Spline Toolbox

For Use with MATLAB®

Carl de Boor

Computation

Visualization

Programming

The MATH WORKS Inc.

User's Guide

Version 3

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

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

`http://www.mathworks.com` Web
`ftp.mathworks.com` Anonymous FTP server
`comp.soft-sys.matlab` Newsgroup

`support@mathworks.com` Technical support
`suggest@mathworks.com` Product enhancement suggestions
`bugs@mathworks.com` Bug reports
`doc@mathworks.com` Documentation error reports
`subscribe@mathworks.com` Subscribing user registration
`service@mathworks.com` Order status, license renewals, passcodes
`info@mathworks.com` Sales, pricing, and general information

Spline Toolbox User's Guide

Printing History:

March 1990	First printing
November 1992	Second printing
January 1998	Third printing
January 1999	Revised for Version 2.0.1 (Release 11) (Online only)
September 2000	New for Version 3.0 (Release 12)

Contents

Preface

1 Tutorial

2 Function Reference

A Glossary

Preface

What Is the Spline Toolbox?

This toolbox contains MATLAB versions of the essential 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), 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 `f`unction. 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 *PGS*.

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, MATLAB's sparse matrix facilities are used in the Spline Toolbox 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 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 demos, all accessible through MATLAB's demo interface, i.e., by the command `demos`, and you are strongly urged to have a look at some of them, like the GUI `splinetool`, before attempting to use this toolbox, or even before reading on.

Related Products

MATLAB 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 Spline Toolbox. 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 outputs values of the interpolating tensor product cubic spline at the grid specified by `y1,...,yd`.

Further, any of the `fn...` commands of the Spline Toolbox can be applied to the output of MATLAB's `spline(x,y)` command, with simple versions of the Spline Toolbox commands `fnval`, `ppmak`, `fnbrk` available directly in MATLAB, as the commands `ppval`, `mkpp`, `unmkpp`, respectively.

Related Products List

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with the Spline Toolbox. In particular, the Spline Toolbox *requires*:

- MATLAB®

Note Version 3.0 of the Spline Toolbox requires Release 11.0 of MATLAB or later, for the GUIs to run properly.

For more information about MATLAB or any of the related products, see either:

- The online documentation for that product if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the "products" section

The products listed below complement the functionality of the Spline toolbox.

Product	Description
Financial Time Series Toolbox	Tool for analyzing time series data in the financial markets
Financial Toolbox	MATLAB functions for quantitative financial modeling and analytic prototyping
Neural Network Toolbox	Comprehensive environment for neural network research, design, and simulation within MATLAB
Optimization Toolbox	Tool for general and large-scale optimization of nonlinear problems, as well as for linear programming, quadratic programming, nonlinear least squares, and solving nonlinear equations

Using This Guide

Expected Background

The toolbox started out as an extension of MATLAB 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 *PGS*.

However, the basic toolbox 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 the one entitled “Some Simple Examples”, 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.

A Glossary at the end of this Manual provides definitions of almost all the mathematical terms used in this document.

Organization of the Document

The main body of this guide comprises two chapters, and these are preceded by the present Notes to the Reader. Here, in tabular form, are the high points.*

Chapter	Description
Chapter 1	Tutorial The first section provides specific examples illustrating simple uses of the toolbox commands. The second section provides an overview of splines and spline approximation methods. The next two sections offer detailed descriptions of the two basic ways used in the toolbox to describe a piecewise-polynomial aka spline: the ppform and the B-form. This is followed by a section concerning the use made in this toolbox of tensor products for multivariate work and a section on NURBs and other rational splines. The remaining sections are meant for users interested in developing their own spline commands. Three specific computational tasks are discussed in detail: •Construction of the Chebyshev spline •Approximate solution of a nonlinear ordinary differential equation boundary value problem with a boundary layer •The use of univariate vector-valued splines in the construction and use of approximations to multivariate data

Chapter	Description
Chapter 2	Reference Begins with an ordered tabulation of the commands in this toolbox, followed by reference pages for those commands.
Glossary	Provides a briefing on the basic technical terms and concepts used in this guide.

Technical Conventions

Vectors. the Spline Toolbox 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*, that is, 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, that is, a d-vector, is always stored as a column vector. In particular, if you wish 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.

Naming conventions. Most of the 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 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 start particular demos

Some of these naming conventions are the result of discussions with J\"org Peters.

Typographical Conventions

We use some or all of these conventions in our manuals.

Item	Convention to Use	Example
Example code	`Monospace font`	To assign the value 5 to `A`, enter `A = 5`
Function names/syntax	`Monospace font`	The cos function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Keys	**Boldface** with an initial capital letter	Press the **Return** key.
Literal strings (in syntax descriptions in Reference chapters)	**`Monospace bold`** for literals.	`f = freqspace(n,'whole')`
Mathematical expressions	Variables in *italics* Functions, operators, and constants in standard text.	This vector represents the polynomial $p = x^2 + 2x + 3$

MATLAB output	`Monospace font`	MATLAB responds with
		`A =`
		`5`

Menu names, menu items, and controls	**Boldface** with an initial capital letter	Choose the **File** menu.
New terms	*Italics*	An *array* is an ordered collection of information.
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd, 'method')`

Tutorial

Some Simple Examples

Here are some simple ways to make use of the commands in this toolbox. More complicated examples are given in later sections. Other examples are available in the various demos, all of which can be reached by MATLAB's `demos` command. In addition, the command `splinetool` provides a GUI for you to try several of the basic spline interpolation and approximation commands from this toolbox on your data; it even provides various instructive data sets.

Check the reference pages if you have specific questions about the use of the commands mentioned. Check the Glossary if you have specific questions about the terminology used; a look into the index may help.

Suppose you want to interpolate to some smooth data, for example to

```
x = (4*pi)*[0 1 rand(1,20)]; y = sin(x);
```

Then you could try the cubic spline interpolant obtained by

```
cs = csapi(x,y);
```

and plotted, along with the data, by

```
fnplt(cs); hold on, plot(x,y,'o')
legend('cubic spline','data'), hold off
```

This produces a figure like the following:

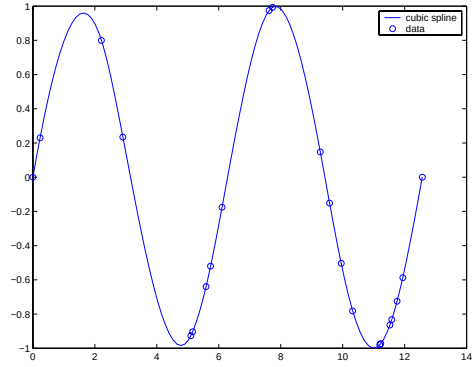


Figure 1-1: Cubic Spline Interpolant to Some Smooth Data

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 MATLAB's command `spline(x,y)`.

We know that the sine function is 2π-periodic. To check how well our interpolant does on that score, we compute, e.g., the difference in the value of its first derivative at the two endpoints,

```
» diff(fnval(fnder(cs),[0 4*pi]))
ans = -.0307
```

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 `cspape` which permits specification of many different kinds of end conditions, including periodic end conditions. So, use instead

```
pcs = csape(x,y,'periodic');
```

for which we get

```
» diff(fnval(fnder(pcs),[0 4*pi]))
ans = -2.2204e-016
```

as the difference of end slopes. Even the difference in end second derivatives is small:

```
» diff(fnval(fnder(pcs,2),[0 4*pi]))
ans = -1.4017e-014
```

Other end conditions can be handled as well. For example,

```
cs = csape(x,y,[1 2],[3 -4]);
```

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.

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 our data, as obtained by the statements

```
sp2 = spapi(2,x,y); fnplt(sp2), hold on
sp3 = spapi(3,x,y); fnplt(sp3,'--')
sp5 = spapi(5,x,y); fnplt(sp5,'-.'), plot(x,y,'o')
legend('linear','quadratic','quartic','data'), hold off
```

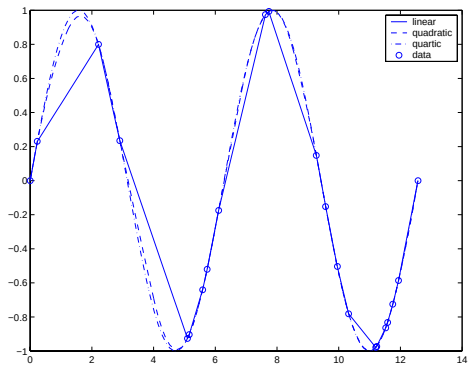


Figure 1-2: Spline Interpolants of Various Orders to Smooth Data

Even the cubic spline interpolant obtained from `spapi` is different from the one provided by `csapi` and `spline`. To emphasize their difference, we compute and plot their second derivatives, as follows:

```
fnplt(fnder(spapi(4,x,y),2)), hold on
fnplt(fnder(csapi(x,y),2),1.5),plot(x,zeros(size(x)),'o')
legend('from spapi','from csapi','data sites'), hold off
```

This gives the following graph:

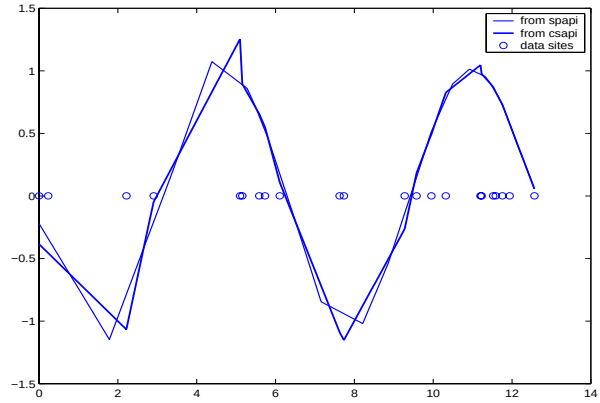


Figure 1-3: Second Derivative of Two Cubic Spline Interpolants to the Same Smooth Data

Since the second derivative of a cubic spline is a broken line, with vertices at the breaks of the spline, we can see clearly that `csapi` places breaks at the data sites, while `spapi` does not, and thereby produces a less jerky second derivative in this example.

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 we 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. One of the first things done in `spapi` is to check whether the data site array is nondecreasing and to reorder it to make it so if need be, in which case the data

value array is reordered in the same way. In this way, `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 the section entitled "The B-form" for a discussion of the command `augknt` used here to generate the appropriate "knot sequence".

What if the data are noisy? For example, suppose that the given values are

```
noisy = y + .1*(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), hold on, plot(x,noisy,'o')
legend('smoothing spline','noisy data'), hold off
```

This produces a figure like this:

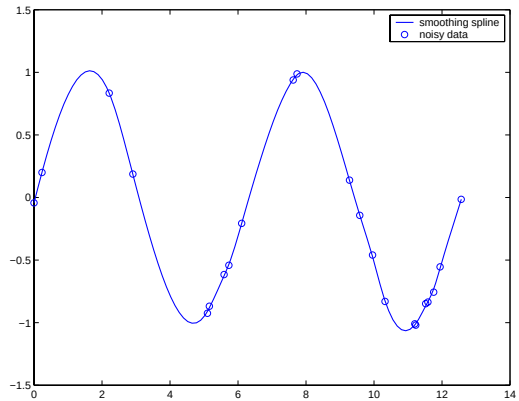


Figure 1-4: Cubic Smoothing Spline to Noisy Data

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), hold on
fnplt(csaps(x,noisy,p/2),'--')
fnplt(csaps(x,noisy,(1+p)/2),':'), plot(x,noisy,'o')
legend('smoothing spline','more smoothed','less smoothed',...
'noisy data'), hold off
```

This produces the following picture.

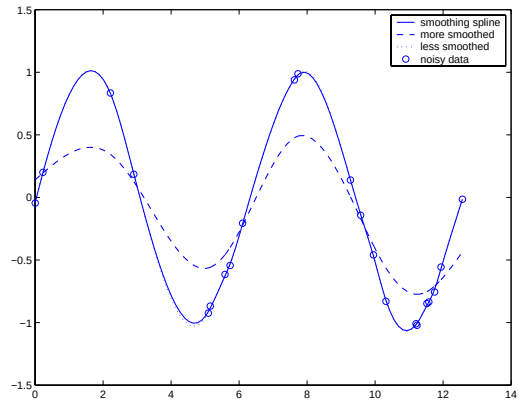


Figure 1-5: Noisy Data More or Less Smoothed

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
```

This spline is provided by the command

```
sp = spaps(x,noisy,tol);
```

If a least-squares approximant is wanted instead, it is provided 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);
```

If `f` is one of these splines `cs`, `ch`, or `sp` so constructed, then, as we saw already, it can be displayed by the statement

```
fnplt(f)
```

Its value at `a` is given by the statement

```
fnval(f,a);
```

Its second derivative is constructed by the statement

```
DDf = fnder(fnder(f));
```

or by the statement

```
DDf = fnder(f,2);
```

Its definite integral over the interval `[a..b]` is supplied by the statement

```
[1 -1]*fnval(fnint(f),[b;a]);
```

and the difference between the spline in `cs` and the one in `ch` can be computed as

```
fncmb(cs,'-',sp);
```

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 statements

```
xy = [x;y]; df = diff(xy.').';
t = cumsum([0, sqrt([1 1]*(df.*df))]);
cv = csapi(t,xy);
```

provide such a spline curve, using chord-length parametrization and cubic spline interpolation with the not-a-knot end condition, as can be verified by the statements

```
fnplt(cv), hold on, plot(x,y,'o'), hold off
```

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)))
```

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]))
```

Both plots are shown below.

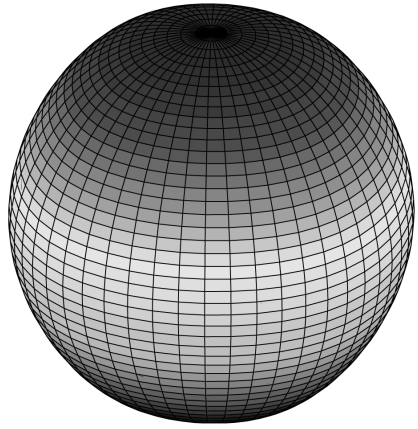


Figure 1-6: A Sphere Made by a 3-D-Valued Bivariate Tensor Product Spline

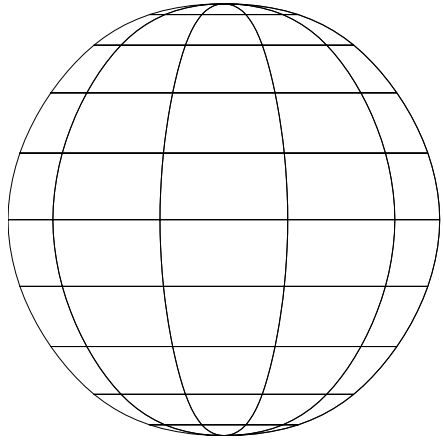


Figure 1-7: A Planar Projection of the Above Spline Sphere

Splines: An Overview

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 $s(x) := p_j(x)$ for $x \in [\xi_j..\xi_{j+1}]$, all j, has several continuous derivatives. Any such smooth piecewise polynomial function is called a *spline*. I.J. Schoenberg coined this term since 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 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 splines are just two different views of the same thing.

The *ppform* of a spline of *order k* provides a description in terms of its *breaks* $\xi_1, \ldots, \xi_{l+1}$ and the *local polynomial coefficients* c_{ji} of its l pieces.

$$p_j(x) = \sum_{i=1}^{k} (x-\xi_j)^{k-i} \; c_{ji}, \qquad i = 1{:}n.$$

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.

The *B-form* has become the standard way to represent a spline during its *construction*, since 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, $B_{j,k} := B(\cdot \mid t_j, \ldots, t_{j+k})$ 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

$$\sum_{j=1}^{n} B_{j,k}(x) = 1 \quad \text{on } [t_k .. t_{n+1}]$$

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. All these properties of the B-spline are nicely visualized and experimented with interactively with the command `bspligui`.

Since $B_{j,k}$ 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, if a spline s of order k with knot sequence $t_1 \le t_2 \le \cdots \le t_{n+k}$ is to be constructed so that $s(x_i) = y_i$ for $i=1,\ldots,n$, then we are led to the linear system

$$\sum_{j=1}^{n} B_{j,k}(x_i) a_j = y_i \qquad i = 1{:}n,$$

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 $t_1, \ldots, t_{n+k}$ if and only if $B_{j,k}(x_j) \neq 0$ for all j (Schoenberg-Whitney Conditions). Computations with B-splines are facilitated by stable *recurrence relations*

$$B_{j,k}(x) = \frac{x - t_j}{t_{j+k-1} - t_j} B_{j,k-1}(x) + \frac{t_{j+k} - x}{t_{j+k} - t_{j+1}} B_{j+1,k-1}(x)$$

that 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.

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 mth 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* $s = s_\lambda$ which, for given data (x_i,y_i) with $x_i \in [a..b]$, all i, and given corresponding positive weights w_i, and for given *smoothing parameter* λ, minimizes

$$\sum_i w_i (y_i - f(x_i))^2 + \lambda \int_a^b (D^m f(t))^2 \, dt$$

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 art of using the

smoothing spline consists in choosing λ so that s contains as much of the information, and as little of the supposed noise, in the data as possible.

Multivariate splines can be obtained from univariate splines by the tensor product construct. For example, a trivariate spline in B-form is given by

$$\sum_{u=1}^{U} \sum_{v=1}^{V} \sum_{w=1}^{W} B_{u,k}(x) B_{v,l}(y) B_{w,m}(z) a_{u,v,w}$$

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 3-space.

The ppform

A univariate *piecewise polynomial* f is specified by its *break sequence* `breaks` and the *coefficient array* `coefs` of the local power form (see (*) below) of its polynomial pieces; -- see the section on Tensor Product Splines for a discussion of multivariate piecewise-polynomials. The coefficients may be `d`-vectors, usually 2-vectors or 3-vectors, in which case f is a curve in 2-space or 3-space. To comply with the standard treatment of vectors in this toolbox, any coefficient and any value of a spline curve is always written as a 1-*column* matrix. 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 `j`-th polynomial piece, from the highest to the lowest power; see (*) below.

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))
```

for $\text{breaks(j)} \le t < \text{breaks(j+1)}$.

Here, `polyval(a,x)` is the MATLAB function; it returns the number

$$\sum_{j=1}^{k} \text{a}(j)\text{x}^{k-j} = \text{a}(1)\ \text{x}^{k-1} + \text{a}(2)\ \text{x}^{k-2} + \cdots + \text{a}(k)\ \text{x}^{0}.$$

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

$$f(t) = \text{polyval(coefs(j,:)}, t\text{-breaks(j))}, \qquad \text{j} = \begin{matrix} 1, & t < \text{breaks}(1) \\ \text{l}, & t \geq \text{breaks(l+1)} \end{matrix}$$

i.e., by extending the first, respectively last, polynomial piece.

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, we might say `pp=ppmak(-5:-1,-22:-11)`, or, more explicitly,

```
breaks = -5:-1;
coefs = -22:-11;
pp = ppmak(breaks,coefs);
```

thus supplying the uniform break sequence `-5:-1` and the coefficient sequence `-22:-11`. Since this break sequence has 5 entries, hence 4 break intervals, while the coefficient sequence has 12 entries, we 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 the Spline Toolbox 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.

Here are some 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

`pj=fnbrk(pp,j)` pulls out the `j`-th polynomial piece

`pc=fnbrk(pp,[a b])` restricts/extends to the interval [a..b]

`fnplt(pp)` plots piecewise-polynomial on its basic interval

`sp = fn2fm(pp,'B-')` converts to B-form

`pr = fnrfn(pp,morebreaks)` inserts additional breaks

Inserting additional breaks comes in handy when one wants to add two piecewise-polynomials with different breaks, as is done in the command `fncmb`.

To illustrate the use of some of these commands, here is a plot of the particular piecewise-polynomial we just made up. First, the basic plot:

```
x = linspace(-5.5,-.5,101);
plot(x, fnval(pp,x),'x')
```

Then add to the plot the breaklines:

```
breaks=fnbrk(pp,'b'); yy=axis; hold on
for j=1:fnbrk(pp,'l')+1
   plot(breaks([j j]),yy(3:4))
end
```

Finally, superimpose on that plot 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
```

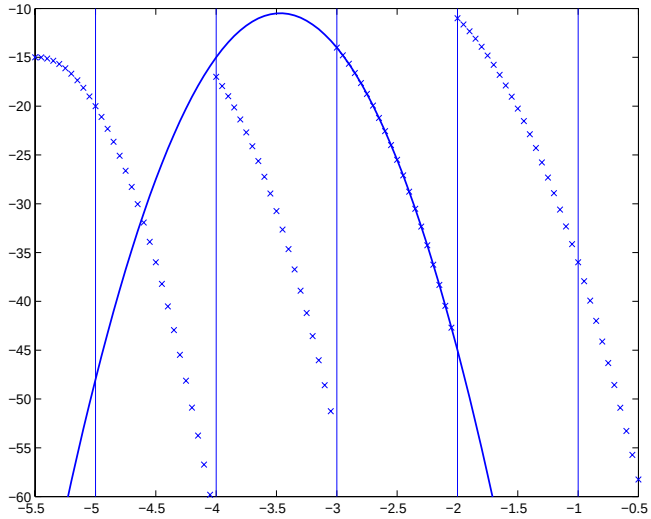


Figure 1-8: A Piecewise-Polynomial Function, Its Breaks, and the Polynomial Giving Its Third Piece

Figure 1-8 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.

The B-form

A univariate spline f is specified by its nondecreasing knot sequence `t` and by its B-spline coefficient sequence `a`; -- see the section on Tensor Product Splines for a discussion of multivariate splines. The coefficients may be `d`-vectors, usually 2-vectors or 3-vectors and required to be written as 1-column matrices, in which case f is a curve in 2-space or 3-space and the coefficients are called the *control points* for the curve.

Roughly speaking, such a spline is 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* since it takes four coefficients to specify a cubic polynomial. These four items, `t`, `a`, `n`, and `k`, make up the B-form of the spline *f*.

This means, explicitly, that

$$f = \sum_{i=1}^{n} B_{ik}\,\mathrm{a}(:, i)$$

with $B_{i,k} = B(\cdot \mid \mathrm{t}(i{:}i{+}\mathrm{k}))$ the *i*th 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 since, outside its basic interval, a piecewise-polynomial is defined by extension of its first or last polynomial piece.

The building blocks for the B-form of a spline are the B-splines. The Figure 1-9 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])
```

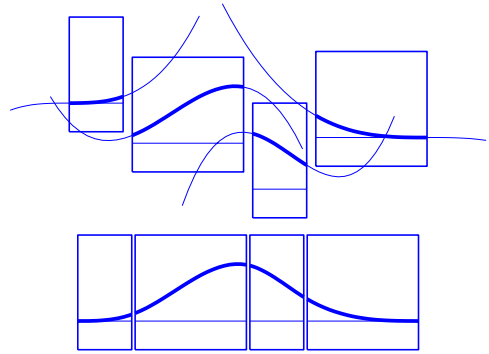


Figure 1-9: A B-Spline of Order 4, and the Four Cubic Polynomials from Which It Is Made

To summarize: The B-spline with knots $t(i) \le \cdots \le t(i+k)$ 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. The rule is:

knot multiplicity + condition multiplicity = order

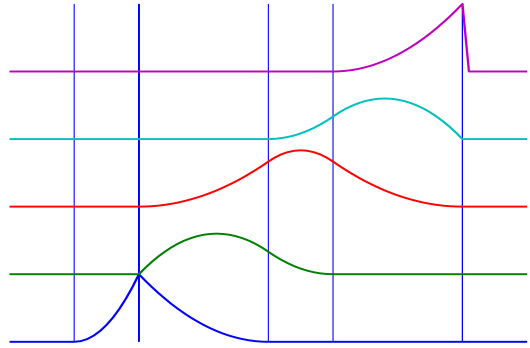


Figure 1-10: All Third-Order B-Splines for a Certain Knot Sequence with Various Knot Multiplicities

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.

Figure 1-10 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 demo `spalldem`.

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.,

$$\text{jump}_{\xi_i} D^j s := D^j s(\xi_i+) - D^j s(\xi_i-) = 0, \qquad 0 \le j < \mu_i, \qquad i = 2, \ldots, l$$

Then, the appropriate knot sequence t should contain the break ξ_i exactly $k - \mu_i$ times, $i = 2, \ldots, l$. 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] );
```

The shorthand

$$f \in S_{\mathrm{k,t}}$$

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 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.

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, we might say

```
sp = spmak(1:10,3:8);
```

thus supplying the uniform knot sequence `1:10` and the coefficient sequence `3:8`. Since there are 10 knots and 6 coefficients, the order must be 4(= 10 - 6), i.e., we 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 Spline Toolbox 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`.

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 (.95, .95)$/\sqrt{1.9}$ would make a visually perfect circle.

Here are more details. The spline curve generated has the form

$\Sigma_{j=1}^{8} B_{j,5}\,\mathrm{a}(:,j)$, with `[-4:8]` the uniform knot sequence, and with its control points `a(:,`*j*`)` the sequence (0, α), (-α, 0), (0, -α), (α, 0), (0, α) (-α, 0), (0, -α), (α, 0) with α = .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, we compute its unsigned curvature. The curvature $\kappa(t)$ at the curve point $\gamma(t)$ of a space curve γ can be computed from the formula

$$\kappa(t) = \mathrm{norm}\{\gamma'(t) \times \gamma''(t)\} / \mathrm{norm}\{\gamma'(t)\}^3,$$

in which norm{a} is the Euclidean length of the 3-vector a, and $a \times b$ is the cross product of the two 3-vectors a and b, and γ' and γ'' are the first and second derivative of the curve with respect to the parameter used. We treat our 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 we see from the next calculation:

```
1/norm(fnval(sp,0))
ans = 1.7864
```

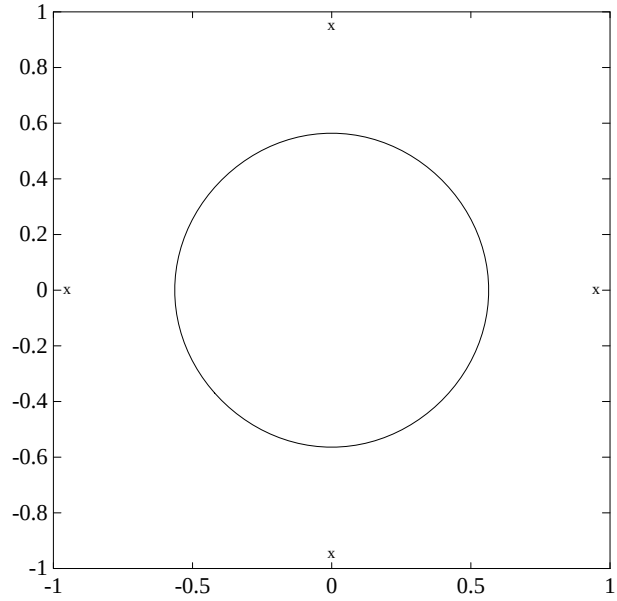


Figure 1-11: Spline Approximation to a Circle; Control Points Are Marked x

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, plot, or refine a spline with the aid of `fnval`, `fnder`, `fndir`, `fnint`, `fnplt`, and `fnrfn`.

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

For display of a spline *curve* with given two-dimensional coefficient sequence and a uniform knot sequence, there is `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 rth derivative at τ of the jth 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 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`.

The remaining commands involving the B-form are utilities, of no interest to the casual user.

Tensor Product Splines

The toolbox provides 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. The same commands are used for their construction and use.

For simplicity, the discussion to follow deals just with bivariate 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, we obtain a bivariate spline as

$$f(x, y) = \sum_{i=1}^{m} \sum_{j=1}^{n} B(x|s_i, \ldots, s_{i+h}) B(y|t_j, \ldots, t_{j+k}) a_{ij}$$

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`, `fn2fm` can be used to plot, evaluate, differentiate and integrate, refine, and convert this form.

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});
```

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.

Here is an example of a surface constructed as a 3-D-valued bivariate spline. The surface is the famous Moebius 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)
hold off
```

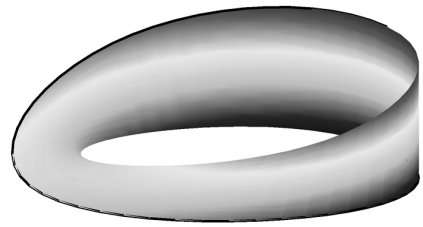


Figure 1-12: A Moebius Band Made by Vector-Valued Bivariate Spline Interpolation

NURBS and Other 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. 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; have a look at the resulting plot, in Figure 1-13.

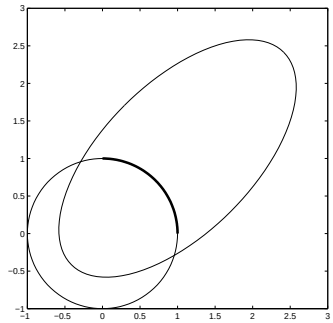


Figure 1-13: A Circle and an Ellipse, Both Given By a Rational Spline

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. We highlight the first such piece, by 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 Figure 1-13, with linewidth `3` to make it stand out.

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 2/3 of that sphere, as shown in Figure 1-14.

```
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
```

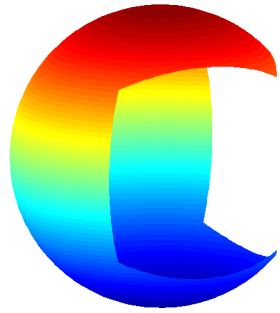


Figure 1-14: Part of a Sphere Formed by Four Rotates of a Quartic Rational

Offhand, the two splines, s and w, in the rational spline $r(x) = s(x)/w(x)$ need not at all 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, we can (and do) 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. It is very easy to obtain $r(x)$ from $R(x)$. 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. 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' 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 we 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 j-th derivative of r. There is a corresponding formula for partial and directional derivatives for multivariate 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 Spline Toolbox, 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.

A special rational spline, 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 B-form, with each coefficient for *s* containing explicitly the corresponding coefficient for *w* as a factor:

$$s = \sum_i B_i v(i) a(:, i), \qquad w = \sum_i B_i v(i)$$

The normalized coefficients $a(:,i)$ for the numerator spline are more readily used as control points than the unnormalized coefficients $v(i)a(:,i)$. 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 commands 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].

Example: A Nonlinear ODE

The following sample can be run via `difeqdem`.

We consider the nonlinear singularly perturbed problem

$$\varepsilon D^2 g(x) + (g(x))^2 = 1 \text{ on } [0..1]$$

$$Dg(0) = g(1) = 0.$$

We seek an approximate solution by collocation from C^1 piecewise cubics with a suitable break sequence; for instance,

```
breaks = [0:4]/4;
```

Since cubics are of order 4, we have

```
k = 4;
```

and since C^1 requires two smoothness conditions across each interior break, we want knot multiplicity =4 - 2 =2, hence use the knot sequence

```
knots = sort([0 0 breaks breaks 1 1]);
```

which we could also have obtained as `knots = augknt(breaks,4,2)`. This gives a quadruple knot at both 0 and 1, which is consistent with the fact that we have cubics, i.e., have order 4.

This implies that we have

```
n = length(knots)-k
```

= 10 degrees of freedom. We 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.

We choose the two Gauss sites for each interval. For the *standard* interval [-1/2 . . 1/2] of length 1, these are the two sites

```
gauss = [-1; 1]/(sqrt(3)*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);
```

```
colpnts = temp(:).';
```

With this, the numerical problem we want to solve is to find $y \in S_{4,\text{knots}}$ that satisfies the nonlinear system

$$\begin{aligned} Dy(0) &= 0 \\ (y(x))^2 + \varepsilon D^2 y(x) &= 1 \quad \text{for } x \in \text{colpnts} \\ y(1) &= 0 \end{aligned}$$

If y is our current approximation to the solution, then the linear problem for the supposedly better solution z by Newton's method reads

$$\begin{aligned} Dz(0) &= 0 \\ w_0(x)z(x) + \varepsilon D^2 z(x) &= b(x) \quad \text{for } x \in \text{colpnts} \\ z(1) &= 0 \end{aligned}$$

with $w_0(x) := 2y(x)$, $b(x) := (y(x))^2 + 1$. In fact, by choosing

$$\begin{aligned} w_0(1) &:= 1, \quad w_1(0) := 1 \\ w_1(x) &:= 0, \quad w_2(x) := \varepsilon \quad \text{for } x \in \text{colpnts} \end{aligned}$$

and choosing all other values of w_0, w_1, w_2, b not yet specified to be zero, we can give our system the uniform shape

$$w_0(x)z(x) + w_1(x)Dz(x) + w_2(x)D^2 z(x) = b(x), \text{ for } x \in \text{sites}$$

with

```
sites = [0,colpnts,1];
```

Since $z \in S_{4,\text{knots}}$, we convert this last system into a system for the B-spline coefficients of z. This requires the values, first, and second derivatives at every $x \in \text{points}$ and for all the relevant B-splines. The command `spcol` was expressly written for this purpose.

We use `spcol` to supply the matrix

```
colmat = ...
```

```
spcol(knots,k,reshape([sites(1 1 1],:),1,(length(sites)));
```

From this, we 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, we need a current approximation y. Initially, we get it by interpolating some reasonable initial guess from our piecewise-polynomial space at the `sites`. We use the parabola $()^2 - 1$ (i.e., the function $x \mapsto x^2-1$) that 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+[1:8]*3,1+9*3],:);
coefs = intmat\[0 colpnts.*colpnts-1 0].';
y = spmak(knots,coefs.');
```

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 *small enough*. We choose a relatively mild $\varepsilon = .1$.

```
epsilon =.1;
tolerance=1.e-9;
while 1
  vtau=fnval(y,colpnts);
  weights = [0 1 0;
             [2*vtau.'zeros(8,1) epsilon*ones(8,1)];
             1 0 0];
  colloc = zeros(10,10);
  for j=1:10
     colloc(j,:)=weights(j,:)*colmat(3*(j-1)+[1:3],:);
  end
  coefs = colloc\[0 vtau.*vtau+1 0].';
  z = spmak(knots,coefs.');
  maxdif = max(abs(z-y))
if maxdif<tolerance, break, end
    y = z;
end
```

The resulting printout of the errors

```
maxdif = 0.2067
maxdif = 0.0121
```

```
maxdif = 3.9515e-005
maxdif = 4.4322e-010
```

shows the quadratic convergence expected from Newton's method. The plot below shows the initial guess and the computed solution, as the two leftmost curves. Note that the computed solution, like the exact solution, does *not* equal -1 at 0.

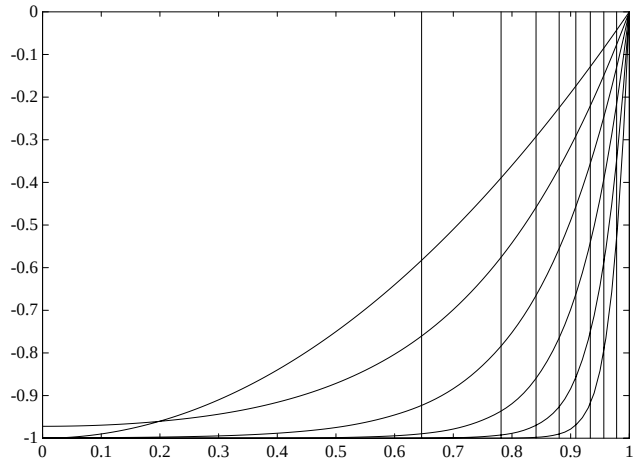


Figure 1-15: Solutions of a Nonlinear ODE with Increasingly Strong Boundary Layer

If we now decrease ε, 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;
```

From the new break sequence, we 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];
```

We use `spcol` to supply the matrix

```
colmat = spcol(knots,k,sort([sites sites sites]));
```

and use our 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, we cut 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 plot above. The plot also shows the final break sequence, as a sequence of vertical bars.

In this example, at least, `newknt` has performed satisfactorily.

Example: Construction of the 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 $C = C_t \in S_{k,t}$ 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 $t_i < \tau_i < t_{k+i}$, 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 $S_{k,t}$ since the norm of the resulting projector is about as small as can be; see the toolbox command `chbpnt`.

In this example, which can be run via `chebdem`, we try to construct C for a particular knot sequence t.

We 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))]);
```

Note the quadruple knot at each end. Since `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);
```

As our initial guess for the τ, we use the knot averages

$$\tau_i = (t_{i+1} + \cdots + t_{i+k-1})/(k-1)$$

recommended as good interpolation site choices. These are supplied by

```
tau=aveknt(t,k);
```

We plot the resulting first approximation to *C*, i.e., the spline *c* that satisfies $c(\tau_i) = (-1)^{n-i}$, all *i*:

```
b = (-ones(1,n)).^[n-1:-1:0];
c = spapi(t,tau,b);
fnplt(c,'-.')
grid
```

Here is the resulting picture:

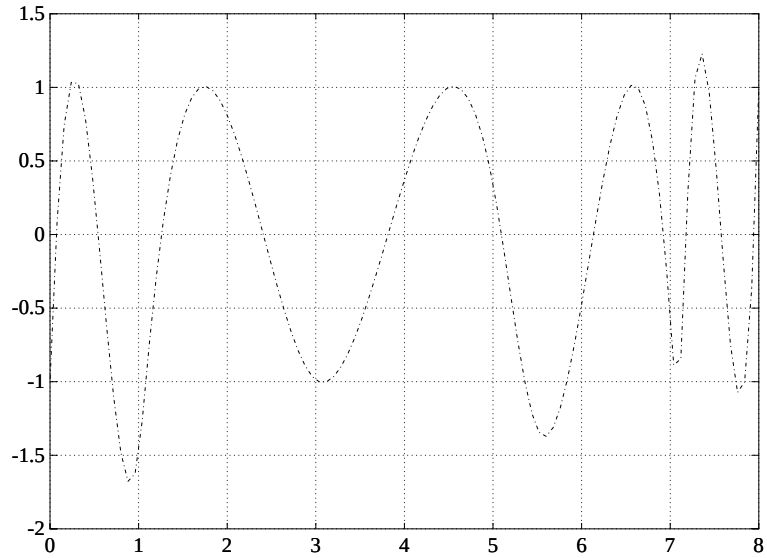


Figure 1-16: First Approximation to a Chebyshev Spline

Starting from this approximation, we use the Remez algorithm to produce a sequence of splines converging to *C*. This means that we construct new τ as the

extrema of our current approximation *c* to *C* and try again. Here is the entire loop.

We find the new interior τ_i as the zeros of *Dc* := the first derivative of *c*, in several steps. First, we differentiate:

```
cp = fnder(c);
```

Next, we take the zeros of the control polygon of *Dc* as our first guess for the zeros of *Dc*. For this, we must take apart the spline `cp`.

```
[knots,coefs,np,kp] = spbrk(cp);
```

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 `cp`:

```
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.

We refine this estimate for the zeros of *Dc* by two steps of the secant method, taking `tau` and this `guess` as our first approximations. First, we 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(cp,sites(1:2,:)),2,n-2);
```

Now come two steps of the secant method. We guard against division by zero by setting the function value difference to 1 in case it is zero. Since *Dc* is strictly monotone near the sites sought, this is harmless:

```
for j=2:3
  rows = [j,j-1];cpd=diff(values(rows,:));
  cpd(find(cpd==0)) = 1;
  sites(j+1,:) = sites(j,:) ...
           -values(j,:).*(diff(sites(rows,:))./cpd);
  values(j+1,:) = fnval(cp,sites(j+1,:));
end
```

The check

```
max(abs(values.'))
  ans = 4.1176 5.7789 0.4644 0.1178
```

shows the improvement.

Now we take these sites as our new tau,

```
tau = [tau(1) sites(4,:) tau(n)];
```

and check the extrema values of our current approximation there:

```
extremes = abs(fnval(c, tau));
```

The difference

```
max(extremes)-min(extremes)
  ans = 0.6905
```

is an estimate of how far we are from total leveling.

We construct a new spline corresponding to our 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)
```

Here is the resulting picture:

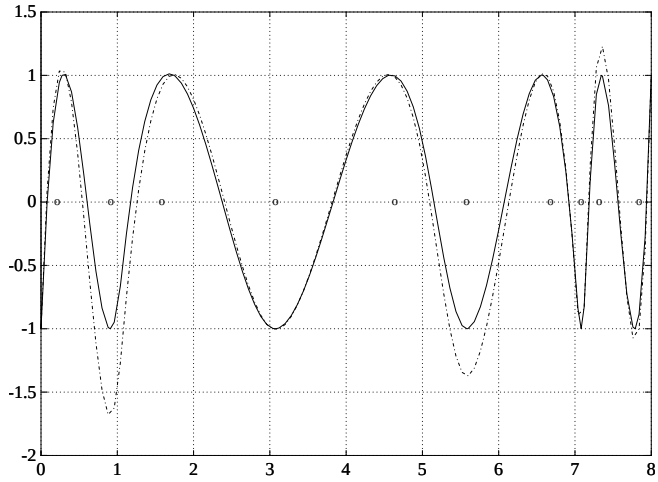


Figure 1-17: A More Nearly Level Spline

If this is not close enough, one simply reiterates the loop. For this example, the next iteration already produces *C* to graphic accuracy.

Example: Approximation by Tensor Product Splines

Since 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, run by `tspdem`, is meant to show.

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.

Consider, for example, least-squares approximation to given data `z(`*i*`,`*j*`)` = *f*(`x(`*i*`),y(`*j*`)`), *i* = `1:Nx`, *j* = `1:Ny`. We 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. We choose a few more data sites in the *x*-direction than the *y*-direction; also, for a better definition, we 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);
```

We 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, we choose to approximate this function by a vector-valued parabolic spline, with three uniformly spaced interior knots. This means that we 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 us with the least-squares approximant to the data:

```
sp = spap2(knotsy,ky,y,z);
```

In effect, we are finding simultaneously the discrete least-squares approximation from $S_{\mathtt{ky},\mathtt{knotsy}}$ to each of the `Nx` data sets

$$(y(j), z(i,j))_{j=1}^{Ny}, \quad i = 1{:}Nx.$$

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*`)`) since `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 MATLAB's matrix-oriented view when plotting an array. This can be a serious problem in bivariate approximation since 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*`)`).

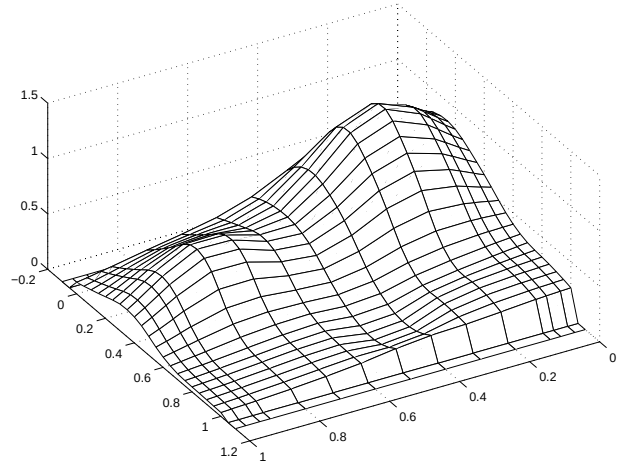


Figure 1-18: A Family of Smooth Curves Pretending to Be a Surface

Note that both the first two and the last two values on each smooth curve are actually zero since 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 we are plotting smooth curves in one direction only.

To get an actual surface, we now have to go a step further. Look at the coefficients coefsy of the spline in sp:

```
coefsy = fnbrk(sp,'c');
```

Abstractly, you can think of the spline in sp as the function

$$y \mapsto \sum_r \text{coefsy}(:,r)\, B_{r,\text{ky}}(y)$$

with the *i*th entry coefsy(*l*, *r*) of the vector coefficient coefsy(:,*r*) corresponding to x(*i*), all *i*. This suggests approximating each coefficient vector coefsy(*r*,:) by a spline of the same order kx and with the same appropriate knot sequence knotsx. Again for no particular reason, we choose this time to 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. Since we wanted to fit the datum coefsy(:,*r*) at x(*r*), all *r*, we had to present spap2 with the *transpose* of coefsy.

Now consider the transpose of the coefficients cxy of the resulting spline *curve*:

```
coefs = fnbrk(sp2,'c').';
```

It provides the *bivariate* spline approximation

$$(x,y) \mapsto \sum_q \sum_r \text{coefs}(q,r)\, B_{q,\text{kx}}(x)\, B_{r,\text{ky}}(y)$$

to the original data

(x(*i*),y(*j*)) ↦ 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.

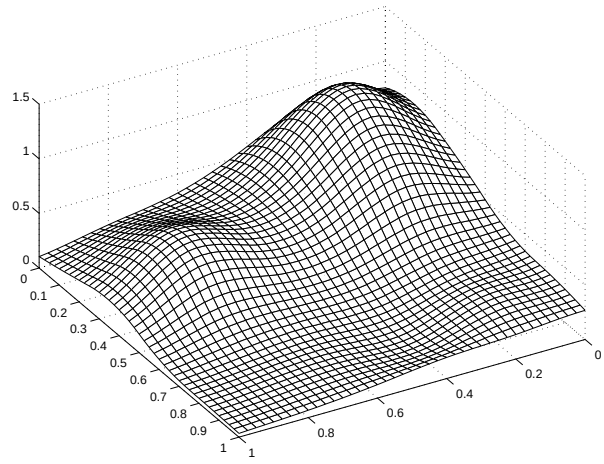


Figure 1-19: Spline Approximation to Franke's Function

This makes good sense since `spcol(knotsx,kx,xv)` is the matrix whose (i,q)-entry equals the value $B_{q,\mathrm{kx}}(\mathrm{xv}(i))$ at `xv`(i) of the qth B-spline of order `kx` for the knot sequence `knotsx`.

Since 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))) )
0.0539
```

This is perhaps not too impressive. On the other hand, we used only a coefficient array of size

```
disp(size(coefs))
8 6
```

to fit a data array of size

```
disp(size(z))
15 11
```

The approach followed here seems *biased*, in the following way. We first think of the given data 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. Here is the numerical experiment.

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 that now become the data values. Correspondingly, we 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 we plot `valsb` using `mesh`:

```
mesh(xv,y,valsb.'), view(150,50)
```

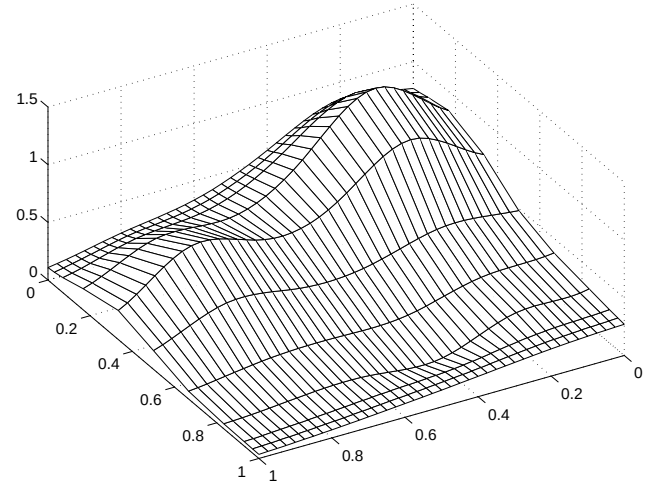


Figure 1-20: Another Family of Smooth Curves Pretending to Be a Surface

Note the ridges. They confirm that we are, once again, plotting smooth curves in one direction only. But this time the curves run in the other direction.

Now comes the second step, to get the actual surface. First, extract the coefficients:

```
coefsx = fnbrk(spb,'c');
```

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, we need to transpose the coefficient array from `spb`, since `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,'c');
```

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))) )
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 $A = A_{\texttt{knots,k,x}}$ so that

$$\texttt{c} = \texttt{y}*A_{\texttt{knots,k,x}}$$

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 *d*-space, 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 =fnpbrk(sp,'c');
```

provide us with the matrix `coefsy` that satisfies

$$\texttt{coefsy} = \texttt{z}*A_{\texttt{knotsy,ky,y}}$$

The subsequent computations

```
sp2 = spap2(knotsx,kx,xx,coefsy.');
coefs = fnbrk(sp2,'c').';
```

generate the coefficient array `coefs`, which, taking into account the two transpositions, satisfies

$$\begin{aligned}\texttt{coefs} &= ((\texttt{z}*A_{\texttt{knotsy,ky,y}})'\ *A_{\texttt{knotsx,kx,x}})' \\ &= (A_{\texttt{knotsx,kx,x}})'\ *\texttt{z}*A_{\texttt{knotsy,ky,y}}\end{aligned}$$

In the second, alternative calculation, we first computed

```
spb = spap2(knotsx,kx,x,z.');
coefsx = fnbrk(spb,'c');
```

hence `coefsx = z'`$*A_{\texttt{knotsx,kx,x}}$. The subsequent calculation

```
spb2 = spap2(knotsy,ky,y,coefsx.');
coefsb = fnbrk(spb,'c');
```

then provided

coefsb = coefsx.' *$A_{\texttt{knotsy,ky,y}}$ = ($A_{\texttt{knotsx,kx,x}}$).'*z*$A_{\texttt{knotsy,ky,y}}$

Consequently, coefsb = coefs.

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 we would start off with

```
coefs = reshape(v,Nx,Ny*Nz);
```

Assuming that nj = knotsj - kj, for j = x, y, z, we would then proceed as follows:

```
sp = spap2(knotsx,kx,x,coefs.');
coefs = reshape(fnbrk(sp,'c'),Ny,Nz*nx);
sp = spap2(knotsy,ky,y,coefs.');
coefs = reshape(fnbrk(sp,'c'),Nz,nx*ny);
sp = spap2(knotsz,kz,z,coefs.');
coefs = reshape(fnbrk(sp,'c'),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.

2

Function Reference

This chapter contains detailed descriptions of the main commands in the Spline Toolbox. It begins with a listing of entries grouped by subject area and continues with the reference entries in alphabetical order. Not all the entries in the initial listing actually have a reference page. Those that do not appear in parentheses in the initial listing.

Information is also available through the online help facility, `help splines`.

Functions Listed by Category

GUIs	
bspligui	Experiment with a B-spline as function of its knots
splinetool	Experiment with some spline approximation methods

Construction of Splines	
csape	Cubic spline interpolation with end conditions
csapi	Cubic spline interpolation
csaps	Cubic smoothing spline
cscvn	'Natural' or periodic interpolating cubic spline curve
getcurve	Interactive creation of a cubic spline curve
ppmak	Put together a spline in ppform
spapi	Spline interpolation
spaps	Smoothing spline
spap2	Least-squares spline approximation
spcrv	Spline curve by uniform subdivision
spmak	Put together a spline in B-form
rpmak	Put together a rational spline in rpform
rsmak	Put together a rational spline in rBform

Operators	
fnbrk	Name and part(s) of a form
fncmb	Arithmetic with function(s)

Operators	
fnder	Differentiate a function
fndir	Directional derivative of a function
fnint	Integrate a function
fnjmp	Jumps, i.e., $f(x+) - f(x-)$
fnplt	Plot a function
fnrfn	Insert additional points into the partition of a form
fntlr	Taylor coefficients or polynomial
fnval	Evaluate a function
fn2fm	Convert to specified form

Work with Breaks, Knots, and Sites	
augknt	Augment a knot sequence
aveknt	Provide knot averages
brk2knt	Breaks with multiplicities into knots
knt2brk	From knots to breaks and their multiplicities
knt2mlt	Knot multiplicities
sorted	Locate sites with respect to meshsites
aptknt	Acceptable knot sequence
newknt	New break distribution
optknt	Knot distribution ‘optimal' for interpolation
chbpnt	Good data sites, the Chebyshev-Demko points

Customized Linear Equation Solver	
slvblk	Solve almost block diagonal linear system
bkbrk	Part(s) of an almost block-diagonal matrix

Information About Splines and the Toolbox	
(spterms)	Explanation of spline toolbox terms

Demonstrations	
(spdemos)	List of demonstrations
(splexmpl)	Some simple examples
(ppalldem)	Introduction to ppform
(spalldem)	Introduction to B-form
bspline	Display a B-spline and its polynomial pieces
(bsplidem)	Some B-splines
(csapidem)	Cubic spline interpolation
(spapidem)	Spline interpolation
(histodem)	Smoothing a histogram
(csapsdem)	Cubic smoothing spline
(pckkntdm)	Knot choices
(spcrvdem)	Spline curve construction
(difeqdem)	A singularly perturbed ODE
(chebdem)	An equi-oscillating spline
(tspdem)	Tensor products

Utilities	
`(franke)`	Franke's bivariate test function.
`(subplus)`	Positive part
`(titanium)`	Titanium heat data
`splpp`	Convert left of 0 from B-form to ppform
`sprpp`	Convert right of 0 from B-form to ppform
`spcol`	B-spline collocation matrix

Function Reference Pages

This section contains function reference pages listed alphabetically. For ease of use, most functions have default arguments. In the reference entry under Syntax, we first list the function with all *necessary* input arguments and then with all *possible* input arguments. The functions can be used with any number of arguments between these extremes, the rule being that if you want to specify an optional argument, you must also specify all other optional arguments (if any) to the left of it in the argument list. The rest are given default values, as specified in the manual.

As always in MATLAB, only the output arguments explicitly specified are returned to the user

Purpose

Acceptable knot sequence

Syntax

```
knots = aptknt(tau,k)
[knots,k] = aptknt(tau,k)
```

Description

Assume that `tau` has at least `k` entries, is nondecreasing, and satisfies `tau(i)<tau(i+k-1)` for all `i`. Then the knot sequence `knots` returned by the command `aptknt(tau,k)` is acceptable for interpolation to data at the sites `tau` in the sense that there is exactly one spline of order `k` with knot sequence `knots` that matches given data 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)`; the `k` used is optionally returned. An error results if `tau` fails to be nondecreasing and/or `tau(i)==tau(i+k-1)` for some `i`.

Examples

If `tau` is equally spaced, e.g., `tau = 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.

Algorithm

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

`newknt`, `aveknt`, `optknt`, `augknt`

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.

augknt

Purpose

Augment a knot sequence

Syntax

```
[augknot,addl] = augknt(knots,k)
[augknot,addl] = augknt(knots,k,mults)
```

Description

`augknt` 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.) Also returns the number `addl` of knots added on the left. (This number may be negative.)

If the third argument is present, the augmented knot sequence 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`.

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`.

Purpose

Provide knot averages

Syntax

```
tstar = aveknt(t,k)
```

Description

`aveknt` returns the averages of successive `k - 1` knots, i.e., the sites

$$t_i^* := (t_{i+1} + \cdots + t_{i+k-1})/(k-1), \qquad i = 1{:}n$$

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, with `t=fnbrk(sp,'knots')` and `k=fnbrk(sp,'order')`, the points `(tstar(`*i*`),a(`*i*`))` with `tstar=aveknt(t,k)` constitute the vertices of the spline's *control polygon*.

See Also

`aptknt`, `chbpnt`, `optknt`

bkbrk

Purpose

Part(s) of an almost block-diagonal matrix

Syntax

```
[nb,rows,ncols,last,blocks] = bkbrk(blokmat)
bkbrk(blokmat)
```

Description

`bkbrk` is a utility used in `slvblk`. It 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]`.

If there are no output arguments, nothing is returned but the details are printed out. This is of use when trying to understand what went wrong with such a matrix.

`spcol` provides the spline collocation matrix in an almost block-diagonal form especially suited for splines, for use with `slvblk`. But `bkbrk` can also decode the almost block-diagonal form used in [1].

See Also

`slvblk`, `spcol`

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.

Purpose

Breaks with multiplicities into knots

Syntax

```
[knots, index] = brk2knt(breaks, mults)
```

Description

The sequence `knots` 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)`.

If, as one would expect, `breaks` is strictly increasing and all multiplicities are positive, then, for each `i`, `index(i)` is the first place in `knots` at which `breaks(i)` appears.

Examples

If `t = [1 1 2 2 2 3 4 5 5]`, then `[xi,m] = knt2brk(t)` gives `[1 2 3 4 5]` for `xi` and `[2 3 1 1 2]` for `m`, and `tt = brk2knt(xi,m)` gives `t` for `tt`.

See Also

`knt2mlt`, `knt2brk`, `augknt`

bspligui

Purpose

Experiment with a B-spline as a function of its knots

Syntax

```
bspligui
```

Description

The command `bspligui` starts a Graphical User Interface (or, 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 $t_0 \le \cdots \le t_k$:

- 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 one 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 indicates 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`, `bsplidem`, `spcol`, `chbpnt`

bspline

Purpose

Display a B-spline and its polynomial pieces

Syntax

```
bspline(t)
bspline(t,window)
pp = bspline(t)
```

Description

`bspline(t)` plots $B(\cdot \mid t)$, i.e., the B-spline with knot sequence `t`, as well as the polynomial pieces of which it is composed.

If the second argument, `window`, is present, the plotting is done in the subplot window specified by `window`; see the MATLAB command `subplot` for details.

If there is an output argument, then nothing is plotted, and the ppform of the B-spline is returned instead.

Examples

The statement `pp=fn2fm(spmak(t,1),'pp')` has the same effect as the statement `pp=bspline(t)`.

See the demo `bsplidem` for typical uses of this command.

See Also

`bsplidem`, `bspligui`

Purpose

Good data sites, the Chebyshev-Demko points

Syntax

```
tau = chbpnt(t,k)
[tau, sp] = chbpnt(t,k,tol)
```

Description

The extreme points of the Chebyshev spline of order `k` with knot sequence `t` are returned. 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.

Optionally, the Chebyshev spline is also returned. The default value for `tol` is `.001`. The iterative process used to construct the Chebyshev spline is terminated when the relative difference between its absolutely largest and its absolutely smallest local extremum is smaller than `tol`.

Examples

`chbpnt([-ones(1,k),ones(1,k)],k)` provides (approximately) the extreme points 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.

Algorithm

The Chebyshev spline for the given knot sequence and order is constructed iteratively, using the Remes algorithm, using as initial guess the spline that takes alternately the values 1 and -1 at the sequence `aveknt(t,k)`. The demo `chebdem` gives a detailed discussion of one version of the process as applied to a particular example.

See Also

`chebdem`, `aveknt`

csape

Purpose

Cubic spline interpolation with end conditions

Syntax

```
pp = csape(x,y)
pp = csape(x,y,conds,valconds)
```

Description

A cubic spline *s* (in ppform) with knot sequence `x` is constructed that satisfies `s(x(j)) = y(:,j)` for all `j`, as well as an additional *end condition* at the first and at the last data site, as specified by *conds* and *valconds*.

conds may be a *string* whose first character matches one of the following: `'complete'` or `'clamped'`, `'not-a-knot'`, `'periodic'`, `'second'`, `'variational'`, with the following meanings.

`'complete'`	Match endslopes (as given in *valconds*, with default as under "default")
`'not-a-knot'`	Make second and second-last sites inactive knots (ignoring *valconds* if given)
`'periodic'`	Match first and second derivatives at first site with those at last site
`'second'`	Match end second derivatives (as given in *valconds*, with default [0 0], i.e., as in `'variational'`)
`'variational'`	Set end second derivatives equal to zero (ignoring *valconds* if given)
default	Match endslopes to the slope of the cubic that matches the first four data at the respective end (i.e., Lagrange)

By giving *conds* as a 1-by-2 matrix instead, it is possible to specify *different* conditions at the two endpoints. Explicitly, $D^i s$ is given the value *valconds*(*j*) at the left (*j* = 1) respectively right (*j* = 2) endpoint in case conds(*j*) = *i*, *i* = 1,2. There are default values for *conds* and/or *valconds*.

Available conditions are:

clamped	$Ds(e) = valconds()$	$conds() = 1$
curved	$D^2s(e) = valconds()$	$conds() = 2$
Lagrange	$Ds(e) = Dp(e)$	default
periodic	$D^r s(a) = D^r s(b), r = 1,2$	$conds() = [0\ \ 0]$
variational	$D^2s(e) = 0$	$conds() = 2$ and $valconds(j) = 0$

Here, e = a(b) is the first (last) data site in case $j = 1$ ($j = 2$), and (in the Lagrange condition) p is the cubic polynomial that interpolates to the given data at e and the three sites nearest e.

If $conds(j)$ is not specified or is different from 0, 1, or 2, then it is taken to be 1 and the corresponding $valconds(j)$ is taken to be the corresponding default value.

The default value for $valconds(j)$ is the derivative of the cubic interpolant at the nearest four sites in case $conds(j) = 1$, and is 0 otherwise.

It is possible (and, in the case of gridded data required) to specify *valconds* as part of `y`. Specifically, if `size(y)==[d,ny]` and `ny==length(x)+2`, then *valconds* is taken to be `y(:,[1 end])`, and `y(:,i+1)` is matched at `x(i)`, `i=1:length(x)`.

It is also possible to handle gridded data, by having `x` be a cell array containing m univariate meshes and, correspondingly, having `y` be an m-dimensional array (or an m+1-dimensional array if the function is to be vector-valued). Correspondingly, *conds* is a cell array with m entries, but the information normally specified by *valconds* is now expected to be part of `y`.

This command calls on a much expanded version of the Fortran routine `CUBSPL` in *PGS*.

Examples

`csape(x,y)` provides the cubic spline interpolant with the Lagrange end conditions, while `csape(x,y,[2 2])` provides the variational, or *natural* cubic spline interpolant, as does `csape(x,y,'v')`. `csape([-1 1],[-1 1], [1 2],[3 6])` provides the cubic polynomial p for which $p(-1) = -1$, $Dp(-1) = 3$,

$p(1) = 1$, $D^2p(1) = 6$, i.e., $p(x) = x^3$. Finally, `csape([-1 1],[-1 1])` provides the straight line p for which p(1) = 1, i.e.$p(x) = x$.

As a multivariate vector-valued example, here is a sphere, done as a parametric bicubic spline, 3d-valued, using prescribed slopes in one direction and periodic side conditions in the other:

```
x = 0:4; y=-2:2; s2 = 1/sqrt(2);
 clear v
 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'});
 values = fnval(sph,{0:.1:4,-2:.1:2});

 surf(squeeze(values(1,:,:)),squeeze(values(2,:,:)),...
  squeeze(values(3,:,:))); axis equal, axis off
```

The lines involving `fnval` and `surf` could have been replaced by the simple command: `fnplt(sph)`. Note that `v` is a 3-dimensional array, with `v(:,i,j)` the 3-vector to be matched at `(x(i),y(j))`, `i=1:5`, `j=1:5`. Note further that, in accordance with *conds*`{1}` being `'clamped'`, `size(v,2)` is 7 (and not 5), with the first and last entry of `v(r,:,j)` specifying the end slopes to be matched.

End conditions other than the ones listed earlier can be handled along the following lines. Suppose that we want to enforce the condition

$$\lambda(s) := aDs(e) + bD^2s(e) = c$$

for given scalars a, b, and c, and with e := x(1). Then one could compute the cubic spline interpolant s_1 to the given data using the default end condition as well as the cubic spline interpolant s_0 to zero data and some (nontrivial) end condition at e, and then obtain the desired interpolant in the form

$$s = s_1 + ((c - \lambda(s_1))/\lambda(s_0))s_0$$

Here are the (not inconsiderable) details (in which the first polynomial piece of s_1 and s_0 is pulled out to avoid differentiating all of s_1 and s_0):

```
pp1 = csape(x,y);
dp1 = fnder(fnbrk(pp1,1));
pp0 = csape(x,zeros(size(y)),[1,0],[1,0]);
dp0 = fnder(fnbrk(pp0,1));
```

```
e = x(1);
lam1 = a*fnval(dp1,e) + b*fnval(fnder(dp1),e);
lam0 = a*fnval(dp0,e) + b*fnval(fnder(dp0),e);
pp = fncmb(pp0,(c-lam1)/lam0,pp1);
```

Algorithm

The relevant tridiagonal linear system is constructed and solved using MATLAB's sparse matrix capabilities.

See Also

`csapi`, `spapi`, `spline`

Cautionary Note

If the sequence `x` is not nondecreasing, both `x` and `y` will be reordered in concert to make it so. Also, if the value sequence `y` is vector-valued, then *`valconds`*`(:,j)`, `j=1:2`, must be vectors of that same length (if explicitly given).

csapi

Purpose

Cubic spline interpolation

Syntax

```
values = csapi(x,y,xx)
pp = csapi(x,y)
```

Description

A cubic spline s with knot sequence `x` is constructed that satisfies s(`x(j)`)=`y(j)` for all `j`, as well as the not-a-knot end conditions, $\text{jump}_{\texttt{x(2)}}\, D^3 s = 0 = \text{jump}_{\texttt{x(end-1)}}\, D^3 s$ (with $D^3 s$ the third derivative of s).

The call `csapi(x,y,xx)` returns the values s(`xx`) of this interpolating cubic spline at the given argument sequence `xx`.

The alternative call `csapi(x,y)` returns instead the ppform of the cubic spline, for later use with `fnval`, `fnder`, etc.

If `x` is a cell array, containing sequences `x1`, ..., `xm`, of lengths `n1`, ..., `nm` respectively, then `y` is expected to be an array, of size `[n1,...,nm]` (or of size `[d,n1,...,nm]` if the interpolant is to be `d`-vector-valued), and the output will be an `m`-cubic spline interpolant to such data. Precisely, if there are only two input arguments, then the output will be the ppform of this interpolant, while, if there is a third input argument, `xx`, then the output will be the values of the interpolant at the sites specified by `xx`. If `xx` is a cell array with `m` sequence entries, then the corresponding `m`- (or (`m`+1)-)dimensional array of grid values is returned. Otherwise, `xx` must be a list of `m`-vectors and, the corresponding list of values of the interpolant at these sites is returned.

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 values and can handle gridded data.

Examples

See the demo `csapidem` for various examples.

Up to rounding errors, and assuming that `x` has at least four entries, the statement `pp = csapi(x,y)` should put the same spline into `pp` as the statements

```
n = length(x);
pp = fn2fm(spapi(augknt(x([1 3:(n-2) n]),4),x,y),'pp');
```

except that the description of the spline obtained the second way will use no break at x(2) and x(n - 1).

Here is a simple bivariate example, a bicubic spline interpolant to the Mexican Hat function being plotted:

```
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])
```

Note the reversal of `x` and `y` in the call to `meshgrid`, needed since MATLAB likes to think of the entry `z(i,j)` as the value at `(x(j),y(i))` while this toolbox follows the Approximation Theory standard of thinking of `z(i,j)` as the value at `(x(i),y(j))`. Similar caution has to be exerted when values of such a bivariate spline are to be plotted with the aid of MATLAB's `mesh`, as is shown here (note the use of the transpose of the matrix of values obtained from `fnval`):

```
xf = linspace(x(1),x(end),41); yf = linspace(y(1),y(end),41);
mesh(xf, yf, fnval( bcs, {xf, yf}).')
```

Algorithm

The relevant tridiagonal linear system is constructed and solved, using MATLAB's sparse matrix capability.

The not-a-knot end condition is used, thus 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`, `tspdem`

Cautionary Note

If the sequence `x` is not nondecreasing, both `x` and `y` will be reordered in concert to make it so.

csaps

Purpose

Cubic smoothing spline

Syntax

```
[values,p] = csaps(x,y,p,xx)
[values,p] = csaps(x,y,p,xx,w)
[pp,p] = csaps(x,y)
[pp,p] = csaps(x,y,p)
[pp,p] = csaps(x,y,p,[],w)
```

Description

The cubic smoothing spline *s* to the given data `x,y` is constructed, for the specified *smoothing parameter* `p` $\in$ `[0..1]` and the optionally specified weight `w`. If the smoothing parameter is negative or none is specified, `p` is chosen in the "interesting range" discussed below; in any case, the value of `p` used is returned as the optional second output argument.
The smoothing spline minimizes

$$\mathrm{p}\sum_i \mathrm{w}(i)(\mathrm{y}(i)-s(\mathrm{x}(i)))^2 + (1-\mathrm{p})\int \lambda(t)(D^2 s)(t)^2\,dt$$

with `w=ones(size(x))` the default value for the weight vector `w` in the *error measure*, and 1 the default for the piecewise constant weight function λ in the *roughness measure*. For `p` = 0, *s* is the least-squares straight line fit to the data, while, on the other extreme, i.e., for `p` = 1, *s* 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 interesting range of `p` is often near $1/(1+h^3/6)$, with *h* the average spacing of the data sites. For uniformly spaced data, one would expect a close following of the data for $\mathrm{p} = 1/(1 + h^3/60)$ and some satisfactory smoothing for $\mathrm{p} = 1/(1 + h^3/.6)$.

The call `csaps(x,y,p,xx)` and `csaps(x,y,p,xx,w)` return the values *s*(`xx`) of this cubic smoothing spline at the given argument sequence `xx` instead of the smoothing spline itself.

The alternative call `csaps(x,y,p)` returns instead the ppform of the cubic spline, for later use with `fnval`, `fnder`, etc.

It is in general difficult to choose the parameter `p` without experimentation (such as specifying a negative `p` on a first try). For that reason, use of `spaps` is encouraged since there `p` is (implicitly) chosen so as to produce the smoothest spline within a specified tolerance of the data.

The weight function λ in the roughness measure can, optionally, be specified as a (non-negative) piecewise constant function, with breaks at the data sites `x` , by inputing for `p` a *vector* whose `i`-th entry provides 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` continues to be used as the desired value of the smoothness parameter `p`. In this way, it is possible to insist that the resulting smoothing spline be smoother in some parts of the interval than in others.

It is also possible to smooth data on a rectangular grid and obtain smoothed values on a rectangular grid or at scattered sites, by the calls

```
values = csaps( {x1,...,xm}, y, p, xx, w)
```

or

```
pp = csaps( {x1,...,xm}, y, p, [], w )
```

in which `y` is expected to have size `[d,length(x1),...,.length(xm)]` (or `[length(x1),...,.length(xm)]` if the function is to be scalar-valued), and `p` is either a scalar or an m-vector of scalars, and `xx` is either a list of m-vectors `xx(:,j)` or else a cell-array `{xx1,...,xxm}` specifying the m-dimensional grid at which to evaluate the interpolant, and, correspondingly, `w`, if given, is cell array of weight sequences for the m dimensions (with `w{i}` empty the indication that the default weights are to be used with the `i`th variable).

Algorithm

This is an implementation of the Fortran routine `SMOOTH` from *PGS*.

See Also

`spaps`, `csape`, `spap2`

Cautionary Note

If the sequence `x` is not nondecreasing, both `x` and `y` will be reordered in concert to make it so.

cscvn

Purpose

'Natural' or periodic interpolating cubic spline curve

Syntax

```
curve = cscvn(points)
```

Description

`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 jth point is chosen by Eugene Lee's [1] centripetal scheme, i.e., as accumulated squareroot 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 a periodic cubic spline curve is constructed. However, double points result in corners.

Examples

The following provides the plot of a questionable curve through some points (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
```

Here is a closed curve, good for 14 February, with one double point:

```
c=fnplt(cscvn([0 .82 .92 0 0 -.92 -.82 0; .66 .9 0 -.83 -.83 ...
0.9.66])); fill(c(2,:),'r'), axis equal
```

Algorithm

The break sequence `t` is determined as

```
t = cumsum([0;((diff(points.').^2)*ones(d,1)).^(1/4)]).';
```

and `csape` (with either periodic or variational side conditions) is used to construct the smooth pieces between double points (if any).

See Also

`csape`, `getcurve`, `spcrvdem`, `fnplt`

References

[1] E.T.Y. Lee, Choosing nodes in parametric curve interpolation, *Computer-Aided Design* **21** (1989), 363–370.

Purpose

Convert to specified form

Syntax

```
g = fn2fm(f, form)
sp = fn2fm(pp,'B-',sconds)
```

Description

The output describes the same function as the input, but in the specified form. Choices for `form` are `'B-'` (or `'sp'`), `'pp'`, `'BB'`, for the B-form, the ppform, and the BBform, 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 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.

In addition, for backward compatibility, `form` may be the string `'MA'`, in case `f` describes a univariate function, and then `g` contains the ppform of that function, but in the terms understood by earlier versions of `ppval`.

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. But the user can assist by supplying a tolerance (strictly between 0 and 1) in the optional argument *sconds*.

Alternatively, the user can supply, in *sconds*`(i)`, the correct number of smoothness conditions to be used across the *i*th *interior* break, but must then do so for all interior breaks. If the function in `f` is a tensor product, then *sconds*, if given, must be a cell array.

Examples

`sp = fn2fm(spline(x,y),'sp')` will give the interpolating cubic spline provided by MATLAB's `spline`, but in B-form (i.e., described as a linear combination of B-splines). The subsequent command `pp = fn2fm(sp,'MA')` recovers the original output from `spline(x,y)` (assuming all the sites in `x` are active knots).

As another example,

```
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).

Algorithm

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 the section “Splines: An Overview” in the Tutorial. Without further information, such a conversion has to ascertain the actual smoothness across each interior break of the function in `f`.

See Also

`ppalldem`, `spalldem`, `spmak`, `ppmak`

Cautionary Note

When going from B-form to ppform, any jump discontinuity at the first and last knot, `t(1)` or `t(n+k)`, 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, $x \mapsto 1$.

Purpose

Name and part(s) of a form

Syntax

```
out = fnbrk(f,part)
pp = fnbrk (pp,[a b])
pp = fnbrk(pp,j)
fnbrk(f)
```

Description

`out = fnbrk(f, part)` returns the part of the form in `f` specified by *part*. These are the parts used when the form was put together, in `spmak` or `ppmak` or `rsmak` or `spmak`, but also other parts derived from these. In particular, `out = fnbrk(f,'form')` returns a string indicating the form contained in `f`.

If the form in `f` is a B-form, then possible choices for *part* are: `'knots'` or `'t'`, `'coefs'`, `'number'`, `'order'`, `'dim'`ension, and `'interval'` (returning the knot sequence, the B-spline coefficient sequence, the number of coefficients, the polynomial order, the (vector) dimension of the coefficients, and the basic interval, respectively).

Exactly the same is returned in case `f` is in BBform. Exactly the same is returned in case `f` is in rBform, except that the dimension returned is that of the target of the function, hence it is one less than the (vector) dimension of the coefficients.

If the form in `f` is a ppform, then the possible choices for *part* are: `'breaks'`, `'coefs'`, `'pieces'` or `'l'`, `'order'`, `'dim'`ension, and `'interval'` (returning the break sequence, the local polynomial coefficients, the number of polynomial pieces, the polynomial order, the (vector) dimension of the coefficients, and the basic interval, respectively). The string `'guide'` also elicits the coefficients, but in the form needed for `PPVALU` in PGS. Finally, *part* can also be a positive integer, `j`, in which case the output is the ppform of the *j*th polynomial piece of the piecewise-polynomial function in `f`.

Exactly the same is returned in case `f` is in rpform, except that the dimension returned is that of the target of the function, hence it is one less than the (vector) dimension of the coefficients.

Finally, for any form, *part* can also be a 1-by-2 matrix specifying an interval, in which case the output is the restriction/extension of the function in `f` to that interval and in the same form.

If the function in `f` is multivariate, then the corresponding multivariate parts are returned. This means, e.g., that knots and breaks are cell arrays, the coefficient array is, in general, higher than two-dimensional, and order, number and pieces are vectors.

If no output is specified, then there should be only one input argument and, in that case, nothing is returned, but a description of the various parts of the form is printed on the screen 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 4th and 5th entry in the break sequence for the first variable.

See Also

`ppmak`, `spmak`, `ppalldem`, `spalldem`

Purpose

Arithmetic with function(s)

Syntax

```
fn = fncmb(function,matrix)
fn = fncmb(function,function)
fn = fncmb(function,matrix,function)
fn = fncmb(function,matrix,function,matrix)
fn = 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. More than that, a matrix may be applied to a vector-valued function, and even two (univariate) functions in different forms may be added or multiplied pointwise.

Examples

`fncmb(fn,3.5)` multiplies (the coefficients of) the function in `fn` by 3.5, while `fncmb(f,g)` returns the sum of the function in `f` and in `g`, and `fncmb(f,3,g,-4)` returns the linear combination, with weights 3 and -4, of the function in `f` and the function in `g`. Also, `fncmb(f,3,g)` adds 3 times the function in `f` to the function in `g`.

Assuming, more generally, that the function *f* in `f` is `d`-*vector*-valued for some `d`, and that, correspondingly, `A` is a matrix of size `[r,d]` for some `r`, then the statement `fncmb(f,A)` returns the description of the function

$$\mathrm{IR} \to \mathrm{IR}^r : x \mapsto \mathrm{A} * f(x)$$

As a simple example, 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 $(f(x),2f(x),3f(x))$. Note that, by the convention throughout this toolbox, the subsequent statement `fnval(f3, x)` returns a 1-*column*-matrix. As another simple example, if `f` describes a surface in 3-space, 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. As another example, if `t` is a knot sequence of length `n+k` and `a` is a matrix with `n` columns, then `fncmb(spmak(t,eye(n,n)),a)` is the same as `spmak(t,a)`.

Finally, `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 $x \mapsto B(x \mid 0,1,2,3,4) + x$ (but has no active break at 0 or 1, hence differs from this function outside the interval [`0 .. 4`]), while

`fncmb(spmak([0:4],1),'-',0)` has the same effect as `fn2fm(spmak([0:4],1),'pp')`.

Algorithm

The coefficients are extracted (via `fnbrk`) and operated on by the specified matrix (and, possibly, added), then recombined with the rest of the function description (via `ppmak` or `spmak`). 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

$$x \mapsto f1(x) \text{ op } f2(x)$$

with `op` one of `+`, `-`, `*`, and `f1`, `f2` of arbitrary form. In addition, if `f2` is a scalar, it is taken to be the function that is constantly equal to that scalar.

Limitations

`fncmb` only works for *univariate* functions, except for the case 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)`.

Cautionary Note

This matching condition is not checked for explicitly. But, MATLAB will issue an error message about incompatible sizes if the two coefficient arrays involved do not agree in size.

Purpose

Differentiate a function

Syntax

```
fprime = fnder(f)
fprime = fnder(f,dorder)
```

Description

`fnder(f,dorder)` is the description of the `dorder`th derivative of the function whose description is contained 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, i.e., they are both ppforms or both B-forms.

If the function in `f` is multivariate, say *m*-variate, then `dorder` must be given, and must be of length *m*.

Examples

If `f` is in ppform, or in B-form with its last knot of sufficiently high multiplicity, then, up to rounding errors, `f` and `fnint(fnder(f))` are the same.

If `f` is in ppform, then, up to rounding errors, `f` and `fnder(fnint(f))` 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 left-most knot, then, up to rounding errors, `fnint(fnder(f))` contains the B-form of $f\text{-}f(t_1)$. However, its left-most 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 demos `spalldem` and `ppalldem` for examples.

Algorithm

For differentiation of either form, the derivatives are found in the piecewise-polynomial sense. This means that, in effect, each polynomial piece is differentiated separately, and jump discontinuities between polynomial pieces are ignored during differentiation.

For the B-form, the formulas [*PGS*; (X.10)] for differentiation are used.

See Also

`fndir`, `fnint`, `fnval`, `fnplt`, `ppalldem`, `spalldem`

Purpose

Directional derivative of a function

Syntax

```
df = fndir(f,direction)
```

Description

Let f be the function described by `f`, let y be the sole column of the matrix `direction`, and assume that y has as many entries as the function f has arguments. Then `df` is the ppform of the directional derivative of f in the direction y, i.e., of the function $D_y f(x) := \lim_{t->0}(f(x+ty)-f(x))/t$.

If the matrix direction has n columns and f is m-valued, then the function in `df` is (mn)-valued. Its value at x, reshaped as a matrix of size (m,n), has in its jth column the directional derivative of f at x in the direction of the jth column of direction.

Examples

For example, if `f` describes a `d`-variate `m`-vector-valued function and `x` is some point in its domain, then

```
reshape(fnval(fndir(f,eye(d)),x),m,d)
```

is the Jacobian of that function at that point.

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,:,:)))
```

Algorithm

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

`fnder`, `fnint`

fnint

Purpose

Integrate a function

Syntax

```
intgrf = fnint(f)
intgrf = fnint(f,value)
```

Description

`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.

Indefinite integration of a *multivariate* function, in coordinate directions only, is available via `fnder(f,dorder)` with `dorder` having nonpositive entries.

Examples

The statement `fnval(fnint(f),[a b])*[-1;1]` 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 `fnint(fnder(f))` are the same.

If `f` is in ppform, then, up to rounding errors, `f` and `fnder(fnint(f))` 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 left-most knot, then, up to rounding errors, `fnint(fnder(f))` contains the B-form of $f - f(t_1)$. However, its left-most 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 demos `spalldem` and `ppalldem` for examples.

Algorithm For the B-form, the formula [*PGS*; (X.22)] for integration is used.

See Also `fnder`, `fnval`, `fnplt`, `ppalldem`, `spalldem`

fnjmp

Purpose

Jumps, i.e., $f(x+) - f(x-)$

Syntax

```
jumps = fnjmp(f,x)
```

Description

This is a function for spline specialists. It returns, for the *univariate* function *f* described by `f`, the value $f(x+) - f(x-)$ of the jump across `x` made by *f*. If `x` is a matrix, then `jumps` is a matrix of the same size containing the jumps of *f* across the sites in `x`.

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` differs from `x` by roundoff, 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)`.

See Also

`ppalldem`, `spalldem`

Purpose

Plot a function

Syntax

```
fnplt(f)
fnplt(f,arg1,arg2,arg3,arg4)
points = fnplt(f)
```

Description

Plots the function *f* described by `f` on the interval `[a..b]` specified by an optional argument of the form `[a b]` with `a` and `b` scalars (default is the basic interval), using the `symbol` (optionally) specified by a (legal) string (default is `'-'`), and using the linewidth (optionally) specified by a scalar (default is `1`), and using, for a univariate function, `NaN`s in order to plot any jumps correctly, but only if one of the optional arrguments is a string that starts with `'j'`. Up to four optional arguments may appear, in any order. The plot depends strongly on whether the function is univariate or multivariate and also on the dimension of its target, i.e., whether it is scalar-valued, 2-vector-valued or, more generally, d-vector-valued.

If *f* is univariate, the following will be plotted:

- If *f* is scalar-valued, the graph of *f* is plotted;
- If *f* is 2-vector-valued, the planar curve is plotted;
- If *f* is *d*-vector-valued with $d > 2$, the space curve given by the first three components of *f* is plotted.

If *f* is bivariate, the following will be plotted:

- If *f* is scalar-valued, the graph of *f* is plotted (via `surf`);
- If *f* is 2-vector-valued, the image in the plane of a regular grid in its domain is plotted;
- If *f* is *d*-vector-valued with $d>2$, then the parametric surface given by the first three components of its values is plotted (via `surf`).

If *f* is a function of more than two variables, then the bivariate function, obtained by choosing the midpoint of the basic interval in each of the variables other than the first two, is plotted.

Nothing is plotted if an output argument is specified, but the two-dimensional points or three-dimensional points it would have plotted are returned instead.

Algorithm

The univariate function *f* described by `f` is evaluated at 101 equally spaced sites `x` filling out the plotting interval. If *f* is real-valued, the points (`x`, *f*(`x`)) are plotted. If *f* is vector-valued, then the first two or three components of *f*(`x`) are plotted.

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`

Cautionary Note

The basic interval for *f* in B-form is the interval containing *all* the knots. This means that, e.g., *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.

Purpose

Insert additional points into the partition of a form

Syntax

```
g = fnrfn(f,addpts)
```

Description

The form in `f` is refined by the insertion of the entries of `addpts` into the knot sequence or break sequence of the form. 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.

If the function in `f` is m-variate, then `addknts` 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

See `fncmb` for the use of `fnrfn` to refine the knot or break sequences of two splines to a common refinement before forming their sum.

Algorithm

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`, `spmak`, `ppmak`

fntlr

Purpose

Taylor coefficients or polynomial

Syntax

```
taylor = fntlr(f,dorder,x)
p = fntlr(f,dorder,x, interv)
```

Description

The first command returns the unnormalized Taylor coefficients at the given `x` of the function described in `f` up to the given order `dorder`.

For a univariate function and a scalar `x`, this is the vector

$$(f(x);Df(x);\ldots;D^{\text{dorder-1}}f(x))$$

For a matrix `x`, this is the corresponding matrix in which each entry has been replaced by the corresponding Taylor vector.

For a multivariate function, the Taylor vector of order `dorder` becomes the Taylor array of order `dorder`. Assuming that the function is m-variate for some $m>1$, this means that `dorder` is expected to have length m, and the output provides, for each m-vector `x(:,j)` in its `x` input, the array of size `dorder` whose $(i_1,i_2,\ldots,i_m)$ entry is

$$D_1^{i_1-1}D_2^{i_2-1}\cdot D_m^{i_m-1}f(x)$$

However, the output contains this array as the equivalent vector `taylor(:,j)`, of length $d*i_1*i_2*\ldots*i_m$, with `d` the dimension of the target of the function f described by `f`.

The second command returns instead a ppform of the Taylor polynomial at `x` of order `dorder` for the function described by `f`. The basic interval for this ppform is as specified by `interv`. In this case and assuming that the function described by f is `m`-variate, both `x` is expected to be of size `[m,1]`, and `interv` is either of size `[m,2]` or else a cell array of length `m` containing `m` vectors of size `[1,2]`.

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 that `v(3:4,j)` is a vector tangent to the circle at the point `v(1:2,j)`, we use MATLAB's `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 Figure 2-1.

```
quiver(v(1,:),v(2,:),v(5,:),v(6,:)), axis equal, hold off
```

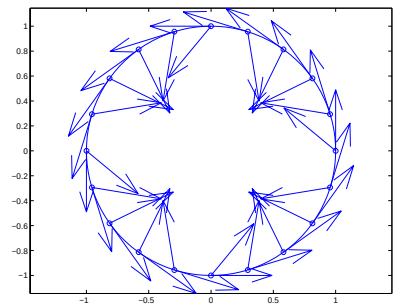


Figure 2-1: First and Second Derivative of a Rational Spline Giving a Circle

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 at the end of the first section of the Tutorial, 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.

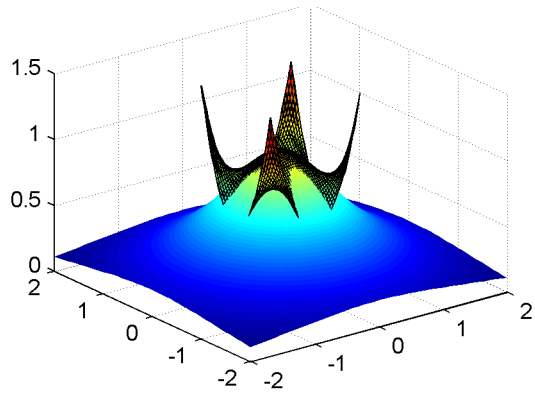


Figure 2-2: The Function 1/(1+x^2+y^2) and Its Taylor Polynomial of Order [3,3] at the Origin

As a final example, we start with a bivariate version of the Runge function, obtaining, for variety, a ppform for its denominator, $1/(1+x^2+y^2)$ by bicubic spline interpolation:

```
w = csapi({-1:1,-1:1},[3 2 3;2 1 2;3 2 3]);
```

Next, we make up the coefficient array for the numerator, 1, using exactly the same size, and put the two together into a rational spline:

```
wcoefs = fnbrk(w,'coef');
scoefs = zeros(size(wcoefs)); scoefs(end)=1;
runge2 = rpmak(fnbrk(w,'breaks'),[scoefs;wcoefs]);
```

Then we enlarge the basic interval for this rational spline, plot it and plot, on top of it, its Taylor polynomial of order [3,3] at (0,0).

```
fnplt(fnbrk(runge2,{[-2 2],[-2 2]}));  shading interp, hold on
fnplt(fntlr(runge2,[3 3], [0;0], [-.5 .5; -.5 .5])), hold off
```

Since we shaded the function but not the Taylor polynomial, we can easily distinguish the two in Figure 2-2. We can also see that, in contrast to the function, the Taylor polynomial fails to be rotationally symmetric. This is due to the fact that it is a polynomial of order [3,3] rather than a polynomial of total order 3.

To obtain the Taylor polynomial of order 3, we get the Taylor polynomial of order [3,3], but with (0,0) the left point of its basic interval, set all its coefficients of total order bigger than 3 equal to zero, and then reconstruct the polynomial, and plot it, choosing a different view in order to show off the Taylor polynomial better. Here are the commands and the resulting Figure 2-3.

```
taylor = fntlr(runge2,[3 3],[0;0],[0 1;0 1]);
tcoef = fnbrk(taylor,'coe'); tcoef([1 2 4]) = 0;
taylor2 = fnbrk(ppmak(fnbrk(taylor,'br'),tcoef),{[-1 1],[-1
1]});
fnplt(fnbrk(runge2,{[-2 2],[-2 2]}));  shading interp, hold on
fnplt(taylor2), view(-28,-26), axis off, hold off
```

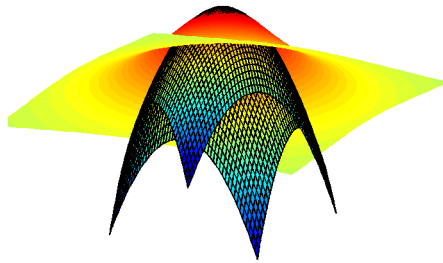


Figure 2-3: The Function 1/(1+x^2+y^2) and Its Taylor Polynomial of Order 3 at the Origin

See Also

`fnder`, `fndir`

fnval

Purpose

Evaluate a function

Syntax

```
values = fnval(f,x)
values = fnval(x,f)
values =  fnval(f,x,'l')
```

Description

Both `fnval(f,x)` and `fnval(x,f)` provide the matrix *f*(x), with *f* the function whose description is contained in `f`. The output (and input) depends on whether *f* is univariate or multivariate.

If the function in `f` is *univariate*, then the output is a matrix of size `[d*m,n]`, with `[m,n]` the size of `x` and `d` the dimension of *f*'s target (e.g., `d` = 2 if *f* maps into the plane).

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 *f*'s basic interval; for it, the value *f*(x-), i.e., the limit from the left, is returned.

If the optional third input argument is present and is a string beginning with `'l'`, then, *f* is instead made to be 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 it, the value *f*(x+) is returned.

If the function is *multivariate*, then the above statements concerning continuity from the left and right apply coordinatewise. Further, if the function is, more precisely, `m`-variate for some `m`>1, then `x` must be either a list of `m`-vectors, i.e., of size `[m,n]`, or a cell array `{x1,...,xm}` containing `m` vectors. In the first case, the output is of size `[d*m,n]` and contains the values of the function at the sites in `x`. In the second case, the output is of size `[d,length(x1),...,length(xm)]` (or of size `[length(x1),...,length(xm)]` in case `d` is 1), and contains the values of the function at the `m`-dimensional grid specified by `x`.

Examples

The statement `fnval(csapi(x,y),xx)` has the same effect as the statement `csapi(x,y,xx)`.

Algorithm

For each entry of `x`, the relevant break- or knot-interval is determined and the relevant information assembled. Depending on whether `f` is in ppform or in B-form, nested multiplication or the B-spline recurrence (see, e.g., [*PGS*; X.(3)]) is then used vector-fashion for the simultaneous evaluation at all entries of `x`. Evaluation of a multivariate function takes full advantage of the tensor product structure.

See Also

`ppmak`, `spmak`, `fnbrk`

getcurve

Purpose

Interactive creation of a cubic spline curve

Syntax

```
[xy, spcv] = getcurve
```

Description

`getcurve` displays a gridded window and asks for input. As the user clicks on points in the gridded window, the broken line connecting these points is displayed. When the user is done, (indicated by clicking outside the gridded window), 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 the last point is *close* to the initial point, a closed curve is drawn. Clicking twice (or more times) in succession at a point permits the curve to have a corner at that point.

See Also

`cscvn`

Purpose

From knots to breaks and/or their multiplicities

Syntax

```
[breaks, mults] = knt2brk(knots)
[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.

The statement `knt2brk(knots)` returns the distinct elements in `knots`, and in increasing order. The optional second output argument provides the multiplicity with which each distinct element occurs in `knots`. In particular, the two outputs, `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`.

The statement `m = knt2mlt(t)` returns a vector of the same length as `t`, with `m(i)` counting the number of entries to the left of the *i*th entry in `sortedt = sort(t)` 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 $D^{m(i)}s(\mathrm{t}(i)) = \mathrm{z}(i)$, all *i*. The optional second argument 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

Purpose

New break distribution

Syntax

```
newknots = newknt(f)
newknots = newknt(f,newl)
[newknots,distfn] = newknt(f,newl)
```

Description

`newknt` 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 (whose ppform is returned in `distfn` if requested) related to the high derivative of `f` is equidistributed. The default value for `newl` is the number of polynomial pieces in `f`.

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 informtion about *g* to make this feasible.

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 the last part of the demo `difeqdem`.

Algorithm

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

$$\int_{\text{newknots(i)}}^{\text{newknots(i+1)}} |D^k f|^{1/k} = \text{const}, \quad \text{i} = k{:}k + \text{newl} - 1$$

See Also

`difeqdem`

Purpose

Knot distribution 'optimal' for interpolation

Syntax

```
knots = optknt(tau,k)
knots = optknt(tau,k, maxiter)
```

Description

`t= optknt(tau,k)` 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 [3] and [2], 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 $\|g-Rg\| \le \text{const}_R \|D^k g\|$ for all *smooth* functions *g*.

Here, $\|f\| := \sup_{\text{tau}(1) < x < \text{tau}(n)} |f(x)|$. 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

$$\int_{\text{tau}(1)}^{\text{tau}(n)} f(x)h(x)dx = 0 \quad \text{for all} \quad f \in S_{k,t}$$

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*).

Examples

See the last part of the demo `spapidem` 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.

Algorithm

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 in (**3**) 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 .

See Also

`aptknt`, `aveknt`, `newknt`, `spapidem`

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.

Purpose

Put together a spline in ppform

Syntax

```
ppmak
ppmak(breaks,coefs)
pp = ppmak(breaks,coefs,d)
```

Description

ppmak puts together a piecewise-polynomial function in ppform, from minimal information, with the rest inferred from the input. fnbrk returns 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 commands using the construct.

If there are no arguments, you will be prompted for breaks and coefs. However, the casual user is not likely to use ppmak explicitly, relying instead on the various spline construction commands in the toolbox to construct splines that satisfy certain conditions.

The action taken by ppmak 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, which 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 determined as l=length(breaks)-1, and the basic interval is, correspondingly, the interval [breaks(1) .. breaks(l+1)].
2. The order k and the dimension d of the function's target are inferred as follows:
   a. If the dimension d is not given explicitly, then coefs(:,i*k+j) is assumed to contain the jth coefficient of the (i+1)st polynomial piece (with the first coefficient the highest and the kth coefficient the lowest or constant coefficient). Thus the dimension d is obtained from [d,kl] = size(coefs) and the order k of the piecewise-polynomial is obtained as k = fix(kl/l).
   b. If d is explicitly specified, then coefs(i*d+j,:) is assumed to contain the jth components of the coefficient vector for the (i+1)st polynomial piece. This corresponds to the format used internally, while the earlier format seems easier to handle when specifying such a piecewise-polynomial

explicitly. In particular, it is the format in which `fnbrk` returns the coefficient array, hence `d` must be explicitly specified when the input `coefs` is the result of a call to `fnbrk`.

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`, and any third input argument is ignored.
   a. If `coefs` is an m-dimensional array, then the function is taken to be scalar-valued, i.e., `d` = 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 (`m`+1)-dimensional array, then `d` is taken to be `size(coefs,1)`, and the *i*th entry of `k` is computed as `size(coefs,i+1)/l(i)`, `i=1:m`.

Since, MATLAB suppresses trailing singleton dimensions, you must use the optional third input argument to supply the desired size of the input array `coefs` in case it has one or more trailing singleton dimensions.

The coefficient array is internally treated 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))` entry the coefficient of

```
(x(1)-breaks{1}(i(1)))^(k(1)-r(1))*...*(x(m)-breaks{m}(i(m)))..
^(k(m)-r(m))
```

in the local polynomial representation of the function on the (hyper)rectangle

```
[breaks{1}(i(1))..breaks{1}(i(1)+1)]x...x[breaks{m}(i(m))..
breaks{m}(i(m)+1)]
```

Examples

`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 $x \mapsto x^2 + 2x + 3$, while the second piece is $x \mapsto 4(x-1)^2 + 5(x-1) + 6$.

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 $x \mapsto (1,4)x^2 + (2,5)x + (3,6)$. 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 (i.e., `d = 2`) curve, but this one is piecewise linear; its first polynomial piece is $x \mapsto (1,5)x + (2,6)$.

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 $x \mapsto x^3 + 2x^2 + 3x + 4$.

See `ppalldem` for other examples.

See Also

`fnbrk`, `ppalldem`

rpmak, rsmak

Purpose

Put together a rational spline

Syntax

```
rp = rpmak(breaks,coefs)
rp = rpmak(breaks,coefs,d)
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.

The command `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.

The commands `ppmak(breaks,coefs,d)` and `rpmak(breaks,coefs,d-1)` are similarly related. 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`.

The commands `spmak(knots,coefs)` and `rsmak(knots,coefs)` are also similarly related. 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.

Finally, the command `rsmak(`*shape*`,`*parameters*`)` provides a rational spline in rBform that describes the shape being specified by the string *shape* and the optional additional *parameters*. Specific choices are:

```
rsmak('circle',radius,center)
rsmak('cone',radius,halfheight)
rsmak('cylinder',radius,height)
rsmak('southcap',radius,center)
```

From these, one may generate related shapes by affine transformations.

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.

Figure 2-4, 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
```

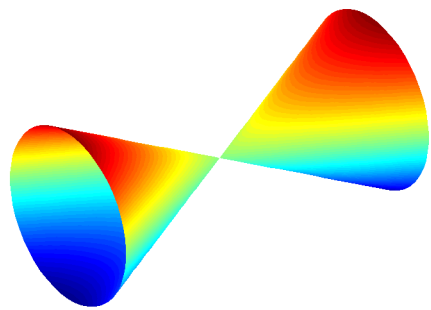


Figure 2-4: A Rotated Cone Given By a Rational Quadratic Spline

For further, illustrated examples, see the section on "NURBS and Other Rational Splines" in the Tutorial.

See Also

`ppmak`, `spmak`, `fnbrk`

slvblk

Purpose

Solve an almost block-diagonal linear system

Syntax

```
x = slvblk(blokmat,b)
x = slvblk(blokmat,b,w)
```

Description

`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 system is overdetermined (i.e., has more equations than unknowns but is of full rank), then the least-squares solution is returned. In this case, the optional third argument, `w`, may be supplied to ensure that the solution minimizes the *weighted* sum $\Sigma_j w(j)((Ax-b)(j))^2$.

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`.

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., satisfies `s(x)=y`.

Algorithm

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 backsubstitution.

See Also

`bkbrk`, `spcol`, `spapi`, `spap2`

Purpose

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.

The vector `pointer=sorted(meshsites,sites)` is the integer sequence for which, for all `j`, `pointer(j)` equals the number of entries in `meshsites` that are ≤ `ssites(j)`, with `ssites=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])
```

Algorithm

The indexing output from `sort([meshsites(:).',sites(:).'])` is used.

spap2

Purpose

Least-squares spline approximation

Syntax

```
sp = spap2(knots,k,x,y)
sp = spap2(l,k,x,y)
sp = spap2(knots,k,x,y,w)
```

Description

Returns the spline *f* of order `k` with knot sequence `knots` for which

```
(*)   y(:,j) = f(x(j)), all j
```

in the weighted mean-square sense, meaning that the sum Σ_j `w(j)*norm(y(:,j)-f(x(j)))^2` is minimized, with default weights equal to 1. If the sites `x` satisfy the (Schoenberg-Whitney) conditions

```
(**)   knots(j) < x(j) < knots(j + k)
          j=1,...,length(x) = length(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`.

Since the proper choice of the knot sequesnce may be a challenge at times, it is also acceptable to input, instead of the sequence `knots`, the integer `1` giving the number of polynomial pieces to be used, in which case `spap2` provides a knot sequence suitable for that.

It is also possible to fit to gridded data. If `knots` is a cell array with `m` entries, then also `x` must be a cell array with `m` entries, as must `w` be (if given). Further, then also `k` must be an `m`-vector, and `y` must be an (`m`+1)-dimensional array, with `y(:,i1,...,im)` the datum to be fitted at the `m`-vector `[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)` the datum to be fitted at the `m`-vector `[x{1}(i1),...,x{m}(im)]`, all `i1`, ..., `im`.

Examples

```
sp = spap2(augknt([a,xi,b],4),4,x,y)
```

is the least-squares approximant 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 in some fashion. In that case, the approximant consists of `l=length(xi)+1` polynomial

pieces. If you do not want to worry about the conditions (**) but merely want to get a cubic spline approximant consisting of 1 polynomial pieces, use instead

```
sp = spap2(1,4,x,y);
```

If the resulting approximation is not satisfactory, try using a larger 1. Else use

```
sp = spap2(newknt(sp),4,x,y);
```

for a possible better distribution of the knot sequence. In fact, if that helps, repeating it may help even more.

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.

Algorithm

`spcol` is called on to provide the almost block-diagonal collocation matrix $(B_{j,k}(x))$, and `slvblk` solves the linear system $(*)$ in the (weighted) least-squares sense, using a block QR factorization.

If only the number of polynomial pieces is specified, then an appropriate knot sequence is obtained by applying `aptknt` to an appropriate subsequence of the data sites x.

Gridded data is 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

`spapi`, `slvblk`, `spcol`

spapi

Purpose

Spline interpolation

Syntax

```
spline = spapi(knots,x,y)
spline = spapi(k,x,y)
```

Description

Returns the spline f (if any) of order

```
k = length(knots) - length(x)
```

with knot sequence `knots` for which

```
(*)   y(:,i) = f(x(i)), all i.
```

This is taken in the osculatory sense in case some `x` are repeated, i.e., in the sense that $D^{m(i)}f(\mathrm{x}(i)) = \mathrm{y}(:,i)$ in case the `x` are in nondecreasing order, with m = `knt2mlt(x)`, i.e., $m(i) := \#\{j < i : \mathrm{x}(j) = \mathrm{x}(i)\}$. Thus m-fold repetition of a site z in `x` corresponds to the prescribing of value and the first m - 1 derivatives of f at z.

It is possible to merely specify the desired spline order, `k`, as the first argument instead of the knot sequence, `knots`, in which case `aptknt` is used to determine a workable (though not necessarily optimal) knot sequence for the given sites `x`.

It is also possible to interpolate to gridded data. If `knots` is a cell array with `m` entries, then also `x` must be a cell array with `m` entries, and `y` must be an (`m`+1)-dimensional array, with `y(:,i1,...,im)` the datum to be fitted at the `m`-vector `[x{1}(i1),...,x{m}(im)]`, all `i1`, ..., `im` , unless the spline is to be scalar-valued, in which case, in contrast to the univariate case, `y` is permitted to be an `m`-dimensional array.

Examples

`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 [0..2] with exactly one interior knot, at 1, that satisfies the five conditions

$f(0+) = 2$, $f(1) = 0$, $Df(1) = 1$, $D^2f(1) = 2$, $f(2-) = -1$

Since the given values are reordered in concert with the given sites if the latter are not already in nondecreasing order, it is, e.g., possible to carry out interpolation to values `y` and slopes `s` at the increasing site sequence `x` by a quintic spline by the command

```
sp = spapi(augknt(x,6,2),[x x x([1 end])],[y,s,ddy0,ddy1]);
```

with ddy0 and ddy1 values for the second derivative at the endpoints.

As a related example, if the function sin(x) is to interpolated at the distinct data sites x by a cubic spline and its slope is also to be matched at a subsequence x(s), then this can be accomplished by the command

```
sp = spapi(4,[x x(s)], [sin(x) cos(x(s))]);
```

in which a suitable knot sequence is supplied iwth ahe aid of aptknt. In fact, if you wanted to interpolate the same data by quintic splines, simply change the 4 to 6. The command spapi(k,x,y) has the same effect as the more explicit command spapi(aptknt(x,k),x,y).

As a final example, here is the bivariate interpolant.

```
x = -2:.5.2; x=-1:.25:1; [xx, yy] = ndgrid(x,y);
z = exp(-(xx.^2+yy.^2));
sp = spapi({3,4},{x,y},z); fnplt(sp)
```

The command spapi(k,x,y) has the same effect as the more explicit command spapi(aptknt(x,k),x,y).

Algorithm

spcol is called on 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.

Gridded data are fitted, in tensor-product fashion, one variable at a time, taking advantage of the fact that a univariate spline fit depends linearly on the values being fitted.

See Also

spaps, spap2, csapi, spline

Limitations

The given (univariate) knots and sites must satisfy the Schoenberg-Whitney conditions for the interpolant to be defined. Assuming the site sequence x to be nondecreasing, this means that we must have

knots(j) < x(j) < knots(j+k), all j,

(with equality possible at knots(1) and knots(end)). In the multivariate case, these conditions must hold in each variable separately.

Cautionary Note

If the (univariate) sequence x is not nondecreasing, both x and y will be reordered in concert to make it so. In the multivariate case, this is done in each variable separately. A positive side effect of this was noted above in the examples.

Purpose

Smoothing spline

Syntax

```
sp = spaps(x,y,tol)
[sp, values, rho] = spaps(x,y,tol,arg1,arg2)
```

Description

Returns the smoothest function that lies within the given tolerance `tol` of the given data and, optionally, its values at the given `x`.

Here, *smoothest* means that the following roughness measure is minimized:

$$F(D^m f) = \int_{x(1)}^{x(end)} \lambda(t) D^m f(t)^2 dt$$

Further, the distance of the function f from the given data is measured by

$$E(f) = \sum_{j=1}^{n} w(j)\left(y(j) - f(x(j))\right)^2$$

The default value for `m` is 2, leading to the cubic smoothing spline. However, the choices `m=1`, for the linear smoothing spline, and `m=3`, for the quintic, smoothing spline, are available, too, by setting one of the optional inputs *arg i* equal to `1` or `3`. Further, the default value for the weight vector `w` makes $E(f)$ the composite trapezoidal rule approximation to $\int_{x(1)}^{x(n)} (y-f)^2$. But the weight vector may also be supplied as one of the optional inputs *arg i* (as a positive vector of the same length as `x`). Finally, the default for the piecewise constant nonnegative weight function λ in the roughness measure is 1, but its constant value on the interval (`x(i-1)` .. `x(i)`) may also be supplied as the `i`-th entry, `i=2:length(x)`, of the *vector* `tol`, in which case `tol(1)` continues to be used as the specified tolerance.

The spline f is determined as the unique minimizer of the expression $\rho E(f) + D(f)$, with the smoothing parameter ρ (optionally returned) so chosen that $E(f) = \text{tol}$ in case `tol` is nonnegative; in the contrary case, ρ is taken to be `-tol`. Hence, when `m` is 2, then, after conversion to ppform, the result should be the same (up to roundoff) as obtained by `csaps(x,y, ρ/(ρ+1))`.

If x is not increasing, then both x and y (as well as w if given) will be reordered in concert to make x increasing. After that, x must be strictly increasing.

The data being fitted may be d-vector-valued, and this is indicated by having y be of size [d,n]. In this case, both the measure of roughness and the distance of the (d-vector-valued) function *f* from the data are the sum of the componentwise measures. For example, if *f(x)* is the d-vector ($f1(x)$, ...,$fd(x)$), then $E(f) = E(f1) + ... + E(fd)$.

It is also possible to obtain a smoothing spline for *gridded data*. When x is a cell array of length r, then y is expected to supply the corresponding gridded values, with size(y) equal to [length(x{1}),...,length(x{r})] in case the function is scalar-valued, and equal to [d,length(x{1}),...,length(x{r})] in case the function is d-vector-valued. Further, the optional input for m must be an r-vector (with entries from the set {1,2,3}), and the optional argument 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 x{i}.

Examples

The statements

```
w = ones(size(x)); w([1 end]) = 100;
sp = spaps(x,y, 1.e-2, w, 3);
```

give a quintic smoothing spline approximation to the given data that close to interpolates the first and last datum, while being within about 1.e-2 of the rest.

Algorithm

Reinsch's approach [1] is used (including his clever 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).

See Also

spap2, spapi, csaps

References

[1] C. Reinsch, "Smoothing by spline functions", *Numer. Math.* 10 (1967), 177–183.

Purpose

B-spline collocation matrix

Syntax

```
spcol(knots,k,tau)
colloc = spcol(knots,k,tau,arg1,arg2)
```

Description

`spcol` constructs the matrix

$$\texttt{colloc} := (D^{m(i)}B_j(\texttt{tau}(i)))$$

with B_j the jth B-spline of order `k` for the knot sequence `knots`, `tau` a sequence of sites, assumed to be *nondecreasing*, and m = `knt2mlt(tau)`, i.e.,

$$m(i) := \#\{j < i : \texttt{tau}(j) = \texttt{tau}(i)\}$$

If one of the optional arguments is a string 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 optional arguments is a string with the same first two letters as in `'sparse'`, then the matrix is returned in MATLAB's `sparse` format.

If one of the optional arguments is a string with the same first two letters as in `'noderiv'`, multiplicities are ignored, i.e., $m(i) = 1$ for all i.

Examples

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.

By adding the optional argument `'sl'`, the output is instead a one-dimensional array containing the same information in storage-saving form. The command `bkbrk` decodes this information.

The statement `bkbrk(spcol([1:6],3,.1+[2:4],'sl'))`; provides the following detailed information about the block structure of the matrix encoded in the information returned by `spcol([1:6],3,.1+[2:4],'sl')`:

```
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)
```

Algorithm

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 :=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 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 determining the B-form of a piecewise-polynomial function from interpolation or other approximation conditions.

See Also

`slvblk`, `spapi`, `spap2`

Limitations

The sequence `tau` is assumed to be nondecreasing.

Purpose

Spline curve by uniform subdivision

Syntax

```
spcrv(c)
curve = 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

$$f{:}t \mapsto \sum_{j=1}^{n} B(t - k/2 \mid j,\ldots,j+k) * c(j), \qquad \frac{k}{2} \le t \le n + \frac{k}{2},$$

with $B(\cdot \mid a,\ldots,z)$ the B-spline with knots $a,\ldots,z$, and `n` the number of coefficients in `c`, i.e., `[d,n]:=size(c)`.

The default value for `k` is 4. 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
```

Algorithm

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

`spcrvdem`, `fnplt`

splinetool

Purpose

Experiment with some spline approximation methods

Syntax

```
splinetool
splinetool(x,y)
```

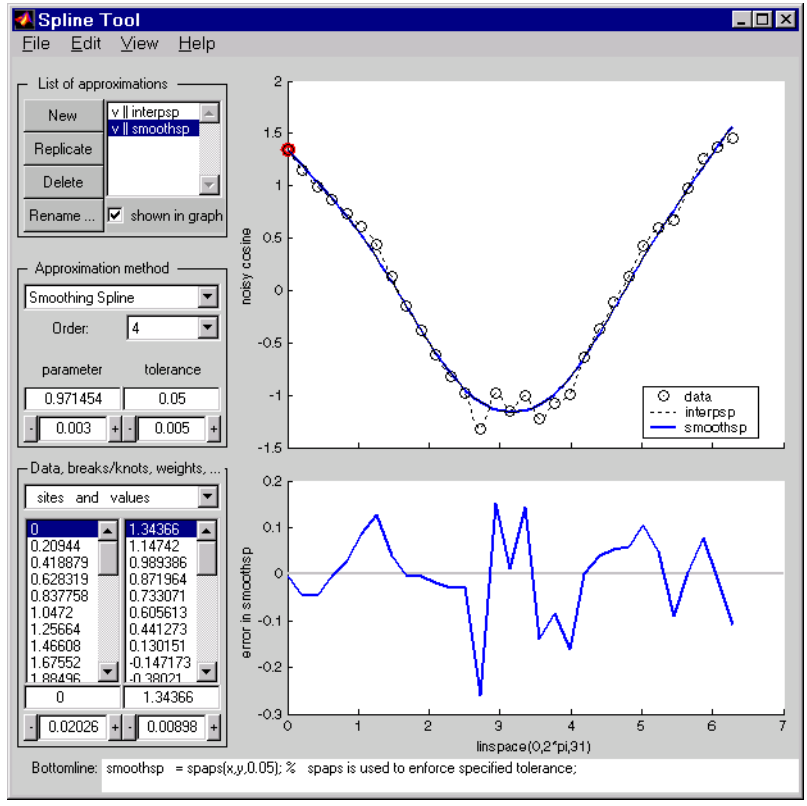
Description

`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.

Remarks

The Spline Tool is shown below 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.

Table 2-1:

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 9. 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 9. The default order is 4, which gives cubic spline interpolants. If the default knots supplied are not satisfactory, they can be moved 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
  - 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 choosing any one of the items in the **View** menu.

- 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 Graph** 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 add, delete, or move data, knots, and breaks by right-clicking in the graph, or selecting the appropriate item in the **Edit** menu.

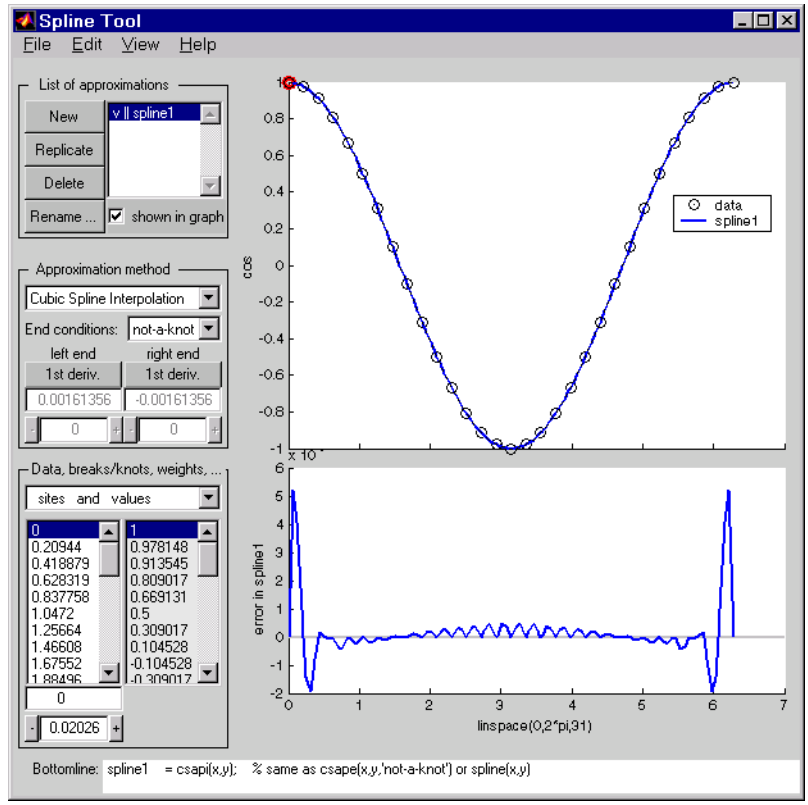
Examples

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 **Provide your own data** from the initial screen, and accept the default function. You should see the following display.



The default approximation shown is cubic spline interpolation with the not-a-knot end condition.

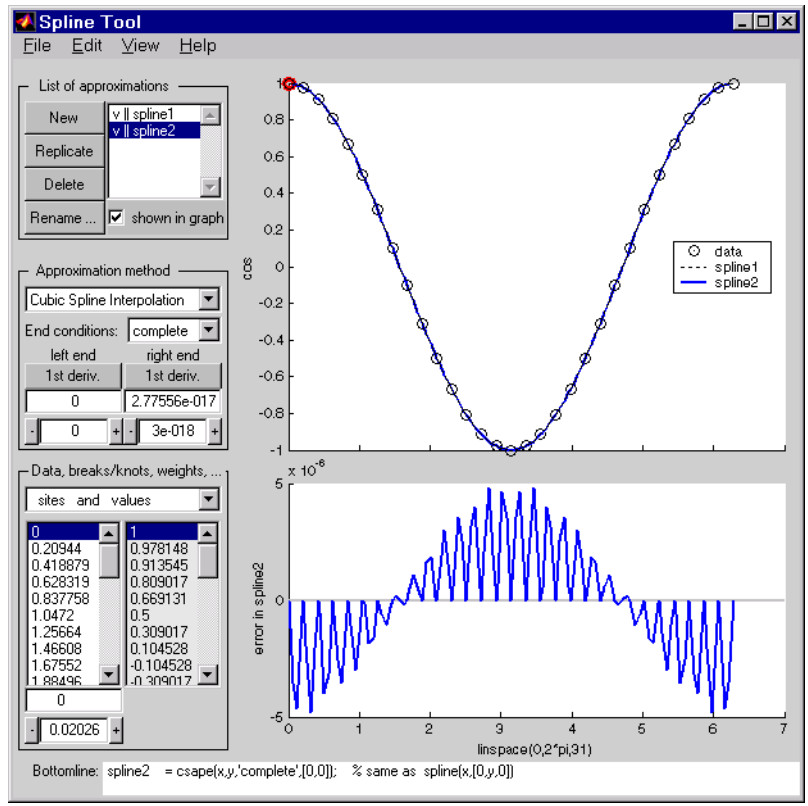
The data has sites at `x=linspace(0,2*pi,21)` and values given by `cos(x)`. This differs from simply providing the vector `y` of values in that the cosine function is explicitly recorded as the underlying function. Therefore, the error shown in the graph is the error in the spline as an approximation 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. 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.

3 In **Approximation method**, select **complete** from the list of **End conditions**.

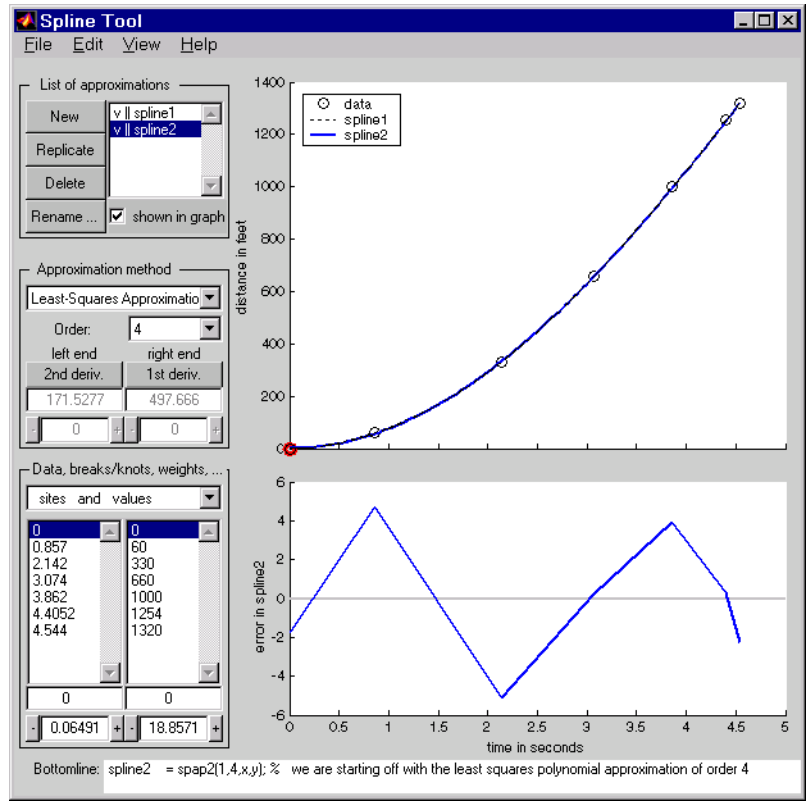
4 Adjust the initial speed by changing the first derivative at the left endpoint to zero.

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

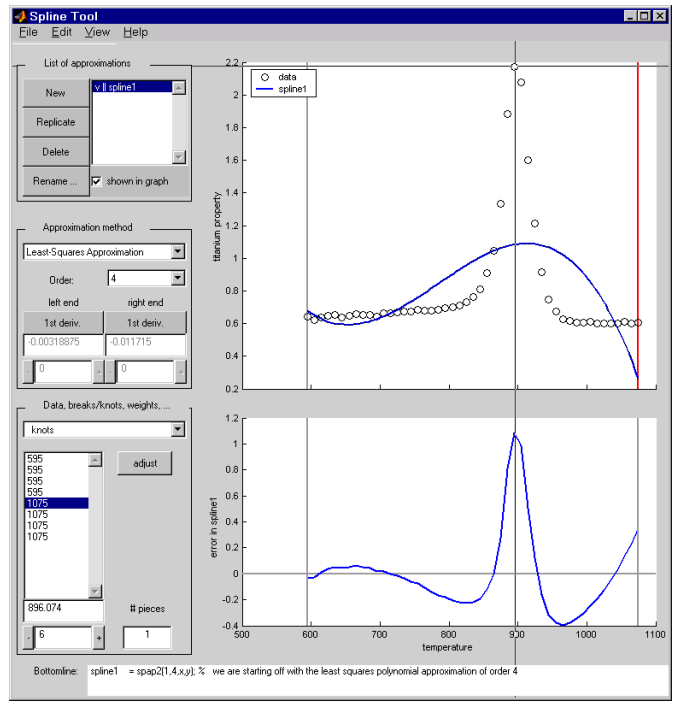
This example encourages you to place five interior knots in such a way that the least-squares approximation to these data by cubic splines has an absolute error no bigger than `.04` everywhere.

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.



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 **sites and error weights** from **Data, breaks/knots, weights**.
- Click on the highest data point in the graph and notice its site, which is indicated in the sites list box.
- 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.

8 This step modifies the weights in the roughness measure to force the approximation to balance the error with the smoothness of the second derivative.

- Choose **sites and 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 across 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.

  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.

See Also `csapi`, `csape`, `csaps`, `spap2`, `spaps`, `spapi`

Purpose

Convert locally from B-form to ppform

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.

Each row `a(.,:)` of `a` is taken to contain the B-coefficients of some spline *s* (and its order `k` is taken to be the column number of `a`). In `splpp`, the polynomial piece associated with the knot interval `[tx(.,k-1)..tx(.,k)]` is focused on. Repeated knot insertion (of the knot 0) is used to derive from the given information the B-spline coefficients `b(.,1:k)` for the same polynomial, relevant for the interval `[tx(.,k-1)..0]` (with respect to the knot sequence `[tx(.,1:k-1),0,...,0]`).

From this, the numbers $v(j) := D^{k-j} s(0-)/(k-j)!$, $j = 1, ..., k$ are computed, for each of the splines *s* described by the given knots `tx(.,:)` and coefficients `a(.,:)`.

The command `sprpp` carries out exactly the same job, but for the interval `[0..tx(.,k)]`, and therefore ends up with the values $v(j) := D^{k-j} s(0+)/(k-j)!$, $j = 1, ..., k$.

Examples

The statement `[v,b]=splpp([-2 -1 0 1],[0 1 0])` provides the sequence

```
v = -1.0000 -1.0000 0.5000
```
$= D^2 s(0-)/2, Ds(0-), s(0-)$

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 that the spline *s* in question be

$$s = 0*B(\cdot\,|\,[?,-2,-1,0]) + 1*B(\cdot\,|\,[-2,-1,0,1]) + 0*B(\cdot\,|\,[-1,0,1,?])$$

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

$$0*B(\cdot\,|\,[?,-2,-1,0]) + 1*B(\cdot\,|\,[-2,-1,0,0]) + 5*B(\cdot\,|\,[-1,0,0,?])$$

The statement `[v,b]=sprpp([-1 0 1 2],[1 0 0])` provides the sequence

$$\texttt{v = [0.5000 -1.0000 0.5000]} = [D^2s(0+)/2,\ Ds(0+),\ s(0+)]$$

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 left-most 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

$$.5*B(\cdot\,|\,[0,0,0,1]) + 0*B(\cdot\,|\,[0,0,1,2]) + 0*B(\cdot\,|\,[0,1,2,?])$$

Cautionary Note

It is assumed that `tx(.,k-1)` $<$ `0` $\le$ `tx(.,k)` for `splpp` and `tx(.,k-1)` $\le$ `0` $<$ `tx(.,k)` for `sprpp`.

Purpose

Put together a spline in B-form

Syntax

```
spmak
sp = spmak(knots,coefs)
sp = spmak(knots,coefs,sizec)
```

Description

`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 commands using the construct.

If there are no arguments, you will be prompted for `knots` and `coefs`.

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 2-space or 3-space).

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.

If `knots` is a sequence (required to be non-decreasing), then the spline is taken to be univariate, of order `k = length(knots) - size(coefs,2)`. This means that each column of `coefs` is taken to be a B-spline coefficient of the spline. This follows the general agreement in this package that, in case of a vector-valued spline, any vector in its target, be it a coefficient, or the value of the spline at a site, is written as a *1-column* matrix. In particular, the spline is `d`-vector-valued, with `d = size(coefs,1)`. Finally, the basic interval of the B-form is [`knots(1)` .. `knots(end)`].

Knot multiplicity is held to be $\le$ `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)==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 (`m`+1)-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 this array is immediately reshaped by

```
coefs = reshape(coefs, [1,size(coefs)]);
```

With this, the *i*th entry of `m`-vector `k` is computed as `length(knots{i}) - size(coef,i+1)`, `i=1:m`, and the *i*th entry of the cell array of basic intervals is set to `[knots{i}(1), knots{i}(end)]`.

Since, MATLAB suppresses trailing singleton dimensions, you must use the optional third input argument to supply the desired size of the input array `coefs` in case it has one or more trailing singleton dimensions.

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 $B(\cdot \mid t)$ in ppform. See `spalldem` for other examples.

If the intent is to construct a 2-vector valued bivariate polynomial on the rectangle $[-1..1] \times [0..1]$, 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 fail, producing a scalar-valued function of (illegal) order [2 0], while proper use of that third argument, as in

```
sp = spmak({[-1 -1 1 1],[0 1]},coefs,[2 2 1]);
```

will succeed.

See `spalldem` for other examples.

See Also

`spbrk`, `spalldem`

Diagnostics

There will be an error return if the proposed knot sequence fails to be nondecreasing or if there are not more knots than there are coefficients, or if the coefficient array is empty.

Limitations

The size of a multidimensional array created by the statement `reshape(1,[1,1,1,...,1])` will be reported as `[1 1]`. This means that the B-form created by the statement `sp = spmak(knots,1)`, with `knots` a cell array of length `m`>2, will not be interpreted correctly, by `fnval` and other commands, as the tensor-product B-spline for the given knots.

A

Glossary

Introduction

This glossary provides a thumbnail sketch of the basic mathematical terms used in this guide. But, in contrast to standard glossaries, the terms do not appear here in alphabetical order. This is not much of a disadvantage since the glossary is quite short (and all the terms appear in the Index in any case). The order is carefully chosen to have the explanation of each term only use terms discussed earlier.

In this way, the reader may, the first time around, choose to read the entire glossary from start to finish, for a cohesive introduction to these terms.

List of Terms

Intervals

Since MATLAB uses the notation `[a,b]` to indicate a matrix with the two columns, `a` and `b`, we use in this guide the notation [a .. b] to indicate the closed interval with endpoints `a` and `b`. We do the same for open and half-open intervals. For example, [a .. b) denotes the interval that includes its left endpoint, `a`, and excludes its right endpoint, `b`.

Vectors

A *d-vector* is a list of *d* real numbers, i.e., a point in R^d. In MATLAB, a *d*-vector is stored as a matrix of size [1,*d*], i.e., as a row-vector, or as a matrix of size [*d*,1], i.e., as a column-vector.

Functions

In this toolbox, the term *function* is used in its mathematical sense, and so describes any rule that associates, to each element of a certain set called its *domain*, some element in a certain set called its *target*. Common examples in this toolbox are polynomials and splines. But even a point `x` in R^d, i.e., a d-vector, may be thought of as a function, namely the function, with domain the set {1,...,d} and target the real numbers R, that, for `i=1:d`, associates to `i` the real number `x(i)`.

The *range* of a function is the set of its values.

We distinguish between scalar-valued and vector-valued functions. Scalar-valued functions have the real numbers R (or, more generally, the complex numbers) as their target, while d-vector-valued functions have R^d as their target. We also distinguish between univariate and multivariate functions. The former have some real interval, or, perhaps, all of R as their domain, while *m*-variate functions have some subset, or perhaps all, of R^m as their domain.

Placeholder notation

If f is a *bivariate* function, and y is some specific value of its second variable, then

$$f(\cdot,y)$$

is the *univariate* function whose value at x is $f(x,y)$.

Curves and surfaces vs functions

In this toolbox, the term *function* usually refers to a scalar-valued function. A vector-valued function is called here a:

- *curve* if its domain is some interval
- *surface* if its domain is some rectangle

To be sure, to a mathematician, a curve is *not* a vector-valued function on some interval but, rather, the range of such a (continuous) function, with the function itself being just one of infinitely many possible parametrizations of that curve.

Tensor products

A bivariate *tensor product* is any weighted sum of products of a function in the first variable with a function in the second variable, i.e., any function of the form

$$f(x, y) = \sum_i \sum_j a(i, j) g_i(x) h_j(y)$$

More generally, an m-variate tensor product is any weighted sum of products $g_1(x_1)g_2(x_2)\cdots g_m(x_m)$ of m univariate functions.

Polynomials

A univariate scalar-valued polynomial is specified by the list of its polynomial coefficients. The length of that list is the order of that polynomial, and, in this toolbox, the list is always stored as a row vector. Hence an m-list of polynomials of order k is always stored as a matrix of size $[m, k]$.

The coefficients in a list of polynomial coefficients are listed from "highest" to "lowest", to conform to MATLAB's convention, as in the command `polyval(a,x)`. To recall: assuming that `x` is a scalar and that `a` has `k` entries, this command returns the number

$$a(1)x^{k-1} + a(2)x^{k-2} + \cdots + a(k-1)x^1 + a(k)$$

In other words, the command treats the list `a` as the coefficients in a *power form*. For reasons of numerical stability, such a coefficient list is treated in this toolbox, more generally, as the coefficients in a *shifted*, or, *local* power form, for some given *center* `c`. This means that the value of the polynomial at some point `x` is supplied by the command `polyval(a,x-c)`.

A vector-valued polynomial is treated in exactly the same way, except that now each polynomial coefficient is a vector, say a d-vector. Correspondingly, the coefficient list now becomes a matrix of size [d,k].

Multivariate polynomials appear in this toolbox only as *tensor products*. Assuming first, for simplicity, that the polynomial in question is scalar-valued but m-variate, this means that its coefficient "list" a is an m-dimensional array, of size [k1,...,km] say, and its value at some m-vector x is, correspondingly, given by

$$\sum_{i1=1}^{k1} \cdots \sum_{im=1}^{km} a(i1,\ldots,im)(x(i1)-c(i1))^{k1-i1}\cdots(x(im)-c(im))^{km-im}$$

for some "center" c.

Piecewise-polynomials

A *piecewise-polynomial* function refers to a function put together from polynomial pieces. If the function is univariate, then, for some strictly increasing sequence $\xi_1 < \cdots < \xi_{l+1}$, and for i=1:l, it agrees with some polynomial p_i on the interval $[\xi_i..\xi_{i+1})$. Outside the interval $[\xi_1..\xi_{l+1})$, its value is given by its first, respectively its last, polynomial piece.The ξ_i are its *breaks*. All the multivariate piecewise-polynomials in this toolbox are tensor products of univariate ones.

B-splines

In this toolbox, the term *B-spline* is used in its original meaning only, as given to it by its creator, I. J. Schoenberg, and further amplified in his basic 1966 article with Curry, and used in *PGS* and many other books on splines. According to Schoenberg, the B-spline with knots t_j, ..., t_{j+k} is given by the following somewhat obscure formula (see, e.g., IX(1) in *PGS*):

$$B_{j,k}(x) = B(x|t_j,\ldots,t_{j+k}) = (t_{j+k}-t_j)[t_j,\ldots,t_{j+k}](x-\cdot)_+^{k-}$$

To be sure, this is only one of several reasonable normalizations of the B-spline, but it is the one used in this toolbox. It is chosen so that

$$\sum_{j=1}^{n} B_{j,k}(x) = 1, \qquad t_k \le x \le t_{n+1}$$

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 first-hand experience with this intriguing function. Its most important property for the purposes of this toolbox is also the reason Schoenberg used the letter B in its name:

Every space of (univariate) piecewise-polynomials of a given order has a B*asis consisting of* B*-splines.*

Splines

Consider the set

$$S := \Pi^{\mu}_{\xi,\,k}$$

of all (scalar-valued) piecewise-polynomials of order k with breaks $\xi_1 < \cdots < \xi_{l+1}$ that, for i=2:l, may have a jump across ξ_i in its μ_i-th derivative but have no jump there in any lower order derivative. This set is a linear space, in the sense 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 $B(\cdot\,|\,t_j,\ldots,t_{j+k})$, j=1:n, with the knot sequence t obtained from the break sequence ξ and the sequence μ by the following recipe:

- 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.

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. 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 $[s(x);w(x)]$. 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. Since $s = wr$, Leibniz' rule tells us that

$$D^m s = \sum_{j=0}^{m} \binom{m}{j} D^j w D^{m-j} r$$

Hence, if `v(:,j)` contains $D^{j-1}R(x)$, j=1:`m`+1, then

$$\left(\text{v(1:end-1,m+1)} - \sum_{\text{j}=1}^{\text{m}} \binom{\text{m}}{\text{j}} \text{v(end,j+1)*v(1:end-1,j+1)} \right) / \text{v(end,1)}$$

provides the value of $D^m r(x)$.

Interpolation

Interpolation is 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". Many considerations might enter that choice. One of these considerations is sure to be that one can match in this way arbitrary data. For example, polynomial interpolation is popular since, for arbitrary n *data points* (x_i,y_i) with distinct data sites, there is exactly one polynomial of order n that matches these data.

This says that choosing the f_j in the above "model" to be $f_j(x) = x^{j-1}$, $j=1:n$, guarantees exactly one such interpolant to arbitrary n data points with distinct data sites.

In spline interpolation, one chooses the f_j to be the n consecutive B-splines $B_j(x) = B(x \mid t_j,\ldots,t_{j+k})$, $j=1:n$, of order k for some knot sequence $t_1 \le t_2 \le \cdots \le t_{n+k}$. For this choice, we have the following important theorem.

Schoenberg-Whitney Theorem

Let $x_1 < x_2 < \cdots < x_n$. For arbitrary corresponding values y_i, $i=1:n$, there exists exactly one spline f of order k with knot sequence t_j, $j=1:n+k$, so that $f(x_i) = y_i$, $i = 1:n$, if and only if the sites satisfy the Schoenberg-Whitney conditions of order k with respect to that knot sequence t, namely

$$t_i \le x_i \le t_{i+k}, \qquad i = 1:n,$$

with equality allowed only if the knot in question has multiplicity k, i.e., appears k times in t. In that case, the spline being constructed may have a jump discontinuity across that knot, and it is its limit from the right or left at that knot that matches the value given there.

Least-squares approximation

In least-squares approximation, the data may be matched only approximately. Specifically, the linear system

$$f(x_i) = \sum_j f_j(x_i) a_j = y_i, \qquad i = 1:n,$$

is solved in the least-squares sense. In this, some weighting is involved, i.e., the coefficients a_j are determined so as to minimize the error measure

$$E(f) = \sum_i w_i (y_i - f(x_i))^2$$

for certain nonnegative weights w_i at the user's disposal, with the default being to have all these weights the same.

Smoothing

In spline smoothing, one also tries to make such an error measure small, but tries, at the same time, to keep the following roughness measure small,

$$F(D^m f) = \int_{x_1}^{x_n} \lambda(x)(D^m f(x))^2 \, dx$$

with λ a nonnegative weight function that is usually just the constant function 1, and $D^m f$ the m-th derivative of f. The competing claims of small $E(f)$ and small $F(D^m f)$ are mediated by a smoothing parameter, for example by minimizing

$$\rho E(f) + F(D^m f)$$

for some choice of ρ, and over all f for which this expression makes sense.

Remarkably, if the roughness weight λ is constant, then the unique minimizer f is a spline of order $2m$, with knots only at the data sites, and all the interior knots simple, and with its derivatives of orders $m,\ldots,2m$-2 equal to zero at the two extreme data sites, the so-called “natural” end conditions. The larger the smoothing parameter ρ used, the more closely f matches the given data and the larger is its m-th derivative.

Index

Numerics

A

B

C

M

N

O

P

Q

R

S

T

U

V

W