data, the `prepareCurveData` function ensures you can use any data that you can fit in the Curve Fitting app, by reshaping if necessary to column doubles and removing `NaN`s, `Inf`s, or the imaginary parts of complex numbers.

Examples

Reshape Rows to Columns for Curve Fitting

The following commands load the example `titanium` data in which x and y are row vectors. Attempting to use rows as inputs to the `fit` function produces an error. The `prepareCurveData` function reshapes x and y to columns for use with the `fit` function.

```
[x,y] = titanium();
[x,y] = prepareCurveData(x,y);
```

Create and plot a fit using the reshaped data.

```
f = fit(x,y,'smoothingspline');
plot(f,x,y)
```
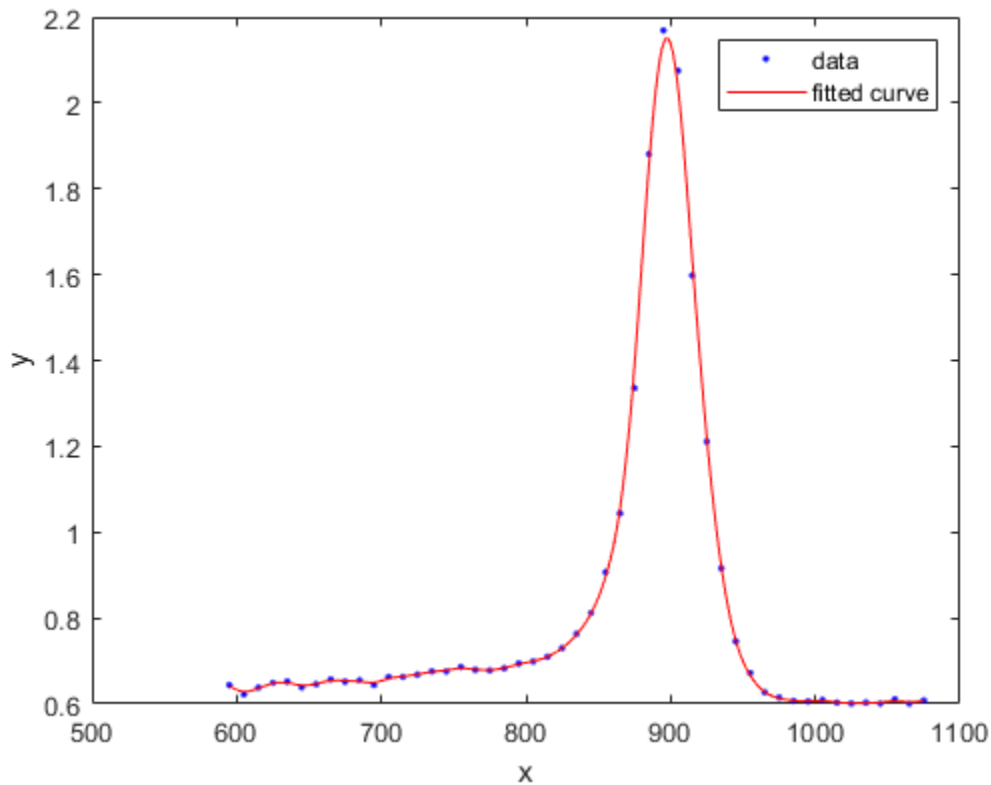
Input Arguments

XIn — X data variable for curve fitting
vector | matrix

X data variable for curve fitting, specified as a scalar, a vector or a matrix. XIn can be empty. Specify empty ([ ]) when you want to fit curves to *y* data against index (x=1:length(y)). See YOut.

Data Types: double

YIn — Y data variable for curve fitting
vector | matrix

Y data variable for curve fitting, specified as a scalar, a vector or a matrix.

Data Types: double

WIn — Weights for curve fitting
vector | matrix

Weights variable for curve fitting specified, as a scalar, a vector or a matrix.

Data Types: double

Output Arguments

XOut — X data column variable for curve fitting
vector

X data column variable prepared for curve fitting, returned as a vector.

If `XIn` is empty, then `XOut` is a vector of indices into `YOut`. The `fit` function can use the vector `XOut` for the *x* data when there is only *y* data.

Data Types: `double`

YOut — Y data column variable for curve fitting
vector

Y data column variable prepared for curve fitting, returned as a vector.

Data Types: `double`

WOut — Weights column variable for curve fitting
vector

Weights column variable prepared for curve fitting, returned as a vector.

Data Types: `double`

See Also
`excludedata` | `fit` | `prepareSurfaceData`

Introduced in R2013a

prepareSurfaceData

Prepare data inputs for surface fitting

Syntax

```
[XOut, YOut, ZOut] = prepareSurfaceData(XIn, YIn, ZIn)
[XOut, YOut, ZOut, WOut] = prepareSurfaceData(XIn, YIn, ZIn, WIn)
```

Description

`[XOut, YOut, ZOut] = prepareSurfaceData(XIn, YIn, ZIn)` transforms data, if necessary, for surface fitting with the `fit` function. The function transforms data as follows:

- For grid vectors, transform row (`YIn`) and column (`XIn`) headers into arrays `YOut` and `XOut` that are the same size as `ZIn`. Warn if `XIn` and `YIn` are reversed.
- Return data as columns regardless of the input shapes. Error if the number of elements do not match. Warn if the number of elements match, but the sizes are different.
- Convert complex to real (remove imaginary parts) and warn of this conversion.
- Remove `NaN` or `Inf` from data and warn of this removal.
- Convert nondouble to double and warn of this conversion.

`[XOut, YOut, ZOut, WOut] = prepareSurfaceData(XIn, YIn, ZIn, WIn)` transforms data including weights (`WIn`) for surface fitting with the `fit` function.

Use `prepareSurfaceData` if your data is not in column vector form. For example, you have 3 matrices. You can also use `prepareSurfaceData` if you have grid vectors, where `length(XIn) = n`, `length(YIn) = m` and `size(ZIn) = [m,n]`. You must process grid vector data for use with the `fit` function by using `prepareSurfaceData`. If you use Curve Fitting app, you can select grid vector data and it automatically converts the data for you.

If your data is in a MATLAB table, you do not need to use `prepareSurfaceData`. You can specify variables in a MATLAB table as inputs to the `fit` function using the form `tablename.varname`.

When you generate code from Curve Fitting app, the generated code includes a call to `prepareSurfaceData` (or `prepareCurveData` for curve fits). You can call the generated file from the command line with your original data or new data as input arguments, to recreate your fits and plots. If you call the generated file with new data, the `prepareCurveData` function ensures you can use any data that you can fit in Curve Fitting app, by reshaping if necessary and removing `NaNs`, `Infs`, or the imaginary parts of complex numbers.

Examples

### Prepare surface data for the fit function

Create some data that is unsuitable for the fit function without preprocessing, because it is nondouble, noncolumn, and contains some `Nan` and `Inf` values.

```
x = int32(1:4);
y = int32(1:5);
z = rand(5,4);
z(13) = Inf;
z(3) = NaN;
```

Use the `prepareSurfaceData` to convert rows to columns, nondoubles to doubles, and remove `Nan` and `Inf`.

```
[xo,yo,zo] = prepareSurfaceData(x,y,z);
```

The function displays the same warnings that you see if you select this data in the Curve Fitting app. The warnings tell you how your data is processed to be suitable for the `fit` function.

Use `whos` to check that the `prepareSurfaceData` converted the variables to column vectors that are doubles.

```
whos xo yo zo
```

Input Arguments

XIn — X data variable to prepare for surface fitting
vector | matrix

X data variable to prepare for surface fitting, specified as a scalar, a vector or a matrix.

Data Types: `double`

YIn — Y data variable to prepare for surface fitting
vector | matrix

Y data variable to prepare for surface fitting, specified as a scalar, a vector or a matrix.

Data Types: `double`

ZIn — Z data variable to prepare for surface fitting
vector | matrix

Z data variable to prepare for surface fitting, specified as a scalar, a vector or a matrix.

Data Types: `double`

WIn — Weights variable
scalar | vector | matrix

Weights variable to prepare for surface fitting, specified as a scalar, a vector or a matrix.

Data Types: `double`

Output Arguments

XOut — X data column variable prepared for surface fitting
vector

X data column variable prepared for surface fitting, returned as a vector.

Data Types: `double`

13-215

YOut — Y data column variable prepared for surface fitting
vector

Y data column variable prepared for surface fitting, returned as a vector.

Data Types: `double`

ZOut — Z data column variable prepared for surface fitting
vector

Z data column variable prepared for surface fitting, returned as a vector.

Data Types: `double`

WOut — Weights column variable
vector

Weights column variable prepared for surface fitting, returned as a vector.

Data Types: `double`

See Also
`excludedata` | `fit` | `prepareCurveData`

Topics
"Selecting Compatible Size Surface Data" on page 2-9

Introduced in R2010b

probnames

Problem-dependent parameter names of `cfit`, `sfit`, or `fittype` object

Syntax

```
pnames = probnames(fun)
```

Description

`pnames = probnames(fun)` returns the names of the problem-dependent (fixed) parameters of the `cfit`, `sfit`, or `fittype` object `fun` as a cell array of character vectors.

Examples

```
f = fittype('(x-a)^n + b','problem',{'a','b'});
coeffnames(f)
ans =
    'n'
probnames(f)
ans =
    'a'
    'b'

load census

c = fit(cdate,pop,f,'problem',{cdate(1),pop(1)},...
        'StartPoint',2);
coeffvalues(c)
ans =
    0.9877
probvalues(c)
ans =
  1.0e+003 *
    1.7900    0.0039
```

See Also
coeffnames | fittype | probvalues

Introduced in R2006b

probvalues

Problem-dependent parameter values of `cfit` or `sfit` object

Syntax

```
pvals = probvalues(fun)
```

Description

`pvals = probvalues(fun)` returns the values of the problem-dependent (fixed) parameters of the `cfit` object `fun` as a row vector.

Examples

```
f = fittype('(x-a)^n + b','problem',{'a','b'});
coeffnames(f)
ans =
    'n'
probnames(f)
ans =
    'a'
    'b'

load census

c = fit(cdate,pop,f,'problem',{cdate(1),pop(1)},...
        'StartPoint',2);
coeffvalues(c)
ans =
    0.9877
probvalues(c)
ans =
  1.0e+003 *
    1.7900    0.0039
```

See Also
`fit` | `fittype` | `probnames`

Introduced in R2006b

quad2d

Numerically integrate `sfit` object

Syntax

```
Q = quad2d(FO, a, b, c, d)
[Q,ERRBND] = quad2d(...)
[Q,ERRBND] = QUAD2D(FO,a,b,c,d,PARAM1,VAL1,PARAM2,VAL2,...)
```

Description

`Q = quad2d(FO, a, b, c, d)` approximates the integral of the surface fit object `FO` over the planar region $a \leq x \leq b$ and $c(x) \leq y \leq d(x)$. The bounds `c` and `d` can each be a scalar, a function handle, or a curve fit (`cfit`) object.

`[Q,ERRBND] = quad2d(...)` also returns an approximate upper bound on the absolute error, ERRBND.

`[Q,ERRBND] = QUAD2D(FO,a,b,c,d,PARAM1,VAL1,PARAM2,VAL2,...)` performs the integration with specified values of optional parameters.

See the MATLAB function `quad2d` for details of the upper bound and the optional parameters.

See Also
`cfit` | `fit` | `quad2d` | `sfit`

Introduced in R2009b

rpmak

Put together rational spline

Syntax

```
rp = rpmak(breaks,coefs)
rp = rpmak(breaks,coefs,d)
rpmak(breaks,coefs,sizec)
rs = rsmak(knots,coefs)
rs = rsmak(shape,parameters)
```

Description

Both `rpmak` and `rsmak` put together a rational spline from minimal information. `rsmak` is also equipped to provide rational splines that describe standard geometric shapes. A rational spline must be scalar- or vector-valued.

`rp = rpmak(breaks,coefs)` has the same effect as the command `ppmak(breaks, coefs)` except that the resulting ppform is tagged as a rational spline, i.e., as a rpform.

To describe what this means, let R be the piecewise-polynomial put together by the command `ppmak(breaks,coefs)`, and let $r(x) = s(x)/w(x)$ be the rational spline put together by the command `rpmak(breaks,coefs)`. If `v` is the value of R at x, then `v(1:end-1)/v(end)` is the value of r at x. In other words, $R(x) = [s(x);w(x)]$. Correspondingly, the dimension of the target of r is one less than the dimension of the target of R. In particular, the dimension (of the target) of R must be at least 2, i.e., the coefficients specified by `coefs` must be d-vectors with `d > 1`. See `ppmak` for how the input arrays `breaks` and `coefs` are being interpreted, hence how they are to be specified in order to produce a particular piecewise-polynomial.

`rp = rpmak(breaks,coefs,d)` has the same effect as `ppmak(breaks,coefs,d+1)`, except that the resulting ppform is tagged as being a rpform. Note that the desire to have that optional third argument specify the dimension of the target requires different values for it in `rpmak` and `ppmak` for the same coefficient array `coefs`.

`rpmak(breaks,coefs,sizec)` has the same effect as `ppmak(breaks,coefs,sizec)` except that the resulting ppform is tagged as being a rpform, and the target dimension is taken to be `sizec(1)-1`.

`rs = rsmak(knots,coefs)` is similarly related to `spmak(knots,coefs)`, and `rsmak(knots,coefs,sizec)` to `spmak(knots,coefs,sizec)`. In particular, `rsmak(knots,coefs)` puts together a rational spline in B-form, i.e., it provides a rBform. See `spmak` for how the input arrays `knots` and `coefs` are being interpreted, hence how they are to be specified in order to produce a particular piecewise-polynomial.

`rs = rsmak(shape,parameters)` provides a rational spline in rBform that describes the shape being specified by the character vector `shape` and the optional additional `parameters`. Specific choices are:

```
rsmak('arc',radius,center,[alpha,beta])
rsmak('circle',radius,center)
```

```
rsmak('cone',radius,halfheight)
rsmak('cylinder',radius,height)
rsmak('southcap',radius,center)
rsmak('torus',radius,ratio)
```

with 1 the default value for `radius`, `halfheight` and `height`, and the origin the default for `center`, and the arc running through all the angles from `alpha` to `beta` (default is `[-pi/2,pi/2]`), and the cone, cylinder, and torus centered at the origin with their major circle in the (x,y)-plane, and the minor circle of the torus having radius `radius*ratio`, the default for `ratio` being 1/3.

From these, one may generate related shapes by affine transformations, with the help of `fncmb(rs,transformation)`.

All `fn...` commands except `fnint`, `fnder`, `fndir` can handle rational splines.

Examples

The commands

```
runges = rsmak([-5 -5 -5 5 5 5],[1 1 1; 26 -24 26]);
rungep = rpmak([-5 5],[0 0 1; 1 -10 26],1);
```

both provide a description of the rational polynomial $r(x) = 1/(x^2 + 1)$ on the interval [-5 .. 5]. However, outside the interval [-5 .. 5], the function given by `runges` is zero, while the rational spline given by `rungep` agrees with $1/(x^2 + 1)$ for every x.

The figure of a rotated cone is generated by the commands

```
fnplt(fncmb(rsmak('cone',1,2),[0 0 -1;0 1 0;1 0 0]))
axis equal, axis off, shading interp
```
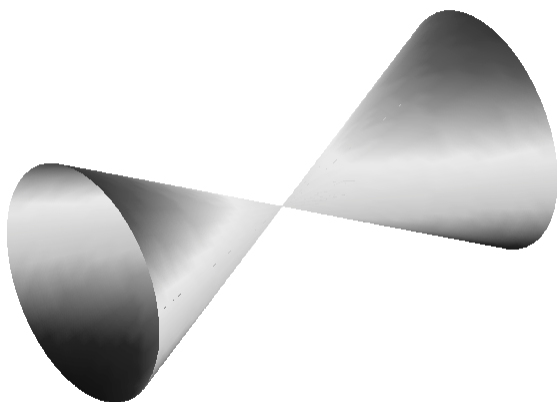

A Rotated Cone Given by a Rational Quadratic Spline

"A Helix" on page 13-222, showing a helix with several windings, is generated by the commands

```
arc = rsmak('arc',2,[1;-1],[0 7.3*pi]);
[knots,c] = fnbrk(arc,'k','c');
helix = rsmak(knots, [c(1:2,:);aveknt(knots,3).*c(3,:);
c(3,:)]);
fnplt(helix)
```
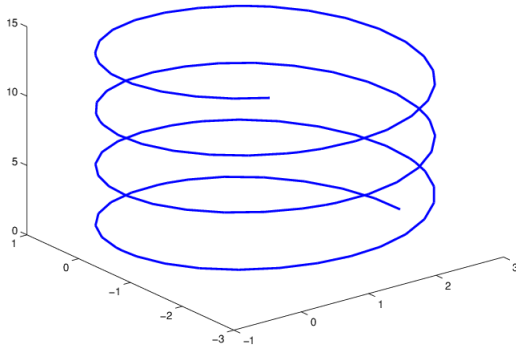

A Helix

For further illustrated examples, see "NURBS and Other Rational Splines" on page 10-23

See Also
`fnbrk` | `ppmak` | `rsmak` | `spmak`

rscvn

Piecewise biarc Hermite interpolation

Syntax

```
c = rscvn(p,u)
c = rscvn(p)
```

Description

`c = rscvn(p,u)` returns a planar piecewise biarc curve (in quadratic rBform) that passes, in order, through the given points `p(:,j)` and is constructed in the following way (see "Construction of a Biarc" on page 13-224). Between any two distinct points `p(:,j)` and `p(:,j+1)`, the curve usually consists of two circular arcs (including straight-line segments) which join with tangent continuity, with the first arc starting at `p(:,j)` and normal there to `u(:,j)`, and the second arc ending at `p(:,j+1)` and normal there to `u(:,j+1)`, and with the two arcs written as one whenever that is possible. Thus the curve is tangent-continuous everywhere except, perhaps, at repeated points, where the curve may have a corner, or when the angle, formed by the two segments ending at `p(:,j)`, is unusually small, in which case the curve may have a cusp at that point.

`p` must be a real matrix, with two rows, and at least two columns, and any column must be different from at least one of its neighboring columns.

`u` must be a real matrix with two rows, with the same number of columns as `p` (for two exceptions, see below), and can have no zero column.

`c = rscvn(p)` chooses the normals in the following way. For `j=2:end-1`, `u(:,j)` is the average of the (normalized, right-turning) normals to the vectors `p(:,j)-p(:,j-1)` and `p(:,j+1)-p(:,j)`. If `p(:,1)==p(:,end)`, then both end normals are chosen as the average of the normals to `p(:,2)-p(:,1)` and `p(:,end)-p(:,end-1)` thus preventing a corner in the resulting closed curve. Otherwise, the end normals are so chosen that there is only one arc over the first and last segment (not-a-knot end condition).

`rscvn(p,u)`, with `u` having exactly two columns, also chooses the interior normals as for the case when `u` is absent but uses the two columns of `u` as the end-point normals.

Examples

Example 1. The following code generates a description of a circle, using just four pieces. Except for a different scaling of the knot sequence, it is the same description as is supplied by `rsmak('circle',1,[1;1])`.

```
p = [1 0 -1 0 1; 0 1 0 -1 0]; c = rscvn([p(1,:)+1;p(2,:)+1],p);
```

The same circle, but using just two pieces, is provided by

```
c2 = rscvn([0,2,0; 1,1,1]);
```

Example 2. The following code plots two letters. Note that the second letter is the result of interpolation to just four points. Note also the use of translation in the plotting of the second letter.

```
p = [-1 .8 -1 1 -1 -1 -1; 3 1.75 .5 -1.25 -3 -3  3];
i = eye(2); u = i(:,[2 1 2 1 2 1 1]); B = rscvn(p,u);
S = rscvn([1 -1 1 -1; 2.5 2.5 -2.5 -2.5]);
fnplt(B), hold on, fnplt(fncmb(S,[3;0])), hold off
axis equal, axis off
```
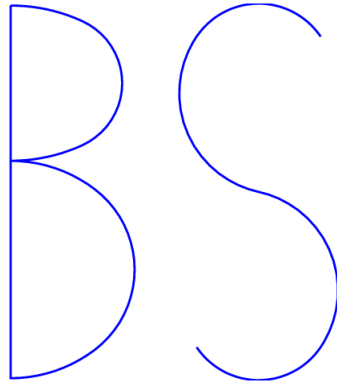

Two Letters Composed of Circular Arcs

Example 3. The following code generates the "Construction of a Biarc" on page 13-224, of use in the discussion below of the biarc construction used here. Note the use of `fntlr` to find the tangent to the biarc at the beginning, at the point where the two arcs join, and at the end.

```
p = [0 1;0 0];   u = [.5 -.1;-.25 .5];
plot(p(1,:),p(2,:),'k'), hold on
biarc = rscvn(p,u); breaks = fnbrk(biarc,'b');
fnplt(biarc,breaks(1:2),'b',3), fnplt(biarc,breaks(2:3),'r',3)
vd = fntlr(biarc,2,breaks);
quiver(vd(1,:),vd(2,:),vd(4,:),-vd(3,:)), hold off
```
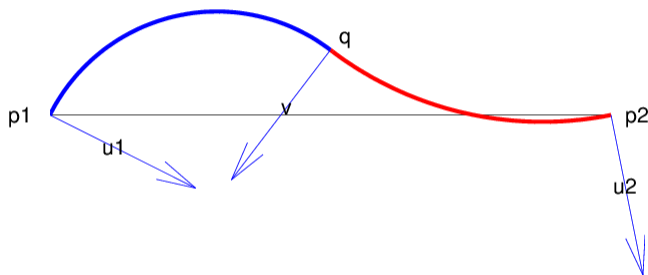

Construction of a Biarc

Algorithms

Given two distinct points, `p1` and `p2`, in the plane and, correspondingly, two nonzero vectors, `u1` and `u2`, there is a one-parameter family of biarcs (i.e., a curve consisting of two arcs with common tangent at their join) starting at `p1` and normal there to `u1` and ending at `p2` and normal there to `u2`. One way to parametrize this family of biarcs is by the normal direction, `v`, at the point `q` at which the two arcs join. With a nonzero `v` chosen, there is then exactly one choice of `q`, hence the entire biarc is then determined. In the construction used in `rscvn`, `v` is chosen as the reflection, across the perpendicular to the segment from `p1` to `p2`, of the average of the vectors `u1` and `u2`, -- after both vectors have been so normalized that their length is 1 and that they both point to the right of the segment from `p1` to `p2`. This choice for `v` seems natural in the two standard cases: (i) `u2` is the reflection of `u1` across the perpendicular to the segment from `p1` to `p2`; (ii) `u1` and `u2` are parallel. This choice of `v` is validated by "Biarcs as a Function of the Left Normal" on page 13-225 which shows the resulting biarcs when `p1`, `p2`, and `u2 = [.809;.588]` are kept fixed and only the normal at `p1` is allowed to vary.
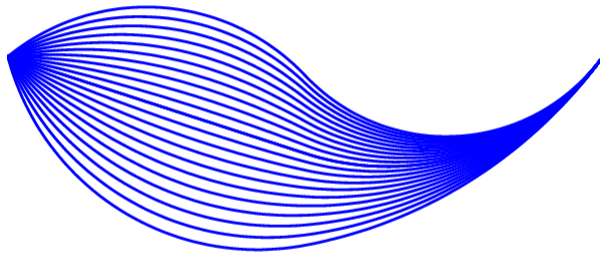
Biarcs as a Function of the Left Normal

But it is impossible to have the interpolating biarc depend continuously at all four data, `p1`, `p2`, `u1`, `u2`. There has to be a discontinuity as the normal directions, `u1` and `u2`, pass through the direction from `p1` to `p2`. This is illustrated in "Biarcs as a Function of One Endpoint" on page 13-226 which shows the biarcs when one point, `p1 = [0;0]`, and the two normals, `u1 = [1;1]` and `u2 = [1;-1]`, are held fixed and only the other point, `p2`, moves, on a circle around `p1`.
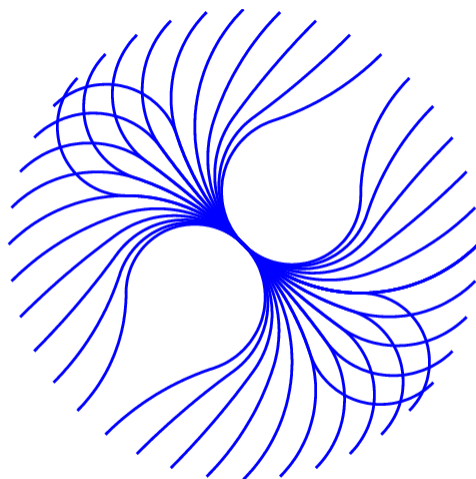
Biarcs as a Function of One Endpoint

See Also

cscvn | rsmak

rsmak

Put together rational spline for standard geometric shapes

Syntax

```
rs = rsmak(shape,parameters)
```

Description

`rs = rsmak(shape,parameters)` provides a rational spline in rBform that describes the shape being specified by the character vector `shape` and the optional additional `parameters`. Specific choices for `shape` are:

```
rsmak('arc',radius,center,[alpha,beta])
rsmak('circle',radius,center)
rsmak('cone',radius,halfheight)
rsmak('cylinder',radius,height)
rsmak('southcap',radius,center)
rsmak('torus',radius,ratio)
```

with `1` the default value for `radius`, `halfheight` and `height`, and the origin the default for `center`, and the arc running through all the angles from `alpha` to `beta` (default is `[-pi/2,pi/2]`), and the cone, cylinder, and torus centered at the origin with their major circle in the (x,y)-plane, and the minor circle of the torus having radius `radius*ratio`, the default for `ratio` being `1/3`.

From these, one may generate related shapes by affine transformations, with the help of `fncmb(rs,transformation)`.

See `rpmak` for more information on other options.

See Also
rpmak

set

Assign values in fit options structure

Syntax

```
set(options)
s = set(options)
set(options,field1,val1,field2,val2,...)
set(options,fields,vals)
```

Description

set(options) displays all property names of the fit options structure options. If a property has a finite list of possible character vector values, these values are also displayed.

s = set(options) returns a structure s with the same property names as options. If a property has a finite list of possible character vector values, the value of the property in s is a cell array containing the possible character vector values. If a property does not have a finite list of possible character vector values, the value of the property in s is an empty cell array.

set(options,field1,val1,field2,val2,...) sets the properties specified by the character vectors *field1*, *field2*, ... to the values val1, val2, ..., respectively.

set(options,fields,vals) sets the properties specified by the cell array of character vectors *fields* to the corresponding values in the cell array vals.

Examples

Set Property Values of a `fitoptions` Structure

Create a custom nonlinear model, and create a default fit options structure for the model:

```
f = fittype('a*x^2+b*exp(n*c*x)','problem','n');
options = fitoptions(f);
```

Set the Robust and Normalize properties of the fit options structure using property name/value pairs:

```
set(options,'Robust','LAR','Normalize','On')
```

Set the Display, Lower, and Algorithm properties of the fit options structure using cell arrays of property names/values:

```
set(options,{'Disp','Low','Alg'},...
        {'Final',[0 0 0],'Levenberg'})
```

Input Arguments

options — fitoptions object
fitoptions (default)

`fitoptions` object of which you want to set the property names and values, specified as a `fitoptions` object.

field1,val1,field2,val2,... — Property field
character vector (default) | scalar | vector | matrix

Property fields, specified as a character vector, of the `fitoptions` object `options` that you want to set to the values `val1`, `val2`, ..., respectively, specified as a scalar, a vector, or a matrix.

Data Types: char

fields — Property fields
cell array of character vector (default)

Property field of the `fitoptions` object `options` that you want to set, specified as a cell array of character vector.

Data Types: cell

vals — Property values
cell array (default)

Values of the `fitoptions` object `options` that you want to set the fields to, specified as a cell array.

Data Types: cell

Output Arguments

s — fitoptions structure
fitoptions

Copy of the `fitoptions` object `options`, returned as a `fitoption` object.

See Also
fitoptions | get

Introduced before R2006a

setoptions

Set model fit options

Syntax

```
FT = setoptions(FT, options)
```

Description

`FT = setoptions(FT, options)` sets the fit options of `FT` to `options`, where `FT` is a `fittype`, `cfit`, or `sfit` object. The `FT` output argument must match the `FT` input argument.

See Also

`fit` | `fitoptions` | `fittype`

Introduced in R2009b

sfit

Constructor for `sfit` object

Syntax

```
surfacefit = sfit(fittype,coeff1,...,coeffn)
```

Description

`surfacefit = sfit(fittype,coeff1,...,coeffn)` constructs the `sfit` object `surfacefit` using the model type specified by the `fittype` object and the coefficient values `coeff1`, `coeff2`, etc.

An `sfit` object encapsulates the result of fitting a surface to data. They are normally constructed by calling the `fit` function, or interactively by exporting a fit from the Curve Fitting app to the workspace. You can get and set coefficient properties of the `sfit` object.

You can treat an `sfit` object as a function to make predictions or evaluate the surface at values of X and Y.

Like the `cfit` class, `sfit` inherits all `fittype` methods.

Note `sfit` is called by the `fit` function when fitting `fittype` objects to data. To create a `sfit` object that is the result of a regression, use `fit`.

You should only call `sfit` directly if you want to assign values to coefficients and problem parameters of a `fittype` object *without* performing a fit.

Methods of `sfit` objects:

argnames	Input argument names of `cfit`, `sfit`, or `fittype` object
category	Category of fit of `cfit`, `sfit`, or `fittype` object
coeffnames	Coefficient names of `cfit`, `sfit`, or `fittype` object
dependnames	Dependent variable of `cfit`, `sfit`, or `fittype` object
formula	Formula of `cfit`, `sfit`, or `fittype` object
indepnames	Independent variable of `cfit`, `sfit`, or `fittype` object
islinear	Determine if `cfit`, `sfit`, or `fittype` object is linear
numargs	Number of input arguments of `cfit`, `sfit`, or `fittype` object
numcoeffs	Number of coefficients of `cfit`, `sfit`, or `fittype` object
probnames	Problem-dependent parameter names of `cfit`, `sfit`, or `fittype` object
probvalues	Problem-dependent parameter values of `cfit` or `sfit` object
quad2d	Numerically integrate `sfit` object
setoptions	Set model fit options
sfit	Constructor for `sfit` object
type	Name of `cfit`, `sfit`, or `fittype` object

Examples

Make predictions or evaluate the surface values of X and Y

You can treat an `sfit` object as a function to make predictions or evaluate the surface at values of X and Y.

```
x = 3 - 6 * rand( 49, 1 );
y = 3 - 6 * rand( 49, 1 );
z = peaks( x, y );
sf = fit( [x, y], z, 'poly32' );
zhat = sf( mean( x ), mean( y ) )
```

Input Arguments

`fittype` — Model type
`fittype` (default)

Model type the `sfit` function uses to construct the `sfit` object, specified as a `fittype` constructed with the `fittype` function.

Example: `fittype('linearinterp')`

`coeff1,...,coeffn` — Coefficient values
scalar (default)

Coefficient values of the `sfit` object, specified as scalars.

Data Types: `single` | `double`

Output Arguments

surfacefit — sfit object
sfit

Function output, returned as an sfit object.

See Also
cfit | feval | fit | fittype

Topics
"Evaluate a Surface Fit" on page 7-25
"Fit Postprocessing"

Introduced in R2009a

sftool

Open Curve Fitting app

Syntax

```
sftool
sftool(x,y,z)
sftool(x,y,z,w)
sftool(filename)
```

Description

Note `sftool` will be removed in a future release. Use `cftool` instead. See **Curve Fitting**.

`sftool` opens Curve Fitting app or brings focus to the tool if it is already open.

`sftool(x,y,z)` creates a fit to x and y inputs (or predictor data) and z output (or response data). `sftool` opens Curve Fitting app if necessary.

x, y, and z must be numeric, have two or more elements, and have compatible sizes. Sizes are compatible if either:

- x, y, and z all have the same number of elements, or
- x and y are vectors, z is a 2D matrix, where `length(x) = n`, `length(y) = m`, and `[m,n] = size(z)`.

`sftool(x,y,z,w)` creates a fit with weights w. w must be numeric and have the same number of elements as z.

`sftool(filename)` loads the surface fitting session in *filename* into Curve Fitting app. The *filename* should have the extension `.sfit`.

`Infs`, `NaNs`, and imaginary parts of complex numbers are ignored in the data.

Curve Fitting app provides a flexible interface where you can interactively fit curves and surfaces to data and view plots. You can:

- Create, plot, and compare multiple fits
- Use linear or nonlinear regression, interpolation, local smoothing regression, or custom equations
- View goodness-of-fit statistics, display confidence intervals and residuals, remove outliers and assess fits with validation data
- Automatically generate code for fitting and plotting surfaces, or export fits to workspace for further analysis

See Also

Apps
Curve Fitting

Topics
"Interactive Curve and Surface Fitting" on page 2-2

Introduced in R2009a

smooth

Smooth response data

Syntax

```
yy = smooth(y)
yy = smooth(y,span)
yy = smooth(y,method)
yy = smooth(y,span,method)
yy = smooth(y,'sgolay',degree)
yy = smooth(y,span,'sgolay',degree)
yy = smooth(x,y, ___ )
gpuarrayYY = smooth(gpuarrayY, ___ )
gpuarrayYY = smooth(gpuarrayX,gpuarrayY, ___ )
```

Description

`yy = smooth(y)` smooths the response data in column vector `y` using a moving average filter.

The first few elements of `yy` follow.

```
yy(1) = y(1)
yy(2) = (y(1) + y(2) + y(3))/3
yy(3) = (y(1) + y(2) + y(3) + y(4) + y(5))/5
yy(4) = (y(2) + y(3) + y(4) + y(5) + y(6))/5
...
```

Because of the way `smooth` handles endpoints, the result differs from the result returned by the `filter` function.

`yy = smooth(y,span)` sets the span of the moving average to `span`.

`yy = smooth(y,method)` smooths the data in `y` using the method specified by `method` and the default `span`.

`yy = smooth(y,span,method)` sets the span of `method` to `span`.

`yy = smooth(y,'sgolay',degree)` uses the Savitzky-Golay method with the polynomial degree specified by `degree`.

`yy = smooth(y,span,'sgolay',degree)` uses the number of data points specified by `span` in the Savitzky-Golay calculation. `span` must be odd and `degree` must be less than `span`.

`yy = smooth(x,y, ___ )` specifies values for the independent variable x. You can use this syntax with any of the arguments in the previous syntaxes.

`gpuarrayYY = smooth(gpuarrayY, ___ )` performs the operation on a GPU using `gpuArray` data. You can use `gpuArray` response data with all previous syntaxes. This syntax requires Parallel Computing Toolbox™.

`gpuarrayYY = smooth(gpuarrayX,gpuarrayY, ___ )` performs the operation on a GPU using `gpuArray` input data. This syntax requires Parallel Computing Toolbox.

Note Using gpuArray x and y inputs with the smooth function is only recommended if you use the default method, 'moving'. Using GPU data with other methods does not offer any performance advantage.

Examples

Smooth Data Using Moving Average Filter

Smooth data by linear index and by each column separately, using a moving average filter. Plot and compare the results.

Load the data in count.dat. The 24-by-3 array count contains traffic counts at three intersections for each hour of the day.

```
load count.dat
```

Suppose that the data are from a single intersection over three consecutive days. Smoothing all the data together would then indicate the overall cycle of traffic flow through the intersection. Use a moving average filter with a 5-hour span to smooth all the data simultaneously (by linear index).

```
c = smooth(count(:));
C1 = reshape(c,24,3);
```

However, the data are in fact from three different intersections. Thus, smoothing columnwise gives a more meaningful picture of the traffic through each intersection in a day. Use the same moving average filter to smooth each column of the data separately.

```
C2 = zeros(24,3);
for I = 1:3
    C2(:,I) = smooth(count(:,I));
end
```

Plot the original data and the data smoothed by linear index and by each column separately. Then, plot the difference between the two smoothed data sets. The two methods give different results near the endpoints.

```
subplot(3,1,1)
plot(count,':');
hold on
plot(C1,'-');
title('Smooth C1 (All Data)')

subplot(3,1,2)
plot(count,':');
hold on
plot(C2,'-');
title('Smooth C2 (Each Column)')

subplot(3,1,3)
plot(C2 - C1,'o-')
title('Difference C2 - C1')
```
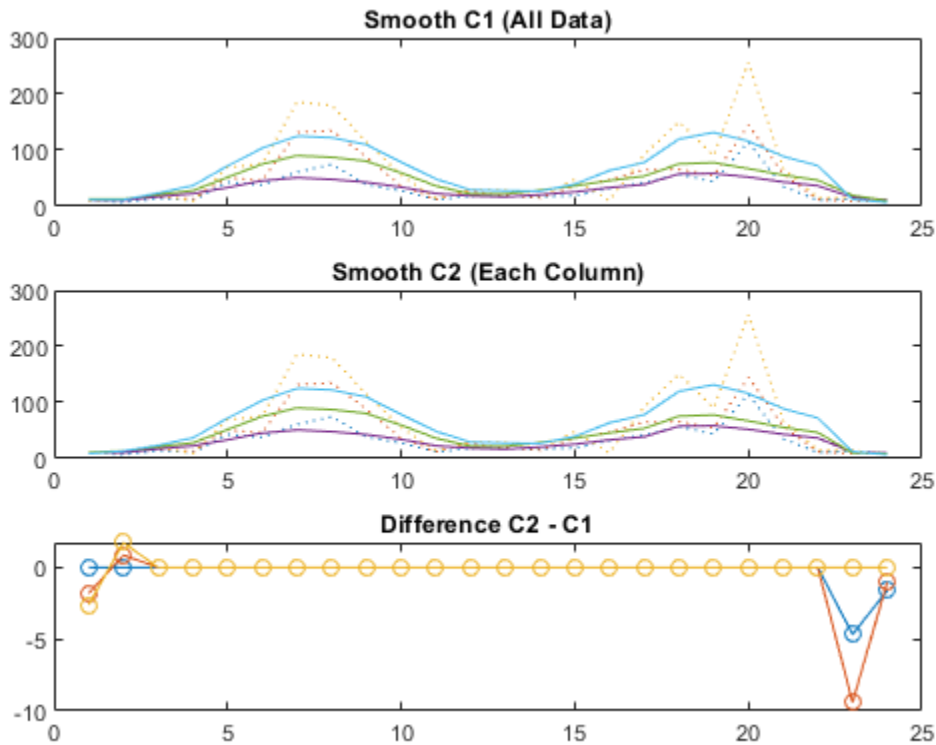
Smooth Data Using `loess` and `rloess`

Plot and compare the results of data smoothed using the `loess` and `rloess` methods. Then determine which method is less sensitive to outliers.

Create noisy data with two outliers.

```
x = (0:0.1:15)';
y = sin(x) + 0.5*(rand(size(x))-0.5);
y([90,110]) = 3;
```

Smooth the data with the `loess` and `rloess` methods. Use a span of 10% of the total number of data points.

```
yy1 = smooth(x,y,0.1,'loess');
yy2 = smooth(x,y,0.1,'rloess');
```

Plot the original and smoothed data. The outliers have less influence with the robust method `rloess`.

```
subplot(2,1,1)
plot(x,y,'b.',x,yy1,'r-')
set(gca,'YLim',[-1.5 3.5])
legend('Original data','Smoothed data using ''loess''',...
        'Location','NW')
```

```
subplot(2,1,2)
plot(x,y,'b.',x,yy2,'r-')
set(gca,'YLim',[-1.5 3.5])
legend('Original data','Smoothed data using ''rloess''',...
       'Location','NW')
```
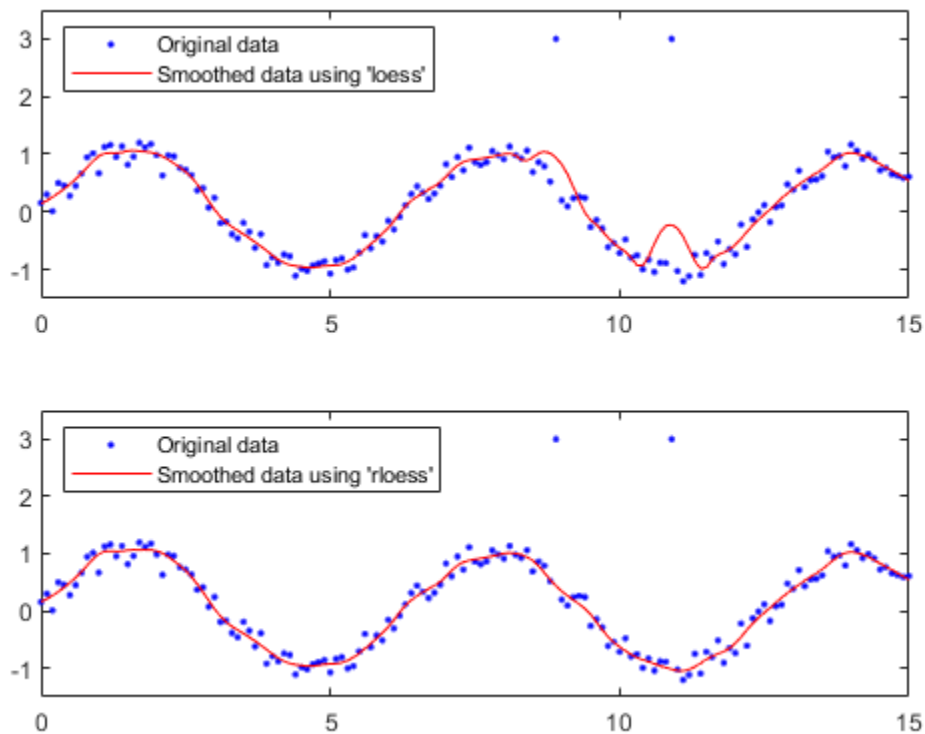

Input Arguments

y — Data to smooth
column vector

Data to smooth, specified as a column vector.

If your data contains NaNs and you do not specify x, your data is treated as nonuniform and the smoothing method `'lowess'` is used.

Data Types: `single` | `double`

span — Number of data points for calculating the smoothed value
5 (default) | integer | scalar value in the range (0,1)

Number of data points for calculating the smoothed value, specified as an integer or as a scalar value in the range (0,1) denoting a fraction of the total number of data points.

If you use the moving average or Savitzky-Golay methods, the number of data points for calculating the smoothed value must be an odd integer. If you specify `span` as an even number or as a fraction that results in an even number of data points, `span` is automatically reduced by 1.

Example: `7`; `0.6`

`method` — Smoothing method
character vector

Smoothing method to smooth the response data, specified as one of the following methods.

Method	Description
`'moving'` (default)	Moving average. A lowpass filter with filter coefficients equal to the reciprocal of the span.
`'lowess'`	Local regression using weighted linear least squares and a 1st degree polynomial model.
`'loess'`	Local regression using weighted linear least squares and a 2nd degree polynomial model.
`'sgolay'`	Savitzky-Golay filter. A generalized moving average with filter coefficients determined by an unweighted linear least-squares regression and a polynomial model of specified degree (default is 2). The method can accept nonuniform predictor data.
`'rlowess'`	A robust version of `'lowess'` that assigns lower weight to outliers in the regression. The method assigns zero weight to data outside six mean absolute deviations.
`'rloess'`	A robust version of `'loess'` that assigns lower weight to outliers in the regression. The method assigns zero weight to data outside six mean absolute deviations.

`degree` — Polynomial degree for Savitzky-Golay method
scalar value

Polynomial degree for the model used in the Savitzky-Golay method, specified as a scalar value. `degree` must be less than `span`.

Example: `3`

`x` — Independent variable for response data `y`
vector

Independent variable for the response data `y`, specified as a column vector. If you do not provide `x`, methods that require `x` assume `x = 1:length(y)`. Specify `x` data when `y` is not sorted or uniformly spaced. If `x` is not uniform and you do not specify `method`, `lowess` is used. If you specify a smoothing method that requires `x` to be sorted, the function automatically sorts the `x` data.

Data Types: `single` | `double`

`gpuarrayY` — Data to smooth
gpuArray vector

Data to smooth, specified as a `gpuArray` column vector.

Data Types: `single` | `double`

gpuarrayX — Input data for response data gpuarrayY
gpuArray vector

Input data for the response data gpuarrayY, specified as a gpuArray column vector. If you do not provide gpuarrayX, methods that require gpuarrayX assume gpuarrayX = 1:length(y). Specify gpuarrayX data when the data are not uniformly spaced or sorted. If the gpuarrayX data is not uniform and you do not specify the smoothing method, 'lowess' is used. If you specify a smoothing method that requires gpuarrayX to be sorted, the function automatically sorts the gpuarrayX data.

Data Types: single | double

Output Arguments

yy — Smoothed data
vector

Smoothed data, returned as a column vector.

gpuarrayYY — Smoothed data
gpuArray vector

Smoothed data, returned as a gpuArray column vector.

Tips

- For more options for smoothing data, including the moving median and Gaussian methods, see smoothdata.
- You can generate a smooth fit to your data using a smoothing spline. For more information, see fit.

Extended Capabilities

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Using gpuArray x and y inputs with the smooth function is only recommended if you use the default method, 'moving'. Using GPU data with other methods does not offer any performance advantage.

For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

See Also
fit | sort

Topics
"Filtering and Smoothing Data" on page 6-27
"Smoothing Splines" on page 6-9
"Lowess Smoothing" on page 6-16
"Nonparametric Fitting" on page 6-2

Introduced before R2006a

slvblk

Solve almost block-diagonal linear system

Syntax

```
x = slvblk(blokmat,b)
x = slvblk(blockmat,b,w)
```

Description

`x = slvblk(blokmat,b)` returns the solution (if any) of the linear system `Ax = b`, with the matrix A stored in `blokmat` in the spline almost block-diagonal form. At present, only the command `spcol` provides such a description, of the matrix whose typical entry is the value of some derivative (including the 0th derivative, i.e., the value) of a B-spline at some site. If the linear system is overdetermined (i.e., has more equations than unknowns but is of full rank), then the least-squares solution is returned.

The right side `b` may contain several columns, and is expected to contain as many rows as there are rows in the matrix described by `blokmat`.

`x = slvblk(blockmat,b,w)` returns the vector `x` that minimizes the *weighted* sum $\Sigma_j w(j)((Ax - b)(j))^2$.

Examples

`sp=spmak(knots,slvblk(spcol(knots,k,x,1),y.'))` provides in `sp` the B-form of the spline s of order `k` with knot sequence `knots` that matches the given data `(x,y)`, i.e., for which `s(x)` equals `y`.

Algorithms

The command `bkbrk` is used to obtain the essential parts of the coefficient matrix described by `blokmat` (in one of two available forms).

A QR factorization is made of each diagonal block, after it was augmented by the equations not dealt with when factoring the preceding block. The resulting factorization is then used to solve the linear system by back substitution.

See Also
bkbrk | spap2 | spapi | spcol

sorted

Locate sites with respect to mesh sites

Syntax

```
pointer = sorted(meshsites,sites)
```

Description

Various commands in this toolbox need to determine the index *j* for which a given *x* lies in the interval $[t_j..t_{j+1}]$, with (t_i) a given nondecreasing sequence, e.g., a knot sequence. This job is done by `sorted` in the following fashion.

`pointer = sorted(meshsites,sites)` is the integer row vector whose j-th entry equals the number of entries in `meshsites` that are ≤ `ssites(j)`, with `ssites` the vector `sort(sites)`. Thus, if both `meshsites` and `sites` are nondecreasing, then

```
meshsites(pointer(j)) ≤ sites(j) < meshsites(pointer(j)+1)
```

with the obvious interpretations when

```
pointer(j) < 1     or     length(meshsites) < pointer(j) + 1
```

Specifically, having `pointer(j) < 1` then corresponds to having `sites(j)` strictly to the left of `meshsites(1)`, while having `length(meshsites) < pointer(j)+1` then corresponds to having `sites(j)` at, or to the right of, `meshsites(end)`.

Examples

The statement

```
sorted([1 1 1 2 2 3 3 3],[0:4])
```

will generate the output `0  3  5  8  8`, as will the statement

```
sorted([3 2 1 1 3 2 3 1],[2 3 0 4 1])
```

Algorithms

The indexing output from `sort([meshsites(:).',sites(:).'])` is used.

spap2

Least-squares spline approximation

Syntax

```
spline = spap2(knots,k,x,y)
spap2(l,k,x,y)
spline = spap2(...,x,y,w)
spap2({knorl1,...,knorlm},k,{x1,...,xm},y)
spap2({knorl1,...,knorlm},k,{x1,...,xm},y,w)
```

Description

`spline = spap2(knots,k,x,y)` returns the B-form of the spline *f* of order k with the given knot sequence `knots` for which

```
(*)    y(:,j) = f(x(j)), all j
```

in the weighted mean-square sense, meaning that the sum

$$\sum_j w(j)\,|y(:,j) - f(x(j))|^2$$

is minimized, with default weights equal to 1. The data values `y(:,j)` can be scalars, vectors, matrices, or ND-arrays, and $|z|^2$ is the sum of the squares of all the entries of *z*. Data points with the same site are replaced by their average.

If the sites x satisfy the Schoenberg-Whitney conditions

$$\text{knots}(j) < x(j) < \text{knots}(j + k)$$

$$(**)\qquad\qquad j = 1, ..., \text{length}(x) = \text{length}(\text{knots}) - k$$

then there is a unique spline of the given order and knot sequence satisfying (*) exactly. No spline is returned unless (**) is satisfied for some subsequence of x.

`spap2(l,k,x,y)` , with l a positive integer, returns the B-form of a least-squares spline approximant, but with the knot sequence chosen for you. The knot sequence is obtained by applying `aptknt` to an appropriate subsequence of x. The resulting piecewise-polynomial consists of l polynomial pieces and has `k-2` continuous derivatives. If you feel that a different distribution of the interior knots might do a better job, follow this up with

```
sp1 = spap2(newknt(spline),k,x,y));
```

`spline = spap2(...,x,y,w)` lets you specify the weights w in the error measure (given above). w must be a vector of the same size as x, with nonnegative entries. All the weights corresponding to data points with the same site are summed when those data points are replaced by their average.

`spap2({knorl1,...,knorlm},k,{x1,...,xm},y)` provides a least-squares spline approximation to *gridded* data. Here, each `knorli` is either a knot sequence or a positive integer. Further, k must be an m-vector, and y must be an (r+m)-dimensional array, with `y(:,i1,...,im)` the datum to be fitted at the `site [x{1}(i1),...,x{m}(im)]`, all `i1`, ..., `im`. However, if the spline is

to be scalar-valued, then, in contrast to the univariate case, y is permitted to be an m-dimensional array, in which case y(i1,...,im) is the datum to be fitted at the site [x{1}(i1),...,x{m}(im)], all i1, ..., im.

spap2({knorl1,...,knorlm},k,{x1,...,xm},y,w) also lets you specify the weights. In this m-variate case, w must be a cell array with m entries, with w{i} a nonnegative vector of the same size as xi, or else w{i} must be empty, in which case the default weights are used in the ith variable.

Examples

Calculate the Least-Squares Approximation of a Cubic Spline

This example shows how to compute the least-squares approximation to the data x, y, by cubic splines with two continuous derivatives, basic interval [a..b], and interior breaks xi, provided xi has all its entries in (a..b) and the conditions (**) are satisfied.

```
sp = spap2(augknt([a,xi,b],4),4,x,y)
```

In that case, the approximation consists of length(xi)+1 polynomial pieces. If you only want to get a cubic spline approximation consisting of l polynomial pieces, use instead

```
sp = spap2(l,4,x,y);
```

If the resulting approximation is not satisfactory, try using a larger l. Else use

```
sp = spap2(newknt(sp),4,x,y);
```

for a better distribution of the knot sequence. Repeat this process multiple times to increase the fidelity of the distribution.

As another example, spap2(1,2,x,y); provides the least-squares straight-line fit to data x,y, while

```
w = ones(size(x));
w([1 end]) = 100;
spap2(1,2,x,y,w);
```

forces that fit to come very close to the first data point and to the last.

Compute and Plot the Least Square Approximation of a Bivariate Function

This example shows how to create a bivariate function, and compute and plot its least-square approximation.

Generate the data for the approximation and the bivariate function.

```
x = -2:.2:2;
y=-1:.25:1;
[xx, yy] = ndgrid(x,y);
z = exp(-(xx.^2+yy.^2));
```

Compute the least-square approximation and plot it.

```
sp = spap2({augknt([-2:2],3),2},[3 4],{x,y},z);
fnplt(sp)
```
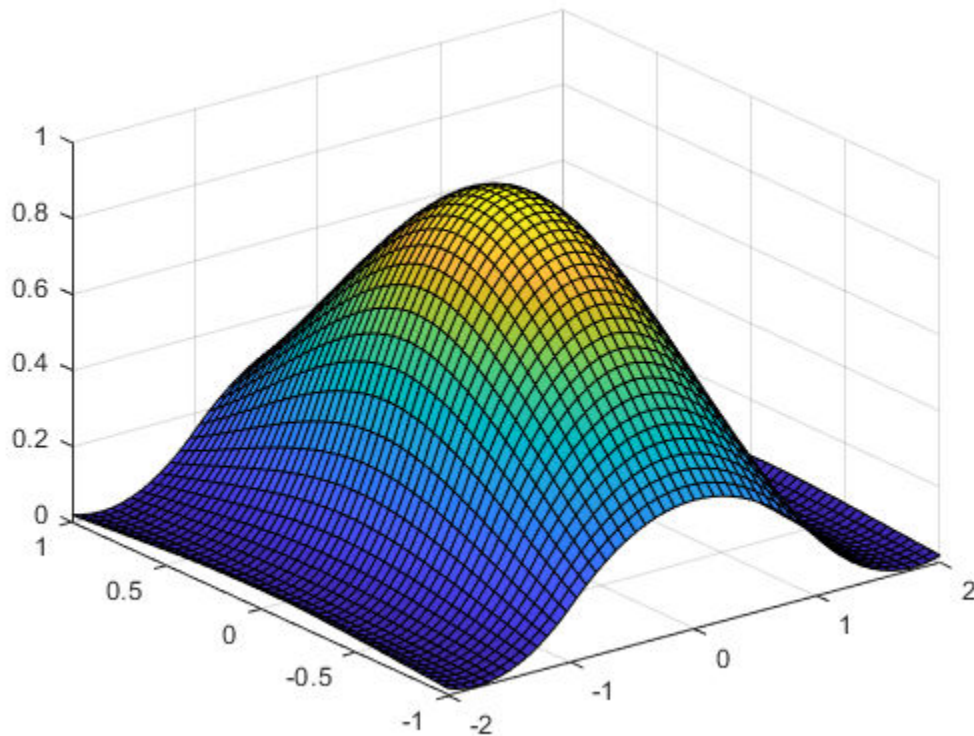
Input Arguments

knots — Sequence of knots
vector | cell array

Knot sequence of the spline, specified as a nondecreasing vector.

k — Spline order
scalar

Order of the spline f, specified as a scalar.

l — Number of polynomial pieces
positive integer

Number of polynomial pieces of the resulting piecewise-polynomial, specified as a positive integer.

w — Error weights
nonnegative vector

Weights in the error measure, specified as a nonnegative vector of the same size as x.

x — Data sites
vector | cell array

Data sites of data values y to be fit, specified as a vector or as a cell array for multivariate data. Spline f is created with knots at each data site x such that $f(\texttt{x(j)}) = \texttt{y(:,j)}$ for all values of j.

For multivariate, gridded data, you can specify x as a cell array that specifies the data site in each variable dimension: $f(\texttt{x1(i),x2(j),...xn(k)}) = \texttt{y(:,i,j,...,k)}$.

y — Data values to fit
scalar | vector | matrix | array

Data values to fit during creation of the spline, specified as a vector, matrix, or array. Data values y(:,j) can be scalars, matrices, or n-dimensional arrays. Data values given at the same data site x are averaged.

Data Types: single | double

Output Arguments

spline — Spline structure
spline structure

Spline, returned as a structure with these fields.

Form — Form of spline
B-

Form of the spline, returned as B-. B- indicates it is given in B-form.

Knots — Knot locations of spline
vector | cell array

Knot positions of the spline, returned as a vector or as a cell array of vectors for multivariate data. Vectors contain strictly increasing elements that represent the start and end of each of the intervals over which the polynomial pieces are defined.

Coefs — Coefficients of polynomials
matrix | array

Coefficients of polynomials for each piece, returned as a matrix or as an array for multivariate data.

Number — Number of polynomial pieces
scalar | vector

Number of polynomial pieces describing the spline, returned as a scalar or as a vector of numbers of pieces in each variable for multivariate data.

Order — Order of polynomials
scalar | vector

Order of the polynomial function describing each polynomial piece of the spline, returned as a scalar or as a vector containing the order in each variable for multivariate data.

Dim — Dimensionality
scalar

Dimensionality of the target function, returned as a scalar.

Algorithms

spcol is called on to provide the almost block-diagonal collocation matrix ($B_{j,k}(x_i)$), and slvblk solves the linear system (*) in the (weighted) least-squares sense, using a block QR factorization.

Gridded data are fitted, in tensor-product fashion, one variable at a time, taking advantage of the fact that a univariate weighted least-squares fit depends linearly on the values being fitted.

See Also

slvblk | spapi | spcol

Topics
About Splines in Curve Fitting on page 8-2

Introduced before R2006a

spapi

Spline interpolation

Syntax

```
spline = spapi(knots,x,y)
spapi(k,x,y)
spapi({knork1,...,knorkm},{x1,...,xm},y)
spapi(...,'noderiv')
```

Description

`spline = spapi(knots,x,y)` returns the spline f (if any) of order

`k = length(knots) - length(x)`

with knot sequence `knots` for which

```
(*)   f(x(j)) = y(:,j), all j.
```

If some of the entries of `x` are the same, then:

$$D^{m(j)}f(x(j)) = y(:, j)$$

with $m(j) = \{i < j : x(i) = x(j)\}$ and $D^m f$ the m-th derivative of f. In this case, the r-fold repetition of a site z in `x` corresponds to the prescribing of value and the first $r - 1$ derivatives of f at z. To match the average of all data values with the same data instead, call `spapi` with an additional fourth argument.

The data values, `y(:,j)`, can be scalars, vectors, matrices, or ND-arrays.

`spapi(k,x,y)`, with `k` a positive integer, specifies the desired spline order, `k`. In this case the `spapi` function calls the `aptknt` function to determine a workable, but not necessarily optimal, knot sequence for the given sites `x`. In other words, the command `spapi(k,x,y)` has the same effect as the more explicit command `spapi(aptknt(x,k),x,y)`.

`spapi({knork1,...,knorkm},{x1,...,xm},y)` returns the B-form of a tensor-product spline interpolant to *gridded* data. Here, each `knorki` is either a knot sequence, or a positive integer specifying the polynomial order used in the `i`-th variable. The `spapi` function then provides a corresponding knot sequence for the `i`-th variable. Further, `y` must be an (`r+m`)-dimensional array, with `y(:,i1,...,im)` the datum to fit at the `site [x{1}(i1),...,x{m}(im)]`, for all `i1`, ..., `im`. In contrast to the univariate case, if the spline is scalar-valued, then `y` can be an `m`-dimensional array.

`spapi(...,'noderiv')` with the character vector `'noderiv'` as a fourth argument, has the same effect as `spapi(...)` except that data values sharing the same site are interpreted differently. With the fourth argument present, the average of the data values with the same data site is interpolated at such a site. Without it, data values with the same data site are interpreted as values of successive derivatives to be matched at such a site, as described above, in the first paragraph of this Description.

Examples

Bivariate Spline Interpolant and Obsculatory Interpolation to Gridded Data

The function `spapi([0 0 0 0 1 2 2 2 2],[0 1 1 1 2],[2 0 1 2 -1])` produces the unique cubic spline *f* on the interval [0...2] with exactly one interior knot, at 1, that satisfies the five conditions

$$f(0+) = 2, f(1) = 0, \mathrm{D}f(1) = 1, D^2f(1) = 2, f(2-) = -1 \,.$$

These include 3-fold matching at 1, i.e., matching there to prescribed values of the function and its first two derivatives.

Here is an example of osculatory interpolation, to values `y` and slopes `s` at the sites `x` by a quintic spline:

```
sp = spapi(augknt(x,6,2),[x,x,min(x),max(x)],[y,s,ddy0,ddy1]);
```

with `ddy0` and `ddy1` values for the second derivative at the endpoints.

As a related example, if you want to interpolate the `sin(x)` function at the distinct data sites by a cubic spline, and to match its slope at a subsequence `x(s)`, then call the `spapi` function with these arguments:

```
sp = spapi(4,[x x(s)], [sin(x) cos(x(s))]).
```

The aptknt function will provide a suitable knot sequence. If you want to interpolate the same data by quintic splines, then simply change the value 4 to 6.

As a bivariate example, here is a bivariate interpolant.

```
x = -2:.5:2; y = -1:.25:1; [xx, yy] = ndgrid(x,y);
z = exp(-(xx.^2+yy.^2));
sp = spapi({3,4},{x,y},z); fnplt(sp)
```
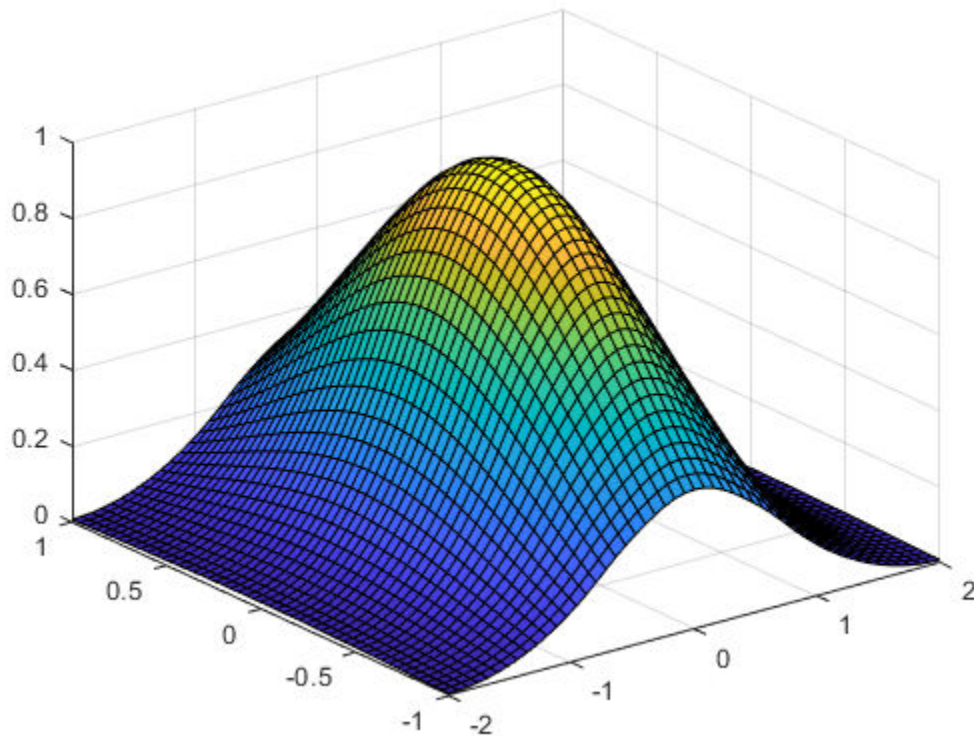
As an illustration of osculatory interpolation to gridded data, here is complete bicubic interpolation, with the data explicitly derived from the bicubic polynomial $g(u, v) = u^3 v^3$. This is helpful to see exactly where the slopes, and slopes of slopes (the cross derivatives), must be placed in the data values supplied. Since g is a bicubic polynomial, its interpolant, f, must be g itself. Test this:

```
sites = {[0,1],[0,2]}; coefs = zeros(4,4); coefs(1,1) = 1;
g = ppmak(sites,coefs);
Dxg = fnval(fnder(g,[1,0]),sites);
Dyg = fnval(fnder(g,[0,1]),sites);
Dxyg = fnval(fnder(g,[1,1]),sites);
f = spapi({4,4}, {sites{1}([1,2,1,2]),sites{2}([1,2,1,2])}, ...
          [fnval(g,sites), Dyg ; ...
           Dxg.'          , Dxyg]);
if any( squeeze( fnbrk(fn2fm(f,'pp'), 'c') ) - coefs )
'something went wrong', end
```

Input Arguments

knots — Sequence of knots
vector | cell array

Knot sequence of the spline, specified as a nondecreasing vector.

k — Evaluation points
vector | matrix | cell

Points at which you want to evaluate the spline function f, specified as a vector, matrix or cell array.

x — Data sites
vector | cell array

Data sites of data values y to be fit, specified as a vector or as a cell array for multivariate data. Spline *f* is created with knots at each data site x such that $f(x(j)) = y(:,j)$ for all values of j.

For multivariate, gridded data, you can specify x as a cell array that specifies the data site in each variable dimension: $f(x1(i),x2(j),...xn(k)) = y(:,i,j,...,k)$.

y — Data values to fit
vector | matrix | array

Data values to fit during creation of the spline, specified as a vector, matrix, or array. Data values y(:,j) can be scalars, matrices, or *n*-dimensional arrays. Data values given at the same data site x are averaged.

Data Types: single | double

Output Arguments

spline — Spline structure
spline structure

Spline, returned as a structure with these fields.

Form — Form of spline
pp | B-

Form of the spline, returned as pp or B-. pp indicates that the spline is given in piecewise polynomial form, B- indicates it is given in B-form.

Knots — Knot locations of spline
vector | cell array

Knot positions of the spline, returned as a vector or as a cell array of vectors for multivariate data. Vectors contain strictly increasing elements that represent the start and end of each of the intervals over which the polynomial pieces are defined.

Coefs — Coefficients of polynomials
matrix | array

Coefficients of polynomials for each piece, returned as a matrix or as an array for multivariate data.

Number — Number of polynomial pieces
scalar | vector

Number of polynomial pieces describing the spline, returned as a scalar or as a vector of numbers of pieces in each variable for multivariate data.

Order — Order of polynomials
scalar | vector

Order of the polynomial function describing each polynomial piece of the spline, returned as a scalar or as a vector containing the order in each variable for multivariate data.

`Dim` — Dimensionality
scalar

Dimensionality of the target function, returned as a scalar.

Limitations

The given (univariate) knots and sites must satisfy the Schoenberg-Whitney conditions for the interpolant to be defined. If the site sequence x is nondecreasing, then

$$\text{knots}(j) < x(j) < \text{knots}(j + k), \text{ all } j$$

with equality possible at `knots`(1) and `knots`(end)). In the multivariate case, these conditions must hold in each variable separately.

Algorithms

The function calls `spcol` to provide the almost-block-diagonal collocation matrix ($B_{j,k}(x)$) (with repeats in x denoting derivatives, as described above), and `slvblk` solves the linear system (*), using a block QR factorization.

The function fits gridded data, in tensor-product fashion, one variable at a time, taking advantage of the fact that a univariate spline fit depends linearly on the values that are being fitted.

See Also

csapi | spap2 | spaps | spline

Topics
About Splines in Curve Fitting on page 8-2
"The B-form" on page 10-13
"The ppform" on page 10-8

Introduced in R2006b

spaps

Smoothing spline

Syntax

```
sp = spaps(x,y,tol)
[sp,values] = spaps(x,y,tol)
[sp,values,rho] = spaps(x,y,tol)
[...] = spaps(x,y,tol,w,m)
[...] = spaps({x1,...,xr},y,tol,...)
```

Description

Note For a simpler but less flexible method to generate smoothing splines, try the Curve Fitting app or the `fit` function.

`sp = spaps(x,y,tol)` returns the B-form of the smoothest function f that lies within the given tolerance `tol` of the given data points `(x(j), y(:,j)), j=1:length(x)`. The data values `y(:,j)` are scalars, vectors, matrices, or even ND-arrays. Data points with the same data site are replaced by their weighted average, with its weight the sum of the corresponding weights, and the tolerance `tol` is reduced accordingly.

`[sp,values] = spaps(x,y,tol)` also returns the smoothed values. `values` is the same as `fnval(sp,x)`.

Here, the distance of the function f from the given data is measured by

$$E(f) = \sum_{j=1}^{n} w(j) |(y(:,j) - f(x(j)))|^2$$

with the default choice for the weights w making $E(f)$ the composite trapezoidal rule approximation to $\int_{\min(x)}^{\max(x)} |y - f|^2$, and $|z|^2$ denoting the sum of squares of the entries of z.

Further, *smoothest* means that the following roughness measure is minimized:

$$F(D^m f) = \int_{\min(x)}^{\max(x)} \lambda(t) |D^m f(t)|^2 dt$$

where $D^m f$ denotes the mth derivative of f. The default value for m is 2, the default value for the roughness measure weight λ is the constant 1, and this makes f a cubic smoothing spline.

When `tol` is nonnegative, then the spline f is determined as the unique minimizer of the expression $\rho E(f) + F(D^m f)$, with the smoothing parameter ρ (optionally returned) so chosen that $E(f)$ equals `tol`. Hence, when m is 2, then, after conversion to ppform, the result should be the same (up to round-off) as obtained by csaps($x,y,\rho/(\rho + 1)$). Further, when `tol` is zero, then the "natural" or variational spline

interpolant of order 2*m* is returned. For large enough `tol`, the least-squares approximation to the data by polynomials of degree `<m` is returned.

When `tol` is negative, then ρ is `-tol`.

The default value for the weight function λ in the roughness measure is the constant function 1. But you can choose it to be, more generally, a piecewise constant function, with breaks only at the data sites. Assuming the vector `x` to be strictly increasing, you specify such a piecewise constant λ by inputting `tol` as a vector of the same size as `x`. In that case, `tol(i)` is taken as the constant value of λ on the interval `(x(i-1) .. x(i))`, `i=2:length(x)`, while `tol(1)` continues to be used as the specified tolerance.

`[sp,values,rho] = spaps(x,y,tol)` also returns the actual value of ρ used as the third output argument.

`[...] = spaps(x,y,tol,w,m)` lets you specify the weight vector `w` and/or the integer `m`, by supplying them as an `argi`. For this, `w` must be a nonnegative vector of the same size as `x`; `m` must be 1 (for a piecewise linear smoothing spline), or 2 (for the default cubic smoothing spline), or 3 (for a quintic smoothing spline).

If the resulting smoothing spline, sp, is to be evaluated outside its basic interval, it should be replaced by `fnxtr(sp,m)` to ensure that its `m`-th derivative is zero outside that interval.

`[...] = spaps({x1,...,xr},y,tol,...)` returns the B-form of an `r`-variate tensor-product smoothing spline that is roughly within the specified tolerance to the given *gridded data*. For *scattered* data, use `tpaps`. Now `y` is expected to supply the corresponding gridded values, with `size(y)` equal to `[length(x1),...,length(xr)]` in case the function is scalar-valued, and equal to `[d,length(x1),...,length(xr)]` in case the function is `d`-valued. Further, `tol` must be a cell array with `r` entries, with `tol{i}` the tolerance used during the `i`-th step when a univariate (but vector-valued) smoothing spline in the `i`-th variable is being constructed. The optional input for `m` must be an `r`-vector (with entries from the set `{1,2,3}`), and the optional input for `w` must be a cell array of length `r`, with `w{i}` either empty (to indicate that the default choice is wanted) or else a positive vector of the same length as `xi`.

Examples

Compare Two Cubic Smoothing Splines Obtained from Noisy Data

This code returns a fit to the noisy data that is expected to be quite close to the underlying noisefree data since the latter come from a slowly varying function and since the used `TOL` is of the size appropriate for the size of the noise.

```
x = linspace(0,2*pi,21); y = sin(x) + (rand(1,21)-.5)*.2;
sp = spaps(x,y, (.05)^2*(x(end)-x(1)) );
fnplt(sp)
```
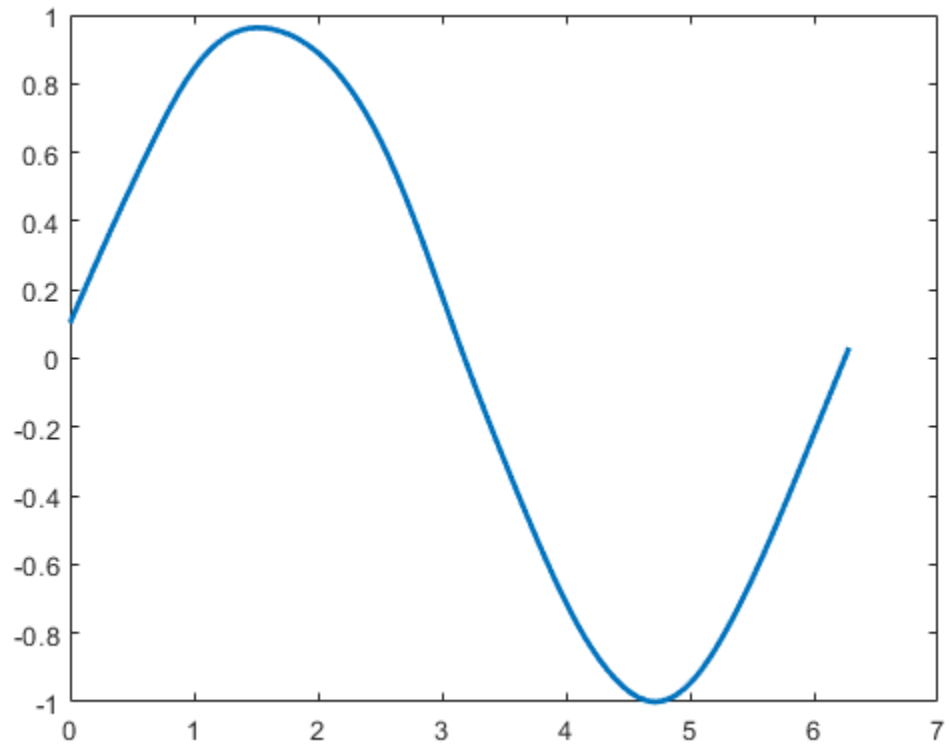
This code uses the same data and tolerance as before, but chooses the roughness weight to be only 0.1 in the right half of the interval and gives, correspondingly, a rougher but better fit there.

```
sp1 = spaps(x,y, [(.025)^2*(x(end)-x(1)),ones(1,10),repmat(.1,1,10)] );
fnplt(sp1)
```
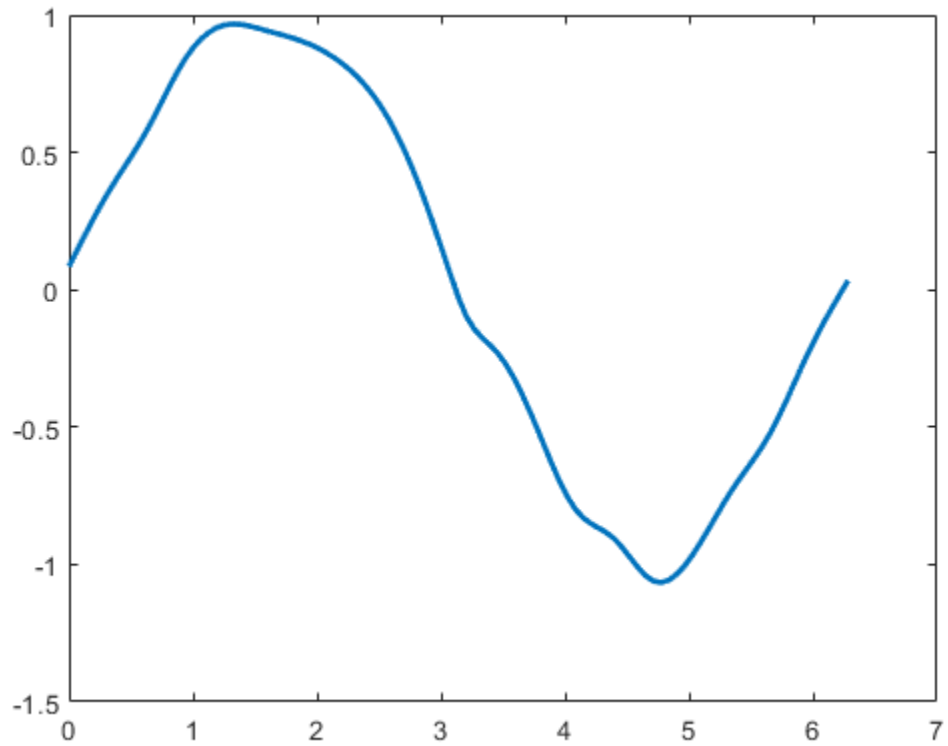
Finally, compare the two cubic smoothing splines obtained previously.

```
fnplt(sp);
hold on
fnplt(sp1,'r')
plot(x,y,'ok')
hold off
title('Two cubic smoothing splines')
xlabel('The red one has reduced smoothness requirement in right half.')
```

Two cubic smoothing splines

The red one has reduced smoothness requirement in right half.
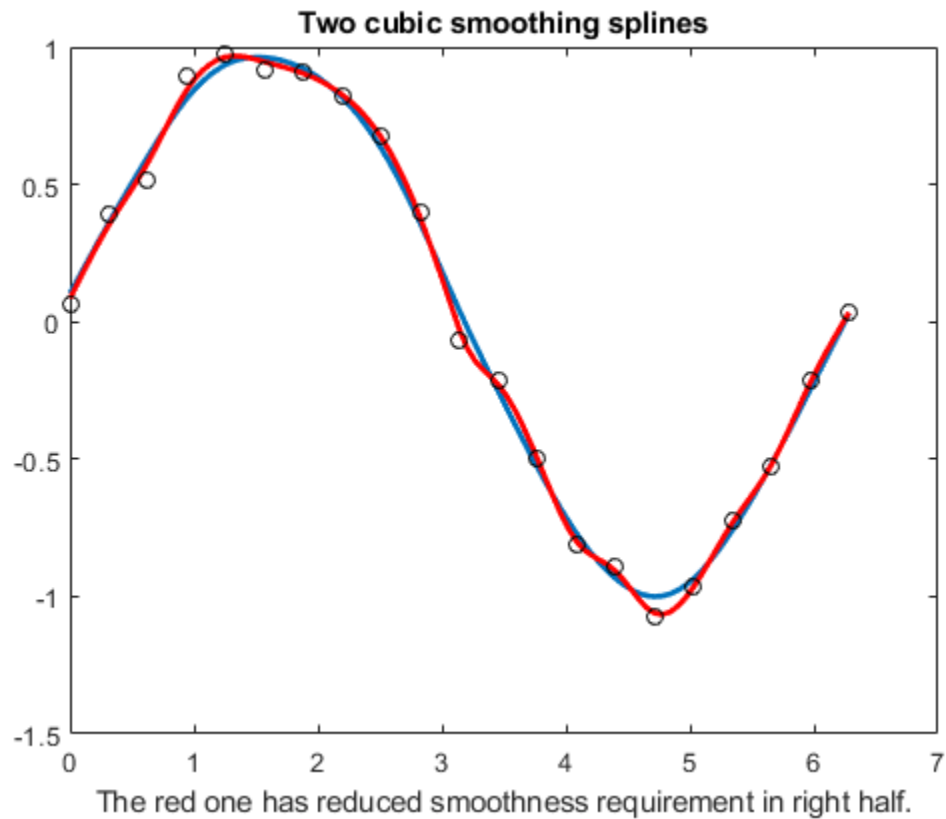
Smooth Approximant to Noisy Data of a Bivariate Function

This example produces a smooth approximant to noisy data from a smooth bivariate function. Note the use of `ndgrid` here; use of `meshgrid` would produce an error.

```
x = -2:.2:2; y=-1:.25:1;
[xx,yy] = ndgrid(x,y);
rng(39);
z = exp(-(xx.^2+yy.^2)) + (rand(size(xx))-.5)/30;
sp = spaps({x,y},z,8/(60^2));
fnplt(sp)
axis off
```

Input Arguments

x — Data sites
vector | cell array

Data sites of data values y to be fit, specified as a vector or as a cell array for multivariate data. Spline *f* is created with knots at each data site x such that f(x(j)) = y(:,j) for all values of j.

For multivariate, gridded data, you can specify x as a cell array that specifies the data site in each variable dimension: f(x1(i),x2(j),...xn(k)) = y(:,i,j,...,k).

Data Types: `single` | `double`

y — Data values to fit
vector | matrix | array

Data values to fit during creation of the spline, specified as a vector, matrix, or array. Data values y(:,j) can be scalars, matrices, or *n*-dimensional arrays. Data values given at the same data site x are averaged.

Data Types: `single` | `double`

`tol` — Tolerance
scalar

Tolerance of the given data points, specified as a scalar.

Data Types: `single` | `double`

w — Data weights
vector | cell array

Weights *w* of the data points, specified as a vector of nonnegative entries of the same size as `x`.

m — Derivative order
scalar

Derivative order, specified as a scalar. This value must be either 1 for a piecewise linear smoothing spline, 2 for the default cubic smoothing spline, or 3 for a quintic smoothing spline.

Data Types: `single` | `double`

Output Arguments

sp — Spline in B-form
spline structure

Spline in B-form, returned as a structure with these fields.

Form — Form of spline
pp

Form of the spline, returned as B-. B- indicates that the spline is given in B-form.

Knots — Knot locations of spline
vector | cell array

Knot positions of the spline, returned as a vector or as a cell array of vectors for multivariate data. Vectors contain strictly increasing elements that represent the start and end of each of the intervals over which the polynomial pieces are defined.

Coefs — Coefficients of polynomials
matrix | array

Coefficients of polynomials for each piece, returned as a matrix or as an array for multivariate data.

Number — Number of polynomial pieces
scalar | vector

Number of polynomial pieces describing the spline, returned as a scalar or as a vector of numbers of pieces in each variable for multivariate data.

Order — Order of polynomials
scalar | vector

Order of the polynomial function describing each polynomial piece of the spline, returned as a scalar or as a vector containing the order in each variable for multivariate data.

Dim — Dimensionality
scalar

Dimensionality of the target function, returned as a scalar.

values — Evaluated spline
vector | matrix | array

Values of the spline at the points in x, returned as a vector or as a matrix or array for multivariate data.

rho — Smoothing parameter
scalar | cell array

Smoothing parameter used to calculate the spline, returned as a scalar or as a cell array of scalar values for multivariate data.

Algorithms

This function uses the Reinsch's approach [1] , including his way of choosing the equation for the optimal smoothing parameter in such a way that a good initial guess is available and Newton's method is guaranteed to converge and to converge fast.

References

[1] C. Reinsch. "Smoothing by spline functions." *Numer. Math*. 10 (1967), 177–183.

See Also
csaps | spap2 | spapi | tpaps

Topics
About Splines in Curve Fitting on page 8-2

Introduced before R2006a

spcol

B-spline collocation matrix

Syntax

```
colmat = spcol(knots,k,tau)
colmat = spcol(knots,k,tau,arg1,arg2,...)
```

Description

`colmat = spcol(knots,k,tau)` returns the matrix, with `length(tau)` rows and `length(knots)-k` columns, whose (*i,j*)th entry is

$$D^{m(i)}B_j(\text{tau}(i))$$

This is the value at tau(*i*) of the *m*(*i*)th derivative of the *j*th B-spline of order `k` for the knot sequence `knots`. Here, `tau` is a sequence of sites, assumed to be *nondecreasing*, and *m* = knt2mlt(tau), i.e., *m*(*i*) is #{*j* < *i*:tau(*j*) = tau(*i*)}, all *i*.

`colmat = spcol(knots,k,tau,arg1,arg2,...)` also returns that matrix, but gives you the opportunity to specify some aspects.

If one of the `argi` is a character vector with the same first two letters as in `'slvblk'`, the matrix is returned in the almost block-diagonal format (specialized for splines) required by `slvblk` (and understood by `bkbrk`).

If one of the `argi` is a character vector with the same first two letters as in `'sparse'`, then the matrix is returned in the `sparse` format of MATLAB.

If one of the `argi` is a character vector with the same first two letters as in `'noderiv'`, multiplicities are ignored, i.e., *m*(*i*) is taken to be 1 for all *i*.

Examples

To solve approximately the non-standard second-order ODE

$$D^2y(t) = 5 \cdot (y(t) - \sin(2t))$$

on the interval [0..π], using cubic splines with 10 polynomial pieces, you can use `spcol` in the following way:

```
tau = linspace(0,pi,101); k = 4;
knots = augknt(linspace(0,pi,11),k);
colmat = spcol(knots,k,brk2knt(tau,3));
coefs = (colmat(3:3:end,:)/5-colmat(1:3:end,:))\(-sin(2*tau).');
sp = spmak(knots,coefs.');
```

You can check how well this spline satisfies the ODE by computing and plotting the residual, $D^2y(t)$ – 5· ($y(t)$ – sin(2t)), on a fine mesh:

```
t = linspace(0,pi,501);
yt = fnval(sp,t);
D2yt = fnval(fnder(sp,2),t);
plot(t,D2yt - 5*(yt-sin(2*t)))
title(['residual error; to be compared to max(abs(D^2y)) = ',...
        num2str(max(abs(D2yt)))])
```

The statement spcol([1:6],3,.1+[2:4]) provides the matrix

```
ans =

    0.5900    0.0050         0
    0.4050    0.5900    0.0050
         0    0.4050    0.5900
```

in which the typical row records the values at 2.1, or 3.1, or 4.1, of all B-splines of order 3 for the knot sequence 1:6. There are three such B-splines. The first one has knots 1,2,3,4, and its values are recorded in the first column. In particular, the last entry in the first column is zero since it gives the value of that B-spline at 4.1, a site to the right of its last knot.

If you add the character vector 'sl' as an additional input to spcol, then you can ask bkbrk to extract detailed information about the block structure of the matrix encoded in the resulting output from spcol. Thus, the statement bkbrk(spcol(1:6,3,.1+2:4,'sl')) gives:

```
block 1 has 2 row(s)
    0.5900    0.0050         0
    0.4050    0.5900    0.0050
next block is shifted over 1 column(s)
block 2 has 1 row(s)
    0.4050    0.5900    0.0050
next block is shifted over 2 column(s)
```

Limitations

The sequence tau is assumed to be nondecreasing.

Algorithms

This is the most complex command in this toolbox since it has to deal with various ordering and blocking issues. The recurrence relations are used to generate, simultaneously, the values of all B-splines of order k having anyone of the tau(i) in their support.

A separate calculation is carried out for the (presumably few) sites at which derivative values are required. These are the sites tau(i) with $m(i) > 0$. For these, and for every order $k - j, j = j_0, j_0 - 1,...,0$, with j_0 equal to max(m), values of all B-splines of that order are generated by recurrence and used to compute the jth derivative at those sites of all B-splines of order k.

The resulting rows of B-spline values (each row corresponding to a particular tau(i)) are then assembled into the overall (usually rather sparse) matrix.

When the optional argument 'sl' is present, these rows are instead assembled into a convenient almost block-diagonal form that takes advantage of the fact that, at any site tau(i), at most k B-splines of order k are nonzero. This fact (together with the natural ordering of the B-splines) implies that the collocation matrix is almost block-diagonal, i.e., has a staircase shape, with the individual blocks or steps of varying height but of uniform width k.

The command `slvblk` is designed to take advantage of this storage-saving form available when used, in `spap2`, `spapi`, or `spaps`, to help determine the B-form of a piecewise-polynomial function from interpolation or other approximation conditions.

See Also
slvblk | spap2 | spapi

spcrv

Spline curve by uniform subdivision

Syntax

```
spcrv(c,k)
spcrv(c)
spcrv(c,k,maxpnt)
```

Description

`spcrv(c,k)` provides a dense sequence f(tt) of points on the uniform B-spline curve f of order k with B-spline coefficients c. Explicitly, this is the curve

$$f : t \Big| \rightarrow \sum_{j=1}^{n} B(t - k/2 \Big| j, ..., j+k) \, c(j), \ \frac{k}{2} \le t \le n + \frac{k}{2}$$

with $B(\cdot|a,...,z)$ the B-spline with knots $a,...,z$, and n the number of coefficients in c, i.e., [d,n] equals `size(c)`.

`spcrv(c)` chooses the order k to be 4.

`spcrv(c,k,maxpnt)` makes sure that at least `maxpnt` points are generated. The default value for the maximum number of sites tt to be generated is `100`.

The parameter interval that the site sequence tt fills out uniformly is the interval [k/2 .. (n-k/2)].

The output consists of the array f(tt).

Examples

The following would show a questionable broken line and its smoothed version:

```
points = [0 0 1 1 0 -1 -1 0 0 ;
          0 0 0 1 2 1 0 -1 -2];
plot(points(1,:),points(2,:),':')
values = spcrv(points,3);
hold on, plot(values(1,:),values(2,:)), hold off
```

Algorithms

Repeated midpoint knot insertion is used until there are at least `maxpnt` sites. There are situations where use of `fnplt` would be more efficient.

See Also
fnplt

splinetool

Experiment with some spline approximation methods

Syntax

```
splinetool
splinetool(x,y)
```

Description

`splinetool` is a graphical user interface (GUI), whose initial menu provides you with various choices for data including the option of importing some data from the workspace.

`splinetool(x,y)` brings up the GUI with the specified data `x` and `y`, which are vectors of the same length.

Examples

Start the GUI and Experiment With Noisy Data

Run this code to start the splinetool GUI and experiment with noisy data.

```
x = linspace(1,pi,101);
y = cos(x)+(rand(size(x))-.5)/10;
splinetool(x,y)
```

Exploring End Conditions For Cubic Spline Interpolation

The purpose of this example is to explore the various end conditions available with cubic spline interpolation:

1 Type `splinetool` at the command line.

2 Select **Import your own data** from the initial screen, and accept the default function. You should see the following display.
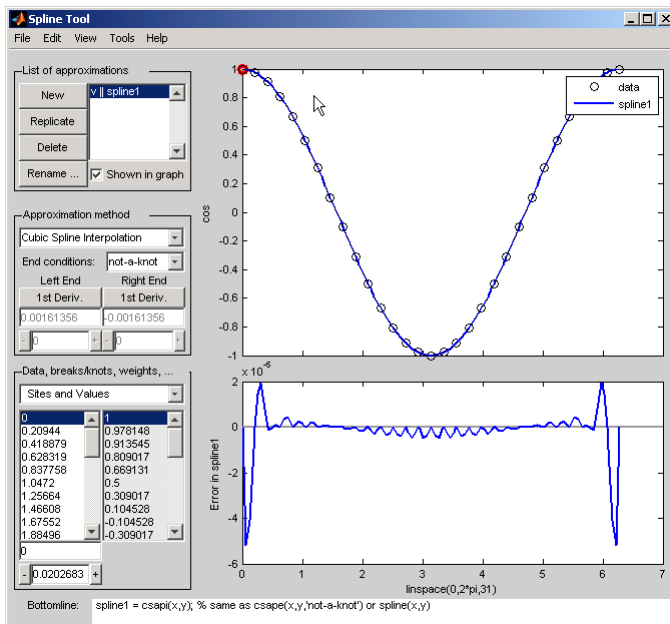
The default approximation shown is the cubic spline interpolant with the not-a-knot end condition.

The vector x of data sites is `linspace(0,2*pi,31)` and the values are `cos(x)`. This differs from simply providing the vector y of values in that the cosine function is explicitly recorded as the underlying function. Therefore, the error shown in the graph is the error in the spline as an approximation to the cosine rather than as an approximation to the given values. Notice the resulting relatively large error, about `5e-5`, near the endpoints.

3 For comparison, follow these steps:

- Click on **New** in the **List of approximations**.
- In **Approximation method**, select **complete** from the list of **End conditions**.
- Since the first derivative of the cosine function is sine, adjust the first-derivative values to their known values of zero at both the left end and the right end.

This procedure results in the display shown below (after the mouse is used to move the Legend further down). Note that the right end slope is zero only up to round-off. **Bottomline** tells you that the toolbox function `csape` was used to create the spline.
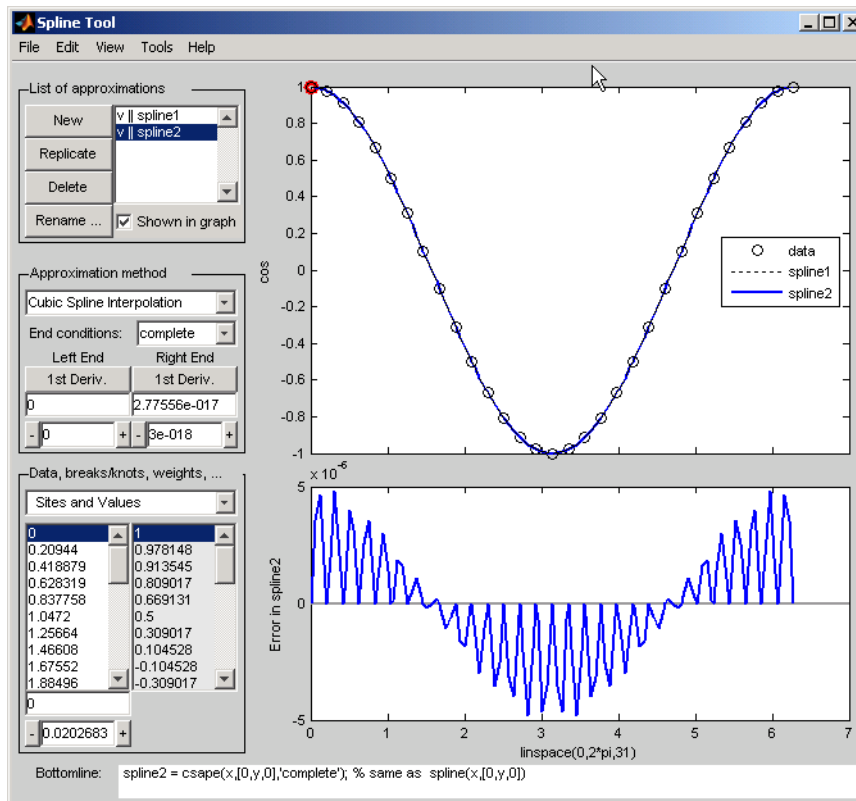
Be impressed by the improvement in the error, which is only about `5e-6`.

4 For further comparison, follow these steps:

- Click on **New** in the **List of approximations**.

- In **Approximation method**, select **natural** from the list of **End conditions**.

Note the deterioration of the approximation near the ends, an error of about `2e-3`, which is much worse than with the not-a-knot end conditions.

5 For a final comparison, follow these steps:

- Click on **New** in the **List of approximations**.

- Since we know that the cosine function is periodic, in **Approximation method**, select **periodic** from the list of **End conditions**.

Note the dramatic improvement in the approximation, back to an error of about `5e-6`, particularly compared to the **natural** end conditions.

Estimating the Second Derivative at an Endpoint

This example uses cubic spline interpolation and least-squares approximation to determine an estimate of the initial acceleration for a drag car:

1 Type `splinetool` at the command line or if the GUI is already running, click on **File > Restart**.

2 Choose **Richard Tapia's drag racing data**. These data show the distance traveled by a drag car as a function of time. The message window asks you to estimate the initial acceleration by setting the initial speed to zero. Click on **OK,** or use **Space** or **Enter,** to remove the message window.

3 In **Approximation method**, select **complete** from the list of **End conditions**.

4 Adjust the initial speed by changing the first derivative at the left endpoint to zero.

5 Look for the value of the initial acceleration, which is given by the value of the second derivative at the left endpoint. You can toggle between the first derivative and the second derivative at this endpoint by clicking on the **left end** button. The value of the second derivative should be around 187 in the units chosen. Choose **View > Show 2nd Derivative** to see this graphically.

6 For comparison, click on **New**, then choose **Least-Squares Approximation** as the **Approximation method**. With this method, you can no longer specify end conditions. Instead, you may vary the order of the method. Verify that the initial acceleration is close to the cubic interpolation value.

The results of this procedure are shown below.
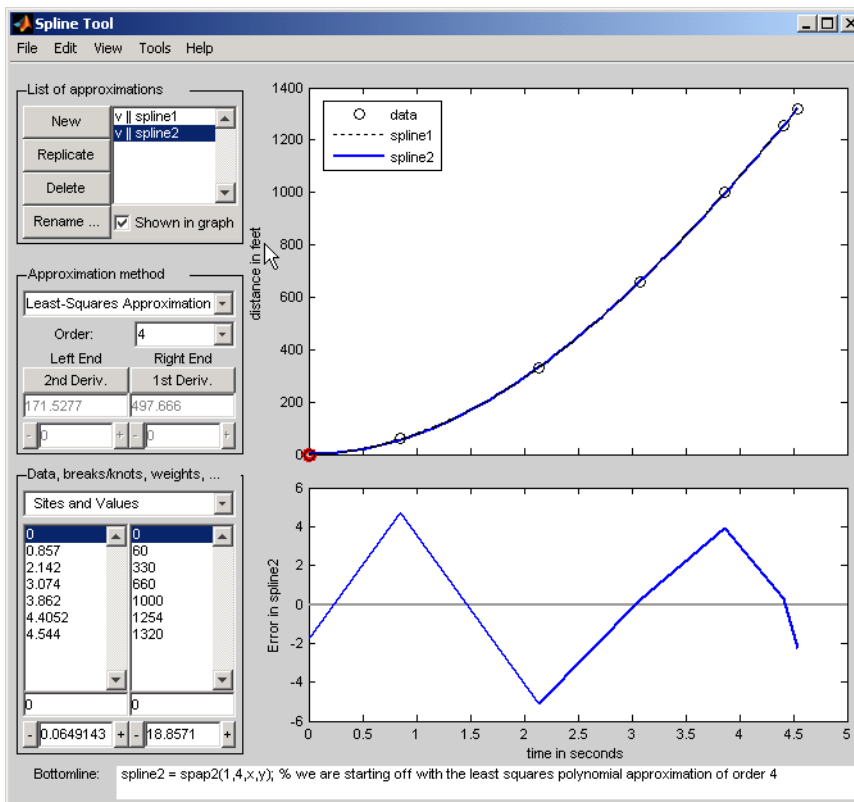
Least-Squares Approximation

This example encourages you to place five interior knots in such a way that the least-squares approximation to these data by cubic splines has an absolute error no bigger than .04 everywhere:

1 Type `splinetool` at the command line or if the GUI is already running, click on **File > Restart**.

2 Choose **Titanium heat data**.

3 Select **Least-Squares Approximation** as the **Approximation method.**

4 Notice how poorly this approximates the data since there are no interior knots. To view the current knots and add new knots, choose **knots** from **Data, breaks/knots, weights**. The knots are now listed in **knots**, and also displayed in the data graph as vertical lines. Notice that there are just the two end knots, each with multiplicity 4.

5 Right-click in the data graph and choose **Add Knot**. This brings up crosshairs for you to move with the mouse. Its precise horizontal location is shown in the edit field below the list of knots. A mouse click places a new knot at the current location of the crosshairs. One possible strategy is to place the additional knot at the place of maximum absolute error, as shown in the auxiliary graph below the data graph.
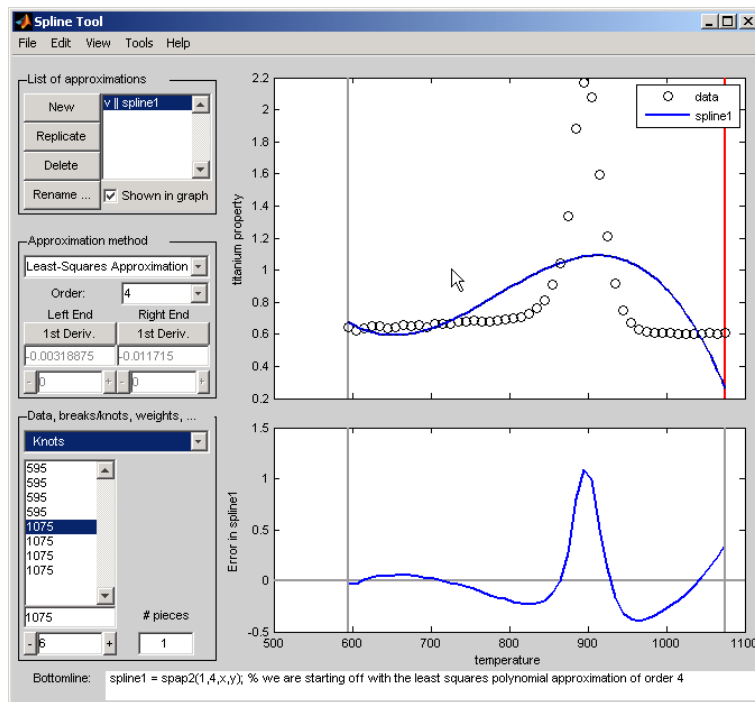
If you right-click and choose **Replicate Knot**, you will increase the multiplicity of the current knot, which is shown by its repeated occurrence in **Knots**. If you don't like a particular knot, you can delete it. To delete a specific knot, you must first select it in either the list of knots or the data graph, and then right-click in the graph and choose **Delete Knot**.

6 You could also ask for an approximation using six polynomial pieces, which corresponds to five interior knots. To do this, enter 6 as **# pieces** in **Data, breaks/knots, weights**.

7 After you have the five interior knots, try to make the error even smaller by moving the knots. To do this, select the knot you want to move by clicking on its vertical line in the graph, then use the interface control below **Knots** in **Data, breaks/knots, weights** and observe how the error changes with the movement of the knot. You can also use the edit field to overwrite the current knot location. You could also try **adjust**, which redistributes the current knot sequence.

8 Use **Replicate** in **List of approximations** to save any good knot distribution for later use. Rename the replicated approximation to `lstsqr` using **Rename**. To return to the original approximation, click on its name in **List of approximations**.

Smoothing Spline

This example experiments with smoothing splines:

1 Type `splinetool` at the command line or, if the GUI is already running, click on **File > Restart**.

2 Choose **Titanium heat data**.

3 In **Approximation method**, choose **Smoothing Spline**.

4 Vary **Parameter** between 0 and 1, which changes the approximation from the least-squares straight-line approximation to the "natural" cubic spline interpolant.

5 Vary **Tolerance** between 0 and some large value, even `inf`. The approximation changes from the best possible one, the "natural" cubic spline interpolant, to the least-squares straight-line approximation.

6 As you increase the **Parameter** value or decrease the **Tolerance** value, the error decreases. However, a smaller error corresponds to more roughness, as measured by the size of the second derivative. To see this, choose **View > Show 2nd Derivative** and vary the **Parameter** and **Tolerance** values once again.

7 This step modifies the weights in the error measure to force the approximation to pass through a particular data point.

- Set **Tolerance** to `0.2`. Notice that the approximation does not pass through the highest data point. To see the large error at this site, choose **View > Error**.

- To force the smoothing spline to go through this point, choose **Error Weights** from **Data, breaks/knots, weights**.

- Click on the highest data point in the graph and notice its site, which is indicated in **Sites and Values**.

- Use the edit field beneath the list of weights to change the current weight to `1000`. Notice how much closer the smoothing spline now comes to that highest data point, and the decrease in the error at that site. Turn on the grid, by **Tools > Grid**, to locate the error at that site more readily.

8 This step modifies the weights in the roughness measure to permit a more accurate but less smooth approximation in the peak area while insisting on a smoother, hence less accurate, approximation away from the peak area.

- Choose **Jumps in Roughness Weight** from **Data, breaks/knots, weights**.

- Choose **View > Show 2nd Derivative**

- Select any data point to the left of the peak in the data.

- Set the jump at the selected site to `-1` by changing its value in the edit field below it. Since the roughness weight for the very first site interval is 1, you have just set the roughness weight to the right of the highlighted site to 0. Correspondingly, the second derivative has become relatively small to the left of that site.

- Select any data point to the right of the peak in the data.

- Set the jump across the selected site to `1`. Since the roughness weight just to the left of the highlighted site is 0, you have just set the roughness weight to the right of the highlighted site to 1. Correspondingly, the second derivative has become relatively small to the right of that site. The total effect is a very smooth but not very accurate fit away from the peak, while in the peak area, the spline fit is much better but the second derivative is much larger, as is shown in the auxiliary graph below.

At the sites where there is a jump in the roughness weight, there is a corresponding jump in the second derivative. If you increase the **Parameter** value, the error across the peak area decreases but the second derivative remains quite large, while the opposite holds true away from the peak area.
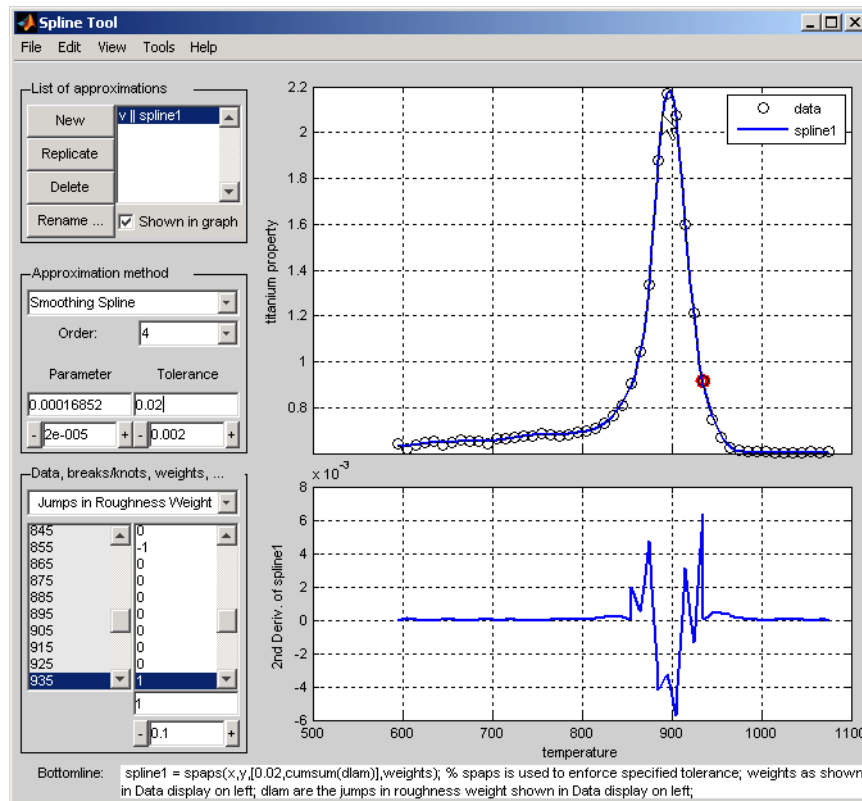
Input Arguments

x — Data sites
vector

Data sites, specified as a vector of the same length of y. The data sites do not need to be distinct or ordered, but there must be at least two distinct sites.

y — Data values
vector

Data values, specified as a vector of the same length of x.

Tips

The Spline Tool is shown in the following figure comparing cubic spline interpolation with a smoothing spline on sample data created by adding noise to the cosine function.
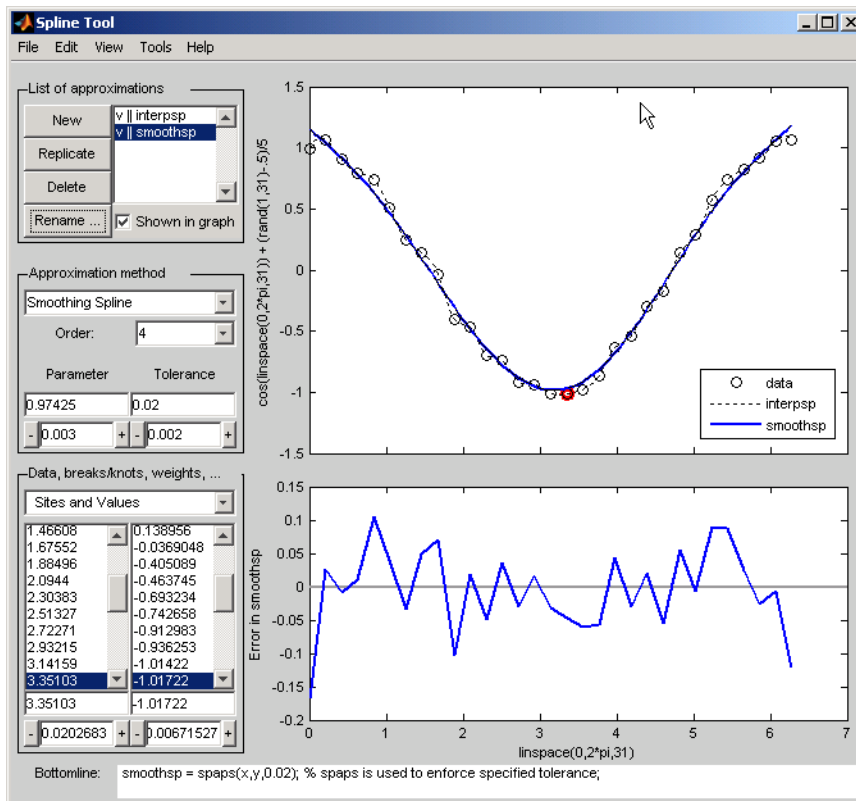
Approximation Methods

The approximation methods and options supported by the GUI are shown below.

Approximation Method	Option
Cubic Interpolating Spline	Adjust the type and values of the end conditions.
Smoothing Spline	Choose between cubic (order 4) and quintic (order 6) splines. Adjust the value of the tolerance and/or smoothing parameter. Adjust the weights in the error and roughness measures.
Least-Squares Approximation	Vary the order from 1 to 14. The default order is 4, which gives cubic approximating splines. Modify the number of polynomial pieces. Add and move knots to improve the fit. Adjust the weights in the error measure.
Spline Interpolation	Vary the order from 2 to 14. The default order is 4, which gives cubic spline interpolants. If the default knots supplied are not satisfactory, you can move them around to vary the fit.

Graphs

You can generate and compare several approximations to the same data. One of the approximations is always marked as "current" using a thicker line width. The following displays are available:

- Data graph. It shows:
  - The data

- The approximations chosen for display in **List of approximations**
- The current knot sequence or the current break sequence

- Auxiliary graph (if viewed) for the current approximation. You can invoke this graph by selecting any one of the items in the **View** menu. It shows one of the following:

  - The first derivative
  - The second derivative
  - The error

By default, the error is the difference between the given data values and the value of the approximation at the data sites. In particular, the error is zero (up to round-off) when the approximation is an interpolant. However, if you provide the data values by specifying a function, then the error displayed is the difference between that function and the current approximation. This also happens if you change the y-label of the data graph to the name of a function.

Menu Options

You can annotate and print the graphs with the **File > Print to Figure** menu.

You can export the data and approximations to the workspace for further use or analysis with the **File > Export Data** and **File > Export Spline** menus, respectively.

You can create, with the **File > Generate Code** menu, a function file that you can use to generate, from the original data, any or all graphs currently shown. This file also provides you with a written record of the commands used to generate the current graph(s).

You can save, with the **Replicate** button, the current approximation before you experiment further. If, at a later time, you click on the approximation so saved, `splinetool` restores everything to the way it was, including the data used in the construction of the saved approximation. This is true even if, since saving this approximation, you have edited the data while working on other approximations.

You can add, delete, or move data, knots, and breaks by right-clicking in the graph, or selecting the appropriate item in the **Edit** menu.

You can toggle the grid or the legend in the graph(s) with the **Tools** menu.

See Also
csape | csapi | csaps | spap2 | spapi | spaps

Topics
About Splines in Curve Fitting on page 8-2

Introduced before R2006a

splpp, sprpp

Taylor coefficients from local B-coefficients

Syntax

```
[v,b] = splpp(tx,a)
[v,b] = sprpp(tx,a)
```

Description

These are utility commands of use in the conversion from B-form to ppform (and in certain evaluations), but of no interest to the casual user.

`[v,b] = splpp(tx,a)` provides the matrices v and b, both of the same size `[r,k]` as a, and related to the input in the following way.

For `i=1:r`, `b(i,:)` are the B-coefficients, with respect to the knot sequence `[tx(i,1:k-1),0,...,0]`, of the polynomial of order k on the interval `[tx(i,k-1) .. tx(i,k)]` whose k B-spline coefficients, with respect to the knot sequence `tx(i,:)`, are in `a(i,:)`. This is done by repeated knot insertion (of the knot 0). It is assumed that `tx(i,k-1)<0<=tx(i,k)`.

For `i=1:r`, `v(i,:)` are the polynomial coefficients for that polynomial, i.e., `v(i,j)` is the number $D^{k-j}s(0-)/k-j)!$, `j=1:k`, with s having the knots `tx(i,:)` and the B-coefficients `a(i,:)`.

`[v,b] = sprpp(tx,a)` carries out exactly the same job, except that now `b(i,:)` are the B-coefficients for that polynomial with respect to the knot sequence `[0,...,0,tx(i,k: 2*(k-1))]`, and, correspondingly, `v(i,j)` is $D^{k-j}s(0+)/k-j)!$, `j=1:k`. Also, now it is assumed that `tx(i,k-1)<=0<tx(i,k)`.

Examples

The statement `[v,b]=splpp([-2 -1 0 1],[0 1 0])` provides the sequence

$$v = \text{-1.0000 -1.0000 0.5000} = D^2s(0-)/2, Ds(0-), s(0-)$$

with s the B-spline with knots -2, -1, 0, 1. This is so because the `l` in `splpp` indicates the limit from the left, and the second argument, `[0 1 0]`, indicates the spline s in question to be

$$s = 0 \times B(\cdot\,|[?, -2, -1, 0]) + 1 \times B(\cdot\,|[-2, -1, 0, 1]) + 0 \times B(\cdot\,|[-1, 0, 1, ?])$$

i.e., this particular linear combination of the third-order B-splines for the knot sequence ..., -2, -1,0,1,... (Note that the values calculated do not depend on the knots marked ?.) The above statement also provides the sequence `b = 0 1.0000 0.5000` of B-spline coefficients for the polynomial piece of s on the interval [-1. .0], and with respect to the knot sequence ?, -2, -1, 0, 0, ?.

In other words, on the interval [-1. .0], the B-spline with knots 2, -1, 0, 1 can be written

$$0 \times B(\cdot\,|[?, -2, -1, 0]) + 1 \times B(\cdot\,|[-2, -1, 0, 0]) + 5 \times B(\cdot\,|[-1, 0, 0, ?])$$

The statement `[v,b]=sprpp([-1 0 1 2],[1 0 0])` provides the sequence

`v = [0.5000 -1.0000 0.5000]` $= D^2 s(0+)/2, Ds(0+), s(0+)$

with s the B-spline with knots ?,-1,0,1. Its polynomial piece on the interval [0..1] is independent of the choice of ?, so we might as well think of ? as -2, i.e., we are dealing with the same B-spline as before. Note that the last two numbers agree with the limits from the left computed above, while the first number does not. This reflects the fact that a quadratic B-spline with simple knots is continuous with continuous first, but discontinuous second, derivative. (It also reflects the fact that the leftmost knot of a B-spline is irrelevant for its right-most polynomial piece.) The sequence `b = 0.5000 0 0` also provided states that, on the interval [0. .1], the B-spline $B(\cdot|[?,-1,0,1])$ can be written

$$0.5 \times B(\cdot|[0,0,0,1]) + 0 \times B(\cdot|[0,0,1,2]) + 0 \times B(\cdot|[0,1,2,?])$$

spmak

Put together spline in B-form

Syntax

```
spmak(knots,coefs)
spmak(knots,coefs,sizec)
spmak
sp = spmak(knots,coeffs)
```

Description

The command `spmak(...)` puts together a spline function in B-form, from minimal information, with the rest inferred from the input. `fnbrk` returns all the parts of the completed description. In this way, the actual data structure used for the storage of this form is easily modified without any effect on the various `fn...` commands that use this construct.

`spmak(knots,coefs)` returns the B-form of the spline specified by the knot information in `knots` and the coefficient information in `coefs`.

The action taken by `spmak` depends on whether the function is univariate or multivariate, as indicated by `knots` being a sequence or a cell array. For the description, let `sizec` be `size(coefs)`.

If `knots` is a sequence (required to be non-decreasing), then the spline is taken to be univariate, and its order k is taken to be `length(knots)-sizec(end)`. This means that each `column' `coefs(:,j)` of `coefs` is taken to be a B-spline coefficient of the spline, hence the spline is taken to be `sizec(1:end-1)`-valued. The basic interval of the B-form is [`knots(1)` .. `knots(end)`].

Knot multiplicity is held to be ≤ k. This means that the coefficient `coefs(:,j)` is simply ignored in case the corresponding B-spline has only one distinct knot, i.e., in case `knots(j)` equals `knots(j+k)`.

If `knots` is a cell array, of length m, then the spline is taken to be m-variate, and `coefs` must be an (r+m)-dimensional array, – except when the spline is to be scalar-valued, in which case, in contrast to the univariate case, `coefs` is permitted to be an m-dimensional array, but `sizec` is reset by

```
sizec = [1, sizec]; r = 1;
```

The spline is `sizec(1:r)`-valued. This means the output of the spline is an array with r dimensions, e.g., if `sizec(1:2) = [2, 3]` then the output of the spline is a 2-by-3 matrix.

The spline is `sizec(1:r)`-valued, the ith entry of the m-vector k is computed as `length(knots{i}) - sizec(r+i)`, i=1:m, and the ith entry of the cell array of basic intervals is set to [`knots{i}(1)`, `knots{i}(end)`].

`spmak(knots,coefs,sizec)` lets you supply the intended size of the array `coefs`. Assuming that `coefs` is correctly sized, this is of concern only in the rare case that `coefs` has one or more trailing singleton dimensions. For, MATLAB suppresses trailing singleton dimensions, hence, without this explicit specification of the intended size of `coefs`, `spmak` would interpret `coefs` incorrectly.

`spmak` prompts you for `knots` and `coefs`.

`sp = spmak(knots,coeffs)` returns the spline `sp`.

Examples

`spmak(1:6,0:2)` constructs a spline function with basic interval [1. .6], with 6 knots and 3 coefficients, hence of order 6 - 3 = 3.

`spmak(t,1)` provides the B-spline $B(\cdot|t)$ in B-form.

The coefficients may be d-vectors (e.g., 2-vectors or 3-vectors), in which case the resulting spline is a curve or surface (in R^2 or R^3).

If the intent is to construct a 2-vector-valued bivariate polynomial on the rectangle [–1..1] × [0..1], linear in the first variable and constant in the second, say

```
 coefs = zeros([2 2 1]); coefs(:,:,1) = [1 0;0 1];
```

then the straightforward

```
sp = spmak({[-1 -1 1 1],[0 1]},coefs);
```

will result in the error message `'There should be no more knots than coefficients'`, because the trailing singleton dimension of `coefs` will not be perceived by `spmak`, while proper use of that third argument, as in

```
sp = spmak({[-1 -1 1 1],[0 1]},coefs,[2 2 1]);
```

will succeed. Replacing here `[2 2 1]` by `size(coefs)` would not work.

See the example "Intro to B-form" for other examples.

Diagnostics

There will be an error return if the proposed knot sequence fails to be nondecreasing, or if the coefficient array is empty, or if there are not more knots than there are coefficients. If the spline is to be multivariate, then this last diagnostic may be due to trailing singleton dimensions in `coefs`.

See Also
`fnbrk`

spterms

Explain spline terms

Syntax

```
spterms(term)
expl = spterms(term)
[...,term] = spterms(...)
```

Description

`spterms(term)` provides, in a message box, an explanation of the technical term indicated by the character vector `term` as used in the Curve Fitting Toolbox spline functions and, specifically, in the GUI `splinetool`. Only the first few (but at least two) letters of the desired term need to be specified, and the full term is shown in the title of the message box.

`expl = spterms(term)` returns, in `expl`, the character vector, or cell array of character vectors, comprising the explanation of the desired term.

`[...,term] = spterms(...)` also returns, in `term`, the fully spelled-out term actually used.

Examples

`spterms('sp')` gives an explanation of the term `spline', while `spterms('spline i')` explains the terms `spline interpolation'.

`help spterms` provides the list of all available terms.

See Also
`splinetool`

stcol

Scattered translates collocation matrix

Syntax

```
colmat = stcol(centers,x,type)
colmat = stcol(...,'tr')
```

Description

`colmat = stcol(centers,x,type)` is the matrix whose (`i,j`)th entry is

$$\psi_j(x(:,i)), \quad i = 1:\text{size}(x, 2), j = 1:n$$

with the bivariate functions ψ_j and the number `n` depending on the `centers` and the character vector `type`, as detailed in the description of `stmak`.

`centers` and `x` must be matrices with the same number of rows.

The default for `type` is the character vector `'tp'`, and for this default, `n` equals `size(centers,2)`, and the functions ψ_j are given by

$$\psi_j(x) = \psi(x - \text{centers}(:, j)), \quad j = 1:n$$

with ψ the thin-plate spline basis function

$$\psi(x) = |x|^2 \log|x|^2$$

and with $|x|$ denoting the Euclidean norm of the vector x.

Note See `stmak` for a description of other possible values for `type`.

The matrix `colmat` is the coefficient matrix in the linear system

$$\sum_j a_j \psi_j(x(:, i)) = y_i, \quad i = 1:\text{size}(x, 2)$$

that the coefficients a_j of the function $f = \Sigma_j a_j \psi_j$ must satisfy in order that f interpolate the value y_i at the site `x(:,i)`, all `i`.

`colmat = stcol(...,'tr')` returns the transpose of the matrix returned by `stcol(...)`.
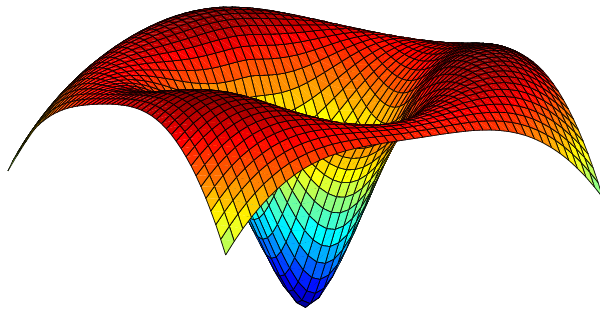
Examples

Example 1. The following evaluates and plots the function

$$f(x) = \psi(x - c_1) + \psi(x - c_2) + \psi(x - c_3) - 3.5\psi(x)$$

on a regular mesh, with ψ the above thin-plate basis function, and with c_1, c_2, c_3 three points on the unit circle; see the figure below.

```
a = [0,2/3*pi,4/3*pi]; centers = [cos(a), 0; sin(a), 0];
[xx,yy] = ndgrid(linspace(-2,2,45));
xy = [xx(:) yy(:)].';
coefs = [1 1 1 -3.5];
zz = reshape( coefs*stcol(centers,xy,'tr') , size(xx));
surf(xx,yy,zz), view([240,15]), axis off
```


Example 2. The following also evaluates, on the same mesh, and plots the length of the gradient of the function in Example 1.

```
zz = reshape( sqrt(...
        ([coefs,0]*stcol(centers,xy,'tp10','tr')).^2 + ...
        ([coefs,0]*stcol(centers,xy,'tr','tp01')).^2),
size(xx));
figure, surf(xx,yy,zz), view([220,-15]), axis off
```

See Also
spcol | stmak

stmak

Put together function in stform

Syntax

```
stmak(centers,coefs)
st = stmak(centers,x,type)
st = stmak(centers,coefs,type,interv)
```

Description

`stmak(centers,coefs)` returns the stform of the function f given by

$$f(x) = \sum_{j=1}^{n} \text{coefs}(:,j) \cdot \psi(x - \text{centers}(:,j))$$

with

$$\psi(x) = |x|^2 \log|x|^2$$

the thin-plate spline basis function, and with $|x|$ denoting the Euclidean norm of the vector x.

`centers` and `coefs` must be matrices with the same number of columns.

`st = stmak(centers,x,type)` stores in `st` the stform of the function f given by

$$f(x) = \sum_{j=1}^{n} \text{coefs}(:,j) \cdot \psi_j(x)$$

with the ψ_j as indicated by the character vector `type`, which can be one of the following:

- `'tp00'`, for the thin-plate spline;
- `'tp10'`, for the first derivative of a thin-plate spline with respect to its first argument;
- `'tp01'`, for the first derivative of a thin-plate spline with respect to its second argument;
- `'tp'`, the default.

Here are the details.

`'tp00'`	$\psi_j(x) = \varphi(|x - c_j|^2)$, $c_j =$`centers(:,j)`, `j=1:n-3`
	with $\varphi(t) = t\log(t)$
	$\psi_{n-2}(x) = x(1)$
	$\psi_{n-1}(x) = x(2)$
	$\psi_n(x) = 1$

|	
`'tp10'`	$\psi_j(x) = \varphi(\|x - c_j\|^2)$, $c_j =$ `centers(:,j)`, `j=1:n-1` with $\varphi(t) = (D_1 t)(\log t + 1)$, and $D_1 t$ the partial derivative of $t = t(x) = \|x - c_j\|^2$ with respect to $x(1)$ $\psi_n(x) = 1$
`'tp01'`	$\psi_j(x) = \varphi(\|x - c_j\|^2)$, $c_j =$ `centers(:,j)`, `j=1:n-1` with $\varphi(t) = (D_2 t)(\log t + 1)$, and $D_2 t$ the partial derivative of $t = t(x) = \|x - c_j\|^2$ with respect to $x(2)$ $\psi_n(x) = 1$
`'tp'` (default)	$\psi_j(x) = \varphi(\|x - c_j\|^2)$, $c_j =$ `centers(:,j)`, `j=1:n` with $\varphi(t) = t\log(t)$

`st = stmak(centers,coefs,type,interv)` also specifies the basic interval for the stform, with `interv{j}` specifying, in the form `[a,b]`, the range of the `j`th variable. The default for `interv` is the smallest such box that contains all the given centers.

Examples

Example 1. The following generates the figure below, of the thin-plate spline basis function, $\psi(x) = |x|^2 \log|x|^2$, but suitably restricted to show that this function is negative near the origin. For this, the extra lines are there to indicate the zero level.

```
inx = [-1.5 1.5]; iny = [0 1.2];
fnplt(stmak([0;0],1),{inx,iny})
hold on, plot(inx,repmat(linspace(iny(1),iny(2),11),2,1),'r')
view([25,20]),axis off, hold off
```
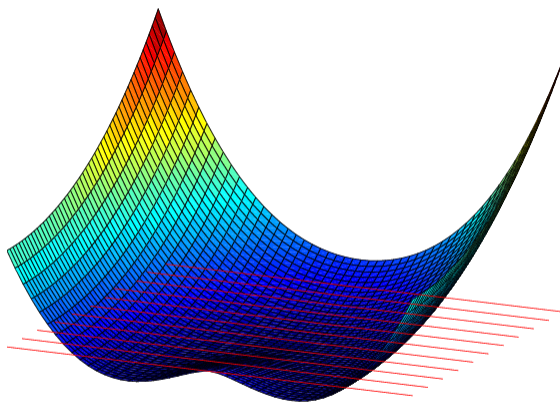
Example 2. We now also generate and plot, on the very same domain, the first partial derivative $D_2\psi$ of the thin-plate spline basis function, with respect to its second argument.

```
inx = [-1.5 1.5]; iny = [0 1.2];
fnplt(stmak([0;0],[1 0],'tp01',{inx,iny}))
view([13,10]),shading flat,axis off
```

Note that, this time, we have explicitly set the basic interval for the stform.

The resulting figure, below, shows a very strong variation near the origin. This reflects the fact that the *second* derivatives of ψ have a logarithmic singularity there.
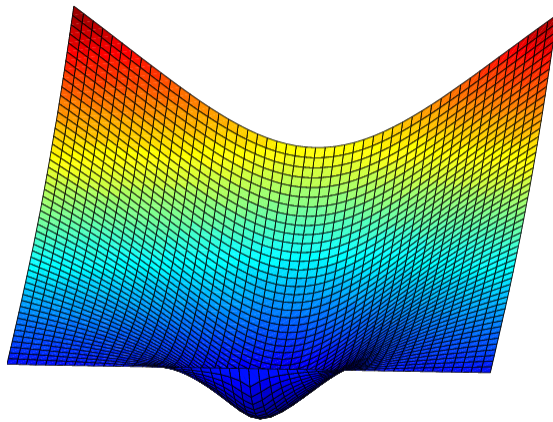
See Also
```
stcol
```

subplus

Calculate the positive part of a function

Syntax

```
xp = subplus(x)
```

Description

`xp = subplus(x)` returns the positive part of x, $(x)_+$, which is x if x is nonnegative, and `0` if x is negative. In other words, `xp` equals `max(x,0)`. If x is an array, this operation is applied entry by entry.

Examples

Plot the Positive Part of a Simple Function

This code generates and plot the essential part of the subplus function.

```
x = -2:2; plot(x,subplus(x),'linew',4), axis([-2,2,-.5,2.5])
```
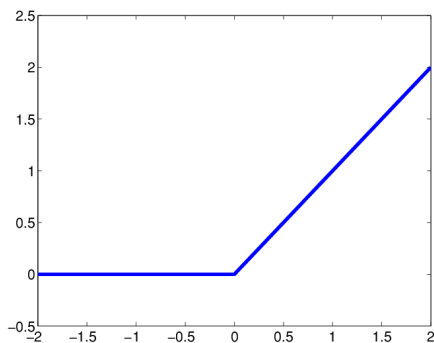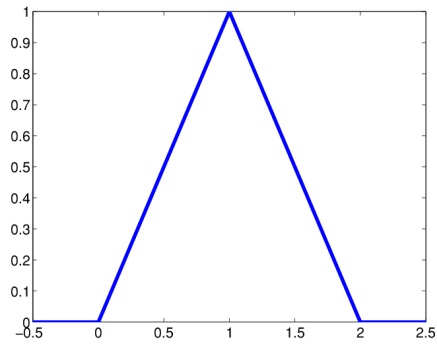

Plot the Hat Function Using the subplus Function

The following anonymous function describes the so-called hat function:

```
hat = @(x) subplus(x) - 2*subplus(x-1) + subplus(x-2);
```

This is the spline also given by `spmak(0:2,1)`, as the following plot shows.

Input Arguments

x — Input function
scalar | vector | matrix

Function of which you want to calculate the positive part, specified as a scalar, vector, or matrix.

Data Types: `single` | `double`

Output Arguments

xp — Positive part
scalar | vector | matrix

Positive part of `x`, returned as a scalar, vector, or matrix.

See Also
fit on page 13-77 | `spline`

Topics
About Splines in Curve Fitting on page 8-2

Introduced in R2006b

titanium

Titanium test data

Syntax

```
[x,y] = titanium
```

Description

`[x,y] = titanium` returns measurements of a certain property of titanium as a function of temperature. Since their use in [1], these data have become a standard test for data fitting since they are hard to fit by classical techniques and have a significant amount of noise.

Examples

The plot of the data shown below is generated by the following commands:

```
[x,y] = titanium; plot(x,y,'ok'), set(gca,'Fontsize',16)
```
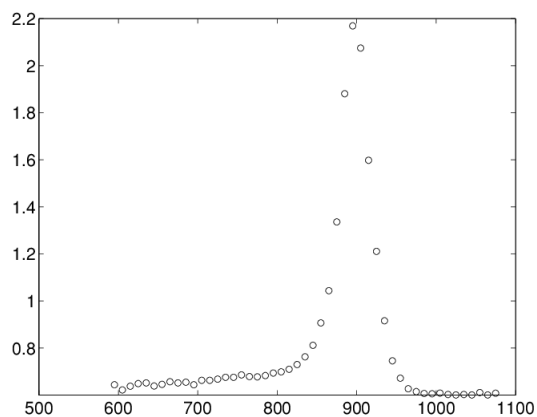

References

[1] C. de Boor and J. R. Rice. "Least squares cubic spline approximation II - Variable knots." CSD TR 21, Comp.Sci., Purdue Univ., April 1968.

tpaps

Thin-plate smoothing spline

Syntax

```
st = tpaps(x,y)
st = tpaps(x,y,p)
[...,P] = tpaps(...)
```

Description

`st = tpaps(x,y)` is the stform of a thin-plate smoothing spline *f* for the given data sites `x(:,j)` and the given data values `y(:,j)`. The `x(:,j)` must be distinct points in the plane, the values can be scalars, vectors, matrices, even ND-arrays, and there must be exactly as many values as there are sites.

The thin-plate smoothing spline *f* is the unique minimizer of the weighted sum

$$pE(f) + (1 - p)R(f)$$

with *E*(*f*) the error measure

$$E(f) = \sum_{j} |y(:, j) - f(x(:, j))|^2$$

and *R*(*f*) the roughness measure

$$R(f) = \int (|D_1D_1f|^2 + 2|D_1D_2f|^2 + |D_2D_2f|^2)$$

Here, the integral is taken over all of R^2, $|z|^2$ denotes the sum of squares of all the entries of *z*, and D_if denotes the partial derivative of *f* with respect to its *i*-th argument, hence the integrand involves second partial derivatives of *f*. The function chooses the smoothing parameter `p` so that `(1-p)/p` equals the average of the diagonal entries of the matrix `A`, with `A + (1-p)/p*eye(n)` the coefficient matrix of the linear system for the `n` coefficients of the smoothing spline to be determined. This ensures staying in between the two extremes of interpolation (when `p` is close to `1` and the coefficient matrix is essentially `A`) and complete smoothing (when `p` is close to `0` and the coefficient matrix is essentially a multiple of the identity matrix). This serves as a good first guess for `p`.

`st = tpaps(x,y,p)` also inputs the *smoothing parameter*, `p`, a number between 0 and 1. As the smoothing parameter varies from 0 to 1, the smoothing spline varies, from the least-squares approximation to the data by a linear polynomial when `p` is `0`, to the thin-plate spline interpolant to the data when `p` is `1`.

`[...,P] = tpaps(...)` also returns the value of the smoothing parameter used in the final spline result whether or not you specify `p`. This syntax is useful for experimentation in which you can start with `[pp,P] = tpaps(x,y)` and obtain a reasonable first guess for `p`.

Examples

Recover the Underlying Exact Smooth Values from Noised Data

The following code obtains values of a smooth function at 31 randomly chosen sites, adds some random noise to these values, and then uses `tpaps` to recover the underlying exact smooth values. To illustrate how well `tpaps` does in this case, the code plots, in addition to the smoothing spline, the exact values (as black balls) as well as each arrow leading from a smoothed value to the corresponding noisy value.

```
rng(23); nxy = 31;
xy = 2*(rand(2,nxy)-.5); vals = sum(xy.^2);
noisyvals = vals + (rand(size(vals))-.5)/5;
st = tpaps(xy,noisyvals); fnplt(st), hold on
avals = fnval(st,xy);
plot3(xy(1,:),xy(2,:),vals,'wo','markerfacecolor','k')
quiver3(xy(1,:),xy(2,:),avals,zeros(1,nxy),zeros(1,nxy), ...
        noisyvals-avals,'r'), hold off
```
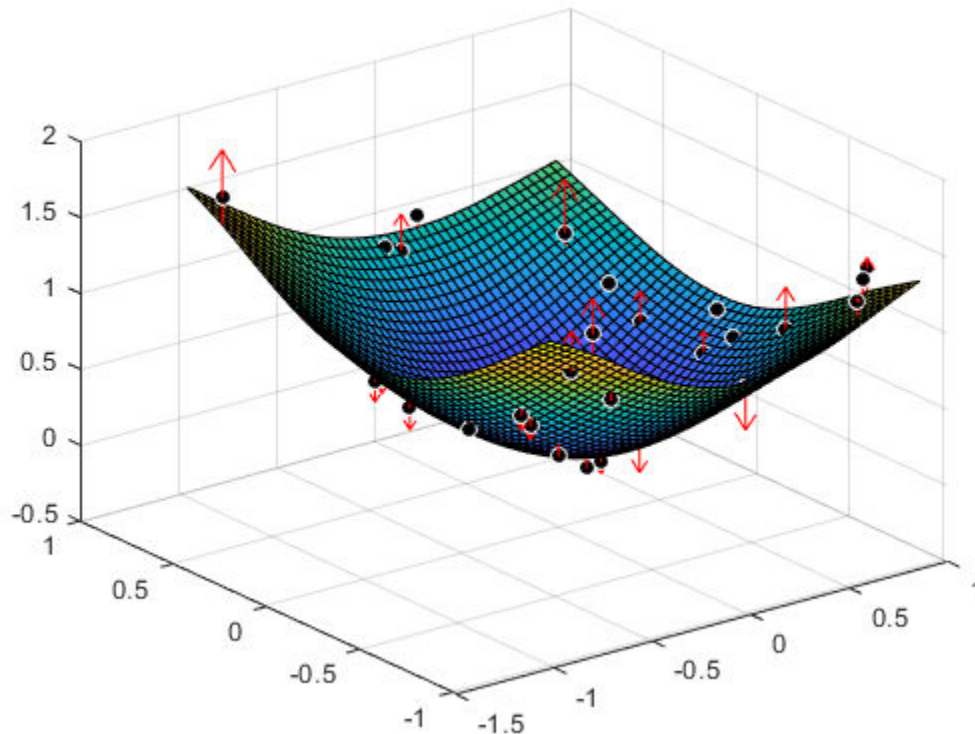

Use an Interpolating Thin-Plate Spline to Construct a Map

The following code uses an interpolating thin-plate spline to vector-valued data values to construct a map, from the plane to the plane, that carries the unit square $\{x: |x(j)| \leq 1, j = 1:2\}$ approximately onto the unit disk $\{x: x(1)^2 + x(2)^2 \leq 1\}$.

```
n = 64; t = linspace(0,2*pi,n+1); t(end) = [];
values = [cos(t); sin(t)];
centers = values./repmat(max(abs(values)),2,1);
st = tpaps(centers, values, 1);
fnplt(st), axis equal
```
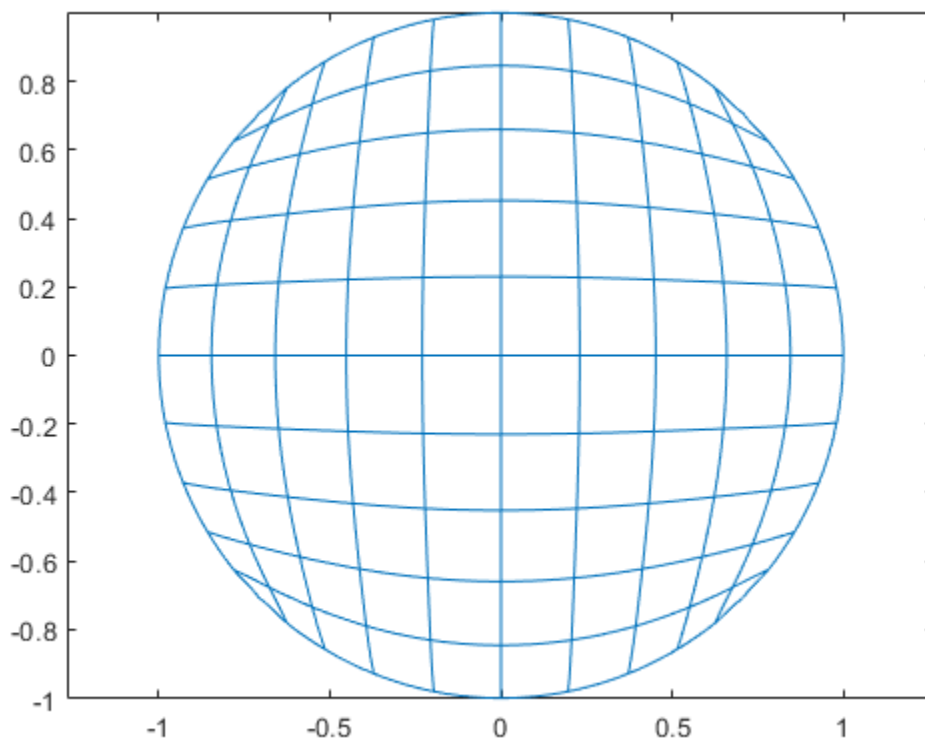

Note the choice of 1 for the smoothing parameter here, to obtain interpolation.

Input Arguments

x — Data sites
vector | cell array

Data sites of data values y to be fit, specified as a vector or as a cell array for multivariate data. Spline *f* is created with knots at each data site x such that $f(x(j)) = y(:,j)$ for all values of j.

For multivariate, gridded data, you can specify x as a cell array that specifies the data site in each variable dimension: $f(x1(i),x2(j),...xn(k)) = y(:,i,j,...,k)$.

y — Data values to fit
vector | matrix | array

Data values to fit during creation of the spline, specified as a vector, matrix, or array. Data values $y(:,j)$ can be scalars, matrices, or *n*-dimensional arrays. Data values given at the same data site x are averaged.

Data Types: `single` | `double`

p — Smoothing parameter
scalar in the range `[0,1]` | vector | cell array | empty array

Smoothing parameter, specified as a scalar value between 0 and 1 or as a cell array of values for multivariate data. You can also specify values for the roughness measure weights λ by providing `p` as a vector. To provide roughness measure weights for multivariate data, use a cell array of vectors. If you provide an empty array, the function chooses a default value for `p` based on the data sites `x` and the default value of 1 for the roughness measure weight λ.

The smoothing parameter determines the relative weight to place on the contradictory demands of having *f* be smooth or having *f* be close to the data. For `p = 0`, *f* is the least-squares straight-line fit to the data. For `p = 1`, *f* is the variational, or natural, cubic spline interpolant. As `p` moves from 0 to 1, the smoothing spline changes from one extreme to the other.

The favorable range for `p` is often near $1/(1 + h^3/6)$, where *h* is the average spacing of the data sites. The function chooses a default value for `p` within this range. For uniformly spaced data, you can expect a close fit with $p = 1(1 + h^3/60)$ and some satisfactory smoothing with $p = 1/(1 + h^3/0.6)$. You can input `p > 1`, but this choice leads to a smoothing spline even rougher than the variational cubic spline interpolant.

If the input `p` is negative or empty, then the function uses the default value for `p`.

You can specify the roughness measure weights λ alongside the smoothing parameter by providing `p` as a vector. This vector must be the same size as `x`, with the `i`th entry the value of λ on the interval $(x(i-1)...x(i))$, for `i = 2:length(x)`. The first entry of the input vector `p` is the desired value of the smoothness parameter `p`. By providing roughness measure weights, you can make the resulting smoothing spline smoother (with larger weight values) or closer to the data (with smaller weight values) in different parts of the interval. Roughness measure weights must be nonnegative.

If you have difficulty choosing `p` but have some feeling for the size of the noise in `y`, consider using `spaps(x,y,tol)` instead. This function chooses `p` such that the roughness measure is as small as possible, subject to the condition that the error measure does not exceed `tol`. In this case, the error measure usually equals the specified value for `tol`.

Data Types: `single` | `double`

Output Arguments

st — Spline structure
spline structure

Spline, returned as a structure with these fields.

Form — Form of spline
st-tp00 | st-tp10 | st-tp01 | st-tp

Form of the spline, returned as `st-tp00`, `st-tp10`, `st-tp01`, or `st-tp`.

Centers — Sequence of sites
matrix | array

Sequence of sites, returned as a matrix or as an array for multivariate data.

Coefs — Coefficients of polynomials
matrix | array

Coefficients of polynomials for each piece, returned as a matrix or as an array for multivariate data.

Ncenters — Number of centers
scalar | vector

Number of sequence of sites.

Number — Number of polynomial pieces
scalar | vector

Number of polynomial pieces describing the spline, returned as a scalar or as a vector of numbers of pieces in each variable for multivariate data.

Dim — Dimensionality
scalar

Dimensionality of the target function, returned as a scalar.

Interv — Basic interval
cell array

Basic interval for the stform that contains all the given centers, returned as an array.

P — Smoothing parameter
scalar | cell array

Smoothing parameter used to calculate the spline, returned as a scalar or as a cell array of scalar values for multivariate data. P is between 0 and 1.

Limitations

The determination of the smoothing spline involves the solution of a linear system with as many unknowns as there are data points. Since the matrix of this linear system is full, the solving can take a long time even if, as is the case here, an iterative scheme is used when there are more than 728 data points. The convergence speed of that iteration is strongly influenced by p, and is slower the larger p is. So, for large problems, use interpolation, i.e., p equal to 1, only if you can afford the time.

See Also
csaps | spaps | stmak

Topics
About Splines in Curve Fitting on page 8-2
"Constructing and Working with stform Splines" on page 10-28
"Multivariate and Rational Splines" on page 10-6

Introduced in R2006b

type

Name of `cfit`, `sfit`, or `fittype` object

Syntax

```
name = type(fun)
```

Description

`name = type(fun)` returns the custom or library name `name` of the `cfit`, `sfit`, or `fittype` object `fun` as a character array.

Examples

```
f = fittype('a*x^2+b*exp(n*x)');
category(f)
ans =
custom
type(f)
ans =
customnonlinear

g = fittype('fourier4');
category(g)
ans =
library
type(g)
ans =
fourier4
```

See Also
category | fittype

Topics
"List of Library Models for Curve and Surface Fitting" on page 4-10

Introduced in R2006b

Bibliography

[1] Barber, C. B., D. P. Dobkin, and H. T. Huhdanpaa. "The Quickhull Algorithm for Convex Hulls." *ACM Transactions on Mathematical Software*. Vol. 22, No. 4, 1996, pp. 469–483.

[2] Bevington, P. R., and D. K. Robinson. *Data Reduction and Error Analysis for the Physical Sciences*. 2nd ed. London: McGraw-Hill, 1992.

[3] Branch, M. A., T. F. Coleman, and Y. Li. "A Subspace, Interior, and Conjugate Gradient Method for Large-Scale Bound-Constrained Minimization Problems." *SIAM Journal on Scientific Computing*. Vol. 21, No. 1, 1999, pp. 1–23.

[4] Carroll, R. J., and D. Ruppert. *Transformation and Weighting in Regression*. London: Chapman & Hall, 1988.

[5] Chambers, J., W. S. Cleveland, B. Kleiner, and P. Tukey. *Graphical Methods for Data Analysis*. Belmont, CA: Wadsworth International Group, 1983.

[6] Cleveland, W. S. "Robust Locally Weighted Regression and Smoothing Scatterplots." *Journal of the American Statistical Association*. Vol. 74, 1979, pp. 829–836.

[7] Cleveland, W. S., and S. J. Devlin. "Locally Weighted Regression: An Approach to Regression Analysis by Local Fitting." *Journal of the American Statistical Association*. Vol. 83, 1988, pp. 596–610.

[8] Daniel, C., and F. S. Wood. *Fitting Equations to Data*. Hoboken, NJ: Wiley-Interscience, 1980.

[9] DeAngelis, D. J., J. R. Calarco, J. E. Wise, H. J. Emrich, R. Neuhausen, and H. Weyand. "Multipole Strength in ^{12}C from the (e,e'α) Reaction for Momentum Transfers up to 0.61 fm^{-1}." *Physical Review C*. Vol. 52, No. 1, 1995, pp. 61–75.

[10] de Boor, C. *A Practical Guide to Splines*. Berlin: Springer-Verlag, 1978.

[11] Draper, N. R., and H. Smith. *Applied Regression Analysis*. 3rd ed. Hoboken, NJ: Wiley-Interscience, 1998.

[12] DuMouchel, W., and F. O'Brien. "Integrating a Robust Option into a Multiple Regression Computing Environment." *Computing Science and Statistics: Proceedings of the 21st Symposium on the Interface*. (K. Berk and L. Malone, eds.) Alexandria, VA: American Statistical Association, 1989, pp. 297–301.

[13] Goodall, C. "A Survey of Smoothing Techniques." *Modern Methods of Data Analysis*. (J. Fox and J. S. Long, eds.) Newbury Park, CA: Sage Publications, 1990, pp. 126–176.

[14] Holland, P. W., and R. E. Welsch. "Robust Regression Using Iteratively Reweighted Least-Squares." *Communications in Statistics—Theory and Methods*. Vol. A6, 1977, pp. 813–827.

[15] Huber, P. J. *Robust Statistics*. Hoboken, NJ: Wiley-Interscience, 1981.

[16] Hutcheson, M. C. "Trimmed Resistant Weighted Scatterplot Smooth." Master's Thesis. Cornell University, Ithaca, NY, 1995.

[17] Levenberg, K. "A Method for the Solution of Certain Problems in Least Squares." *Quarterly of Applied Mathematics*. Vol. 2, 1944, pp. 164–168.

[18] Marquardt, D. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters." *SIAM Journal on Applied Mathematics*. Vol. 11, 1963, pp. 431–441.

[19] Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[20] Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge, UK: Cambridge University Press, 1993.

[21] Street, J. O., R. J. Carroll, and D. Ruppert. "A Note on Computing Robust Regression Estimates Via Iteratively Reweighted Least Squares." *The American Statistician*. Vol. 42, 1988, pp. 152–154.

[22] Watson, David E. *Contouring: A Guide to the Analysis and Display of Spatial Data*. Tarrytown, NY: Pergamon, 1992.