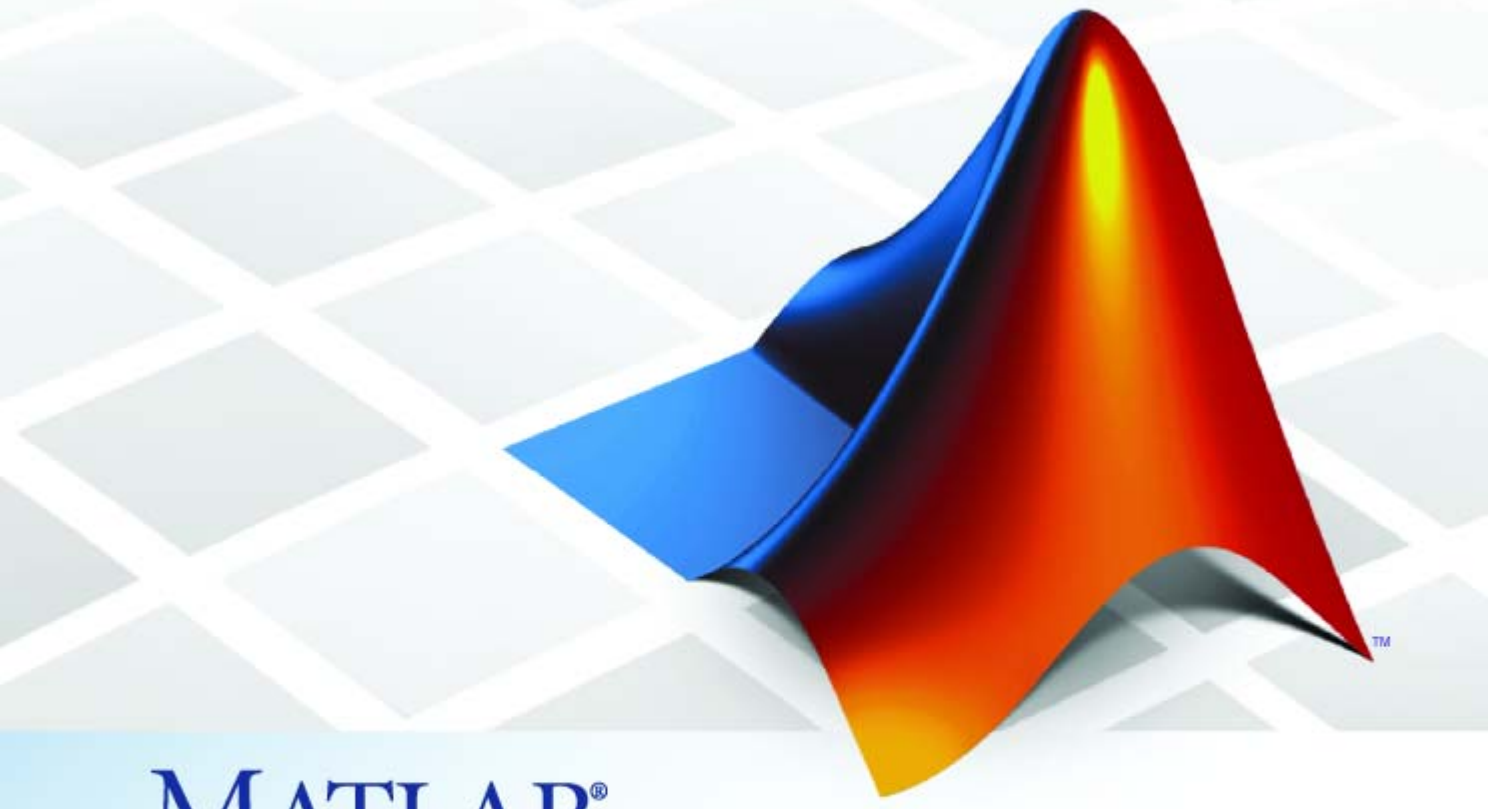
Spline Toolbox™ 3

User's Guide



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Spline Toolbox™ User's Guide

Revision History

March 1990	First printing	New for Version 1.0
November 1992	Second printing	Revised for Version 1.1
January 1998	Third printing	Revised for Version 2.0 (Release 10)
January 1999	Online only	Revised for Version 2.0.1 (Release 11)
September 2000	Fourth printing	Revised for Version 3.0 (Release 12)
May 2001	Fifth printing	Minor revision for Version 3.0 (Release 12.1)
December 2001	Online only	Revised for Version 3.1
February 2003	Sixth printing	Revised for Version 3.2 (Release 13)
June 2004	Seventh printing	Revised for Version 3.2.1 (Release 14)
June 2005	Eighth printing	Minor revision for Version 3.2.1
September 2005	Online only	Minor revision for Version 3.2.2 (Release 14SP3)
March 2006	Ninth printing	Revised for Version 3.3 (Release 2006a)
September 2006	Online only	Minor revision for Version 3.3.1 (Release 2006b)
September 2006	Tenth printing	Version 3.3.1
March 2007	Online only	Revised for Version 3.3.2 (Release 2007a)
May 2007	Eleventh printing	Version 3.3.2
September 2007	Online only	Revised for Version 3.3.3 (Release 2007b)
March 2008	Online only	Revised for Version 3.3.4 (Release 2008a)
October 2008	Online only	Revised for Version 3.3.5 (Release 2008b)
March 2009	Online only	Revised for Version 3.3.6 (Release 2009a)
September 2009	Online only	Revised for Version 3.3.7 (Release 2009b)
March 2010	Online only	Revised for Version 3.3.8 (Release 2010a)

Acknowledgments

The MathWorks™ would like to acknowledge the contributions of **Carl de Boor** to the Spline Toolbox™. Professor de Boor authored the Spline Toolbox from its first release until Version 3.3.4 (2008).

Professor de Boor received the John von Neumann Prize in 1996 and the National Medal of Science in 2003. He is a member of both the American Academy of Arts and Sciences and the National Academy of Sciences. He is the author of *A Practical Guide to Splines* (Springer, 2001).

Contents

1 Getting Started

2 Some Simple Examples

3 Splines: An Overview

4 The ppform

5 The B-form

6 Tensor Product Splines

7 NURBS and Other Rational Splines

8 The stform

9 Advanced Examples

Function Reference

10

Functions – Alphabetical List

11

Glossary

A

1

Getting Started

Product Overview

Spline Toolbox software contains versions of the essential MATLAB® programs of the B-spline package (extended to handle also *vector*-valued splines) as described in *A Practical Guide to Splines,* (Applied Math. Sciences Vol. 27, Springer Verlag, New York (1978), xxiv + 392p; revised edition (2001), xviii+346p), hereafter referred to as *PGS*. The toolbox makes it easy to create and work with piecewise-polynomial functions.

The typical use envisioned for this toolbox involves the construction and subsequent use of a piecewise-polynomial approximation. This construction would involve data fitting, but there is a wide range of possible data that could be fit. In the simplest situation, one is given points (t_i,y_i) and is looking for a piecewise-polynomial function f that satisfies $f(t_i) = y_i$, all i, more or less. An exact fit would involve *interpolation*, an approximate fit might involve *least-squares approximation* or the *smoothing spline*. But the function to be approximated may also be described in more implicit ways, for example as the solution of a differential or integral equation. In such a case, the data would be of the form $(Af)(t_i)$, with A some differential or integral operator. On the other hand, one might want to construct a spline *curve* whose exact location is less important than is its overall shape. Finally, in all of this, one might be looking for functions of more than one variable, such as *tensor product splines*.

Care has been taken to make this work as painless and intuitive as possible. In particular, the user need not worry about just how splines are constructed or stored for later use, nor need the casual user worry about such items as "breaks" or "knots" or "coefficients". It is enough to know that each function constructed is just another variable that is freely usable as input (where appropriate) to many of the commands, including all commands beginning with `fn`, which stands for `function`. At times, it may be also useful to know that, internal to the toolbox, splines are stored in different forms, with the command `fn2fm` available to convert between forms.

At present, the toolbox supports two major forms for the representation of piecewise-polynomial functions, because each has been found to be superior to the other in certain common situations. The B-form is particularly useful during the construction of a spline, while the ppform is more efficient when the piecewise-polynomial function is to be evaluated extensively. These two forms are almost exactly the B-representation and the pp representation used in *PGS*.

But, over the years, the Spline Toolbox product has gone beyond the programs in *PGS*. The toolbox now supports the 'scattered translates' form, or stform, in order to handle the construction and use of bivariate thin-plate splines, and also two ways to represent rational splines, the rBform and the rpform, in order to handle NURBS.

Splines can be very effective for data fitting because the linear systems to be solved for this are banded, hence the work needed for their solution, done properly, grows only linearly with the number of data points. In particular, the MATLAB sparse matrix facilities are used in the Spline Toolbox product when that is more efficient than the toolbox's own equation solver, `slvblk`, which relies on the fact that some of the linear systems here are even almost block diagonal.

All polynomial spline construction commands are equipped to produce bivariate (or even multivariate) piecewise-polynomial functions as tensor products of the univariate functions used here, and the various `fn...` commands also work for these multivariate functions.

There are various examples, all accessible through the Demos tab in the MATLAB Help browser. You are strongly urged to have a look at some of them, or at the GUI `splinetool`, before attempting to use this toolbox, or even before reading on.

MATLAB Splines

The MATLAB technical computing environment provides spline approximation via the command `spline`. If called in the form `cs = spline(x,y)`, it returns the ppform of the cubic spline with break sequence `x` that takes the value `y(i)` at `x(i)`, all `i`, and satisfies the not-a-knot end condition. In other words, the command `cs = spline(x,y)` gives the same result as the command `cs = csapi(x,y)` available in the Spline Toolbox product. But only the latter also works when `x,y` describe multivariate gridded data. In MATLAB, cubic spline interpolation to multivariate gridded data is provided by the command `interpn(x1,...,xd,v,y1,...,yd,'spline')` which returns values of the interpolating tensor product cubic spline at the grid specified by `y1,...,yd`.

Further, any of the Spline Toolbox `fn...` commands can be applied to the output of the MATLAB `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.

Expected Background

The Spline Toolbox product started out as an extension of the MATLAB environment of interest to experts in spline approximation, to aid them in the construction and testing of new methods of spline approximation. Such people will have mastered the material in *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 Chapter 2, “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 guide provides definitions of almost all the mathematical terms used in this document.

Technical Conventions

Vectors

The Spline Toolbox product can handle *vector*-valued splines, i.e., splines whose values lie in R^d. Since MATLAB started out with just one variable type, that of a matrix, there is even now some uncertainty about how to deal with vectors, i.e., lists of numbers. MATLAB sometimes stores such a list in a matrix with just one row, and other times in a matrix with just one column. In the first instance, such a *1-row matrix* is called a row-vector; in the second instance, such a *1-column matrix* is called a column-vector. Either way, these are merely different ways for *storing* vectors, not different *kinds* of vectors.

In this toolbox, *vectors*, i.e., lists of numbers, may also end up stored in a 1-row matrix or in a 1-column matrix, but with the following agreements.

A point in R^d, i.e., a d-vector, is always stored as a column vector. In particular, if you want to supply an `n`-list of `d`-vectors to one of the commands, you are expected to provide that list as the `n` columns of a matrix of size `[d,n]`.

While other lists of numbers (e.g., a knot sequence or a break sequence) may be stored internally as row vectors, you may supply such lists as you please, as a row vector or a column vector.

Naming Conventions

Most of the Spline Toolbox commands in this toolbox have names that follow one of the following patterns:

`cs...` commands construct cubic splines (in ppform)

`sp...` commands construct splines in B-form

`fn...` commands operate on spline functions

`..2..` commands convert something

`..api` commands construct an approximation by interpolation

`..aps` commands construct an approximation by smoothing

`..ap2` commands construct a least-squares approximation

`...knt` commands construct (part of) a particular knot sequence

`...dem` commands are demonstrations now reached via the Demos tag in the MATLAB Help browser.

Some of these naming conventions are the result of a discussion with Jörg Peters, then a graduate student in Computer Sciences at the University of Wisconsin-Madison.

Note See the "Glossary" for information about notation used in this book.

Using Spline Toolbox Functions

For ease of use, most Spline Toolbox functions have default arguments. In the reference entry under Syntax, we usually first list the function with all *necessary* input arguments and then with all *possible* input arguments. When there is more than one optional argument, then, sometimes, but not always, their exact order is immaterial. When their order does matter, you have to specify every optional argument preceding the one(s) you are interested in. In this situation, you can specify the default value for an optional argument by using `[]` (the empty matrix) as the input for it. The description in the reference page tells you the default value for each optional input argument.

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

2

Some Simple Examples

Introduction

These examples provide 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 the Demos tab in the MATLAB Help browser. In addition, the command `splinetool` provides a graphical user interface (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.

Cubic Spline Interpolation

In this section...

Cubic Spline Interpolant of Smooth Data

Suppose you want to interpolate some smooth data, e.g., to

```
rand('seed',6), x = (4*pi)*[0 1 rand(1,15)]; y = sin(x);
```

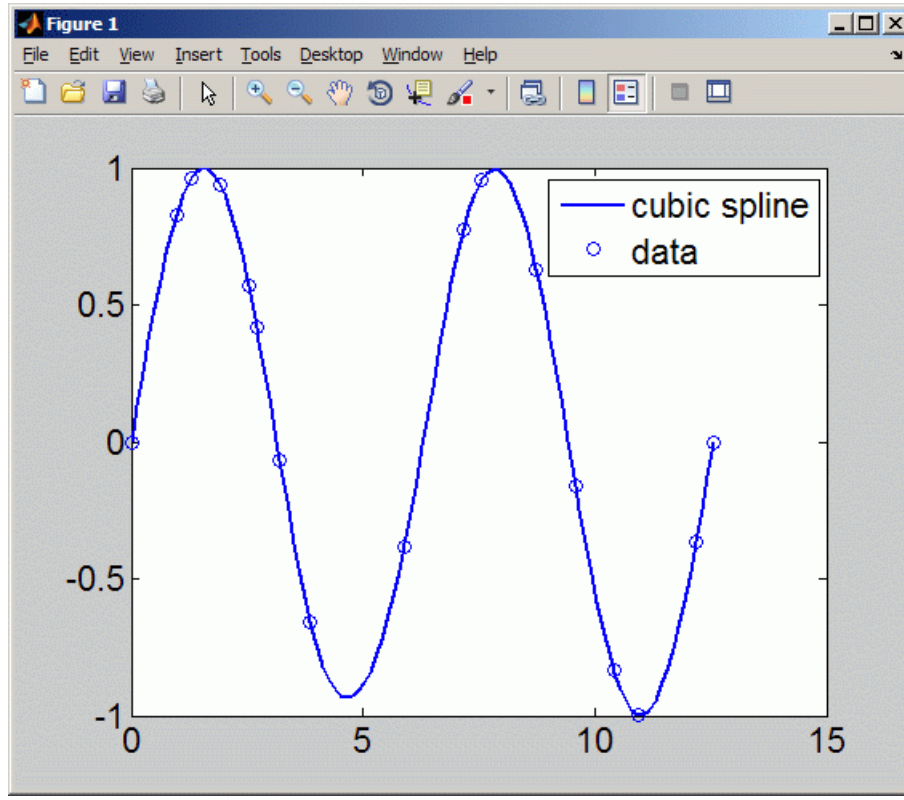
You can use the cubic spline interpolant obtained by

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

and plot the spline, along with the data, with the following code:

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

This produces a figure like the following.



Cubic Spline Interpolant of Smooth Data

This is, more precisely, the cubic spline interpolant with the not-a-knot end conditions, meaning that it is the unique piecewise cubic polynomial with two continuous derivatives with breaks at all *interior* data sites except for the leftmost and the rightmost one. It is the same interpolant as produced by the MATLAB `spline` command, `spline(x,y)`.

Periodic Data

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

```
diff(fnval(fnder(cs),[0 4*pi]))
```

```
ans = -.0100
```

which is not so good. If you prefer to get an interpolant whose first and second derivatives at the two endpoints, `0` and `4*pi`, match, use instead the command `csape` which permits specification of many different kinds of end conditions, including periodic end conditions. So, use instead

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

for which you get

```
diff(fnval(fnder(pcs),[0 4*pi]))
```

Output is `ans = 0` as the difference of end slopes. Even the difference in end second derivatives is small:

```
diff(fnval(fnder(pcs,2),[0 4*pi]))
```

Output is `ans = -4.6074e-015`.

Other End Conditions

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

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

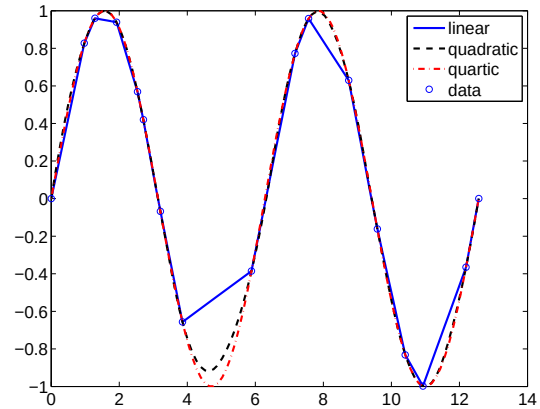
provides the cubic spline interpolant with breaks at the $x(i)$ and with its slope at the leftmost data site equal to 3, and its second derivative at the rightmost data site equal to -4.

General Spline Interpolation

If you want to interpolate at sites other than the breaks and/or by splines other than cubic splines with simple knots, then you use the `spapi` command. In its simplest form, you would say `sp = spapi(k,x,y)`; in which the first argument, `k`, specifies the *order* of the interpolating spline; this is the number of coefficients in each polynomial piece, i.e., 1 more than the nominal degree of its polynomial pieces. For example, the next figure shows a linear, a quadratic, and a quartic spline interpolant to your data, as obtained by the statements

```
sp2 = spapi(2,x,y); fnplt(sp2,2), hold on
```

```
sp3 = spapi(3,x,y); fnplt(sp3,2,'k--'), set(gca,'Fontsize',16)
sp5 = spapi(5,x,y); fnplt(sp5,2,'r-.'), plot(x,y,'o')
legend('linear','quadratic','quartic','data'), hold off
```

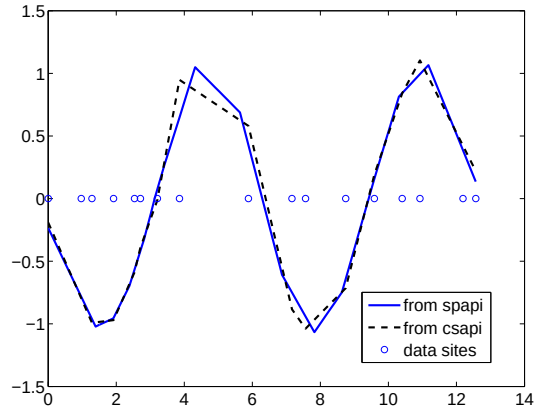


Spline Interpolants of Various Orders of Smooth Data

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

```
fnplt(fnder(spapi(4,x,y),2)), hold on, set(gca,'Fontsize',16)
fnplt(fnder(csapi(x,y),2),2,'k--'),plot(x,zeros(size(x)),'o')
legend('from spapi','from csapi','data sites'), hold off
```

This gives the following graph:



Second Derivative of Two Cubic Spline Interpolants of the Same Smooth Data

Since the second derivative of a cubic spline is a broken line, with vertices at the breaks of the spline, you can see clearly that `csapi` places breaks at the data sites, while `spapi` does not.

Knot Choices

It is, in fact, possible to specify explicitly just where the spline interpolant should have its breaks, using the command `sp = spapi(knots,x,y)`; in which the sequence `knots` supplies, in a certain way, the breaks to be used. For example, recalling that you had chosen `y` to be `sin(x)`, the command

```
ch = spapi(augknt(x,4,2),[x x],[y cos(x)]);
```

provides a cubic Hermite interpolant to the sine function, namely the piecewise cubic function, with breaks at all the `x(i)`'s, that matches the sine function in value *and* slope at all the `x(i)`'s. This makes the interpolant continuous with continuous first derivative but, in general, it has jumps across the breaks in its second derivative. Just how does this command know which part of the data value array `[y cos(x)]` supplies the values and which the slopes? Notice that the data site array here is given as `[x x]`, i.e., each data site appears twice. Also notice that `y(i)` is associated with the first occurrence of `x(i)`, and `cos(x(i))` is associated with the second occurrence of `x(i)`. The data value associated with the first appearance of a data site is taken

to be a function value; the data value associated with the second appearance is taken to be a slope. If there were a third appearance of that data site, the corresponding data value would be taken as the second derivative value to be matched at that site. See Chapter 5, “The B-form” for a discussion of the command `augknt` used here to generate the appropriate "knot sequence".

Smoothing

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

```
noisy = y + .3*(rand(size(x))-.5);
```

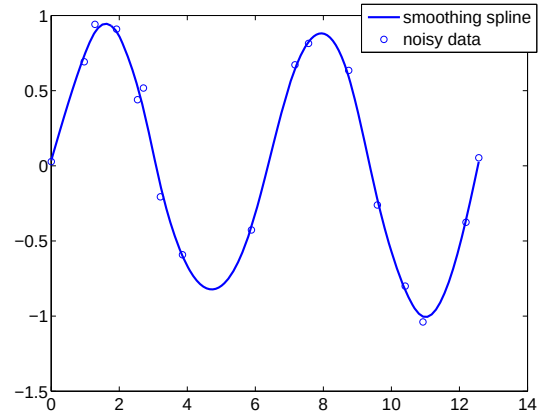
Then you might prefer to approximate instead. For example, you might try the cubic smoothing spline, obtained by the command

```
scs = csaps(x,noisy);
```

and plotted by

```
fnplt(scs,2), hold on, plot(x,noisy,'o'), set(gca,'Fontsize',16)
legend('smoothing spline','noisy data'), hold off
```

This produces a figure like this:

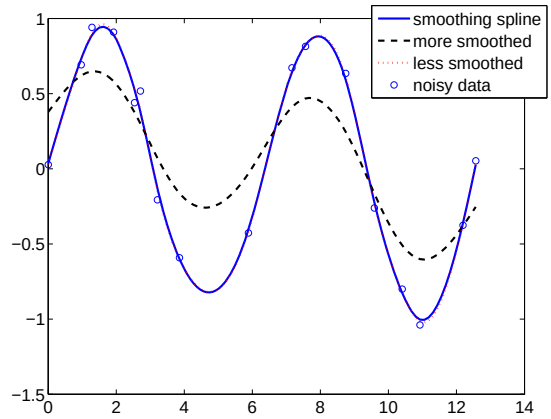


Cubic Smoothing Spline of Noisy Data

If you don't like the level of smoothing done by `csaps(x,y)`, you can change it by specifying the smoothing parameter, `p`, as an optional third argument. Choose this number anywhere between 0 and 1. As `p` changes from 0 to 1, the smoothing spline changes, correspondingly, from one extreme, the least squares straight-line approximation to the data, to the other extreme, the "natural" cubic spline interpolant to the data. Since `csaps` returns the smoothing parameter actually used as an optional second output, you could now experiment, as follows:

```
[scs,p] = csaps(x,noisy); fnplt(scs,2), hold on
fnplt(csaps(x,noisy,p/2),2,'k--'), set(gca,'Fontsize',16)
fnplt(csaps(x,noisy,(1+p)/2),2,'r:'), plot(x,noisy,'o')
legend('smoothing spline','more smoothed','less smoothed',...
'noisy data'), hold off
```

This produces the following picture.



Noisy Data More or Less Smoothed

At times, you might prefer simply to get the smoothest cubic spline `sp` that is within a specified tolerance `tol` of the given data in the sense that `norm(noisy - fnval(sp,x))^2 <= tol`. You create this spline with the command `sp = spaps(x,noisy,tol)` for your defined tolerance `tol`.

Least Squares

If you prefer a least squares approximant, you can obtain it by the statement `sp = spap2(knots,k,x,y)`; in which both the knot sequence `knots` and the order `k` of the spline must be provided.

The popular choice for the order is 4, and that gives you a cubic spline. If you have no clear idea of how to choose the knots, simply specify the number of polynomial pieces you want used. For example,

```
sp = spap2(3,4,x,y);
```

gives a cubic spline consisting of three polynomial pieces. If the resulting error is uneven, you might try for a better knot distribution by using `newknt` as follows:

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

Using the Spline Fits

You can use the following commands with any example spline, such as the `cs`, `ch` and `sp` examples constructed in the section "Cubic Spline Interpolation" on page 2-3.

First construct a spline, for example:

```
sp = spmak(1:6,0:2)
```

To display a plot of the spline:

```
fnplt(sp)
```

To get the value at `a`, use the syntax `fnval(f,a)`, for example:

```
fnval(sp,4)
```

To construct the spline's second derivative:

```
DDf = fnder(fnder(sp))
```

An alternative way to construct the second derivative:

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

To obtain the spline's definite integral over an interval [`a`..`b`], in this example from 2 to 5:

```
diff(fnval(fnint(sp),[2;5]))
```

To compute the difference between two splines, use the form `fncmb(sp1,'-',sp2)`, for example:

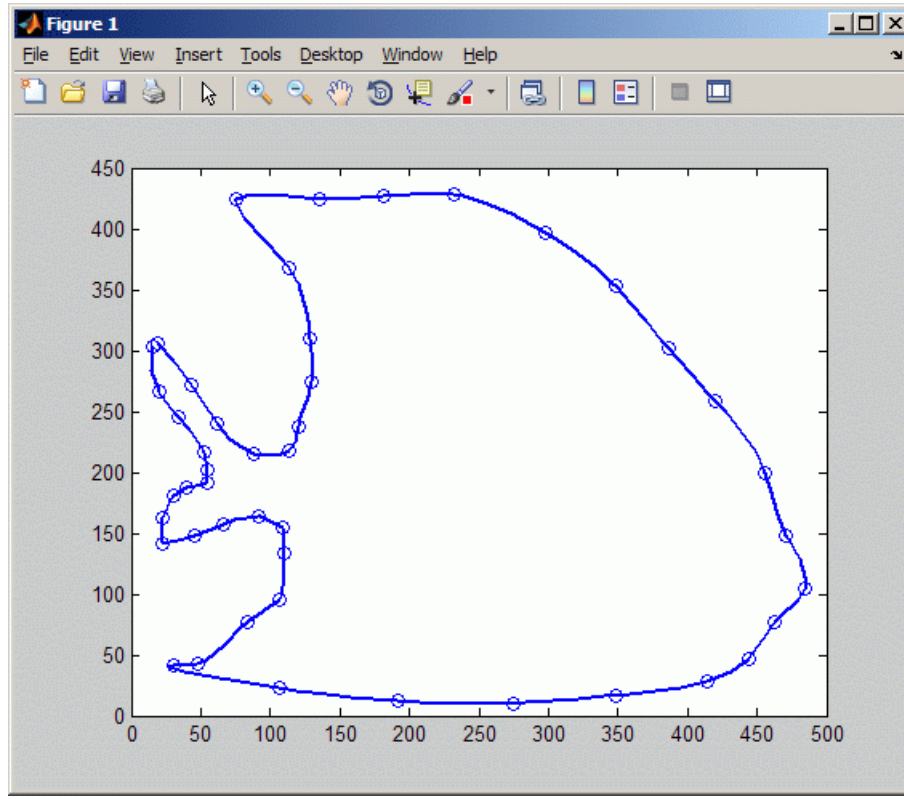
```
fncmb(sp,'-',DDf);
```

Vector-Valued Functions

The toolbox supports *vector-valued* splines. For example, if you want a spline *curve* through given planar points $(x(i), y(i))$, $i = 1, \ldots, n$, then the following code defines some data and then creates and plots such a spline curve, using chord-length parametrization and cubic spline interpolation with the not-a-knot end condition.

```
x=[19 43 62 88 114 120 130 129 113 76 135 182 232 298 ...
 348 386 420 456 471 485 463 444 414 348 275 192 106 ...
 30 48 83 107 110 109 92 66 45 23 22 30 40 55 55 52 34 20 16];
y=[306 272 240 215 218 237 275 310 368 424 425 427 428 ...
 397 353 302 259 200 148 105 77 47 28 17 10 12 23 41 43 ...
 77 96 133 155 164 157 148 142 162 181 187 192 202 217 245 266 303];

xy = [x;y]; df = diff(xy,1,2);
t = cumsum([0, sqrt([1 1]*(df.*df))]);
cv = csapi(t,xy);
fnplt(cv), hold on, plot(x,y,'o'), hold off
```



If you then wanted to know the area enclosed by this curve, you would want to evaluate the integral $\int y(t)\mathrm{d}x(t) = \int y(t)Dx(t)\mathrm{d}t$, with $(x(t),y(t))$ the point on the curve corresponding to the parameter value t. For the spline curve in `cv` just constructed, this can be done exactly in one (somewhat complicated) command:

```
area = diff(fnval(fnint( ...
       fncmb(fncmb(cv,[0 1]),'*',fnder(fncmb(cv,[1 0]))) ...
                          ),fnbrk(cv,'interval')));
```

To explain, `y=fncmb(cv,[0 1])` picks out the second component of the curve in `cv`, `Dx=fnder(fncmb(cv,[1 0]))` provides the derivative of the first component, and `yDx=fncmb(y,'*',Dx)` constructs their pointwise product. Then `IyDx=fnint(yDx)` constructs the indefinite integral of `yDx` and, finally,

`diff(fnval(IyDx,fnbrk(cv,'interval')))` evaluates that indefinite integral at the endpoints of the basic interval and then takes the difference of the second from the first value, thus getting the definite integral of `yDx` over its basic interval. Depending on whether the enclosed area is to the right or to the left as the curve point travels with increasing parameter, the resulting number is either positive or negative.

Further, all the values `Y` (if any) for which the point `(X,Y)` lies on the spline curve in `cv` just constructed can be obtained by the following (somewhat complicated) command:

```
X=250; %Define a value of X
Y = fnval(fncmb(cv,[0 1]), ...
         mean(fnzeros(fncmb(fncmb(cv,[1 0]),'-',X))))
```

To explain: `x = fncmb(cv,[1 0])` picks out the first component of the curve in `cv`; `xmX = fncmb(x,'-',X)` translates that component by `X`; `t = mean(fnzeros(xmX))` provides all the parameter values for which `xmX` is zero, i.e., for which the first component of the curve equals `X`; `y = fncmb(cv,[0,1])` picks out the second component of the curve in `cv`; and, finally, `Y = fnval(y,t)` evaluates that second component at those parameter sites at which the first component of the curve in `cv` equals `X`.

As another example of the use of vector-valued functions, suppose that you have solved the equations of motion of a particle in some specified force field in the plane, obtaining, at discrete times $t_j = t(j), j = 1{:}n$, the position $(x(t_j),y(t_j))$ as well as the velocity $(\dot{x}(t_j),\dot{y}(t_j))$ stored in the 4-vector $z(:,j)$, as you would if, in the standard way, you had solved the equivalent first-order system numerically. Then the following statement, which uses cubic Hermite interpolation, will produce a plot of the particle path:`fnplt(spapi(augknt(t,4,2),t,reshape(z,2,2*n))`.

Fitting Values at N-D Grid

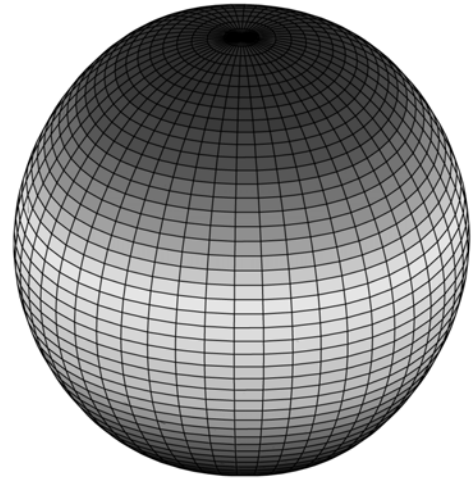
Vector-valued splines are also used in the approximation to *gridded data*, in any number of variables, using *tensor-product* splines. The same spline-construction commands are used, only the form of the input differs. For example, if `x` is an `m`-vector, `y` is an `n`-vector, and `z` is an array of size `[m,n]`, then `cs = csapi({x,y},z);` describes a bicubic spline *f* satisfying *f*(x(*i*),y(*j*))=z(*i*,*j*) for *i*=1:m, *j*=1:n. Such a multivariate spline can be vector-valued. For example,

```
x = 0:4; y=-2:2; s2 = 1/sqrt(2);
z(3,:,:) = [0 1 s2 0 -s2 -1 0].'*[1 1 1 1 1];
z(2,:,:) = [1 0 s2 1 s2 0 -1].'*[0 1 0 -1 0];
z(1,:,:) = [1 0 s2 1 s2 0 -1].'*[1 0 -1 0 1];
sph = csape({x,y},z,{'clamped','periodic'});
fnplt(sph), axis equal, axis off
```

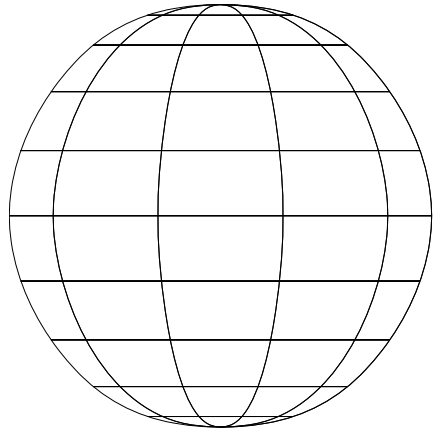
gives a perfectly acceptable sphere. Its projection onto the (x,z)-plane is plotted by

```
fnplt(fncmb(sph,[1 0 0; 0 0 1])), axis equal, axis off
```

Both plots are shown below.



A Sphere Made by a 3-D-Valued Bivariate Tensor Product Spline



Planar Projection of Spline Sphere

Fitting Values at Scattered 2-D Sites

Tensor-product splines are good for gridded (bivariate and even multivariate) data. For work with scattered bivariate data, the toolbox provides the thin-plate smoothing spline. Suppose you have given data values `y(j)` at scattered data sites `x(:,j)`, `j=1:N`, in the plane. To give a specific example,

```
n = 65; t = linspace(0,2*pi,n+1);
x = [cos(t);sin(t)]; x(:,end) = [0;0];
```

provides 65 sites, namely 64 points equally spaced on the unit circle, plus the center of that circle. Here are corresponding data values, namely noisy values of the very nice function $g(x) = (x(1)+1/2)^2 + (x(2)+1/2)^2$.

```
y = (x(1,:)+.5).^2 + (x(2,:)+.5).^2;
noisy = y + (rand(size(y))-.5)/3;
```

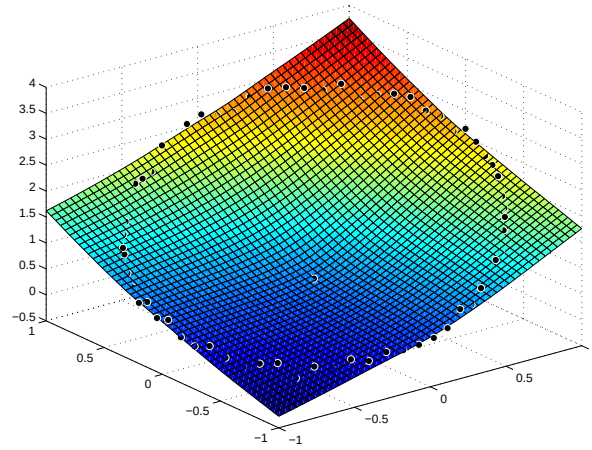
Then you can compute a reasonable approximation to these data by

```
st = tpaps(x,noisy);
```

and plot the resulting approximation along with the noisy data by

```
fnplt(st); hold on
plot3(x(1,:),x(2,:),noisy,'wo','markerfacecolor','k')
hold off
```

and so produce the following picture:



Thin-Plate Smoothing Spline Approximation to Noisy Data

3

Splines: An Overview

Introduction

This chapter provides a quick overview of the mathematics that underlies the various commands in the Spline Toolbox product. In the process, the technical terms and notation used throughout this documentation (and in the online help for individual commands) are introduced. Another source of information about the latter is the Glossary.

Polynomials vs. Splines

Polynomials are the approximating functions of choice when a smooth function is to be approximated locally. For example, the truncated Taylor series

$$\sum_{i=0}^{n} (x-a)^i \, D^i f(a) / i!$$

provides a satisfactory approximation for *f(x)* if *f* is sufficiently smooth and x is sufficiently close to a. But if a function is to be approximated on a larger interval, the degree, n, of the approximating polynomial may have to be chosen unacceptably large. The alternative is to subdivide the interval *[a..b]* of approximation into sufficiently small intervals $[\xi_j..\xi_{j+1}]$, with $a = \xi_1 < \cdots < \xi_{l+1} = b$, so that, on each such interval, a polynomial p_j of relatively low degree can provide a good approximation to *f*. This can even be done in such a way that the polynomial pieces blend smoothly, i.e., so that the resulting patched or composite function *s(x)* that equals $p_j(x)$ for $x \in [\xi_j \; \xi_{j+1}]$, all j, has several continuous derivatives. Any such smooth piecewise polynomial function is called a *spline*. I.J. Schoenberg coined this term because a twice continuously differentiable cubic spline with sufficiently small first derivative approximates the shape of a draftsman's spline.

There are two commonly used ways to represent a polynomial spline, the ppform and the B-form. In this toolbox, a spline in ppform is often referred to as a *piecewise polynomial*, while a piecewise polynomial in B-form is often referred to as a spline. This reflects the fact that piecewise polynomials and (polynomial) splines are just two different views of the same thing.

ppform

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

$$p_j(x) = \sum_{i=1}^{k} \left(x - \xi_j\right)^{k-i} c_{ji}, \quad j = 1:l$$

For example, a cubic spline is of order 4, corresponding to the fact that it requires four coefficients to specify a cubic polynomial. The ppform is convenient for the evaluation and other *uses* of a spline.

B-form

The *B-form* has become the standard way to represent a spline during its *construction*, because the B-form makes it easy to build in smoothness requirements across breaks and leads to banded linear systems. The B-form describes a spline as a weighted sum

$$\sum_{j=1}^{n} B_{j,k} a_j$$

of B-splines of the required order k, with their number, n, at least as big as k–1 plus the number of polynomial pieces that make up the spline. Here, $B_{j,k} = B(\cdot \mid t_j, \ldots, t_{j+k})$ 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

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

Knot Multiplicity

The multiplicity of the knots governs the smoothness, in the following way: If the number τ occurs exactly r times in the sequence $t_j,\ldots t_{j+k}$, then $B_{j,k}$ and its first $k-r-1$ derivatives are continuous across the break τ, while the $(k-r)$th derivative has a jump at τ. You can experiment with all these properties of the B-spline in a very visual and interactive way using the command `bspligui`.

B-Spline Properties

Because $B_{j,k}$ is nonzero only on the interval $(t_j..t_{j+k})$, the linear system for the B-spline coefficients of the spline to be determined, by interpolation or least squares approximation, or even as the approximate solution of some differential equation, is *banded*, making the solving of that linear system particularly easy. For example, to construct a spline s of order k with knot sequence $t_1 \le t_2 \le \cdots \le t_{n+k}$ so that $s(x_i)=y_i$ for $i=1, ..., n$, use the linear system

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

for the unknown B-spline coefficients a_j in which each equation has at most k nonzero entries.

Also, many theoretical facts concerning splines are most easily stated and/or proved in terms of B-splines. For example, it is possible to match arbitrary data at sites $x_1 < \cdots < x_n$ uniquely by a spline of order k with knot sequence $(t_1, ..., t_{n+k})$ if and only if $B_{j,k}(x_j)\neq 0$ for all j (Schoenberg-Whitney Conditions). Computations with B-splines are facilitated by stable *recurrence relations*

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

which are also of help in the conversion from B-form to ppform. The dual functional

$$a_j(s) := \sum_{i<k} (-D)^{k-i-1}\, \Psi_j(\tau) D^i s(\tau)$$

provides a useful expression for the jth B-spline coefficient of the spline s in terms of its value and derivatives at an arbitrary site τ between t_j and t_{j+k}, and with $\Psi_j(t) := (t_{j+1}-t)\cdots(t_{j+k-1}-t)/(k-1)!$ It can be used to show that $a_j(s)$ is closely related to s on the interval $[t_j..t_{j+k}]$, and seems the most efficient means for converting from ppform to B-form.

Constructive vs. Variational

The above *constructive* approach is not the only avenue to splines. In the *variational* approach, a spline is obtained as a *best interpolant*, e.g., as the function with smallest *m*th derivative among all those matching prescribed function values at certain sites. As it turns out, among the many such splines available, only those that are piecewise-polynomials or, perhaps, piecewise-exponentials have found much use. Of particular practical interest is the *smoothing spline* $s = s_p$ which, for given data (x_i,y_i) with $x \in [a..b]$, all i, and given corresponding positive weights w_i, and for given *smoothing parameter* p, minimizes

$$p\sum_i w_i \left|y_i - f(x_i)\right|^2 + (1-p)\int_a^b \left|D^m f(t)\right|^2 dt$$

over all functions f with m derivatives. It turns out that the smoothing spline s is a spline of order $2m$ with a break at every data site. The smoothing parameter, p, is chosen artfully to strike the right balance between wanting the *error measure*

$$E(s) = \sum_i w_i \left|y_i - s(x_i)\right|^2$$

small and wanting the *roughness measure*

$$F\left(D^m s\right) = \int_a^b \left|D^m s(t)\right|^2 dt$$

small. The hope is that s contains as much of the information, and as little of the supposed noise, in the data as possible. One approach to this (used in `spaps`) is to make $F(D^m f)$ as small as possible subject to the condition that $E(f)$ be no bigger than a prescribed tolerance. For computational reasons, `spaps` uses the (equivalent) smoothing parameter $\rho = p/(1-p)$, i.e., minimizes $\rho E(f) + F(D^m f)$. Also, it is useful at times to use the more flexible roughness measure

$$F\left(D^m s\right) = \int_a^b \lambda(t)\left|D^m s(t)\right|^2 dt$$

with λ a suitable positive weight function.

Multivariate Splines

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

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

with $B_{u,k}, B_{v,l}, B_{w,m}$ univariate B-splines. Correspondingly, this spline is of order k in x, of order l in y, and of order m in z. Similarly, the ppform of a tensor-product spline is specified by break sequences in each of the variables and, for each hyper-rectangle thereby specified, a coefficient array. Further, as in the univariate case, the coefficients may be vectors, typically 2-vectors or 3-vectors, making it possible to represent, e.g., certain surfaces in $\Re^3$.

A very different bivariate spline is the *thin-plate spline*. This is a function of the form

$$f(x) = \sum_{j=1}^{n-3} \Psi\left(x - c_j\right) a_j + x(1) a_{n-2} + x(2) a_{n-1} + a_n$$

with $\psi(x) = |x|^2 \log |x|^2$ the thin-plate spline basis function, and $|x|$ denoting the Euclidean length of the vector x. Here, for convenience, denote the independent variable by x, but x is now a *vector* whose two components, $x(1)$ and $x(2)$, play the role of the two independent variables earlier denoted x and y. Correspondingly, the sites c_j are points in $\Re^2$.

Thin-plate splines arise as bivariate *smoothing splines*, meaning a thin-plate spline minimizes

$$p \sum_{i=1}^{n-3} \left| y_i - f{c_i}^2 \right| + (1-p) \int \left(\left| D_1 D_1 f \right|^2 + 2 \left| D_1 D_2 f \right|^2 + \left| D_2 D_2 f \right|^2 \right)$$

over all sufficiently smooth functions f. Here, the y_i are data values given at the data sites c_i, p is the smoothing parameter, and $D_j f$ denotes the partial derivative of f with respect to $x(j)$. The integral is taken over the entire $\Re^2$.

The upper summation limit, n–3, reflects the fact that 3 degrees of freedom of the thin-plate spline are associated with its polynomial part.

Thin-plate splines are functions in stform, meaning that, up to certain polynomial terms, they are a weighted sum of arbitrary or scattered translates $\Psi(\cdot - c)$ of one fixed function, Ψ. This so-called basis function for the thin-plate spline is special in that it is radially symmetric, meaning that $\Psi(x)$ only depends on the Euclidean length, $|x|$, of x. For that reason, thin-plate splines are also known as RBFs or radial basis functions. See Chapter 8, "The stform" for more information.

Rational Splines

A *rational spline* is any function of the form $r(x) = s(x)/w(x)$, with both s and w splines and, in particular, w a scalar-valued spline, while s often is vector-valued.

Rational splines are attractive because it is possible to describe various basic geometric shapes, like conic sections, exactly as the range of a rational spline. For example, a circle can so be described by a quadratic rational spline with just two pieces.

In this toolbox, there is the additional requirement that both s and w be of the same form and even of the same order, and with the same knot or break sequence. This makes it possible to store the rational spline r as the ordinary spline R whose value at x is the vector $[s(x);w(x)]$. Depending on whether the two splines are in B-form or ppform, such a representation is called here the rBform or the rpform of such a rational spline.

It is easy to obtain r from R. For example, if `v` is the value of R at x, then `v(1:end-1)/v(end)` is the value of r at x. There are corresponding ways to express derivatives of r in terms of derivatives of R.

4

The ppform

Introduction

A univariate *piecewise polynomial f* is specified by its *break sequence* `breaks` and the *coefficient array* `coefs` of the local power form (see Equation 4-1 below) of its polynomial pieces; see Chapter 6, "Tensor Product Splines" for a discussion of multivariate piecewise-polynomials. The coefficients may be (column-)vectors, matrices, even ND-arrays. For simplicity, the present discussion deals only with the case when the coefficients are scalars.

The break sequence is assumed to be strictly increasing,

```
breaks(1)
< breaks(2) < ... < breaks(l+1)
```

with `l` the number of polynomial pieces that make up *f*.

While these polynomials may be of varying degrees, they are all recorded as polynomials of the same *order* `k`, i.e., the coefficient array `coefs` is of size `[l,k]`, with `coefs(j,:)` containing the `k` coefficients in the local power form for the `j`th polynomial piece, from the highest to the lowest power; see Equation 4-1 below.

ppform

The items `breaks`, `coefs`, `l`, and `k`, make up the *ppform* of *f*, along with the dimension `d` of its coefficients; usually `d` equals 1. The *basic interval* of this form is the interval [`breaks(1)` .. `breaks(l+1)`]. It is the default interval over which a function in ppform is plotted by the plot command `fnplt`.

In these terms, the precise description of the piecewise-polynomial *f* is

```
f(t) = polyval(coefs(j,:), t - breaks(j))
```
(4-1)

for breaks(j)≤t<breaks(j+1).

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

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

This defines *f(t)* only for *t* in the half-open interval `[breaks(1)..breaks(l+1))`. For any other *t*, *f(t)* is defined by

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

i.e., by extending the first, respectively last, polynomial piece. In this way, a function in ppform has possible jumps, in its value and/or its derivatives, only across the interior breaks, `breaks(2:l)`. The end breaks, `breaks([1,l+1])`, mainly serve to define the basic interval of the ppform.

Construction

A piecewise-polynomial is usually constructed by some command, through a process of interpolation or approximation, or conversion from some other form e.g., from the B-form, and is output as a variable. But it is also possible to make one up from scratch, using the statement

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

For example, if you enter `pp=ppmak(-5:-1,-22:-11)`, or, more explicitly,

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

you specify the uniform break sequence `-5:-1` and the coefficient sequence `-22:-11`. Because this break sequence has 5 entries, hence 4 break intervals, while the coefficient sequence has 12 entries, you have, in effect, specified a piecewise-polynomial of order 3 (= 12/4). The command

```
fnbrk(pp)
```

prints out all the constituent parts of this piecewise-polynomial, as follows:

```
breaks(1:l+1)
    -5 -4 -3 -2 -1
coefficients(d*l,k)
     -22 -21 -20
     -19 -18 -17
     -16 -15 -14
     -13 -12 -11
  pieces number l
    4
order k
  3
dimension d of target
    1
```

Further, `fnbrk` can be used to supply each of these parts separately. But the point of 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.

Available Commands

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
`fnmin(pp,[a,b])`	Finds the minimum value in given interval
`fnzeros(pp,[a,b])`	Finds the zeros in the given interval
`pj = fnbrk(pp,j)`	Pulls out the `j`th polynomial piece
`pc = fnbrk(pp,[a b])`	Restricts/extends to the interval [a..b]
`po = fnxtr(pp,order)`	Extends outside its basic interval by polynomial of specified order
`fnplt(pp,[a,b])`	Plots on given interval
`sp = fn2fm(pp,'B-')`	Converts to B-form
`pr = fnrfn(pp,morebreaks)`	Inserts additional breaks

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

To illustrate the use of some of these commands, execute the following commands to create and plot the particular piecewise-polynomial described in the "Construction" on page 4-4 section.

1. Create the piecewise-polynomial with break sequence `-5:-1` and coefficient sequence `-22:-11`:

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

2. Create the basic plot:

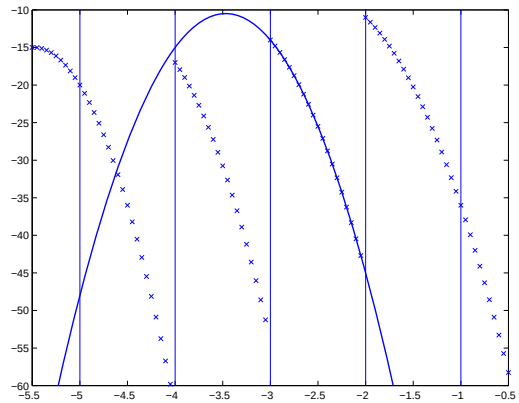
```
x = linspace(-5.5,-.5,101);
plot(x, fnval(pp,x),'x')
```

3 Add the break lines to the plot:

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

4 Superimpose the plot of the polynomial that supplies the third polynomial piece:

```
plot(x,fnval(fnbrk(pp,3),x),'linew',1.3)
set(gca,'ylim',[-60 -10]), hold off
```



A Piecewise-Polynomial Function, Its Breaks, and the Polynomial Giving Its Third Piece

The figure above is the final picture. It shows the piecewise-polynomial as a sequence of points and, solidly on top of it, the polynomial from which its third polynomial piece is taken. It is quite noticeable that the value of a piecewise-polynomial at a break is its limit from the *right*, and that the value of the piecewise-polynomial outside its basic interval is obtained by extending its leftmost, respectively its rightmost, polynomial piece.

While the ppform of a piecewise-polynomial is efficient for evaluation, the *construction* of a piecewise-polynomial from some data is usually more efficiently handled by determining first its *B-form*, i.e., its representation as a linear combination of B-splines.

5

The B-form

Introduction

A univariate spline *f* is specified by its nondecreasing knot sequence `t` and by its B-spline coefficient sequence `a`. See Chapter 6, "Tensor Product Splines" for a discussion of multivariate splines. The coefficients may be (column-)vectors, matrices, even ND-arrays. When the coefficients are 2-vectors or 3-vectors, *f* is a curve in R^2 or R^3 and the coefficients are called the *control points* for the curve.

Roughly speaking, such a spline is piecewise-polynomial of a certain order and with breaks `t`(*i*). But knots are different from breaks in that they may be repeated, i.e., `t` need not be *strictly* increasing. The resulting knot *multiplicities* govern the smoothness of the spline across the knots, as detailed below.

With `[d,n] = size(a)`, and `n+k = length(t)`, the spline is of *order* `k`. This means that its polynomial pieces have degree `< k`. For example, a *cubic* spline is a spline of *order 4* because it takes four coefficients to specify a cubic polynomial.

B-form

These four items, t, a, n, and k, make up the B-form of the spline f.

This means, explicitly, that

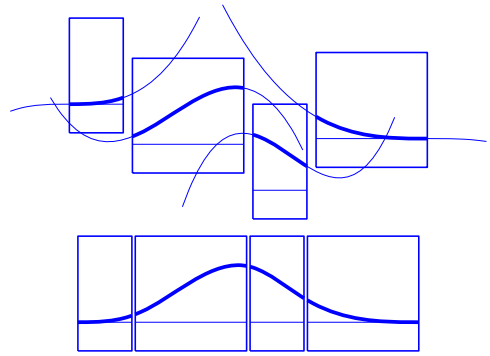
$$f = \sum_{i=1}^{n} B_{i,k} a(:,i)$$

with $B_{i,k}=B(\cdot|t(i:i+k))$ the ith B-spline of order `k` for the given knot sequence `t`, i.e., the B-spline with knots $t(i),...,t(i+k)$. The basic interval of this B-form is the interval $[t(1)..t(n+k)]$. It is the default interval over which a spline in B-form is plotted by the command `fnplt`. Note that a spline in B-form is zero outside its basic interval while, after conversion to ppform via `fn2fm`, this is usually not the case because, outside its basic interval, a piecewise-polynomial is defined by extension of its first or last polynomial piece. In particular, a function in B-form may have jumps in value and/or one of its derivative not only across its interior knots, i.e., across $t(i)$ with $t(1)<t(i)<t(n+k)$, but also across its end knots, $t(1)$ and $t(n+k)$.

B-Splines

The building blocks for the B-form of a spline are the B-splines. A B-Spline of Order 4, and the Four Cubic Polynomials from Which It Is Made on page 5-4 shows a picture of such a B-spline, the one with the knot sequence `[0 1.5 2.3 4 5]`, hence of order 4, together with the polynomials whose pieces make up the B-spline. The information for that picture could be generated by the command

```
bspline([0 1.5 2.3 4 5])
```



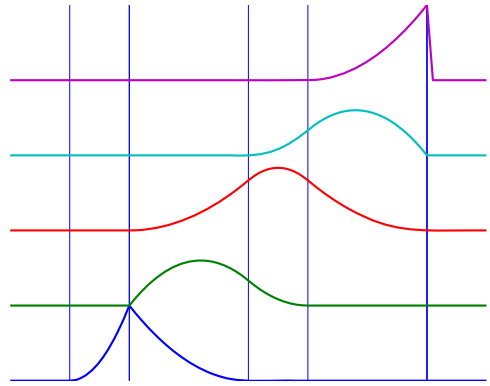
A B-Spline of Order 4, and the Four Cubic Polynomials from Which It Is Made

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

Knot Multiplicity

The rule is

knot multiplicity + condition multiplicity = order



All Third-Order B-Splines for a Certain Knot Sequence with Various Knot Multiplicities

For example, for a B-spline of order 3, a simple knot would mean two smoothness conditions, i.e., continuity of function and first derivative, while a double knot would only leave one smoothness condition, i.e., just continuity, and a triple knot would leave no smoothness condition, i.e., even the function would be discontinuous.

All Third-Order B-Splines for a Certain Knot Sequence with Various Knot Multiplicities on page 5-5 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 “Intro to B-form” available on the Demos tag in the MATLAB Help browser. You can also use the GUI `bspligui` to study the dependence of a B-spline on its knots experimentally.

Choice of Knots

The rule "knot multiplicity + condition multiplicity = order" has the following consequence for the process of choosing a knot sequence for the B-form of a spline approximant. Suppose the spline s is to be of order k, with basic interval $[a..b]$, and with interior breaks $\xi_2 < \cdots < \xi_l$. Suppose, further, that, at ξ_i, the spline is to satisfy μ_i smoothness conditions, i.e.,

$$jump_{\xi_i} D^j s := D^j s(\xi_{i+}) - D^j s(\xi_{i-}) = 0, \quad 0 \le j < \mu_i, \quad i = 2,...,l$$

Then, the appropriate knot sequence t should contain the break ξ_i exactly $k - \mu_i$ times, i=2,...,l. In addition, it should contain the two endpoints, a and b, of the basic interval exactly k times. This last requirement can be relaxed, but has become standard. With this choice, there is exactly one way to write each spline s with the properties described as a weighted sum of the B-splines of order k with knots a segment of the knot sequence t. This is the reason for the *B* in *B-spline*: B-splines are, in Schoenberg's terminology, *basic* splines.

For example, if you want to generate the B-form of a cubic spline on the interval [1 .. 3], with interior breaks 1.5, 1.8, 2.6, and with two continuous derivatives, then the following would be the appropriate knot sequence:

```
t = [1, 1, 1, 1, 1.5, 1.8, 2.6, 3, 3, 3, 3];
```

This is supplied by `augknt([1, 1.5, 1.8, 2.6, 3], 4)`. If you wanted, instead, to allow for a corner at 1.8, i.e., a possible jump in the first derivative there, you would triple the knot 1.8, i.e., use

```
t = [1, 1, 1, 1, 1.5, 1.8, 1.8, 1.8, 2.6, 3, 3, 3, 3];
```

and this is provided by the statement

```
t = augknt([1, 1.5, 1.8, 2.6, 3], 4, [1, 3, 1] );
```

Splines

The shorthand

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

is one of several ways to indicate that f is a spline of order `k` with knot sequence `t`, i.e., *a linear combination of the B-splines* of order `k` for the knot sequence `t`.

A word of caution: The term *B-spline* has been expropriated by the Computer-Aided Geometric Design (CAGD) community to mean what is called here a *spline in B-form*, with the unhappy result that, in any discussion between mathematicians/approximation theorists and people in CAGD, one now always has to check in what sense the term is being used.

Construction

Usually, a spline is constructed from some information, like function values and/or derivative values, or as the approximate solution of some ordinary differential equation. But it is also possible to make up a spline from scratch, by providing its knot sequence and its coefficient sequence to the command `spmak`.

For example, if you enter

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

you supply the uniform knot sequence `1:10` and the coefficient sequence `3:8`. Because there are 10 knots and 6 coefficients, the order must be 4(= 10 – 6), i.e., you get a cubic spline. The command

```
fnbrk(sp)
```

prints out the constituent parts of the B-form of this cubic spline, as follows:

```
knots(1:n+k)
   1 2 3 4 5 6 7 8 9 10
coefficients(d,n)
   3 4 5 6 7 8
number n of coefficients
   6
order k
   4
dimension d of target
   1
```

Further, `fnbrk` can be used to supply each of these parts separately.

But the point of the Spline Toolbox product 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`.

Example: A Spline Curve

As another simple example,

```
points = .95*[0 -1 0 1;1 0 -1 0];
sp = spmak(-4:8,[points points]);
```

provides a planar, quartic, spline curve whose middle part is a pretty good approximation to a circle, as the plot on the next page shows. It is generated by a subsequent

```
plot(points(1,:),points(2,:),'x'), hold on
fnplt(sp,[0,4]), axis equal square, hold off
```

Insertion of additional control points $(\pm 0.95, \pm 0.95)/\sqrt{1.9}$ would make a visually perfect circle.

Here are more details. The spline curve generated has the form $\Sigma_{j=1}^{8} B_{j,5}\alpha(:,j)$, with `-4:8` the uniform knot sequence, and with its control points $\alpha(:,j)$ the sequence (0,α),(–α,0),(0,–α),(α,0),(0,α),(–α,0),(0,–α),(α,0) with α=0.95. Only the curve part between the parameter values 0 and 4 is actually plotted.

To get a feeling for how close to circular this part of the curve actually is, compute its unsigned curvature. The curvature κ(*t*) at the curve point γ(*t*) = (x(*t*), y(*t*)) of a space curve γ can be computed from the formula

$$\kappa = \frac{|x'y'' - y'x''|}{(x'^2 + y'^2)^{3/2}}$$

in which x', x″, y', and y" are the first and second derivatives of the curve with respect to the parameter used (*t*). Treat the planar curve as a space curve in the (*x*,*y*)-plane, hence obtain the maximum and minimum of its curvature at 21 points as follows:

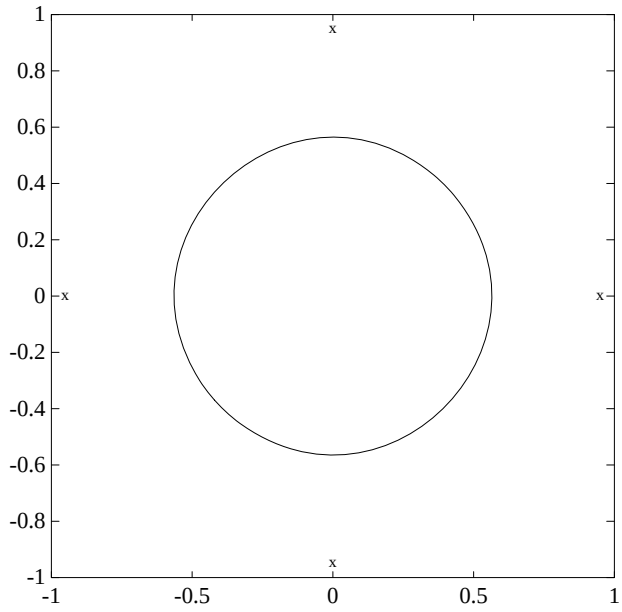
```
t = linspace(0,4,21);zt = zeros(size(t));
dsp = fnder(sp); dspt = fnval(dsp,t); ddspt = fnval(fnder(dsp),t);
kappa = abs(dspt(1,:).*ddspt(2,:)-dspt(2,:).*ddspt(1,:))./...
   (sum(dspt.^2)).^(3/2);
[min(kappa),max(kappa)]
```

```
ans =
     1.6747    1.8611
```

So, while the curvature is not quite constant, it is close to 1/radius of the circle, as you see from the next calculation:

```
1/norm(fnval(sp,0))

ans =
     1.7864
```



Spline Approximation to a Circle; Control Points Are Marked x

Available Commands

The following commands are available for spline work. There is `spmak` and `fnbrk` to make up a spline and take it apart again. Use `fn2fm` to convert from B-form to ppform. You can evaluate, differentiate, integrate, minimize, find zeros of, plot, refine, or selectively extrapolate a spline with the aid of `fnval`, `fnder`, `fndir`, `fnint`, `fnmin`, `fnzeros`, `fnplt`, `fnrfn`, and `fnxtr`.

There are five commands for generating knot sequences:

- `augknt` for providing boundary knots and also controlling the multiplicity of interior knots
- `brk2knt` for supplying a knot sequence with specified multiplicities
- `aptknt` for providing a knot sequence for a spline space of given order that is suitable for interpolation at given data sites
- `optknt` for providing an *optimal* knot sequence for interpolation at given sites
- `newknt` for a knot sequence perhaps more suitable for the function to be approximated

In addition, there is:

- `aveknt` to supply certain knot averages (the Greville sites) as recommended sites for interpolation
- `chbpnt` to supply such sites
- `knt2brk` and `knt2mlt` for extracting the breaks and/or their multiplicities from a given knot sequence

To display a spline *curve* with given two-dimensional coefficient sequence and a uniform knot sequence, use `spcrv`.

You can also write your own spline construction commands, in which case you will need to know the following. The construction of a spline satisfying some interpolation or approximation conditions usually requires a *collocation matrix*, i.e., the matrix that, in each row, contains the sequence of numbers $D^r B_{j,k}(\tau)$, i.e., the 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 an almost-block-diagonal coefficient matrix. If you are interested in seeing how `spcol` and `slvblk` are used in this toolbox, have a look at the commands `spapi`, `spap2`, and `spaps`.

In addition, there are routines for constructing *cubic* splines. `csapi` and `csape` provide the cubic spline interpolant at knots to given data, using the not-a-knot and various other end conditions, respectively. A parametric cubic spline curve through given points is provided by `cscvn`. The cubic *smoothing* spline is constructed in `csaps`.

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

6

Tensor Product Splines

Introduction

The toolbox provides (polynomial) spline functions in any number of variables, as tensor products of univariate splines. These multivariate splines come in both standard forms, the B-form and the ppform, and their construction and use parallels entirely that of the univariate splines discussed in previous sections, Chapter 4, “The ppform” and Chapter 5, “The B-form” The same commands are used for their construction and use.

For simplicity, the following discussion deals just with bivariate splines.

B-form

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 $p\ (x,y)\colon = f\ (x)g(y)$ 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 $a_{ji}:i=1,...,m;j=1,...n)$ a corresponding coefficient array, you obtain a bivariate spline as

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

The B-form of this spline comprises the cell array $\{s,t\}$ of its knot sequences, the coefficient array a, the numbers vector $[m,n]$, and the orders vector $[h,k]$. The command

```
sp = spmak({s,t},a);
```

constructs this form. Further, `fnplt`, `fnval`, `fnder`, `fndir`, `fnrfn`, and `fn2fm` can be used to plot, evaluate, differentiate and integrate, refine, and convert this form.

Construction and Use

You are most likely to construct such a form by looking for an interpolant or approximant to gridded data. For example, if you know the values z(*i*,*j*)=g(*x*(*i*),*y*(*j*)),*i*=1:*m*, *j*=1:*n*, of some function *g* at all the points in a rectangular grid, then, assuming that the strictly increasing sequence `x` satisfies the Schoenberg-Whitney conditions with respect to the above knot sequence *s*, and that the strictly increasing sequence `y` satisfies the Schoenberg-Whitney conditions with respect to the above knot sequence *t*, the command

```
sp=spapi({s,t},[h,k],{x,y},z);
```

constructs the unique bivariate spline of the above form that matches the given values. The command `fnplt(sp)` gives you a quick plot of this interpolant. The command `pp = fn2fm(sp,'pp')` gives you the ppform of this spline, which is probably what you want when you want to evaluate the spline at a fine grid `((xx(i),yy(j))` for `i=1:M, j=1:N)`, by the command:

```
values = fnval(pp,{xx,yy});
```

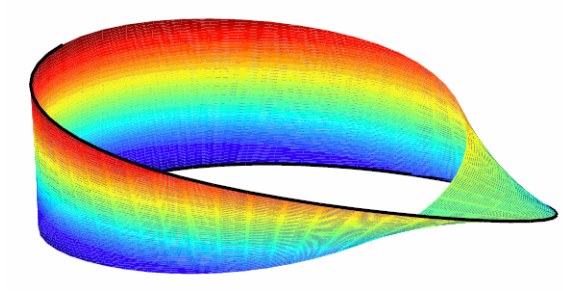
ppform

The ppform of such a bivariate spline comprises, analogously, a cell array of break sequences, a multidimensional coefficient array, a vector of number pieces, and a vector of polynomial orders. Fortunately, the toolbox is set up in such a way that there is usually no reason for you to concern yourself with these details of either form. You use interpolation, approximation, or smoothing to construct splines, and then use the `fn`... commands to make use of them.

Example: The Mobius Band

Here is an example of a surface constructed as a 3-D-valued bivariate spline. The surface is the famous Möbius band, obtainable by taking a longish strip of paper and gluing its narrow ends together, but with a twist. The figure is obtained by the following commands:

```
x = 0:1; y = 0:4; h = 1/4; o2 = 1/sqrt(2); s = 2; ss = 4;
v(3,:,:) = h*[0, -1, -o2, 0, o2, 1, 0;0, 1, o2, 0, -o2, -1, 0];
v(2,:,:) = [ss, 0, s-h*o2, 0, -s-h*o2, 0, ss;...
            ss, 0, s+h*o2, 0,-s+h*o2, 0, ss];
v(1,:,:) = s*[0, 1, 0, -1+h, 0, 1, 0; 0, 1, 0, -1-h, 0, 1, 0];
cs = csape({x,y},v,{'variational','clamped'});
fnplt(cs), axis([-2 2 -2.5 2.5 -.5 .5]), shading interp
axis off, hold on
values = squeeze(fnval(cs,{1,linspace(y(1),y(end),51)}));
plot3(values(1,:), values(2,:), values(3,:),'k','linew',2)
view(-149,28), hold off
```



A Möbius Band Made by Vector-Valued Bivariate Spline Interpolation

7

NURBS and Other Rational Splines

Introduction

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.

Example: Circle

For example,

```
circle = rsmak('circle');
```

provides a rational spline whose values on its basic interval trace out the unit circle, i.e., the circle of radius 1 with center at the origin, as the command

```
fnplt(circle), axis square
```

readily shows; the resulting output is the circle in the figure A Circle and an Ellipse, Both Given by a Rational Spline on page 7-4.

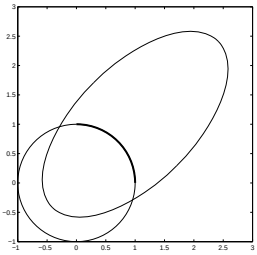
It is easy to manipulate this circle to obtain related shapes. For example, the next commands stretch the circle into an ellipse, rotate the ellipse 45 degrees, and translate it by (1,1), and then plot it on top of the circle.

```
ellipse = fncmb(circle,[2 0;0 1]);
s45 = 1/sqrt(2);
rtellipse = fncmb(fncmb(ellipse, [s45 -s45;s45 s45]), [1;1] );
hold on, fnplt(rtellipse), hold off
```

As a further example, the "circle" just constructed is put together from four pieces. To highlight the first such piece, use the following commands:

```
quarter = fnbrk(fn2fm(circle,'rp'),1);
hold on, fnplt(quarter,3), hold off
```

In the first command, `fn2fm` is used to change forms, from one based on the B-form to one based on the ppform, and then `fnbrk` is used to extract the first piece, and this piece is then plotted on top of the circle in A Circle and an Ellipse, Both Given by a Rational Spline on page 7-4, with linewidth `3` to make it stand out.



A Circle and an Ellipse, Both Given by a Rational Spline

Example: Sphere

As a surface example, the command `rsmak('southcap')` provides a 3-vector valued rational bicubic polynomial whose values on the unit square [-1 .. 1]^2 fill out a piece of the unit sphere. Adjoin to it five suitable rotates of it and you get the unit sphere exactly. For illustration, the following commands generate two-thirds of that sphere, as shown in Part of a Sphere Formed by Four Rotates of a Quartic Rational on page 7-5.

```
southcap = rsmak('southcap'); fnplt(southcap)
xpcap = fncmb(southcap,[0 0 -1;0 1 0;1 0 0]);
ypcap = fncmb(xpcap,[0 -1 0; 1 0 0; 0 0 1]);
northcap = fncmb(southcap,-1);
hold on, fnplt(xpcap), fnplt(ypcap), fnplt(northcap)
axis equal, shading interp, view(-115,10), axis off, hold off
```

Part of a Sphere Formed by Four Rotates of a Quartic Rational

rsform: rpform, rBform

Offhand, the two splines, s and w, in the rational spline $r(x)=s(x)/w(x)$ need not be related to one another. They could even be of different forms. But, in the context of this toolbox, it is convenient to restrict them to be of the same form, and even of the same order and with the same breaks or knots. For, under that assumption, you can represent such a rational spline by the (vector-valued) spline function

$$R(x) = \left[s(x); w(x) \right]$$

whose values are vectors with one more entry than the values of the rational spline r, and call this the *rsform* of the rational spline, or, more precisely, the *rpform* or *rBform*, depending on whether s and w are in ppform or in B-form. Internally, the only thing that distinguishes these rational forms from their corresponding ordinary spline forms, rpform and B-form, is their form part, i.e., the string obtained via `fnbrk(r,'form')`. This is enough to alert the `fn...` commands to act appropriately on a function in one of the rsforms.

For example, as is done in `fnval`, it is very easy to obtain $r(x)$ from $R(x)$. If `v` is the value of R at x, then `v(1:end-1)/v(end)` is the value of r at x. If, in addition, `dv` is $DR(x)$, then `(dv(1:end-1)-dv(end)*v(1:end-1))/v(end)` is $Dr(x)$. More generally, by Leibniz's formula,

$$D^j s = D^j (wr) = \sum_{i=0}^{j} \binom{j}{i} D^i w D^{j-i} r$$

Therefore,

$$D^j r = \left(D^j s - \sum_{i=1}^{j} \binom{j}{i} D^i w D^{j-i} r \right) / w$$

This shows that you can compute the derivatives of r inductively, using the derivatives of s and w (i.e., the derivatives of R) along with the derivatives of r of order less than j to compute the jth derivative of r. This inductive scheme is used in `fntlr` to provide the first so many derivatives of a rational spline.

There is a corresponding formula for partial and directional derivatives for multivariate rational splines.

Available Commands

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 product, with the following exceptions. The integral of a rational spline need not be a rational spline, hence there is no way to extend `fnint` to rational splines. The derivative of a rational spline *is* again a rational spline but one of roughly twice the order. For that reason, `fnder` and `fndir` will not touch rational splines. Instead, there is the command `fntlr` for computing the value at a given `x` of all derivatives up to a given order of a given function. If that function is rational, the needed calculation is based on the considerations given in the preceding paragraph.

The command `r = rsmak(shape)` provides rational splines in rBform that describe exactly certain standard geometric shapes , like `'circle'`, `'arc'`, `'cylinder'`, `'sphere'`, `'cone'`, `'torus'`. The command `fncmb(r,trans)` can be used to apply standard transformations to the resulting shape. For example, if `trans` is a column-vector of the right length, the shape would be translated by that vector while, if `trans` is a suitable matrix like a rotation, the shape would be transformed by that matrix.

The command `r = rscvn(p)` constructs the quadratic rBform of a tangent-continuous curve made up of circular arcs and passing through the given sequence, `p`, of points in the plane.

A special rational spline form, called a NURBS, has become a standard tool in CAGD. A NURBS is, by definition, any rational spline for which both s and w are in the same B-form, with each coefficient for s containing explicitly the corresponding coefficient for w as a factor:

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

The normalized coefficients $a(:,i)$ for the numerator spline are more readily used as control points than the unnormalized coefficients $v(i)a(:,i)$ used in the rBform. Nevertheless, this toolbox provides no special NURBS form, but only the more general rational spline, but in both B-form (called `rBform` internally) and in ppform (called `rpform` internally).

The rational spline `circle` used earlier is put together in `rsmak` by code like the following.

```
x = [1 1 0 -1 -1 -1  0  1 1]; y = [0 1 1  1  0 -1 -1 -1 0];
s45 = 1/sqrt(2); w =[1 s45 1 s45 1 s45 1 s45 1];
circle = rsmak(augknt(0:4,3,2), [w.*x;w.*y;w]);
```

Note the appearance of the denominator spline as the last component. Also note how the coefficients of the denominator spline appear here explicitly as factors of the corresponding coefficients of the numerator spline. The normalized coefficient sequence `[x;y]` is very simple; it consists of the vertices and midpoints, in proper order, of the "unit square". The resulting control polygon is tangent to the circle at the places where the four quadratic pieces that form the circle abut.

For a thorough discussion of NURBS, see [G. Farin, *NURBS*, 2nd ed., AKPeters Ltd, 1999] or [Les Piegl and Wayne Tiller, *The NURBS Book*, 2nd ed., Springer-Verlag, 1997].

8

The stform

Introduction

A multivariate function form quite different from the tensor-product construct is the scattered translates form, or stform for short. As the name suggests, it uses arbitrary or scattered translates $\psi(\cdot - c_j)$ of one fixed function ψ, in addition to some polynomial terms. Explicitly, such a form describes a function

$$f(x) = \sum_{j=1}^{n-k} \psi\left(x - c_j\right) a_j + p(x)$$

in terms of the *basis function* ψ, a sequence (c_j) of sites called *centers* and a corresponding sequence (a_j) of n coefficients, with the final k coefficients, $a_{n-k+1},\ldots,a_n$, involved in the *polynomial part*, p.

When the basis function is radially symmetric, meaning that $\psi(x)$ depends only on the Euclidean length $|x|$ of its argument, x, then ψ is called a *radial basis function*, and, correspondingly, f is then often called an RBF.

At present, the toolbox works with just one kind of stform, namely a bivariate thin-plate spline and its first partial derivatives. For the thin-plate spline, the basis function is $\psi(x) = \varphi(|x|^2)$, with $\varphi(t) = t\log t$, i.e., a radial basis function. Its polynomial part is a linear polynomial, i.e., $p(x)=x(1)a_{n-2}+x(2)a_{n-1}+a_n$. The first partial derivative with respect to its first argument uses, correspondingly, the basis function $\psi(x)=\varphi(|x|^2)$, with $\varphi(t) = (D_1 t)\cdot(\log t+1)$ and $D_1 t = D_1 t(x) = 2x(1)$, and $p(x) = a_n$.

Properties of the stform

A function in stform can be put together from its center sequence `centers` and its coefficient sequence `coefs` by the command

```
f = stmak(centers, coefs, type);
```

with the string `type` one of `'tp00'`, `'tp10'`, `'tp01'`, to indicate, respectively, a thin-plate spline, a first partial of a thin-plate spline with respect to the first argument, and a first partial of a thin-plate spline with respect to the second argument. There is one other choice, `'tp'`; it denotes a thin-plate spline without any polynomial part and is likely to be used only during the construction of a thin-plate spline, as in `tpaps`.

A function *f* in stform depends linearly on its coefficients, meaning that

$$f(x) = \sum_{j=1}^{n} \psi_j(x) a_j$$

with ψ_j either a translate of the basis function Ψ or else some polynomial. Suppose you wanted to determine these coefficients a_j so that the function *f* matches prescribed values at prescribed sites x_i. Then you would need the collocation matrix $(\psi_j(x_i))$. You can obtain this matrix by the command `stcol(centers,x,type)`. In fact, because the stform has a_j as the *j*th *column*, `coefs(:,j)`, of its coefficient array, it is worth noting that `stcol` can also supply the *transpose* of the collocation matrix. Thus, the command

```
values = coefs*stcol(centers,x,type,'tr');
```

would provide the values at the entries of `x` of the st function specified by `centers` and `type`.

The stform is attractive because, in contrast to piecewise polynomial forms, its complexity is the same in any number of variables. It is quite simple, yet, because of the complete freedom in the choice of centers, very flexible and adaptable.

On the negative side, the most attractive choices for a radial basis function share with the thin-plate spline that the evaluation at any site involves

all coefficients. For example, plotting a scalar-valued thin-plate spline via `fnplt` involves evaluation at a 51-by-51 grid of sites, a nontrivial task when there are 1000 coefficients or more. The situation is worse when you want to determine these 1000 coefficients so as to obtain the stform of a function that matches function values at 1000 data sites, as this calls for solving a full linear system of order 1000, a task requiring O(10^9) flops if done by a direct method. Just the construction of the collocation matrix for this linear system (by `stcol`) takes O(10^6) flops.

The command `tpaps`, which constructs thin-plate spline interpolants and approximants, uses iterative methods when there are more than 728 data points, but convergence of such iteration may be slow.

Available Commands

After you have constructed an approximating or interpolating thin-plate spline `st` with the aid of `tpaps` (or directly via `stmak`), you can use the following commands:

- `fnbrk` to obtain its parts or change its basic interval,
- `fnval` to evaluate it
- `fnplt` to plot it
- `fnder` to construct its two first partial derivatives, but no higher order derivatives as they become infinite at the centers.

  This is just one indication that the stform is quite different in nature from the other forms in this toolbox, hence other `fn...` commands by and large don't work with stforms. For example, it makes no sense to use `fnjmp`, and `fnmin` or `fnzeros` only work for univariate functions. It also makes no sense to use `fnint` on a function in stform because such functions cannot be integrated in closed form.
- The command `Ast = fncmb(st,A)` can be used on `st`, provided `A` is something that can be applied to the values of the function described by `st`. For example, `A` might be `'sin'`, in which case `Ast` is the stform of the function whose coefficients are the sine of the coefficients of `st`. In effect, `Ast` describes the function obtained by composing `A` with `st`. But, because of the singularities in the higher-order derivatives of a thin-plate spline, there seems little point to make `fndir` or `fntlr` applicable to such a `st`.

9

Advanced Examples

Least-Squares Approximation by "Natural" Cubic Splines

The construction of a least-squares approximant usually requires that one have in hand a basis for the space from which the data are to be approximated. As the example of the space of "natural" cubic splines illustrates, the explicit construction of a basis is not always straightforward.

This section makes clear that an explicit basis is not actually needed; it is sufficient to have available some means of interpolating in some fashion from the space of approximants. For this, the fact that the Spline Toolbox product supports work with vector-valued functions is essential.

This section discusses these aspects of least-squares approximation by "natural" cubic splines.

Problem

You want to construct the least-squares approximation to given data (`x`,`y`) from the space *S* of "natural" cubic splines with given breaks `b(1)` < ...< `b(l+1)`.

General Resolution

If you know a basis, (f1,f2,...,fm), for the linear space *S* of all "natural" cubic splines with break sequence `b`, then you have learned to find the least-squares approximation in the form `c(1)`f1+ `c(2)`f2+ ... + `c(m)`fm, with the vector `c` the least-squares solution to the linear system `A*c = y`, whose coefficient matrix is given by

```
A(i,j) = fj(x(i)),   i=1:length(x),  j=1:m .
```

In other words, `c = A\y`.

Need for a Basis Map

The general solution seems to require that you know a basis. However, in order to construct the coefficient sequence `c`, you only need to know the matrix `A`. For this, it is sufficient to have at hand a *basis map*, namely an M-file, `F` say, so that `F(c)` returns the spline given by the particular weighted sum `c(1)`f1+`c(2)`f2+... +`c(m)`fm. For, with that, you can obtain, for `j=1:m`, the `j`-th column of `A` as `fnval(F(ej),x)`, with `ej` the `j`-th column of `eye(m)`, the identity matrix of order `m`.

Better yet, the Spline Toolbox product can handle *vector-valued* functions, so you should be able to construct the basis map `F` to handle vector-valued coefficients `c(i)` as well. However, by agreement, in this toolbox, a vector-valued coefficient is a *column* vector, hence the sequence c is necessarily a row vector of column vectors, i.e., a *matrix*. With that, `F(eye(m))` is the vector-valued spline whose `i`-th component is the basis element fi, `i=1:m`. Hence, assuming the vector `x` of data sites to be a row vector, `fnval(F(eye(m)),x)` is the matrix whose `(i,j)`-entry is the value of `fi` at `x(j)`, i.e., the *transpose* of the matrix `A` you are seeking. On the other hand, as just pointed out, your basis map `F` expects the coefficient sequence `c` to be a row vector, i.e., the transpose of the vector `A\y`. Hence, assuming, correspondingly, the vector `y` of data values to be a row vector, you can obtain the least-squares approximation from S to data (`x`,`y`) as

```
F(y/fnval(F(eye(m)),x))
```

To be sure, if you wanted to be prepared for `x` and `y` to be arbitrary vectors (of the same length), you would use instead

```
F(y(:).'/fnval(F(eye(m)),x(:).'))
```

A Basis Map for "Natural" Cubic Splines

What exactly is required of a basis map `F` for the linear space S of "natural" cubic splines with break sequence `b(1) < ...  < b(l+1)`? Assuming the dimension of this linear space is `m`, the map `F` should set up a linear one-to-one

correspondence between m-vectors and elements of *S*. But that is exactly what `csape(b, . ,'var')` does.

To be explicit, consider the following M-file `F`:

```
function s = F(c)
s = csape(b,c,'var');
```

For given vector `c` (of the same length as b), it provides the *unique* "natural" cubic spline with break sequence b that takes the value `c(i)` at `b(i)`, `i=1:l+1`. The uniqueness is key. It ensures that the correspondence between the vector `c` and the resulting spline `F(c)` is one-to-one. In particular, `m` equals `length(b)`. More than that, because the value *f(t)* of a function *f* at a point *t* depends linearly on *f*, this uniqueness ensures that `F(c)` depends linearly on `c` (because `c` equals `fnval(F(c),b)` and the inverse of an invertible linear map is again a linear map).

The One-line Solution

Putting it all together, you arrive at the following code

```
csape(b,y(:).'/fnval(csape(b,eye(length(b)),'var'),x(:).'),...
'var')
```

for the least-squares approximation by "natural" cubic splines with break sequence `b`.

The Need for Proper Extrapolation

Let's try it on some data, the census data, say, which is provided in MATLAB by the command

```
load census
```

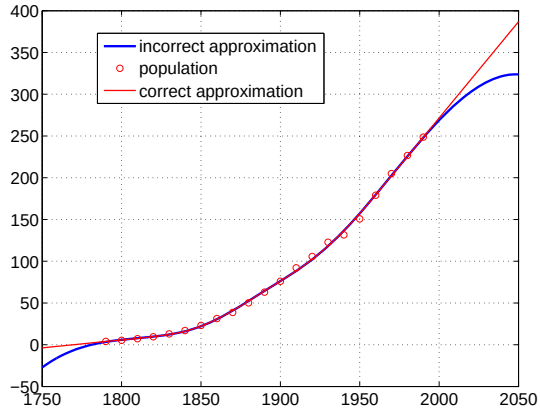
and which supplies the years, `1790:10:1990`, as `cdate` and the values as `pop`. Use the break sequence `1810:40:1970`.

```
b = 1810:40:1970; s = csape(b, ...
pop(:)'/fnval(csape(b,eye(length(b)),'var'),cdate(:)'),'var');
fnplt(s, [1750,2050],2.2), hold on, plot(cdate,pop,'or')
set(gca,'Fontsize',16), hold off
```

Have a look at Least-Squares Approximation by "Natural" Cubic Splines With Three Interior Breaks on page 9-6 which shows, in thick blue, the resulting approximation, along with the given data.

This looks like a good approximation, -- except that it doesn't look like a "natural" cubic spline. A "natural" cubic spline, to recall, must be linear to the left of its first break and to the right of its last break, and this approximation satisfies neither condition. This is due to the following facts.

The "natural" cubic spline interpolant to given data is provided by `csape` in ppform, with the interval spanned by the data sites its basic interval. On the other hand, evaluation of a ppform outside its basic interval is done, in MATLAB `ppval` or Spline Toolbox `fnval`, by using the relevant polynomial end piece of the ppform, i.e., by full-order extrapolation. In case of a "natural" cubic spline, you want instead second-order extrapolation. This means that you want, to the left of the first break, the straight line that agrees with the cubic spline in value and slope at the first break. Such an extrapolation is provided by `fnxtr`. Because the "natural" cubic spline has zero second derivative at its first break, such an extrapolation is even third-order, i.e., it satisfies three matching conditions. In the same way, beyond the last break of the cubic spline, you want the straight line that agrees with the spline in value and slope at the last break, and this, too, is supplied by `fnxtr`.



Least-Squares Approximation by "Natural" Cubic Splines With Three Interior Breaks

The Correct One-Line Solution

The following one-line code provides the correct least-squares approximation to data (x,y) by "natural" cubic splines with break sequence b:

```
fnxtr(csape(b,y(:).'/ ...
    fnval(fnxtr(csape(b,eye(length(b)),'var')),x(:).'),'var'))
```

But it is, admittedly, a rather long line.

The following code uses this correct formula and plots, in a thinner, red line, the resulting approximation on top of the earlier plots, as shown in Least-Squares Approximation by "Natural" Cubic Splines With Three Interior Breaks on page 9-6.

```
 ss = fnxtr(csape(b,pop(:)'/ ...
  fnval(fnxtr(csape(b,eye(length(b)),'var')),cdate(:)'),'var'));
hold on, fnplt(ss,[1750,2050],1.2,'r'),grid, hold off
legend('incorrect approximation','population', ...
'correct approximation')
```

Least-Squares Approximation by Cubic Splines

The one-line solution works perfectly if you want to approximate by the space *S* of all cubic splines with the given break sequence `b`. You don't even have to use the Spline Toolbox product for this because you can rely on the MATLAB `spline`. You know that, with `c` a sequence containing two more entries than does `b`, `spline(b,c)` provides the unique cubic spline with break sequence `b` that takes the value `c(i+1)` at `b(i)`, all `i`, and takes the slope `c(1)` at `b(1)`, and the slope `c(end)` at `b(end)`. Hence, `spline(b,.)` is a basis map for `S`.

More than that, you know that `spline(b,c,xi)` provides the value(s) at `xi` of this interpolating spline. Finally, you know that `spline` can handle vector-valued data. Therefore, the following one-line code constructs the least-squares approximation by cubic splines with break sequence `b` to data `(x,y)` :

```
spline(b,y(:)'/spline(b,eye(length(b)),x(:)'))
```

A Nonlinear ODE

This section discusses these aspects of a nonlinear ODE problem:

The example can be run via the demo "Solving a Nonlinear ODE with a Boundary Layer by Collocation".

Problem

Consider the nonlinear singularly perturbed problem:

$$\varepsilon D^2 g(x) + \left(g(x)\right)^2 = 1 \quad on \quad [0..1]$$

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

Approximation Space

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

```
breaks = (0:4)/4;
```

Because cubics are of order 4, you have

```
k = 4;
```

Obtain the corresponding knot sequence as

```
knots = augknt(breaks,k,2);
```

This gives a quadruple knot at both 0 and 1, which is consistent with the fact that you have cubics, i.e., have order 4.

This implies that you have

```
n = length(knots)-k;
n = 10;
```

i.e., 10 degrees of freedom.

Discretization

You collocate at two sites per polynomial piece, i.e., at eight sites altogether. This, together with the two side conditions, gives us 10 conditions, which matches the 10 degrees of freedom.

Choose the two Gaussian sites for each interval. For the *standard* interval [–0.5,0.5] of length 1, these are the two sites

```
gauss = .5773502692*[-1/2; 1/2];
```

From this, you obtain the whole collection of collocation sites by

```
ninterv = length(breaks)-1;
temp = ((breaks(2:ninterv+1)+breaks(1:ninterv))/2);
temp = temp([1 1],:) + gauss*diff(breaks);
colsites = temp(:).';
```

Numerical Problem

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

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

Linearization

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

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

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

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

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

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

with

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

Linear System to Be Solved

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

Use `spcol` to supply the matrix

```
colmat = ...
spcol(knots,k,brk2knt(sites,3));
```

From this, you get the collocation matrix by combining the row triple of `colmat` for x using the weights $w_0(x), w_1(x), w_2(x)$ to get the row for x of the actual matrix. For this, you need a current approximation y. Initially, you get it by interpolating some reasonable initial guess from your piecewise-polynomial

space at the `sites`. Use the parabola x^2–1, which satisfies the end conditions as the initial guess, and pick the matrix from the full matrix `colmat`. Here it is, in several cautious steps:

```
intmat = colmat([2 1+(1:(n-2))*3,1+(n-1)*3],:);
coefs = intmat\[0 colsites.*colsites-1 0].';
y = spmak(knots,coefs.');
```

Plot the initial guess, and turn hold on for subsequent plotting:

```
fnplt(y,'g');
legend('Initial Guess (x^2-1)','location','NW');
axis([-0.01 1.01 -1.01 0.01]);
hold on
```

Iteration

You can now complete the construction and solution of the linear system for the improved approximate solution z from your current guess y. In fact, with the initial guess y available, you now set up an iteration, to be terminated when the change z–y is *small enough*. Choose a relatively mild $\varepsilon = .1$.

```
tolerance = 6.e-9;
epsilon = .1;
while 1
   vtau = fnval(y,colsites);
   weights=[0 1 0;
            [2*vtau.' zeros(n-2,1) repmat(epsilon,n-2,1)];
            1 0 0];
   colloc = zeros(n,n);
   for j=1:n
      colloc(j,:) = weights(j,:)*colmat(3*(j-1)+(1:3),:);
   end
   coefs = colloc\[0 vtau.*vtau+1 0].';
   z = spmak(knots,coefs.');
   fnplt(z,'k');
   maxdif = max(max(abs(z.coefs-y.coefs)));
   fprintf('maxdif = %g\n',maxdif)
   if (maxdif<tolerance), break, end
% now reiterate
   y = z;
```

```
end
legend({'Initial Guess (x^2-1)' 'Iterates'},'location','NW');
```

The resulting printout of the errors is:

```
maxdif = 0.206695
maxdif = 0.01207
maxdif = 3.95151e-005
maxdif = 4.43216e-010
```

If you now decrease ε, you create more of a boundary layer near the right endpoint, and this calls for a nonuniform mesh.

Use `newknt` to construct an appropriate finer mesh from the current approximation:

```
knots = newknt(z, ninterv+1); breaks = knt2brk(knots);
knots = augknt(breaks,4,2);
n = length(knots)-k;
```

From the new break sequence, you generate the new collocation site sequence:

```
ninterv = length(breaks)-1;
temp = ((breaks(2:ninterv+1)+breaks(1:ninterv))/2);
temp = temp([1 1], :) + gauss*diff(breaks);
colpnts = temp(:).';
sites = [0,colpnts,1];
```

Use `spcol` to supply the matrix

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

and use your current approximate solution `z` as the initial guess:

```
intmat = colmat([2 1+(1:(n-2))*3,1+(n-1)*3],:);
y = spmak(knots,[0 fnval(z,colpnts) 0]/intmat.');
```

Thus set up, divide ε by 3 and repeat the earlier calculation, starting with the statements

```
tolerance=1.e-9;
while 1
```

```
vtau=fnval(y,colpnts);
.
.
.
```

Repeated passes through this process generate a sequence of solutions, for ε = 1/10, 1/30, 1/90, 1/270, 1/810. The resulting solutions, ever flatter at 0 and ever steeper at 1, are shown in the demo plot. The plot also shows the final break sequence, as a sequence of vertical bars. To view the plots, run the demo "Solving a Nonlinear ODE with a Boundary Layer by Collocation".

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

Construction of the Chebyshev Spline

This section discusses these aspects of the Chebyshev spline construction:

What Is a Chebyshev Spline?

The *Chebyshev spline* $C=C_t=C_{k,t}$ of order k for the knot sequence $t=(t_i:\ i=1:n+k)$ is the unique element of $S_{k,t}$ of max-norm 1 that maximally oscillates on the interval $[t_k..t_{n+1}]$ and is positive near t_{n+1}. This means that there is a unique strictly increasing n-sequence τ so that the function $C=C_t \epsilon S_{k,t}$ given by $C(\tau_i)=(-1)^{n-1}$, all i, has max-norm 1 on $[t_k..t_{n+1}]$. This implies that $\tau_1=t_k, \tau_n=t_{n+1}$, and that $t_i < \tau_i < t_{k+i}$, for all i. In fact, $t_{i+1} \le \tau_i \le t_{i+k-1}$, all i. This brings up the point that the knot sequence is assumed to make such an inequality possible, i.e., the elements of $S_{k,t}$ are assumed to be continuous.

In short, the Chebyshev spline C looks just like the Chebyshev polynomial. It performs similar functions. For example, its extreme sites τ are particularly good sites to interpolate at from $S_{k,t}$ because the norm of the resulting projector is about as small as can be; see the toolbox command `chbpnt`.

In this example, which you can run via the demo "Construction of the Chebyshev Spline", you try to construct C for a particular knot sequence t.

Choice of Spline Space

You deal with cubic splines, i.e., with order

```
k = 4;
```

and use the break sequence

```
breaks = [0 1 1.1 3 5 5.5 7 7.1 7.2 8];
lp1 = length(breaks);
```

and use simple interior knots, i.e., use the knot sequence

```
t = breaks([ones(1,k) 2:(lp1-1) lp1(:,ones(1,k))]);
n = length(t)-k;
```

Note the quadruple knot at each end. Because `k = 4`, this makes [0..8] = [`breaks(1)`..`breaks(lp1)`] the interval $[t_k..t_{n+1}]$ of interest, with `n = length(t)-k` the dimension of the resulting spline space $S_{k,t}$. The same knot sequence would have been supplied by

```
t=augknt(breaks,k);
```

Initial Guess

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

$$t_i = (t_{i+1} + ... + t_{i+k-1})/(k-1)$$

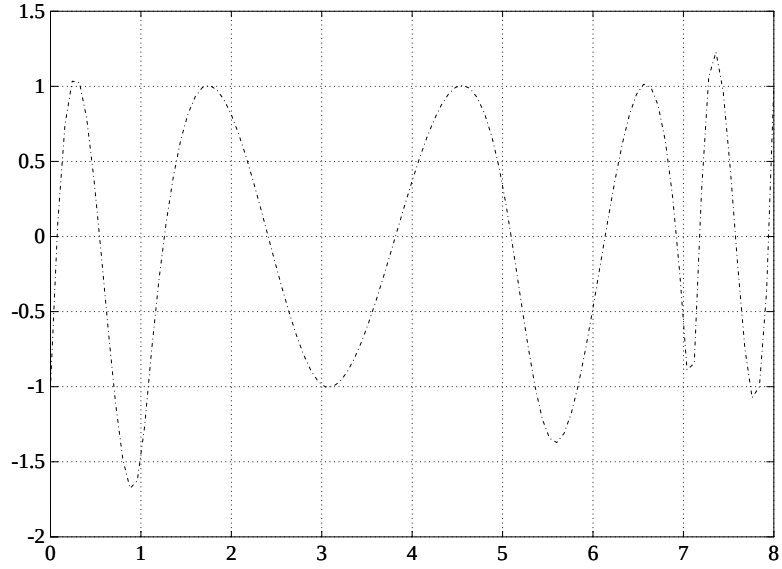
recommended as good interpolation site choices. These are supplied by

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

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

```
b = cumprod(repmat(-1,1,n)); b = b*b(end);
c = spapi(t,tau,b);
fnplt(c,'-.')
grid
```

Here is the resulting plot.



First Approximation to a Chebyshev Spline

Remez Iteration

Starting from this approximation, you use the Remez algorithm to produce a sequence of splines converging to *C*. This means that you construct new τ as the extrema of your current approximation *c* to *C* and try again. Here is the entire loop.

You find the new interior τ_i as the zeros of *Dc*, i.e., the first derivative of *c*, in several steps. First, differentiate:

```
Dc = fnder(c);
```

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

```
[knots,coefs,np,kp] = fnbrk(Dc,'knots','coefs','n','order');
```

The control polygon has the vertices (`tstar(i)`,`coefs(i)`), with `tstar` the knot averages for the spline, provided by `aveknt`:

```
tstar = aveknt(knots,kp);
```

Here are the zeros of the resulting control polygon of `Dc`:

```
npp = (1:np-1);
guess = tstar(npp) -coefs(npp).*(diff(tstar)./diff(coefs));
```

This provides already a very good first guess for the actual zeros.

Refine this estimate for the zeros of *Dc* by two steps of the secant method, taking `tau` and the resulting `guess` as your first approximations. First, evaluate *Dc* at both sets:

```
sites = tau(ones(4,1),2:n-1);
sites(1,:) = guess;
values = zeros(4,n-2);
values(1:2,:) = reshape(fnval(Dc,sites(1:2,:)),2,n-2);
```

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

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

The check

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

shows the improvement.

Now take these sites as your new `tau`,

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

and check the extrema values of your current approximation there:

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

The difference

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

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

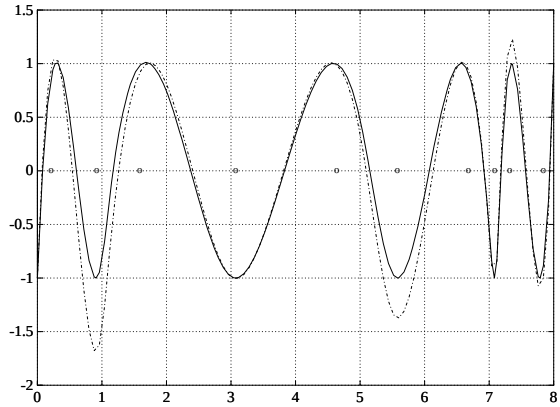
Construct a new spline corresponding to the new choice of `tau` and plot it on top of the old:

```
c = spapi(t,tau,b);
sites = sort([tau (0:100)*(t(n+1)-t(k))/100]);
values = fnval(c,sites);
hold on, plot(sites,values)
```

The following code turns on the grid and plots the locations of the extrema.

```
grid on
plot( tau(2:end-1), zeros( 1, np-1 ), 'o' )
hold off
legend( 'Initial Guess', 'Current Guess', 'Extreme Locations',...
 'location', 'NorthEastOutside' );
```

Following is the resulting figure (legend not shown).



A More Nearly Level Spline

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

Approximation by Tensor Product Splines

Because the toolbox can handle splines with *vector* coefficients, it is easy to implement interpolation or approximation to gridded data by tensor product splines, as the following illustration is meant to show. This example can also be run via the demo "Bivariate Tensor Product Splines".

To be sure, most tensor product spline approximation to gridded data can be obtained directly with one of the spline construction commands, like `spapi` or `csape`, in this toolbox, without concern for the details discussed in this example. Rather, this example is meant to illustrate the theory behind the tensor product construction, and this will be of help in situations not covered by the construction commands in this toolbox.

This section discusses these aspects of the tensor product spline problem:

Choice of Sites and Knots

Consider, for example, least squares approximation to given data $z(i,j)=f(x(i),y(j)),i=1{:}Nx,j=1{:}Ny$. You take the data from a function used extensively by Franke for the testing of schemes for surface fitting (see R. Franke, "A critical comparison of some methods for interpolation of scattered data," *Naval Postgraduate School Techn. Rep. NPS-53-79-003*, March 1979). Its domain is the unit square. You choose a few more data sites in the x-direction than the y-direction; also, for a better definition, you use higher data density near the boundary.

```
x = sort([(0:10)/10,.03 .07, .93 .97]);
```

```
y = sort([(0:6)/6,.03 .07, .93 .97]);
[xx,yy] = ndgrid(x,y); z = franke(xx,yy);
```

Least Squares Approximation as Function of y

Treat these data as coming from a vector-valued function, namely, the function of y whose value at $y(j)$ is the vector $z(:,j)$, all j. For no particular reason, choose to approximate this function by a vector-valued parabolic spline, with three uniformly spaced interior knots. This means that you choose the spline order and the knot sequence for this vector-valued spline as

```
ky = 3; knotsy = augknt([0,.25,.5,.75,1],ky);
```

and then use `spap2` to provide the least squares approximant to the data:

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

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

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

In particular, the statements

```
yy = -.1:.05:1.1;
vals = fnval(sp,yy);
```

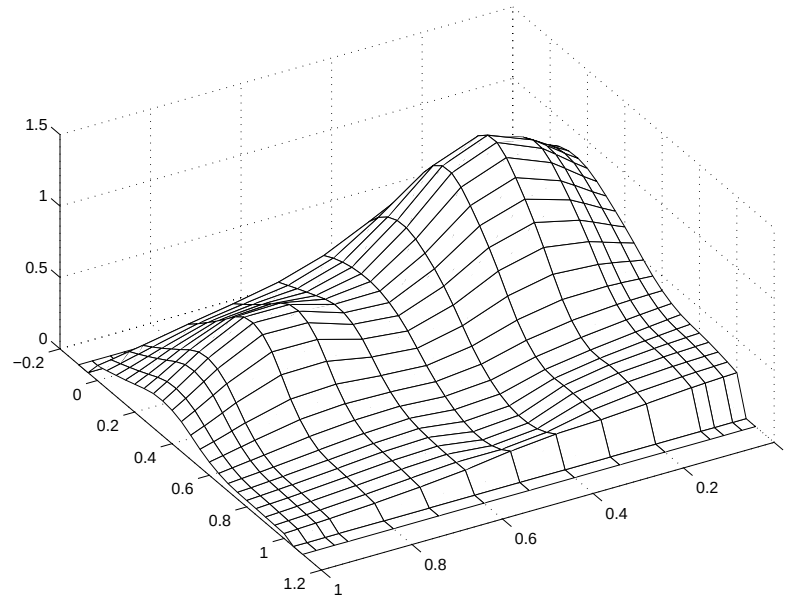
provide the array `vals`, whose entry `vals(i,j)` can be taken as an approximation to the value $f(x(i),yy(j))$of the underlying function f at the mesh-point $x(i),yy(j)$ because `vals(:,j)` is the value at $yy(j)$ of the approximating spline curve in `sp`.

This is evident in the following figure, obtained by the command:

```
mesh(x,yy,vals.'), view(150,50)
```

Note the use of `vals.'`, in the `mesh` command, needed because of the MATLAB matrix-oriented view when plotting an array. This can be a serious problem in bivariate approximation because there it is customary to think of $z(i, j)$ as

the function value at the point $(x(i), y(j))$, while MATLAB thinks of $z(i, j)$ as the function value at the point $(x(j), y(i))$.



A Family of Smooth Curves Pretending to Be a Surface

Note that both the first two and the last two values on each smooth curve are actually zero because both the first two and the last two sites in `yy` are outside the basic interval for the spline in `sp`.

Note also the ridges. They confirm that you are plotting smooth curves in one direction only.

Approximation to Coefficients as Functions of x

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

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

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

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

with the ith entry `coefsy(i,r)` of the vector coefficient `coefsy(:,r)` corresponding to $x(i)$, for all i. This suggests approximating each coefficient vector `coefsy(q,:)` by a spline of the same order `kx` and with the same appropriate knot sequence `knotsx`. For no particular reason, this time use *cubic* splines with *four* uniformly spaced interior knots:

```
kx = 4; knotsx = augknt([0:.2:1],kx);
sp2 = spap2(knotsx,kx,x,coefsy.');
```

Note that `spap2(knots,k,x,fx)` expects `fx(:,j)` to be the datum at `x(j)`, i.e., expects each *column* of `fx` to be a function value. To fit the datum `coefsy(q,:)` at $x(q)$, for all q, present `spap2` with the *transpose* of `coefsy`.

The Bivariate Approximation

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

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

It provides the *bivariate* spline approximation

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

to the original data

$$\big(x(i), y(j)\big) \mapsto z\big(x(i), y(j)\big), i = 1:Nx, j = 1:Ny$$

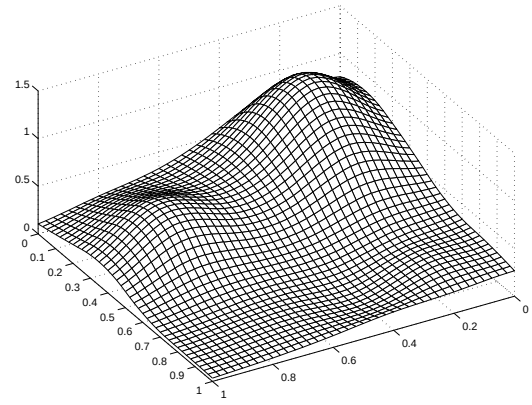
To plot this spline surface over a grid, e.g., the grid

```
xv = 0:.025:1; yv = 0:.025:1;
```

you can do the following:

```
values = spcol(knotsx,kx,xv)*coefs*spcol(knotsy,ky,yv).';
mesh(xv,yv,values.'), view(150,50);
```

This results in the following figure.



Spline Approximation to Franke's Function

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

Because the matrices `spcol(knotsx,kx,xv)` and `spcol(knotsy,ky,yv)` are banded, it may be more efficient, though perhaps more memory-consuming, for *large* `xv` and `yv` to make use of `fnval`, as follows:

```
value2 = ...
  fnval(spmak(knotsx,fnval(spmak(knotsy,coefs),yv).'),xv).';
```

This is, in fact, what happens internally when `fnval` is called directly with a tensor product spline, as in

```
value2 = fnval(spmak({knotsx,knotsy},coefs),{xv,yv});
```

Here is the calculation of the relative error, i.e., the difference between the given data and the value of the approximation at those data sites as compared with the magnitude of the given data:

```
errors = z - spcol(knotsx,kx,x)*coefs*spcol(knotsy,ky,y).';
disp( max(max(abs(errors)))/max(max(abs(z))) )
```

The output is `0.0539`, perhaps not too impressive. However, the coefficient array was only of size `8 6`

```
disp(size(coefs))
```

to fit a data array of size `15 11`.

```
disp(size(z))
```

Switch in Order

The approach followed here seems *biased*, in the following way. First think of the given data `z` as describing a vector-valued function of *y*, and then treat the matrix formed by the vector coefficients of the approximating curve as describing a vector-valued function of *x*.

What happens when you take things in the opposite order, i.e., think of `z` as describing a vector-valued function of *x*, and then treat the matrix made up from the vector coefficients of the approximating curve as describing a vector-valued function of *y*?

Perhaps surprisingly, the final approximation is the same, up to roundoff. Here is the numerical experiment.

Least Squares Approximation as Function of x

First, fit a spline curve to the data, but this time with *x* as the independent variable, hence it is the *rows* of `z` that now become the data values. Correspondingly, you must supply `z.'`, rather than `z`, to `spap2`,

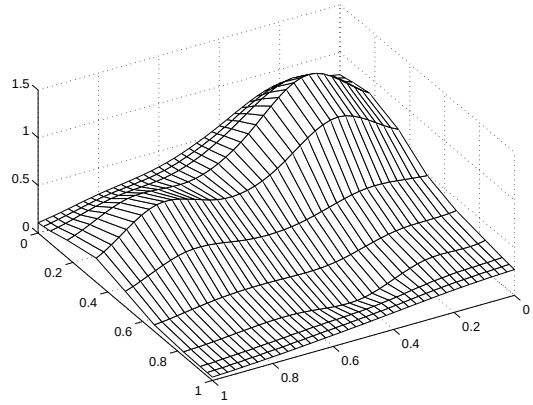
```
spb = spap2(knotsx,kx,x,z.');
```

thus obtaining a spline approximation to all the curves (*x* ; *z* (:, *j*)). In particular, the statement

```
valsb = fnval(spb,xv).';
```

provides the matrix `valsb`, whose entry `valsb(i, j)` can be taken as an approximation to the value $f(xv(i),y(j))$ of the underlying function f at the mesh-point $(xv(i),y(j))$. This is evident when you plot `valsb` using `mesh`:

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



Another Family of Smooth Curves Pretending to Be a Surface

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

Approximation to Coefficients as Functions of y

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

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

Then fit each coefficient vector `coefsx(r,:)` by a spline of the same order `ky` and with the same appropriate knot sequence `knotsy`:

```
spb2 = spap2(knotsy,ky,y,coefsx.');
```

Note that, once again, you need to transpose the coefficient array from `spb`, because `spap2` takes the columns of its last input argument as the data values.

Correspondingly, there is now no need to transpose the coefficient array `coefsb` of the resulting *curve*:

```
coefsb = fnbrk(spb2,'coefs');
```

The Bivariate Approximation

The claim is that `coefsb` equals the earlier coefficient array `coefs`, up to round-off, and here is the test:

```
disp( max(max(abs(coefs - coefsb))) )
```

The output is `1.4433e-15`.

The explanation is simple enough: The coefficients `c` of the spline *s* contained in `sp = spap2(knots,k,x,y)` depend *linearly* on the input values *y*. This implies, given that both `c` and `y` are 1-row matrices, that there is some matrix $A=A_{\mathrm{knots},k,x}$ so that

$$c = yA_{\mathrm{knots},k,x}$$

for any data `y`. This statement even holds when `y` is a *matrix*, of size *d*-by-*N*, say, in which case each datum $y(:,j)$ is taken to be a point in R^d, and the resulting spline is correspondingly *d*-vector-valued, hence its coefficient array `c` is of size *d*-by-`n`, with `n = length(knots)-k`.

In particular, the statements

```
sp = spap2(knotsy,ky,y,z);
coefsy =fnbrk(sp,'coefs');
```

provide us with the matrix `coefsy` that satisfies

$$\mathrm{coefs}y = z.A_{\mathrm{knotsy,ky,y}}$$

The subsequent computations

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

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

$$\begin{aligned}\text{coefs} &= \left(\left(z A_{\text{knotsy,ky,y}}\right)'.A_{\text{knotsx,kx,x}}\right)' \\ &= \left(A_{\text{knotsx,kx,x}}\right)'.z.A_{\text{knotsy,ky,y}}\end{aligned}$$

In the second, alternative, calculation, you first computed

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

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

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

then provided

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

Consequently, `coefsb` = `coefs`.

Comparison and Extension

The second approach is more symmetric than the first in that transposition takes place in each call to `spap2` and nowhere else. This approach can be used for approximation to gridded data in any number of variables.

If, for example, the given data over a *three*-dimensional grid are contained in some three-dimensional array `v` of size `[Nx,Ny,Nz]`, with `v(i,j,k)` containing the value $f(x(i),y(j),z(k))$, then you would start off with

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

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

```
sp = spap2(knotsx,kx,x,coefs.');
```

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

See Chapter 17 of *PGS* or [C. de Boor, "Efficient computer manipulation of tensor products," *ACM Trans. Math. Software 5* (1979), 173–182; Corrigenda, 525] for more details. The same references also make clear that there is nothing special here about using least squares approximation. Any approximation process, including spline interpolation, whose resulting approximation has coefficients that depend linearly on the given data, can be extended in the same way to a multivariate approximation process to gridded data.

This is exactly what is used in the spline construction commands `csapi`, `csape`, `spapi`, `spaps`, and `spap2`, when gridded data are to be fitted. It is also used in `fnval`, when a tensor product spline is to be evaluated on a grid.

10

Function Reference

GUIs

`bspligui`	Experiment with 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 cubic spline curve
`ppmak`	Put together spline in ppform
`rpmak`	Put together rational spline
`rscvn`	Piecewise biarc Hermite interpolation
`rsmak`	Put together rational spline for standard geometric shapes
`spap2`	Least-squares spline approximation
`spapi`	Spline interpolation
`spaps`	Smoothing spline
`spcrv`	Spline curve by uniform subdivision
`spmak`	Put together spline in B-form
`stmak`	Put together function in stform
`tpaps`	Thin-plate smoothing spline

Operators

`fn2fm`	Convert to specified form
`fnbrk`	Name and part(s) of form
`fnchg`	Change part(s) of form
`fncmb`	Arithmetic with function(s)
`fnder`	Differentiate function
`fndir`	Directional derivative of function
`fnint`	Integrate function
`fnjmp`	Jumps, i.e., f(x+)-f(x-)
`fnmin`	Minimum of function in given interval
`fnplt`	Plot function
`fnrfn`	Refine partition of form
`fntlr`	Taylor coefficients or polynomial
`fnval`	Evaluate function
`fnxtr`	Extrapolate function
`fnzeros`	Find zeros of function in given interval
`spbrk`	Part(s) of a B-form or a BBform

Work with Breaks, Knots, and Sites

`aptknt`	Acceptable knot sequence
`augknt`	Augment knot sequence
`aveknt`	Provide knot averages
`brk2knt`	Convert breaks with multiplicities into knots
`chbpnt`	Good data sites, Chebyshev-Demko points
`knt2brk, knt2mlt`	Convert knots to breaks and their multiplicities
`newknt`	New break distribution
`optknt`	Knot distribution “optimal” for interpolation
`sorted`	Locate sites with respect to mesh sites

Customized Linear Equation Solver

`bkbrk`	Part(s) of almost block-diagonal matrix
`slvblk`	Solve almost block-diagonal linear system

Information About Splines and the Toolbox

`bspline`	B-spline and its polynomial pieces
`spterms`	Explanation of Spline Toolbox terms

Utilities

`franke`	Franke's bivariate test function
`spcol`	B-spline collocation matrix
`splpp, sprpp`	Taylor coefficients from local B-coefficients
`stcol`	Scattered translates collocation matrix
`subplus`	Positive part
`titanium`	Titanium test data

Functions – Alphabetical List

This section contains Spline Toolbox reference pages, listed alphabetically. For ease of use, most toolbox functions have default arguments. The reference entry under Syntax usually first lists the function with all necessary input arguments and then with all possible input arguments. When there is more than one optional argument, then, sometimes, but not always, their exact order is immaterial. When their order does matter, you have to specify every optional argument preceding the one(s) you are interested in. In this situation, you can specify the default value for an optional argument by using [ ] (the empty matrix) as the input for it. The description in the reference page tells you the default value for each optional input argument.

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

aptknt

Purpose

Acceptable knot sequence

Syntax

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

Description

`knots = aptknt(tau,k)` returns a knot sequence suitable for interpolation at the data sites `tau` by splines of order `k` with that knot sequence, provided `tau` has at least `k` entries, is nondecreasing, and satisfies `tau(i)<tau(i+k-1)` for all `i`. In that case, there is exactly one spline of order `k` with knot sequence `knots` that matches given values at those sites. This is so because the sequence `knots` returned satisfies the Schoenberg-Whitney conditions

```
knots(i) < tau(i) < knots(i+k),        i=1:length(tau)
```

with equality only at the extreme knots, each of which occurs with exact multiplicity `k`.

If `tau` has fewer than `k` entries, then `k` is reduced to the value `length(tau)`. An error results if `tau` fails to be nondecreasing and/or `tau(i)` equals `tau(i+k-1)` for some `i`.

`[knots,k] = aptknt(tau,k)` also returns the actual `k` used (which equals the smaller of the input `k` and `length(tau)`).

Examples

If `tau` is equally spaced, e.g., equal to `linspace(a,b,n)` for some `n>=4`, and `y` is a sequence of the same size as `tau`, then `sp = spapi(aptknt(tau,4),tau,y)` gives the cubic spline interpolant with the not-a-knot end condition. This is the same cubic spline as produced by the command `spline(tau,y)`, but in B-form rather than ppform.

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

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

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 knot sequence

Syntax

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

Description

`augknt(knots,k)` returns a nondecreasing and augmented knot sequence that has the first and last knot with exact multiplicity `k`. (This may actually shorten the knot sequence.))

`augknt(knots,k,mults)` makes sure that the augmented knot sequence returned will, in addition, contain each interior knot `mults` times. If `mults` has exactly as many entries as there are interior knots, then the *j*th one will appear `mults(j)` times. Otherwise, the uniform multiplicity `mults(1)` is used. If `knots` is strictly increasing, this ensures that the splines of order `k` with knot sequence `augknot` satisfy `k-mults(j)` smoothness conditions across `knots(j+1)`, `j=1:length(knots)-2`.

`[augknot,addl] = augknt(...)` also returns the number `addl` of knots added on the left. (This number may be negative.)

Examples

If you want to construct a cubic spline on the interval [`a..b`], with two continuous derivatives, and with the interior break sequence `xi`, then `augknt([a,b,xi],4)` is the knot sequence you should use.

If you want to use Hermite cubics instead, i.e., a cubic spline with only one continuous derivative, then the appropriate knot sequence is `augknt([a,xi,b],4,2).`

`augknt([1 2 3 3 3],2)` returns the vector `[1 1 2 3 3]`, as does `augknt([3 2 3 1 3],2)`. In either case, `addl` would be `1`.

Purpose

Provide knot averages

Syntax

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

Description

`tstar = aveknt(t,k)` returns the averages of successive `k-1` knots, i.e., the sites

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

which are recommended as good interpolation site choices when interpolating from splines of order `k` with knot sequence $t = (t_i)_{i=1}^{n+k}$.

Examples

`aveknt([1 2 3 3 3],3)` returns the vector `[2.5000 3.0000]`, while `aveknt([1 2 3],3)` returns the empty vector.

With `k` and the strictly increasing sequence `breaks` given, the statements

```
t = augknt(breaks,k); x = aveknt(t);
sp = spapi(t,x,sin(x));
```

provide a spline interpolant to the sine function on the interval `[breaks(1)..breaks(end)]`.

For `sp` the B-form of a scalar-valued univariate spline function, and with `tstar` and `a` computed as

```
tstar = aveknt(fnbrk(sp,'knots'),fnbrk(sp,'order'));
a = fnbrk(sp,'coefs');
```

the points (tstar(*i*), a(*i*)) constitute the *control points* of the spline, i.e., the vertices of the spline's *control polygon*.

See Also

`aptknt`, `chbpnt`, `optknt`

bkbrk

Purpose

Part(s) of almost block-diagonal matrix

Syntax

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

Description

`[nb,rows,ncols,last,blocks] = bkbrk(blokmat)` returns the details of the almost block-diagonal matrix contained in `blokmat`, with `rows` and `last` `nb`-vectors, and `blocks` a matrix of size `[sum(rows),ncols]`.

This utility program is not likely to be of interest to the casual user. It is used in `slvblk` to decode the information, provided by `spcol`, about a spline collocation matrix in an almost block diagonal form especially suited for splines. But `bkbrk` can also decode the almost block-diagonal form used in [1].

`bkbrk(blokmat)` returns nothing, but the details are printed out. This is of use when trying to understand what went wrong with such a matrix.

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

Convert breaks with multiplicities into knots

Syntax

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

Description

`[knots,index] = brk2knt(breaks,mults)` returns the sequence `knots` that is the sequence `breaks` but with `breaks(i)` occurring `mults(i)` times, all `i`. In particular, `breaks(i)` will not appear unless `mults(i)>0`. If, as one would expect, `breaks` is a strictly increasing sequence, then `knots` contains each `breaks(i)` exactly `mults(i)` times.

If `mults` does not have exactly as many entries as does `breaks`, then all `mults(i)` are set equal to `mults(1)`.

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

The statements

```
t = [1 1 2 2 2 3 4 5 5];
[xi,m] = knt2brk(t);
tt = brk2knt(xi,m)
```

give `[1 2 3 4 5]` for `xi`, `[2 3 1 1 2]` for `m`, and, finally, `t` for `tt`.

See Also

`augknt`, `knt2brk`, `knt2mlt`

bspligui

Purpose

Experiment with B-spline as function of its knots

Syntax

```
bspligui
```

Description

`bspligui` starts a graphical user interface (GUI) for exploring how a B-spline depends on its knots. As you add, move, or delete knots, you see the B-spline and its first three derivatives change accordingly.

You observe the following basic facts about the B-spline with knot sequence $t_0 \le \cdots \le t_k$:

- 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 1 every time you add/delete a knot.
- Each knot t_j is a break for the B-spline, but it is permissible for several knots to coincide. Therefore, the number of nontrivial polynomial pieces is maximally k (when all the knots are different) and minimally 1 (when there are no "interior" knots), and any number between 1 and k is possible.
- The smoothness of the B-spline across a break depends on the multiplicity of the corresponding knot. If the break occurs in the knot sequence m times, then the (k–m)th derivative of the B-spline has a jump across that break, while all derivatives of order lower than (k–m) are continuous across that break. Thus, by varying the

multiplicity of a knot, you can control the smoothness of the B-spline across that knot.

- As one knot approaches another, the highest derivative that is continuous across both develops a jump and the higher derivatives become unbounded. But nothing dramatic happens in any of the lower-order derivatives.
- The B-spline is *bell-shaped* in the following sense: if the first derivative is not identically zero, then it has exactly one sign change in the interval $(t_0..t_k)$, hence the B-spline itself is *unimodal*, meaning that it has exactly one maximum. Further, if the second derivative is not identically zero, then it has exactly two sign changes in that interval. Finally, if the third derivative is not identically zero, then it has exactly three sign changes in that interval. This illustrates the fact that, for $j = 0{:}k - 1$, if the jth derivative is not identically zero, then it has exactly j sign changes in the interval $(t_0..t_k)$; it is this property that is meant by the term "bell-shaped". For this claim to be strictly true, one has to be careful with the meaning of "sign change" in case there are knots with multiplicities. For example, the $(k–1)$st derivative is piecewise constant, hence it cannot have $k–1$ sign changes in the straightforward sense unless there are k polynomial pieces, i.e., unless all the knots are simple.

See Also

`bspline`, `chbpnt`, `spcol`

bspline

Purpose

B-spline and its polynomial pieces

Syntax

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

Description

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

`bspline(t,window)` does the plotting in the subplot window specified by `window`; see the MATLAB command `subplot` for details.

`pp = bspline(t)` plots nothing but returns the ppform of the B-spline.

Examples

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

See Also

`bspligui`

Purpose

Good data sites, Chebyshev-Demko points

Syntax

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

Description

`tau = chbpnt(t,k)` are the extreme sites of the Chebyshev spline of order `k` with knot sequence `t`. These are particularly good sites at which to interpolate data by splines of order `k` with knot sequence `t` because the resulting interpolant is often quite close to the best uniform approximation from that spline space to the function whose values at `tau` are being interpolated.

`chbpnt(t,k,tol)` also specifies the tolerance `tol` to be used in the iterative process that constructs the Chebyshev spline. This process is terminated when the relative difference between the absolutely largest and the absolutely smallest local extremum of the spline is smaller than `tol`. The default value for `tol` is `.001`.

`[tau,sp] = chbpnt(...)` also returns, in `sp`, the Chebyshev spline.

Examples

`chbpnt([-ones(1,k),ones(1,k)],k)` provides (approximately) the extreme sites on the interval [–1 .. 1] of the Chebyshev polynomial of degree `k-1`.

If you have decided to approximate the square-root function on the interval [0 .. 1] by cubic splines, with knot sequence `t` as given by

```
k = 4; n = 10; t = augknt(((0:n)/n).^8,k);
```

then a good approximation to the square-root function from that specific spline space is given by

```
x = chbpnt(t,k); sp = spapi(t,x,sqrt(x));
```

as is evidenced by the near equi-oscillation of the error.

Algorithm

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

See Also

`aveknt`

Purpose

Cubic spline interpolation with end conditions

Syntax

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

Description

`pp = csape(x,y)` is the ppform of a cubic spline *s* with knot sequence `x` that satisfies *s*`(x(j)) = y(:,j)` for all `j`, as well as an additional *end condition* at the ends (meaning the leftmost and at the rightmost data site), namely the default condition listed below. The data values `y(:,j)` may be scalars, vectors, matrices, even ND-arrays. Data values at the same data site are averaged.

`pp = csape(x,y,conds)` lets you choose the end conditions to be used, from a rather large and varied catalog, by proper choice of `conds`. If needed, you supply the corresponding end condition values as additional data values, with the first (last) data value taken as the end condition value at the left (right) end. In other words, in that case, *s*`(x(j))` matches `y(:,j+1)` for all `j`, and the variable endcondvals used in the detailed description below is set to `y(:,[1 end])`. For some choices of `conds`, these end condition values need not be present and/or are ignored when present.

`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'` or `'clamped'`	Match endslopes (as given, with default as under "default").
`'not-a-knot'`	Make second and second-last sites inactive knots (ignoring end condition values if given).
`'periodic'`	Match first and second derivatives at left end with those at right end.
`'second'`	Match end second derivatives (as given, with default [0 0], i.e., as in `'variational'`).

`'variational'`	Set end second derivatives equal to zero (ignoring end condition values 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 ends. Explicitly, the ith derivative, D^is, is given the value endcondvals(:,j) at the left (j is 1) respectively right (j is 2) end in case conds(j) is i,i = 1:2. There are default values for `conds` and/or endcondvals.

Available conditions are:

clamped	$Ds(e)$ = endcondvals(:,j)	`if conds(j) == 1`
curved	$D^2s(e)$ = endcondvals(:,j)	`if conds(j) == 2`
Lagrange	$Ds(e) = Dp(e)$	default
periodic	$D^rs(a) = D^rs(b)$, r = 1,2	`if conds == [0 0]`
variational	$D^2s(e) = 0$	`if conds(j) == 2 &` endcondvals(:,j) `== 0`

Here, e is a (e is b), i.e., the left (right) end, in case `j` is 1 (`j` is 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 `endcondvals(:,j)` is taken to be the corresponding default value.

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

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+r-dimensional array if the function is to be r-valued). Correspondingly, `conds` is a cell array with m entries,

and end condition values may be correspondingly supplied in each of the *m* variables. This, as the last example below, of bicubic spline interpolation, makes clear, may require you to supply end conditions for end conditions.

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

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

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

for given scalars a, b, and c, and with e equal to `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

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

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,[1,zeros(1,length(y)),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);
```

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 end 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+1,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.

Here is a bivariate example that shows the need for supplying end conditions of end conditions when supplying end conditions in both variables. You reproduce the bicubic polynomial $g(x,y) = x^3y^3$ by complete bicubic interpolation. You then derive the needed data, including end condition values, directly from g in order to make it easier for you to see just how the end condition values must be placed. Finally, you check the result.

```
sites = {[0 1],[0 2]}; coefs = zeros(4,4); coefs(1,1) = 1;
g = ppmak({bx,by},coefs);
Dxg = fnval(fnder(g,[1 0]),sites);
Dyg = fnval(fnder(g,[0 1]),sites);
Dxyg = fnval(fnder(g,[1 1]),sites);
f = csape(sites,[Dxyg(1,1),   Dxg(1,:),    Dxyg(1,2); ...
```

```
                    Dyg(:,1), fnval(g,sites), Dyg(:,2) ; ...
                    Dxyg(2,1),   Dxg(2,:),    Dxyg(2,2)], ...
                                             {'complete','complete'});
  if any(squeeze(fnbrk(f,'c'))-coefs), 'this is wrong', end
```

Algorithm

The relevant tridiagonal linear system is constructed and solved using the sparse matrix capabilities of MATLAB.

See Also

`csapi`, `spapi`, `spline`

Cautionary Note

`csape` recognizes that you supplied explicit end condition values by the fact that you supplied exactly two more data values than data sites. In particular, even when using different end conditions at the two ends, if you wish to supply an end condition value at one end, you must also supply one for the other end.

csapi

Purpose

Cubic spline interpolation

Syntax

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

Description

`pp=csapi(x,y)` returns the ppform of a cubic spline s with knot sequence `x` that takes the value `y(:,j)` at `x(j)` for `j=1:length(x)`. The values `y(:,j)` can be scalars, vectors, matrices, even ND-arrays. Data points with the same data site are averaged and then sorted by their sites. With `x` the resulting sorted data sites, the spline `s` satisfies the not-a-knot end conditions, namely $\texttt{jump}_{\texttt{x(2)}} D^3 s = 0 = \texttt{jump}_{\texttt{x(end-1)}} D^3 s$ (with $D^3 s$ the third derivative of s).

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`-valued). In that case, `pp` is the ppform of an `m`-cubic spline interpolant s to such data. In particular, now $s(x\texttt{1}(i_1), \ldots, x\texttt{m}(i_\texttt{m}))$ equals $y(:, i_1, \ldots, i_\texttt{m})$ for $i_1 = 1{:}\texttt{n1}, \ldots, i_\texttt{m} = 1{:}\texttt{nm}$.

You can use the structure pp, in `fnval`, `fnder`, `fnplt`, etc, to evaluate, differentiate, plot, etc, this interpolating cubic spline.

`values = csapi(x,y,xx)` is the same as `fnval(csapi(x,y),xx)`, i.e., the values of the interpolating cubic spline at the sites specified by `xx` are 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 data and can handle gridded data.

Examples

See the demo “Spline Interpolation” for various examples.

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

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

except that the description of the spline obtained this second way will 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 because 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 the MATLAB `mesh` function, 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 the MATLAB 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`

Purpose

Cubic smoothing spline

Syntax

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

Description

`pp = csaps(x,y)` returns the ppform of a cubic smoothing spline f to the given data `x,y`, with the value of f at the data site `x(j)` approximating the data value `y(:,j)`, for `j=1:length(x)`. The values may be scalars, vectors, matrices, even ND-arrays. Data points with the same site are replaced by their (weighted) average, with its weight the sum of the corresponding weights.

This smoothing spline f minimizes

$$P\sum_{j=1}^{n} w(j)\left|y(:,j)-f(x(j))\right|^2+(1-p)\int \lambda(t)\left|D^2 f(t)\right|^2\ dt$$

Here, $|z|^2$ stands for the sum of the squares of all the entries of z, n is the number of entries of `x`, and the integral is over the smallest interval containing all the entries of `x`. The default value for the weight vector `w` in the *error measure* is `ones(size(x))`. The default value for the piecewise constant weight function λ in the *roughness measure* is the constant function 1. Further, D^2f denotes the second derivative of the function f. The default value for the *smoothing parameter*, `p`, is chosen in dependence on the given data sites `x`.

If the smoothing spline is to be evaluated outside its basic interval, it must first be properly extrapolated, by the command `pp = fnxtr(pp)`, to ensure that its second derivative is zero outside the interval spanned by the data sites.

`csaps(x,y,p)` lets you supply the smoothing parameter. The smoothing parameter determines the relative weight you would like to place on the contradictory demands of having *f* be smooth *vs* having *f* be close to the data. For `p` = 0, *f* is the least-squares straight line fit to the data, while, at the other extreme, i.e., for `p` = 1, *f* is the variational, or 'natural' cubic spline interpolant. As `p` moves from 0 to 1, the smoothing spline changes from one extreme to the other. The interesting range for `p` is often near $1/(1 + h^3/6)$, with h the average spacing of the data sites, and it is in this range that the default value for `p` is chosen. For uniformly spaced data, one would expect a close following of the data for $p = 1/(1 + h^3/60)$ and some satisfactory smoothing for $p = 1/(1 + h^3/0.6)$. You can input a `p > 1`, but this leads to a smoothing spline even rougher than the variational cubic spline interpolant.

If the input `p` is negative or empty, then the default value for `p` is used.

`[...,p] = csaps(...)` also returns the value of `p` actually used whether or not you specified `p`. This is important for experimentation which you might start with `[pp,p]=csaps(x,y)` in order to obtain a 'reasonable' first guess for `p`.

If you have difficulty choosing `p` but have some feeling for the size of the noise in `y`, consider using instead `spaps(x,y,tol)` which, in effect, chooses `p` in such a way that the roughness measure

$$\int \lambda(t)\left|D^2 s(t)\right|^2 dt$$

is as small as possible subject to the condition that the error measure

$$\sum w(j)\left|y(:,j) - s\left(x(j)\right)\right|^2$$

does not exceed the specified `tol`. This usually means that the error measure equals the specified `tol`.

The weight function λ in the roughness measure can, optionally, be specified as a (nonnegative) 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 (by making the weight function larger) or closer to the data (by making the weight functions smaller) in some parts of the interval than in others.

`csaps(x,y,p,[],w)` lets you specify the weights `w` in the error measure, as a vector of nonnegative entries of the same size as `x`.

`values = csaps(x,y,p,xx)` is the same as `fnval(csaps(x,y,p),xx)`.

`csaps(x,y,p,xx,w)` is the same as `fnval(csaps(x,y,p,[],w),xx)`.

`[...] = csaps({x1,...,xm},y,...)` provides the ppform of an `m`-variate tensor-product smoothing spline to data on a rectangular grid. Here, the first argument is a cell-array, containing the vectors `x1`, ..., `xm`, of lengths `n1`, ..., `nm`, respectively. Correspondingly, `y` is an array of size `[n1,...,nm]` (or of size `[d,n1,...,nm]` in case the data are `d`-valued), with y(:,i_1, ...,i_m) the given (perhaps noisy) value at the grid site xl(i_1), ...,xm(i_m).

In this case, `p` if input must be a cell-array with `m` entries or else an `m`-vector, except that it may also be a scalar or empty, in which case it is taken to be the cell-array whose `m` entries all equal the `p` input. The optional second output argument will always be a cell-array with `m` entries.

Further, `w` if input must be a cell-array with `m` entries, with `w{i}` either empty, to indicate the default choice, or else a nonnegative vector of the same size as `xi`.

Examples

Example 1.

```
x = linspace(0,2*pi,21); y = sin(x)+(rand(1,21)-.5)*.1;
pp = csaps(x,y, .4, [], [ones(1,10), repmat(5,1,10), 0] );
```

returns a smooth fit to the (noisy) data that is much closer to the data in the right half, because of the much larger error weight there, except for the last data point, for which the weight is zero.

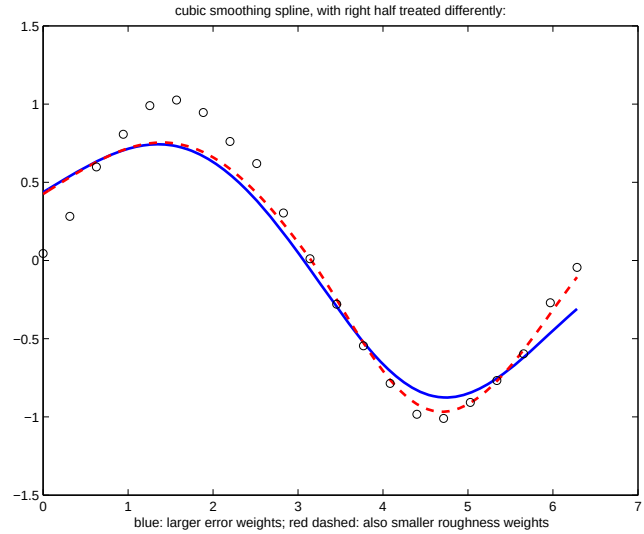
```
pp1 = csaps(x,y, [.4,ones(1,10),repmat(.2,1,10)], [], ...
                 [ones(1,10), repmat(5,1,10), 0]);
```

uses the same data, smoothing parameter, and error weight but chooses the roughness weight to be only .2 in the right half of the interval and gives, correspondingly, a rougher but better fit there, except for the last data point, which is ignored.

A plot showing both examples for comparison can now be obtained by

```
fnplt(pp); hold on, fnplt(pp1,'r--'), plot(x,y,'ok'), hold off
title(['cubic smoothing spline, with right half treated ',...
       'differently:'])
xlabel(['blue: larger error weights; ', ...
        'red dashed: also smaller roughness weights'])
```

The resulting plot is shown below.

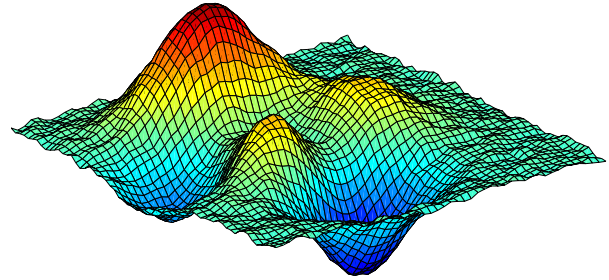


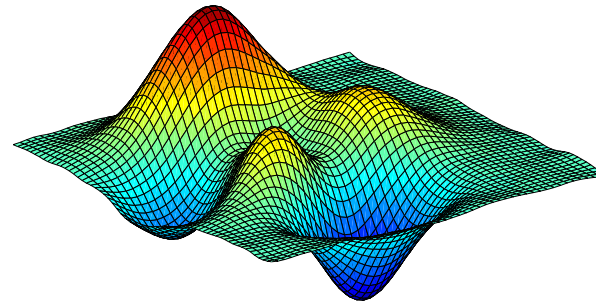
Example 2. This bivariate example adds some uniform noise, from the interval [-1/2 .. 1/2], to values of the MATLAB `peaks` function on a 51-by-61 uniform grid, obtain smoothed values for these data from `csaps`, along with the smoothing parameters chosen by `csaps`, and then plot these smoothed values.

```
x = {linspace(-2,3,51),linspace(-3,3,61)};
[xx,yy] = ndgrid(x{1},x{2}); y = peaks(xx,yy);
rand('state',0), noisy = y+(rand(size(y))-.5);
[smooth,p] = csaps(x,noisy,[],x);
surf(x{1},x{2},smooth.'), axis off
```

Note the need to transpose the array `smooth`. For a somewhat smoother approximation, use a slightly smaller value of `p` than the one, `.9998889`, used above by `csaps`. The final plot is obtained by the following:

```
smoother = csaps(x,noisy,.996,x);
figure, surf(x{1},x{2},smoother.'), axis off
```





Algorithm

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

The default value for `p` is determined as follows. The calculation of the smoothing spline requires the solution of a linear system whose coefficient matrix has the form `p*A + (1-p)*B`, with the matrices `A` and `B` depending on the data sites `x`. The default value of `p` makes `p*trace(A)` equal `(1-p)*trace(B)`.

See Also

`csape`, `spap2`, `spaps`, `tpaps`

Purpose

"Natural" or periodic interpolating cubic spline curve

Syntax

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

Description

`curve = cscvn(points)` returns a parametric variational, or *natural,* cubic spline curve (in ppform) passing through the given sequence points(:,*j*), *j* = 1:end. The parameter value *t*(*j*) for the *j*th point is chosen by Eugene Lee's [1] centripetal scheme, i.e., as accumulated square root of chord length:

$$\sum_{i<j} \sqrt{\left\| \text{points}(:,i+1) - \text{points}\,(:,i) \right\|_2}$$

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(1,:),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 end conditions) is used to construct the smooth pieces between double points (if any).

See Also

`csape`, `fnplt`, `getcurve`, `getcurv2`

References

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

fn2fm

Purpose

Convert to specified form

Syntax

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

Description

`g = fn2fm(f,form)` describes the same function as is described by `f`, but in the form specified by the string `form`. Choices for `form` are `'B-'`, `'pp'`, `'BB'`, `'rB'`, `'rp'`, for the B-form, the ppform, the BBform, and the two rational spline forms, respectively.

The B-form describes a function as a weighted sum of the B-splines of a given order k for a given knot sequence, and the BBform (or, Bernstein-Bézier form) is the special case when each knot in that sequence appears with maximal multiplicity, k. The ppform describes a function in terms of its local polynomial coefficients. The B-form is good for constructing and/or shaping a function, while the ppform is cheaper to evaluate.

Conversion from a polynomial form to the corresponding rational form is possible only if the function in the polynomial form is vector-valued, in which case its last component is designated as the denominator. Converting from a rational form to the corresponding polynomial form simply reverses this process by reinterpreting the denominator of the function in the rational form as an additional component of the piecewise polynomial function.

Conversion to or from the stform is not possible at present.

If `form` is `'B-'` (and `f` is in ppform), then the actual smoothness of the function in `f` across each of its interior breaks has to be guessed. This is done by looking, for each interior break, for the first derivative whose jump across that break is not *small* compared to the size of that derivative nearby. The default tolerance used in this is `1.e-12`.

`sp = fn2fm(f,'B-',sconds)` permits you to supply, as the input argument `sconds`, a tolerance (strictly between 0 and 1) to be used in the conversion from ppform to B-form.

Alternatively, you can input `sconds` as a vector with integer entries, with at least as many entries as the ppform in `f` has *interior* breaks. In that case, `sconds(i)` specifies the number of smoothness conditions to be used across the `i`th *interior* break. If the function in `f` is a tensor product, then `sconds`, if given, must be a cell array.

`fn2fm(f)` converts a possibly old version of a form into its present version.

Examples

`sp = fn2fm(spline(x,y),'B-')` gives the interpolating cubic spline provided by the MATLAB command `spline`, but in B-form rather than in ppform.

```
p0 = ppmak([0 1],[3 0 0]);
p1 = fn2fm(fn2fm(fnrfn(p0,[.4 .6]),'B-'),'pp');
```

gives `p1` identical to `p0` (up to round-off in the coefficients) since the spline has no discontinuity in any derivative across the additional breaks introduced by `fnrfn`, hence conversion to B-form ignores these additional breaks, and conversion to ppform does not retain any knot multiplicities (like the knot multiplicities introduced, by conversion to B-form, at the endpoints of the spline's basic interval).

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 Chapter 3, "Splines: An Overview" Without further information, such a conversion has to ascertain the actual smoothness across each interior break of the function in `f`.

See Also

`ppmak`, `spmak`, `rsmak`, `stmak`

Cautionary Note

When going from B-form to ppform, any jump discontinuity at the first and last knot, `t(1)` or `t(end)`, will be lost since the ppform considers *f* to be defined outside its basic interval by extension of the first, respectively, the last polynomial piece. For example, while `sp=spmak([0 1],1)` gives the characteristic function of the interval [0..1], `pp=fn2fm(spmak([0 1],1),'pp')` is the constant polynomial, $x \mid \rightarrow 1$.

Purpose

Name and part(s) of form

Syntax

```
[out1,...,outn] = fnbrk(f,part1,...,partm)
fnbrk(f,interval)
fnbrk(pp,j)
fnbrk(f)
```

Description

`[out1,...,outn] = fnbrk(f,part1,...,partm)` returns the part(s) of the form in `f` specified by `part1,...,partn` (assuming that `n<=m`). These are the parts used when the form was put together, in `spmak` or `ppmak` or `rpmak` or `rsmak` or `stmak`, but also other parts derived from these.

You only need to specify the beginning character(s) of the revelant string.

Regardless of what particular form `f` is in, `parti` can be one of the following.

`'form'`	The particular form used
`'variables'`	The dimension of the function's domain
`'dimension'`	The dimension of the function's target
`'coefficients'`	The coefficients in that particular form
`'interval'`	The basic interval of that form

Depending on the form in `f`, additional parts may be asked for.

If `f` is in B-form (or BBform or rBform), then additional choices for `parti` are

`'knots'`	The knot sequence
`'coefficients'`	The B-spline coefficients
`'number'`	The number of coefficients
`'order'`	The polynomial order of the spline

If `f` is in ppform (or rpform), then additional choices for `parti` are

`'breaks'`	The break sequence
`'coefficients'`	The local polynomial coefficients
`'pieces'`	The number of polynomial pieces
`'order'`	The polynomial order of the spline
`'guide'`	The local polynomial coefficients, but in the form needed for `PPVALU` in PGS

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

If `f` is in stform, then additional choices for `parti` are

`'centers'`	The centers
`'coefficients'`	The coefficients
`'number'`	Number of coefficients or terms
`'type'`	The particular type

`fnbrk(f,interval)` with `interval` a 1-by-2 matrix `[a b]` with `a<b` does not return a particular part. Rather, it returns a description of the univariate function described by `f` and in the same form but with the basic interval changed, to the interval given. If, instead, `interval` is `[ ]`, `f` is returned unchanged. This is of particular help when the function in `f` is *m*-variate, in which case `interval` must be a cell array with *m* entries, with the *i*th entry specifying the desired interval in the *i*th dimension. If that *i*th entry is `[ ]`, the basic interval in the *i*th dimension is unchanged.

`fnbrk(pp,j)`, with `pp` the ppform of a univariate function and `j` a positive integer, does not return a particular part, but returns the

ppform of the jth polynomial piece of the function in pp. If pp is the ppform of an *m*-variate function, then j must be a cell array of length *m*. In that case, each entry of j must be a positive integer or else an interval, to single out a particular polynomial piece or else to specify the basic interval in that dimension.

fnbrk(f) returns nothing, but a description of the various parts of the form is printed at the command line instead.

Examples

If p1 and p2 contain the B-form of two splines of the same order, with the same knot sequence, and the same target dimension, then

```
p1plusp2 = spmak(fnbrk(p1,'k'),fnbrk(p1,'c')+fnbrk(p2,'c'));
```

provides the (pointwise) sum of those two functions.

If pp contains the ppform of a bivariate spline with at least four polynomial pieces in the first variable, then ppp=fnbrk(pp,{4,[-1 1]}) gives the spline that agrees with the spline in pp on the rectangle [b4 .. b5] x [-1 .. 1] , where b4, b5 are the fourth and fifth entry in the break sequence for the first variable.

See Also

ppmak,rpmak,rsmak, spmak, stmak

fnchg

Purpose

Change part(s) of form

Syntax

```
f = fnchg(f,part,value)
```

Description

`f = fnchg(f,part,value)` returns the given function description `f` but with the specified `part` changed to the specified `value`.

The string `part` can be (the beginning character(s) of) :

`'dimension'`	The dimension of the function's target
`'interval'`	The basic interval of that form

The specified `value` for `part` is not checked for consistency with the rest of the description in `f` in case the string `part` terminates with the letter `z`.

Examples

`fndir(f,directions)` returns a vector-valued function even when the function described by `f` is ND-valued.You can correct this by using `fnchg` as follows:

```
fdir = fnchg( fndir(f,directions),...
           'dim',[fnbrk(f,'dim'),size(directions,2)] );
```

See Also

`fnbrk`

Purpose

Arithmetic with function(s)

Syntax

```
fn = fncmb(function,operation)
f = fncmb(function,function)
fncmb(function,matrix,function)
fncmb(function,matrix,function,matrix)
f = fncmb(function,op,function)
```

Description

The intent is to make it easy to carry out the standard linear operations of scaling and adding within a spline space without having to deal explicitly with the relevant parts of the function(s) involved.

`fn = fncmb(function,operation)` returns (a description of) the function obtained by applying to the values of the function in `function` the operation specified by `operation`. The nature of the operation depends on whether `operation` is a *scalar*, a *vector*, a *matrix*, or a *string*, as follows.

`Scalar`	Multiply the function by that scalar.
`Vector`	Add that vector to the function's values; this requires the function to be vector-valued.
`Matrix`	Apply that matrix to the function's coefficients.
`String`	Apply the function or M-file, specified by that string, to the function's coefficients.

The remaining options only work for *univariate* functions. See Limitations for more information.

`f = fncmb(function,function)` returns (a description of) the pointwise sum of the two functions. The two functions must be of the same form. This particular case of just two input arguments is not included in the above table since it only works for univariate functions.

`fncmb(function,matrix,function)` is the same as `fncmb(fncmb(function,matrix),function)`.

`fncmb(function,matrix,function,matrix)` is the same as `fncmb((fncmb(function,matrix),fncmb(function,matrix))`.

`f = fncmb(function,op,function)` returns the ppform of the spline obtained by the pointwise combining of the two functions, as specified by the string `op`. `op can be` one of the strings `'+'`, `'-'`, `'*'`. If the second function is to be a constant, it is sufficient simply to supply here that constant.

Examples

`fncmb(fn,3.5)` multiplies (the coefficients of) the function in `fn` by 3.5.

`fncmb(f,3,g,-4)` returns the linear combination, with weights 3 and –4, of the function in `f` and the function in `g`.

`fncmb(f,3,g)` adds 3 times the function in `f` to the function in `g`.

If the function *f* in `f` happens to be scalar-valued, then `f3=fncmb(f,[1;2;3])` contains the description of the function whose value at *x* is the 3-vector (*f*(*x*), 2*f*(*x*), 3*f*(*x*)). Note that, by the convention throughout this toolbox, the subsequent statement fnval(*f*3, *x*) returns a 1-*column*-matrix.

If `f` describes a surface in R^3, i.e., the function in `f` is 3-vector-valued bivariate, then `f2 = fncmb(f,[1 0 0;0 0 1])` describes the projection of that surface to the (*x*, *z*)-plane.

The following commands produce the picture of a ... spirochete?

```
c = rsmak('circle');
fnplt(fncmb(c,diag([1.5,1]))); axis equal, hold on
sc = fncmb(c,.4);
fnplt(fncmb(sc,-[.2;-.5]))
fnplt(fncmb(sc,-[.2,-.5]))
hold off, axis off
```

If `t` is a knot sequence of length `n+k` and `a` is a matrix with `n` columns, then `fncmb(spmak(t,eye(n)),a)` is the same as `spmak(t,a)`.

`fncmb(spmak([0:4],1),'+',ppmak([-1 5],[1 -1]))` is the piecewise-polynomial with breaks `-1:5` that, on the interval [0 .. 4],

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

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

Assuming that `sp` describes the B-form of a spline of order `<k`, the output of

```
fn2fm(fncmb(sp,'+',ppmak(fnbrk(sp,'interv'),zeros(1,k))),'B-')
```

describes the B-form of the same spline, but with its order raised to `k`.

Algorithm

The coefficients are extracted (via `fnbrk`) and operated on by the specified matrix or operation (and, possibly, added), then recombined with the rest of the function description (via `ppmak`, `spmak,rpmak,rsmak,stmak`). To be sure, when the function is rational, the matrix is only applied to the coefficients of the numerator. Again, if we are to translate the function values by a given vector and the function is in ppform, then only the coefficients corresponding to constant terms are so translated.

If there are two functions input, then they must be of the same type (see Limitations, below) *except* for the following.

`fncmb(f1,op,f2)` returns the ppform of the function

$$x \mid\to f1(x)\ op\ f2(x)$$

with `op` one of `'+'`, `'-'`, `'*'`, and `f1`, `f2` of arbitrary polynomial form. If, in addition, `f2` is a scalar or vector, it is taken to be the function that is constantly equal to that scalar or vector.

Limitations

`fncmb` only works for *univariate* functions, except for the case `fncmb(function,operation)`, i.e., when there is just one function in the input.

Further, if two functions are involved, then they must be of the same type. This means that they must either both be in B-form or both be in

ppform, and, moreover, have the same knots or breaks, the same order, and the same target. The only exception to this is the command of the form `fncmb(function,op,function)`.

Purpose

Differentiate function

Syntax

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

Description

`fprime = 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 or both stforms. `fnder` does not work for rational splines; for them, use `fntlr` instead. `fnder` works for stforms only in a limited way: if the type is `tp00`, then `dorder` can be `[1,0]` or `[0,1]`.

`fnder(f)` is the same as `fnder(f,1)`.

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 `fnder(fnint(f))` are the same.

If `f` is in ppform and `fa` is the value of the function in `f` at the left end of its basic interval, then, up to rounding errors, `f` and `fnint(fnder(f),fa)` are the same, unless the function described by `f` has jump discontinuities.

If `f` contains the B-form of f, and t_1 is its leftmost knot, then, up to rounding errors, `fnint(fnder(f))` contains the B-form of $f - f(t_1)$. However, its leftmost knot will have lost one multiplicity (if it had multiplicity > 1 to begin with). Also, its rightmost knot will have full multiplicity even if the rightmost knot for the B-form of f in `f` doesn't.

Here is an illustration of this last fact. The spline in `sp = spmak([0 0 1], 1)` is, on its basic interval [0..1], the straight line that is 1 at 0 and 0 at 1. Now integrate its derivative: `spdi = fnint(fnder(sp))`. As

you can check, the spline in `spdi` has the same basic interval, but, on that interval, it agrees with the straight line that is 0 at 0 and –1 at 1.

See the demos “Intro to B-form” and “Intro to ppform” for examples.

Algorithm

For differentiation of either polynomial 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.

For the stform, differentiation relies on knowing a formula for the relevant derivative of the basis function of the particular type.

See Also

`fndir`, `fnint`, `fnplt`, `fnval`

Purpose

Directional derivative of function

Syntax

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

Description

`df = fndir(f,y)` is the ppform of the directional derivative, of the function *f* in `f`, in the direction of the (column-)vector `y`. This means that `df` describes the function $D_y f(x) := \lim_{t \to 0} (f(x+ty) - f(x))/t$.

If `y` is a matrix, with `n` columns, and *f* is `d`-valued, then the function in `df` is `prod(d)*n`-valued. Its value at *x*, reshaped to be of size `[d,n]`, has in its *j*th "column" the directional derivative of *f* at *x* in the direction of the *j*th column of `y`. If you prefer `df` to reflect explicitly the actual size of *f*, use instead

```
df = fnchg( fndir(f,y), 'dim',[fnbrk(f,'dim'),size(y,2)] );
```

`Since fndir` relies on the ppform of the function in `f`, it does not work for rational functions nor for functions in stform.

Examples

For example, if `f` describes an `m`-variate `d`-vector-valued function and `x` is some point in its domain, then, e.g., with this particular ppform `f` that describes a scalar-valued bilinear polynomial,

```
f = ppmak({0:1,0:1},[1 0;0 1]); x = [0;0];
[d,m] = fnbrk(f,'dim','var');
jacobian = reshape(fnval(fndir(f,eye(m)),x),d,m)
```

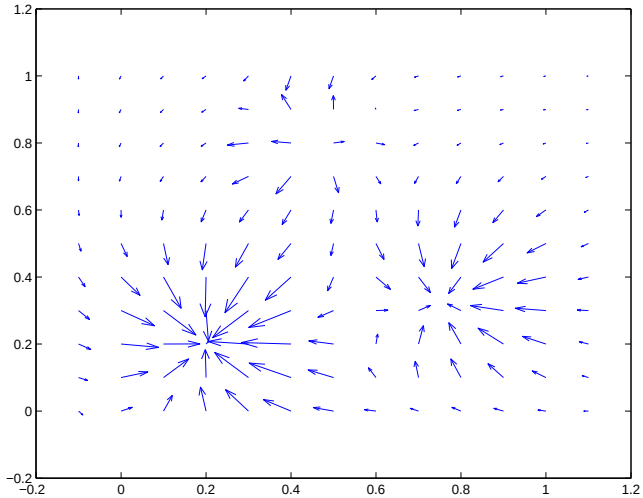
is the Jacobian of that function at that point (which, for this particular *scalar*-valued function, is its gradient, and it is zero at the origin).

As a related example, the next statements plot the gradients of (a good approximation to) the Franke function at a regular mesh:

```
xx = linspace(-.1,1.1,13); yy = linspace(0,1,11);
[x,y] = ndgrid(xx,yy); z = franke(x,y);
pp2dir = fndir(csapi({xx,yy},z),eye(2));
grads = reshape(fnval(pp2dir,[x(:) y(:)].'),...
```

```
    [2,length(xx),length(yy)]);
quiver(x,y,squeeze(grads(1,:,:)),squeeze(grads(2,:,:)))
```

Here is the resulting plot.



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

`fnchg`, `fnder`, `fnint`, `franke`

Purpose

Integrate function

Syntax

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

Description

`intgrf = fnint(f,value)` is the description of an indefinite integral of the *univariate* function whose description is contained in `f`. The integral is normalized to have the specified `value` at the left endpoint of the function's basic interval, with the default value being zero.

The output is of the same type as the input, i.e., they are both ppforms or both B-forms. `fnint` does not work for rational splines nor for functions in stform.

`fnint(f)` is the same as `fnint(f,0)`.

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

Examples

The statement `diff(fnval(fnint(f),[a b]))` provides the definite integral over the interval `[a .. b]` of the function described by `f`.

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

If `f` is in ppform and `fa` is the value of the function in `f` at the left end of its basic interval, then, up to rounding errors, `f` and `fnint(fnder(f),fa)` are the same, unless the function described by `f` has jump discontinuities.

If `f` contains the B-form of f, and t_1 is its leftmost knot, then, up to rounding errors, `fnint(fnder(f))` contains the B-form of $f - f(t_1)$. However, its leftmost knot will have lost one multiplicity (if it had multiplicity > 1 to begin with). Also, its rightmost knot will have full multiplicity even if the rightmost knot for the B-form of f in `f` doesn't.

Here is an illustration of this last fact. The spline in `sp = spmak([0 0 1], 1)` is, on its basic interval `[0..1]`, the straight line that is 1 at 0 and

0 at 1. Now integrate its derivative: `spdi = fnint(fnder(sp))`. As you can check, the spline in `spdi` has the same basic interval, but, on that interval, it agrees with the straight line that is 0 at 0 and -1 at 1.

See the demos “Intro to B-form” and “Intro to ppform” for examples.

Algorithm

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

See Also

`fnder`, `fnplt`, `fnval`

Purpose

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

Syntax

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

Description

`jumps = fnjmp(f,x)` is like `fnval(f,x)` except that it returns the jump $f(x+) - f(x-)$ across `x` (rather than the value at `x`) of the function f described by `f` and that it only works for univariate functions.

This is a function for spline specialists.

Examples

`fnjmp(ppmak(1:4,1:3),1:4)` returns the vector `[0,1,1,0]` since the `pp` function here is 1 on [1 .. 2], 2 on [2 .. 3], and 3 on [3 .. 4], hence has zero jump at 1 and 4 and a jump of 1 across both 2 and 3.

If `x` is `cos([4:-1:0]*pi/4)`, then `fnjmp(fnder(spmak(x,1),3),x)` returns the vector `[12 -24 24 -24 12]` (up to round-off). This is consistent with the fact that the spline in question is a so called perfect cubic B-spline, i.e., has an absolutely constant third derivative (on its basic interval). The modified command

```
fnjmp(fnder(fn2fm(spmak(x,1),'pp'),3),x)
```

returns instead the vector `[0 -24 24 -24 0]`, consistent with the fact that, in contrast to the B-form, a spline in ppform does not have a discontinuity in any of its derivatives at the endpoints of its basic interval. Note that `fnjmp(fnder(spmak(x,1),3),-x)` returns the vector `[12,0,0,0,12]` since `-x`, though theoretically equal to `x`, differs from `x` by 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)`.

fnmin

Purpose

Minimum of function in given interval

Syntax

```
fnmin(f)
fnmin(f,interv)
[minval,minsite] = fnmin(f,...)
```

Description

`fnmin(f)` returns the minimum value of the *scalar-valued univariate* spline in `f` on its basic interval.

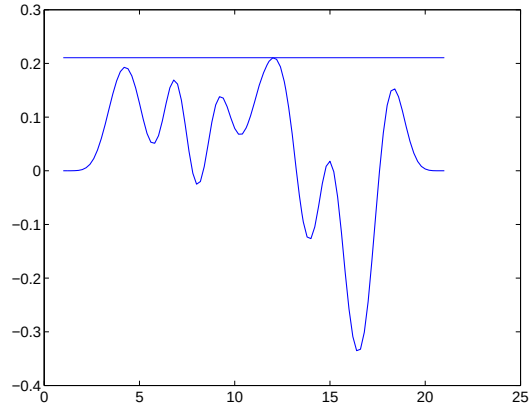
`fnmin(f,interv)` returns the minimum value on the interval `[a..b]` specified by `interv`.

`[minval,minsite] = fnmin(f,...)` also returns a location, `minsite`, at which the function in `f` takes that minimum value, `minval`.

Examples

Example 1. We construct and plot a spline *f* with many local extrema, then compute its *maximum* as the negative of the minimum of –*f*. We indicate this maximum value by adding a horizontal line to the plot at the height of the computed maximum.

```
rand('seed',21);
f = spmak(1:21,rand(1,15)-.5);
fnplt(f)
maxval = -fnmin(fncmb(f,-1));
hold on, plot(fnbrk(f,'interv'),maxval([1 1])), hold off
```



Example 2. Since `spmak(1:5,-1)` provides the negative of the cubic B-spline with knot sequence `1:5`, we expect the command

```
[y,x] = fnmin(spmak(1:5,-1))
```

to return `-2/3` for `y` and `3` for `x`.

Algorithm

`fnmin` first changes the basic interval of the function to the given interval, if any. On the interval, `fnmin` then finds all local extrema of the function as left and right limits at a jump and as zeros of the function's first derivative. It then evaluates the function at these extrema and at the endpoints of the interval, and determines the minimum over all these values.

See Also

`fnval`, `fnzeros`

fnplt

Purpose

Plot function

Syntax

```
fnplt(f)
fnplt(f,arg1,arg2,arg3,arg4)
points = fnplt(f,...)
[points, t] = fnplt(f,...)
```

Description

`fnplt(f)` plots the function, described by `f`, on its basic interval.

If *f* is univariate, the following is 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 is 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.

`fnplt(f,arg1,arg2,arg3,arg4)` permits you to modify the plotting by the specification of additional input arguments. You can place these arguments in whatever order you like, chosen from the following list:

- A *string* that specifies a plotting symbol, such as `'-.'` or `'*'`; the default is `'-'`.
- A *scalar* to specify the linewidth; the default value is `1`.

- A *string* that starts with the letter `'j'` to indicate that any jump in the *univariate* function being plotted should actually appear as a jump. The default is to fill in any jump by a (near-)vertical line.
- A *vector* of the form `[a,b]`, to indicate the interval over which to plot the `univariate` function in `f`. If the function in `f` is *m*-variate, then this optional argument must be a cell array whose ith entry specifies the interval over which the *i*th argument is to vary. In effect, for this `arg`, the command `fnplt(f,arg,...)` has the same effect as the command `fnplt(fnbrk(f,arg),...)`. The default is the basic interval of `f`.
- An empty matrix or string, to indicate use of default(s). You will find this option handy when your particular choice depends on some other variables.

`points = fnplt(f,...)` plots nothing, but the two-dimensional points or three-dimensional points it would have plotted are returned instead.

`[points, t] = fnplt(f,...)` also returns, for a vector-valued `f`, the corresponding vector `t` of parameter values.

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

Refine partition of form

Syntax

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

Description

`g = fnrfn(f,addpts)` describes the same function as does `f`, but uses more terms to do it. This is of use when the sum of two or more functions of different forms is wanted or when the number of degrees of freedom in the form is to be increased to make fine local changes possible. The precise action depends on the form in `f`.

If the form in `f` is a B-form or BBform, then the entries of `addpts` are inserted into the existing knot sequence, subject to the following restriction: The multiplicity of no knot exceed the order of the spline. The equivalent B-form with this refined knot sequence for the function given by `f` is returned.

If the form in `f` is a ppform, then the entries of `addpts` are inserted into the existing break sequence, subject to the following restriction: The break sequence be strictly increasing. The equivalent ppform with this refined break sequence for the function in `f` is returned.

`fnrfn` does not work for functions in stform.

If the function in `f` is m-variate, then `addpts` must be a cell array, `{addpts1,..., addptsm}`, and the refinement is carried out in each of the variables. If the *i*th entry in this cell array is empty, then the knot or break sequence in the *i*th variable is unchanged.

Examples

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`, `ppmak`, `spmak`

fntlr

Purpose

Taylor coefficients or polynomial

Syntax

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

Description

`taylor = fntlr(f,dorder,x)` returns the unnormalized Taylor coefficients, up to the given order `dorder` and at the given `x`, of the function described in `f` .

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

$$T(f,\mathrm{dorder},x) := [f(x); Df(x); ...; D^{\mathrm{dorder}-1} f(x)]$$

If, more generally, the function in `f` is `d`-valued with `d>1` or even `prod(d)>1` and/or is `m`-variate for some `m>1`, then `dorder` is expected to be an `m`-vector of positive integers, `x` is expected to be a matrix with `m` rows, and, in that case, the output is of size `[prod(d)*prod(dorder),size(x,2)]`, with its j-th column containing

$$T(f,\mathrm{dorder},x(:,j))(i1,...,im) = D_1{}^{i1-1} ... D_m{}^{im-1} f(x(:,j))$$

for `i1=1:dorder(1)`, ..., `im=1:dorder(m)`. Here, $D_i f$ is the partial derivative of f with respect to its ith argument.

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

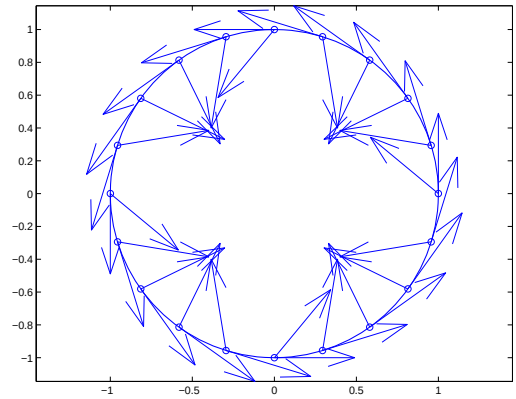
```
fnplt(ci), hold on, plot(v(1,:),v(2,:),'o')
```

Next, to verify that `v(3:4,j)` is a vector tangent to the circle at the point `v(1:2,j)`, we use the MATLAB `quiver` command to add the corresponding arrows to our plot:

```
quiver(v(1,:),v(2,:),v(3,:),v(4,:))
```

Finally, what about `v(5:6,:)`? These are second derivatives, and we add the corresponding arrows by the following `quiver` command, thus finishing First and Second Derivative of a Rational Spline Giving a Circle on page 11-54.

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

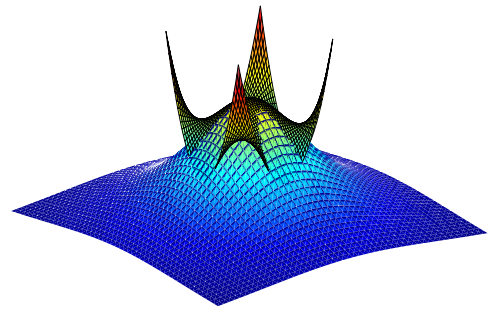


First and Second Derivative of a Rational Spline Giving a Circle

Now, our curve being a circle, you might have expected the 2nd derivative arrows to point straight to the center of that circle, and that would have been indeed the case if the function in `ci` had been using arclength as its independent variable. Since the parameter used is not arclength, we use the formula, given in "Example: A Spline Curve" on page 5-10, 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.



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, $1 + x^2 + y^2$, 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]);
```

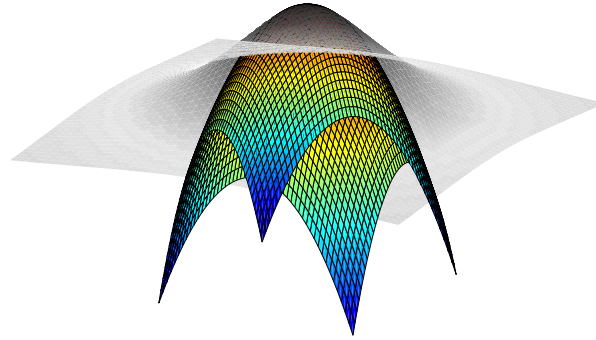
Then we enlarge the basic interval for this rational spline, plot it and plot, on top of it, its Taylor polynomial at (0,0) of order `[3,3]`.

```
fnplt(fnbrk(runge2,{[-2 2],[-2 2]}));  shading interp, hold on
fnplt(fntlr(runge2,[3 3],[0;0],[-.7 .7; -.7 .7]))
axis off, hold off
```

Since we shaded the function but not the Taylor polynomial, we can easily distinguish the two in the previous figure. 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.

```
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
```



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 function

Syntax

```
v = fnval(f,x)
fnval(x,f)
fnval(...,'l')
```

Description

`v = fnval(f,x)` and `v = fnval(x,f)` both provide the value $f(x)$ at the points in `x` of the function f whose description is contained in `f`.

Roughly speaking, the output `v` is obtained by replacing each entry of `x` by the value of f at that entry. This is literally true in case the function in `f` is scalar-valued and univariate, and is the intent in all other cases, except that, for a `d`-valued `m`-variate function, this means replacing `m`-vectors by `d`-vectors. The full details are as follows.

For a univariate f :

- If f is scalar-valued, then v is of the same size as `x`.
- If f is `[d1,...,dr]`-valued, and `x` has size `[n1,...,ns]`, then `v` has size `[d1,...,dr, n1,...,ns]`, with `v(:,...,:, j1,...,js)` the value of f at `x(j1,...,js)`, – except that

  (1) `n1` is ignored if it is `1` and `s` is `2`, i.e., if `x` is a row vector; and

  (2) MATLAB ignores any trailing singleton dimensions of `x`.

For an `m`-variate f with `m>1`, with f `[d1,...,dr]`-valued, `x` may be either an array, or else a cell array `{x1,...,xm}`.

- If `x` is an array, of size `[n1,...,ns]` say, then `n1` must equal `m`, and `v` has size `[d1,...,dr, n2,...,ns]`, with `v(:,...,:, j2,...,js)` the value of f at `x(:,j2,...,js)`, – except that

  (1) `d1`, ..., `dr` is ignored in case f is scalar-valued, i.e., both `r` and `n1` are `1`;

  (2) MATLAB ignores any trailing singleton dimensions of `x`.
- If `x` is a cell array, then it must be of the form `{x1,...,xm}`, with `xj` a vector, of length `nj`, and, in that case, `v` has size `[d1,...,dr,`

n1,...,nm], with v(:,...,:, j1,...,jm) the value of *f* at (x1(j1), ..., xm(jm)), – except that d1, ..., dr is ignored in case *f* is scalar-valued, i.e., both r and n1 are 1.

If *f* has a jump discontinuity at x, then the value *f*(*x* +), i.e., the limit from the right, is returned, except when x equals the right end of the basic interval of the form; for such x, the value *f*(*x*–), i.e., the limit from the left, is returned.

fnval(x,f) is the same as fnval(f,x).

fnval(...,'l') treats *f* as continuous from the left. This means that if *f* has a jump discontinuity at x, then the value *f*(*x*–), i.e., the limit from the left, is returned, except when x equals the left end of the basic interval; for such x, the value *f*(*x* +) is returned.

If the function is *multivariate*, then the above statements concerning continuity from the left and right apply coordinatewise.

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 polynomial spline function takes full advantage of the tensor product structure.

Evaluation of a rational spline follows up evaluation of the corresponding vector-valued spline by division of all but its last component by its last component.

Evaluation of a function in stform makes essential use of stcol, and tries to keep the matrices involved to reasonable size.

See Also

fnbrk, ppmak, rsmak, spmak, stmak

fnxtr

Purpose

Extrapolate function

Syntax

```
g = fnxtr(f,order)
fnxtr(f)
```

Description

`g = fnxtr(f,order)` returns the spline (in ppform) that agrees with the spline in `f` on the latter's basic interval but is a polynomial of the given `order` outside it, with 2 the default for `order`, in such a way that the spline in `g` satisfies at least `order` smoothness conditions at the ends of `f`'s basic interval, i.e., at the new breaks.

f must be in B-form, BBform, or ppform.

While `order` can be any nonnegative integer, `fnxtr` is useful mainly when `order` is positive but less than the order of `f`.

If `order` is zero, then `g` describes the same spline as `fn2fm(f,'B-)` but is in ppform and has a larger basic interval.

If `order` is at least as big as `f`'s order, then `g` describes the same pp as `fn2fm(f,'pp')` but uses two more pieces and has a larger basic interval.

If `f` is m-variate, then `order` may be an m-vector, in which case `order(i)` specifies the matching order to be used in the `i`-th variable, `i` = 1:m.

If `order<0`, then `g` is exactly the same as `fn2fm(f,'pp')`. This unusual option is useful when, in the multivariate case, extrapolation is to take place in only some but not all variables.

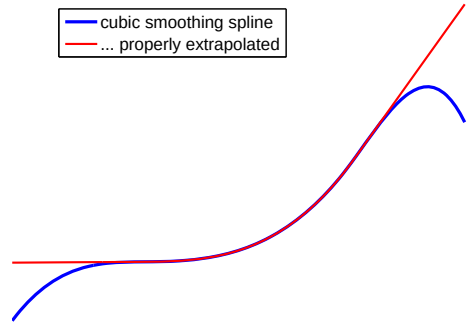
`fnxtr(f)` is the same as `fnxtr(f,2)`.

Examples

Example 1. The cubic smoothing spline for given data `x`,`y` is, like any other 'natural' cubic spline, required to have zero second derivative outside the interval spanned by the data sites. Hence, if such a spline is to be evaluated outside that interval, it should be constructed as `s = fnxtr(csaps(x,y))`. A Cubic Smoothing Spline Properly Extrapolated on page 11-61, generated by the following code, shows the difference.

```
rand('seed',6); x = rand(1,21); s = csaps(x,x.^3); sn = fnxtr(s);
```

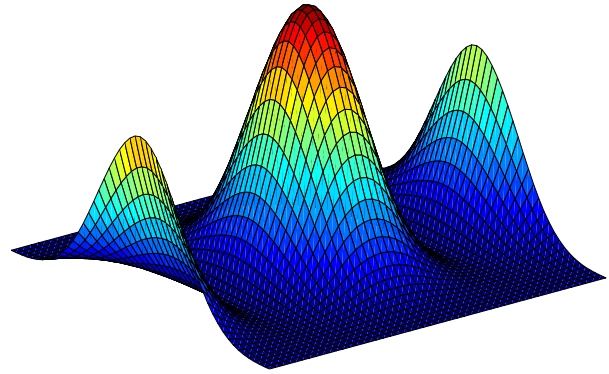
```
fnplt(s,[-.5 1.4],3), hold on, fnplt(sn,[-.5 1.4],.5,'r',2)
legend('cubic smoothing spline','... properly extrapolated')
set(gca,'Fontsize',16), axis off, hold off
```



A Cubic Smoothing Spline Properly Extrapolated

Example 2. Here is the plot of a bivariate B-spline, quadratically extrapolated in the first variable and not at all extrapolated in the second, as generated by

```
fnplt(fnxtr(spmak({0:3,0:4},1),[3,-1]))
```



A Bivariate B-spline Quadratically Extrapolated In One Direction

See Also

`ppmak`, `spmak`, `fn2fm`

Purpose

Find zeros of function in given interval

Syntax

```
z = fnzeros(f,[a b])
z = fnzeros(f)
```

Description

`z = fnzeros(f,[a b])` is an ordered list of the zeros of the univariate spline `f` in the interval `[a .. b]` .

`z = fnzeros(f)` is a list of the zeros in the basic interval of the spline `f`.

A spline zero is either a maximal closed interval over which the spline is zero, or a zero crossing (a point across which the spline changes sign).

The list of zeros, `z`, is a matrix with two rows. The first row is the left endpoint of the intervals and the second row is the right endpoint. Each column `z(:,j)` contains the left and right endpoint of a single interval.

These intervals are of three kinds:

- If the endpoints are different, then the function is zero on the entire interval. In this case the maximal interval is given, regardless of knots that may be in the interior of the interval.
- If the endpoints are the same and coincident with a knot, then the function in `f` has a zero at that point. The spline could cross zero, touch zero or be discontinuous at this point.
- If the endpoints are the same and not coincident with a knot, then the spline has a zero crossing at this point.

If the spline, `f`, touches zero at a point that is not a knot, but does not *cross* zero, then this zero may not be found. If it is found, then it may be found twice.

Examples

Example 1. The following code constructs and plots a piecewise linear spline that has each of the three kinds of zeros: touch zero, cross zero, and zero for an interval. `fnzeros`computes all the zeros, and then the code plots the results on the graph.

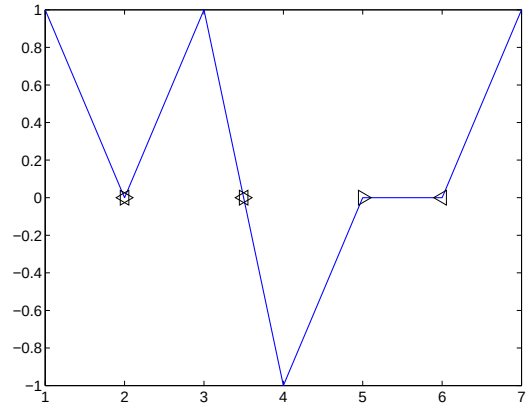
```
sp = spmak(augknt(1:7,2),[1,0,1,-1,0,0,1]);
```

```
fnplt(sp)
z = fnzeros(sp)
nz = size(z,2);
hold on
plot(z(1,:),zeros(1,nz),'>',z(2,:),zeros(1,nz),'<'), hold off
```

This gives the following list of zeros:

```
z =
    2.0000    3.5000    5.0000
    2.0000    3.5000    6.0000
```

In this simple example, even for the second kind of zero, the two endpoints agree to all places.

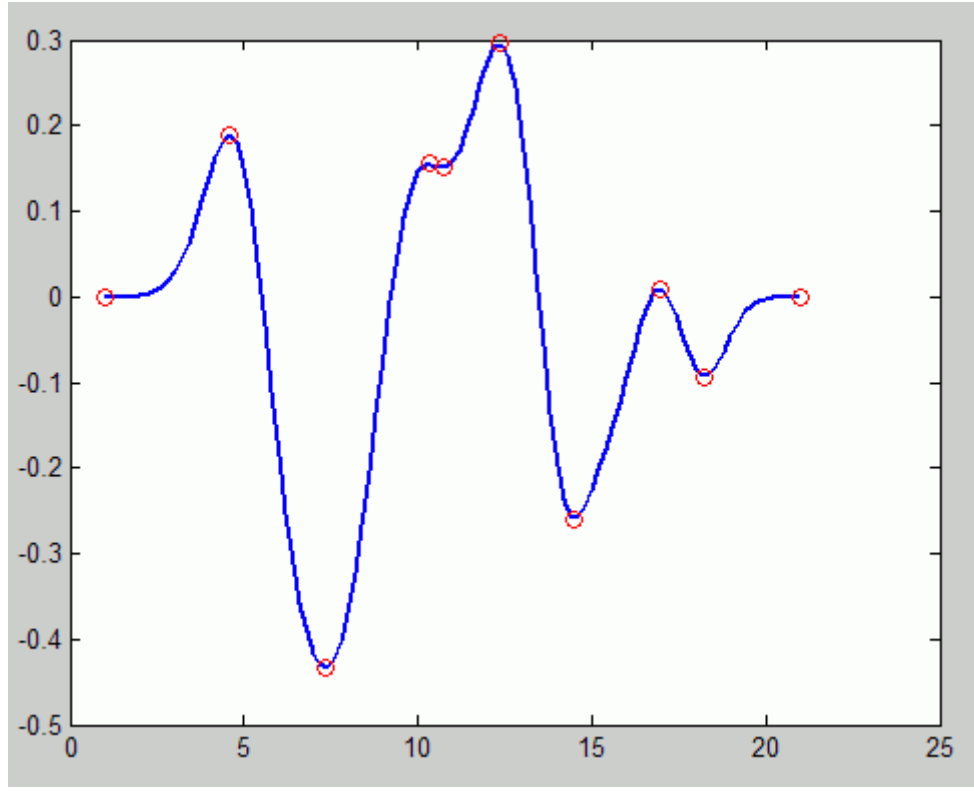


Example 2. The following code generates and plots a spline function with many extrema and locates all extrema by computing the zeros of the spline function's first derivative there.

```
f = spmak( 1:21, rand( 1, 15 )-0.5 );
interval = fnbrk( f, 'interval' );
z = fnzeros( fnder( f ) );
```

```
z = z(1,:);
values = fnval( f, [interval, z] );
min( values )
fnplt(f)
hold on
plot(z,fnval(f,z),'ro')
hold off
```

Your plot will be different to the example following because of the use of `rand` to generate random coefficients.



Example 3. We construct a spline with a zero at a jump discontinuity and in B-form and find all the spline's zeros in an interval that goes beyond its basic interval.

```
sp = spmak([0 0 1 1 2],[1 0 -.2]);
fnplt(sp)
z = fnzeros(sp,[.5, 2.7])
zy = zeros(1,size(z,2));
hold on, plot(z(1,:),zy,'>',z(2,:),zy,'<'), hold off
```

This gives the following list of zeros:

```
z =
    1.0000    2.0000
    1.0000    2.7000
```

Notice the resulting zero interval [2..2.7], due to the fact that, by definition, a spline in B-form is identically zero outside its basic interval, [0..2].

Example 4. The following example shows the use of `fnzeros` with a discontinuous function. The following code creates and plots a discontinuous piecewise linear function, and finds the zeros.

```
sp = spmak([0 0 1 1 2 2],[-1 1 -1 1]);
fnplt(sp);
fnzeros(sp)
```

This gives the following list of zeros, in (1..2) and (0..1) and the jump through zero at 1:

```
ans =

    0.5000    1.0000    1.5000
    0.5000    1.0000    1.5000
```

Algorithm

`fnzeros` first converts the function to B-form. The function performs some preprocessing to handle discontinuities, and then uses the algorithm of Mørken and Reimers.

Reference: Knut Mørken and Martin Reimers, An unconditionally convergent method for computing zeros of splines and polynomials, Math. Comp. 76:845--865, 2007.

See Also

`fnmin`, `fnval`

franke

Purpose

Franke's bivariate test function

Syntax

```
z = franke(x,y)
```

Description

`z = franke(x,y)` returns the value `z(i)` of Franke's function at the site `(x(i),y(i))`, `i=1:numel(x)`, with `z` of the same size as `x` and `y` (which must be of the same size).

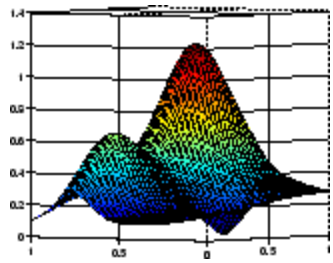
Franke's function is the following weighted sum of four exponentials:

$$\frac{3}{4}e^{-\left((9x-2)^2+(9y-2)^2\right)/4}+\frac{3}{4}e^{-\left((9x+1)^2/49-(9y+1)/10\right)}$$
$$+\frac{1}{2}e^{-\left((9x-7)^2+(9y-3)^2\right)/4}-\frac{1}{5}e^{-\left((9x-4)^2-(9y-7)^2\right)}$$

Examples

The following commands provide a plot of Franke's function:

```
pts = (0:50)/50; [x,y] = ndgrid(pts,pts); z = franke(x,y);
surf(x,y,z), view(145,-2), set(gca,'Fontsize',16)
```



References

[1] Richard Franke. "A critical comparison of some methods for interpolation of scattered data." *Naval Postgraduate School Tech.Rep.* NPS-53-79-003, March 1979.

Purpose

Interactive creation of cubic spline curve

Syntax

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

Description

`[xy,spcv] = getcurve` displays a gridded window and asks you for input. As you click on points in the gridded window, the broken line connecting these points is displayed. To indicate that you are done, click outside the gridded window. Then a cubic spline curve, `spcv`, through the point sequence, `xy`, is computed (via `cscvn`) and drawn. The point sequence and, optionally, the spline curve are output.

If you want a closed curve, place the last point *close* to the initial point.

If you would like the curve to have a corner at some point, click on that point twice (or more times) in succession.

See Also

`cscvn`

Purpose

Convert knots to breaks and their multiplicities

Syntax

```
knt2brk(knots)
[breaks,mults] = knt2brk(knots)
m = knt2mlt(t)
[m,sortedt] = knt2mlt(t)
```

Description

The commands extract the distinct elements from a sequence, as well as their multiplicities in that sequence, with *multiplicity* taken in two slightly different senses.

`knt2brk(knots)` returns the distinct elements in `knots`, and in increasing order, hence is the same as `unique(knots)`.

`[breaks,mults] = knt2brk(knots)` additionally provides, in `mults`, the multiplicity with which each distinct element occurs in `knots`. Explicitly, `breaks` and `mults` are of the same length, and `knt2brk` is complementary to `brk2knt` in that, for any knot sequence `knots`, the two commands `[xi,mlts] = knt2brk(knots); knots1 = brk2knt(xi,mlts);` give `knots1` equal to `knots`.

`m = knt2mlt(t)` returns a vector of the same length as `t`, with `m(i)` counting, in the vector `sort(t)`, the number of entries before its ith entry that are equal to that entry. This kind of multiplicity vector is needed in `spapi` or `spcol` where such multiplicity is taken to specify which particular derivatives are to be matched at the sites in `t`. Precisely, if `t` is nondecreasing and `z` is a vector of the same length, then `sp = spapi(knots, t, z)` attempts to construct a spline s (with knot sequence `knots`) for which $D^{m(i)}s(t(i))$ equals $z(i)$, all i.

`[m,sortedt] = knt2mlt(t)` also returns the output from `sort(t)`.

Neither `knt2brk` nor `knt2mlt` is likely to be used by the casual user of this toolbox.

Examples

`[xi,mlts]=knt2brk([1 2 3 3 1 3])` returns `[1 2 3]` for `xi` and `[2 1 3]` for `mlts`.

`[m,t]=knt2mlt([1 2 3 3 1 3])` returns `[0 1 0 0 1 2]` for `m` and `[1 1 2 3 3 3]` for `t`.

See Also

`brk2knt`, `spapi`, `spcol`

newknt

Purpose

New break distribution

Syntax

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

Description

`newknots = newknt(f,newl)` returns the knot sequence whose interior knots cut the basic interval of `f` into `newl` pieces, in such a way that a certain piecewise linear monotone function related to the high derivative of `f` is equidistributed.

The intent is to choose a knot sequence suitable to the fine approximation of a function g whose rough approximation in `f` is assumed to contain enough information about g to make this feasible.

`newknt(f)` uses for `newl` its default value, namely the number of polynomial pieces in `f`.

`[...,distfn] = newknt(...)` also returns, in `distfn`, the ppform of that piecewise linear monotone function being equidistributed.

Examples

If the error in the least-squares approximation `sp` to some data `x,y` by a spline of order `k` seems uneven, you might try for a more equitable distribution of knots by using

```
spap2(newknt(sp),k,x,y);
```

For another example, see the last part of the demo “Solving an ODE by Collocation”.

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

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

optknt

Purpose

Knot distribution “optimal” for interpolation

Syntax

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

Description

`knots = optknt(tau,k,maxiter)` provides the knot sequence `t` that is *best* for interpolation from $S_{k,t}$ at the site sequence `tau`, with `10` the default for the optional input `maxiter` that bounds the number of iterations to be used in this effort. Here, *best* or *optimal* is used in the sense of Micchelli/Rivlin/Winograd and Gaffney/Powell, and this means the following: For any *recovery scheme* R that provides an interpolant Rg that matches a given g at the sites `tau(1)`, ..., `tau(n)`, we may determine the smallest constant const_R for which $||g - Rg|| \le \text{const}_R \, ||D^k_g||$ 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

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

Gaffney/Powell called this interpolation scheme *optimal* since it provides the *center* function in the band formed by all interpolants to the given data that, in addition, have their kth derivative between M and $-M$ (for large M).

`optknt(tau,k)` is the same as `optknt(tau,k,10)`.

Examples

See the last part of the demo "Spline Interpolation" for an illustration. For the following highly nonuniform knot sequence

```
t = [0, .0012+[0, 1, 2+[0,.1], 4]*1e-5, .002, 1];
```

the command `optknt(t,3)` will fail, while the command `optknt(t,3,20)`, using a high value for the optional parameter `maxiter`, will succeed.

Algorithm

This is the Fortran routine `SPLOPT` in PGS. It is based on an algorithm described in , for the construction of that sign function *h* mentioned above. It is essentially Newton's method for the solution of the resulting nonlinear system of equations, with `aveknt(tau,k)` providing the first guess for `t(k+1)`, ...,`t(n)`, and some damping used to maintain the Schoenberg-Whitney conditions.

See Also

`aptknt`, `aveknt`, `newknt`

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.

ppmak

Purpose

Put together spline in ppform

Syntax

```
ppmak(breaks,coefs)
ppmak
ppmak(breaks,coefs,d)
ppmak(breaks,coefs,sizec)
```

Description

The command `ppmak(...)` puts together a spline in ppform from minimal information, with the rest inferred from that information. `fnbrk` provides any or all of the parts of the completed description. In this way, the actual data structure used for the storage of the ppform is easily modified without any effect on the various `fn...` commands that use this construct. However, the casual user is not likely to use `ppmak` explicitly, relying instead on the various spline construction commands in the toolbox to construct particular splines.

`ppmak(breaks,coefs)` returns the ppform of the spline specified by the break information in `breaks` and the coefficient information in `coefs`. How that information is interpreted depends on whether the function is univariate or multivariate, as indicated by `breaks` being a sequence or a cell array.

If `breaks` is a sequence, it must be nondecreasing, with its first entry different from its last. Then the function is assumed to be univariate, and the various parts of its ppform are determined as follows:

1. The number `l` of polynomial pieces is computed as `length(breaks)-1`, and the basic interval is, correspondingly, the interval `[breaks(1) .. breaks(l+1)]`.

2. The dimension `d` of the function's target is taken to be the number of rows in `coefs`. In other words, each column of `coefs` is taken to be one coefficient. More explicitly, `coefs(:,i*k+j)` is assumed to contain the `j`th coefficient of the `(i+1)`st polynomial piece (with the first coefficient the highest and the `k`th coefficient the lowest, or constant, coefficient). Thus, with `kl` the number of columns of `coefs`, the order `k` of the piecewise-polynomial is computed as `fix(kl/l)`.

After that, the entries of `coefs` are reordered, by the command

```
coefs = reshape(permute(reshape(coefs,[d,k,l]),[1 3 2]),[d*l,k])
```

in order to conform with the internal interpretation of the coefficient array in the ppform for a univariate spline. This only applies when you use the syntax `ppmak(breaks,coefs)` where `breaks` is a sequence (row vector), not when it is a cell-array. The permutation is not made when you use the three-argument forms of `ppmak`. For the three-argument forms only a reshape is done, not a permute.

If `breaks` is a cell array, of length `m`, then the function is assumed to be `m`-variate (tensor product), and the various parts of its ppform are determined from the input as follows:

1. The `m`-vector l has `length(breaks{i})-1` as its `i`th entry and, correspondingly, the `m`-cell array of its basic intervals has the interval `[breaks{i}(1) .. breaks{i}(end)]` as its `i`th entry.

2. The dimension `d` of the function's target and the `m`-vector `k` of (coordinate-wise polynomial) orders of its pieces are obtained directly from the size of `coefs`, as follows.

   a. If `coefs` is an `m`-dimensional array, then the function is taken to be scalar-valued, i.e., `d` is 1, and the `m`-vector `k` is computed as `size(coefs)./l`. After that, `coefs` is reshaped by the command `coefs = reshape(coefs,[1,size(coefs)])`.

   b. If `coefs` is an `(r+m)`-dimensional array, with `sizec = size(c)` say, then `d` is set to `sizec(1:r)`, and the `vector k` is computed as `sizec(r+(1:m))./l`. After that, `coefs` is reshaped by the command `coefs = reshape(coefs,[prod(d),sizec(r+(1:m))])`.

Then, `coefs` is interpreted as an equivalent array of size `[d,l(1),k(1),l(2),k(2),...,l(m),k(m)]`, with its `(:,i(1),r(1),i(2),r(2),...,i(m),r(m))`th entry the coefficient of

$$\prod_{\mu=1}^{m} \left(x(\mu) - \text{breaks}\,|\,\mu](i(\mu))\right)^{(k(\mu)-r(\mu))}$$

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

$$[\text{breaks}\,|\,\mu](i(\mu))\ ..\ \text{breaks}\,|\,\mu](i(\mu)+1)], \qquad \mu = 1:m$$

This is, in fact, the internal interpretation of the coefficient array in the ppform of a multivariate spline.

`ppmak` prompts you for `breaks` and `coefs`.

`ppmak(breaks,coefs,d)` with `d` a positive integer, also puts together the ppform of a spline from the information supplied, but expects the function to be univariate. In that case, `coefs` is taken to be of size `[d*l,k]`, with `l` obtained as `length(breaks)-1`, and this determines the order, `k`, of the spline. With this, `coefs(i*d+j,:)` is taken to be the `j`th components of the coefficient vector for the `(i+1)`st polynomial piece.

`ppmak(breaks,coefs,sizec)` with `sizec` a row vector of positive integers, also puts together the ppform of a spline from the information supplied, but interprets `coefs` to be of size `sizec` (and returns an error when `prod(size(coefs))` differs from `prod(sizec)`). This option is important only in the rare case that the input argument `coefs` is an array with one or more trailing singleton dimensions. For, MATLAB suppresses trailing singleton dimensions, hence, without this explicit specification of the intended size of `coefs`, `ppmak` would interpret `coefs` incorrectly.

Examples

The two splines

```
p1 = ppmak([1 3 4],[1 2 5 6;3 4 7 8]);
p2 = ppmak([1 3 4],[1 2;3 4;5 6;7 8],2);
```

have exactly the same ppform (2-vector-valued, of order 2). But the second command provides the coefficients in the arrangement used internally.

`ppmak([0:2],[1:6])` constructs a piecewise-polynomial function with basic interval [0..2] and consisting of two pieces of order 3, with the sole interior break 1. The resulting function is scalar, i.e., the dimension `d` of its target is 1. The function happens to be continuous at that break since the first piece is $x \mid \rightarrow x^2 + 2x + 3$, while the second piece is $x \mid \rightarrow 4(x - 1)^2 + 5(x–1) + 6$.

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 $x \mid \rightarrow (1,4)x^2 + (2,5)x + (3,6)$. In particular, the dimension `d` of its target is 2. The differently modified statement `ppmak([0:2],[1:4;5:8])` also specifies a planar curve (i.e., `d` is 2), but this one is piecewise linear; its first polynomial piece is $x \mid \rightarrow (1,5)x + (2,6)$.

Explicit specification of the dimension `d` leads, in the univariate case, to a different interpretation of the entries of `coefs`. Now the column number indicates the polynomial order of the pieces, and the row number should equal `d` times the number of pieces. Thus, the statement `ppmak([0:2],[1:4;5:8],2)` is in error, while the statement `ppmak([0:2],[1:4;5:8],1)` specifies a scalar piecewise cubic whose first piece is $x \mid \rightarrow x^3 + 2x^2 + 3x + 4$.

If you wanted to make up the constant polynomial, with basic interval [0..1] say, whose value is the matrix eye(2), then you would have to use the full optional third argument, i.e., use the command

```
pp = ppmak(0:1,eye(2),[2,2,1,1]);
```

Finally, if you want to construct a 2-vector-valued bivariate polynomial on the rectangle [–1 .. 1] x [0 .. 1], linear in the first variable and constant in the second, say

```
coefs = zeros(2,2,1); coefs(:,:,1) = [1 0; 0 1];
```

then the straightforward

```
pp = ppmak({[-1 1],[0 1]},coefs);
```

will fail, producing a scalar-valued function of order 2 in each variable, as will

```
pp = ppmak({[-1 1],[0 1]},coefs,size(coefs));
```

while the following command will succeed:

```
pp = ppmak({[-1 1],[0 1]},coefs,[2 2 1]);
```

See the demo “Intro to ppform” for other examples.

See Also

`fnbrk`

Purpose

Put together rational spline

Syntax

```
rp = rpmak(breaks,coefs)
rp = rpmak(breaks,coefs,d)
rpmak(breaks,coefs,sizec)
rs = rsmak(knots,coefs)
rs = rsmak(shape,parameters)
```

Description

Both `rpmak` and `rsmak` put together a rational spline from minimal information. `rsmak` is also equipped to provide rational splines that describe standard geometric shapes. A rational spline must be scalar- or vector-valued.

`rp = rpmak(breaks,coefs)` has the same effect as the command `ppmak(breaks, coefs)` except that the resulting ppform is tagged as a rational spline, i.e., as a rpform.

To describe what this means, let R be the piecewise-polynomial put together by the command `ppmak(breaks,coefs)`, and let $r(x) = s(x)/w(x)$ be the rational spline put together by the command `rpmak(breaks,coefs)`. If `v` is the value of R at x, then `v(1:end-1)/v(end)` is the value of r at x. In other words, $R(x) = [s(x);w(x)]$. Correspondingly, the dimension of the target of r is one less than the dimension of the target of R. In particular, the dimension (of the target) of R must be at least 2, i.e., the coefficients specified by `coefs` must be `d`-vectors with `d` > 1. See `ppmak` for how the input arrays `breaks` and `coefs` are being interpreted, hence how they are to be specified in order to produce a particular piecewise-polynomial.

`rp = rpmak(breaks,coefs,d)` has the same effect as `ppmak(breaks,coefs,d+1)`, except that the resulting ppform is tagged as being a rpform. Note that the desire to have that optional third argument specify the dimension of the target requires different values for it in `rpmak` and `ppmak` for the same coefficient array `coefs`.

`rpmak(breaks,coefs,sizec)` has the same effect as `ppmak(breaks,coefs,sizec)` except that the resulting ppform is tagged as being a rpform, and the target dimension is taken to be `sizec(1)-1`.

`rs = rsmak(knots,coefs)` is similarly related to `spmak(knots,coefs)`, and `rsmak(knots,coefs,sizec)` to `spmak(knots,coefs,sizec)`. In particular, `rsmak(knots,coefs)` puts together a rational spline in B-form, i.e., it provides a rBform. See `spmak` for how the input arrays `knots` and `coefs` are being interpreted, hence how they are to be specified in order to produce a particular piecewise-polynomial.

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

```
rsmak('arc',radius,center,[alpha,beta])
rsmak('circle',radius,center)
rsmak('cone',radius,halfheight)
rsmak('cylinder',radius,height)
rsmak('southcap',radius,center)
rsmak('torus',radius,ratio)
```

with `1` the default value for `radius`, `halfheight` and `height`, and the origin the default for `center`, and the arc running through all the angles from `alpha` to `beta` (default is `[-pi/2,pi/2]`), and the cone, cylinder, and torus centered at the origin with their major circle in the (x,y)-plane, and the minor circle of the torus having radius `radius*ratio`, the default for `ratio` being `1/3`.

From these, one may generate related shapes by affine transformations, with the help of `fncmb(rs,transformation)`.

All `fn...` commands except `fnint`, `fnder`, `fndir` can handle rational splines.

Examples

The commands

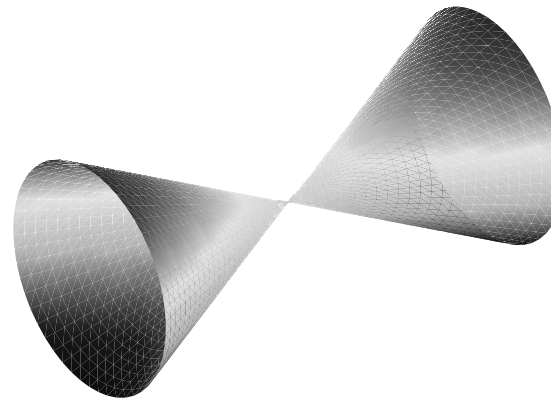
```
runges = rsmak([-5 -5 -5 5 5 5],[1 1 1; 26 -24 26]);
rungep = rpmak([-5 5],[0 0 1; 1 -10 26],1);
```

both provide a description of the rational polynomial $r(x) = 1/(x^2 + 1)$ on the interval [-5 .. 5]. However, outside the interval [-5 .. 5], the function

given by `runges` is zero, while the rational spline given by `rungep` agrees with $1/(x^2 + 1)$ for every x.

The figure of a rotated cone is generated by the commands

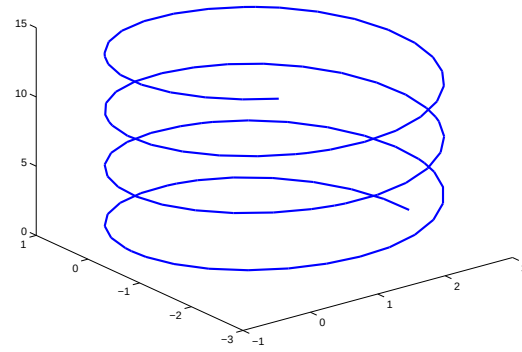
```
fnplt(fncmb(rsmak('cone',1,2),[0 0 -1;0 1 0;1 0 0]))
axis equal, axis off, shading interp
```



A Rotated Cone Given by a Rational Quadratic Spline

A Helix on page 11-84, showing a helix with several windings, is generated by the commands

```
arc = rsmak('arc',2,[1;-1],[0 7.3*pi]);
[knots,c] = fnbrk(arc,'k','c');
helix = rsmak(knots, [c(1:2,:);aveknt(knots,3).*c(3,:);
c(3,:)]);
fnplt(helix)
```



A Helix

For further illustrated examples, see Chapter 7, “NURBS and Other Rational Splines”

See Also

`rsmak`, `fnbrk`, `ppmak`, `spmak`

Purpose

Piecewise biarc Hermite interpolation

Syntax

```
c = rscvn(p,u)
c = rscvn(p)
```

Description

`c = rscvn(p,u)` returns a planar piecewise biarc curve (in quadratic rBform) that passes, in order, through the given points `p(:,j)` and is constructed in the following way (see Construction of a Biarc on page 11-87). Between any two distinct points `p(:,j)` and `p(:,j+1)`, the curve usually consists of two circular arcs (including straight-line segments) which join with tangent continuity, with the first arc starting at `p(:,j)` and normal there to `u(:,j)`, and the second arc ending at `p(:,j+1)` and normal there to `u(:,j+1)`, and with the two arcs written as one whenever that is possible. Thus the curve is tangent-continuous everywhere except, perhaps, at repeated points, where the curve may have a corner, or when the angle, formed by the two segments ending at `p(:,j)`, is unusually small, in which case the curve may have a cusp at that point.

`p` must be a real matrix, with two rows, and at least two columns, and any column must be different from at least one of its neighboring columns.

`u` must be a real matrix with two rows, with the same number of columns as `p` (for two exceptions, see below), and can have no zero column.

`c = rscvn(p)` chooses the normals in the following way. For `j=2:end-1`, `u(:,j)` is the average of the (normalized, right-turning) normals to the vectors `p(:,j)-p(:,j-1)` and `p(:,j+1)-p(:,j)`. If `p(:,1)==p(:,end)`, then both end normals are chosen as the average of the normals to `p(:,2)-p(:,1)`and `p(:,end)-p(:,end-1)`, thus preventing a corner in the resulting closed curve. Otherwise, the end normals are so chosen that there is only one arc over the first and last segment (not-a-knot end condition).

`rscvn(p,u)`, with `u` having exactly two columns, also chooses the interior normals as for the case when `u` is absent but uses the two columns of `u` as the end-point normals.

Examples

Example 1. The following code generates a description of a circle, using just four pieces. Except for a different scaling of the knot sequence, it is the same description as is supplied by `rsmak('circle',1,[1;1])`.

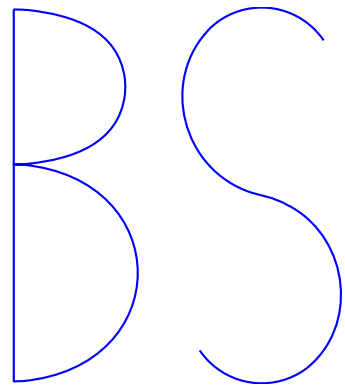
```
p = [1 0 -1 0 1; 0 1 0 -1 0]; c = rscvn([p(1,:)+1;p(2,:)+1],p);
```

The same circle, but using just two pieces, is provided by

```
c2 = rscvn([0,2,0; 1,1,1]);
```

Example 2. The following code plots two letters. Note that the second letter is the result of interpolation to just four points. Note also the use of translation in the plotting of the second letter.

```
p = [-1 .8 -1 1 -1 -1 -1; 3 1.75 .5 -1.25 -3 -3  3];
i = eye(2); u = i(:,[2 1 2 1 2 1 1]); B = rscvn(p,u);
S = rscvn([1 -1 1 -1; 2.5 2.5 -2.5 -2.5]);
fnplt(B), hold on, fnplt(fncmb(S,[3;0])), hold off
axis equal, axis off
```

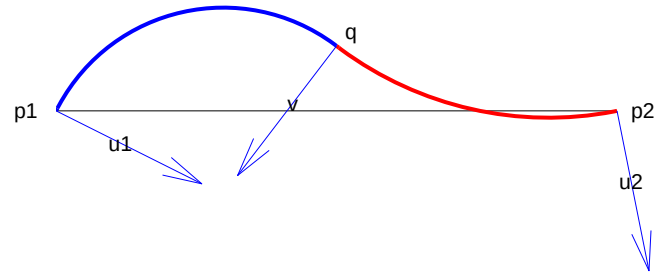


Two Letters Composed of Circular Arcs

Example 3. The following code generates the Construction of a Biarc on page 11-87, of use in the discussion below of the biarc construction

used here. Note the use of `fntlr` to find the tangent to the biarc at the beginning, at the point where the two arcs join, and at the end.

```
p = [0 1;0 0];  u = [.5 -.1;-.25 .5];
plot(p(1,:),p(2,:),'k'), hold on
biarc = rscvn(p,u); breaks = fnbrk(biarc,'b');
fnplt(biarc,breaks(1:2),'b',3), fnplt(biarc,breaks(2:3),'r',3)
vd = fntlr(biarc,2,breaks);
quiver(vd(1,:),vd(2,:),vd(4,:),-vd(3,:)), hold off
```

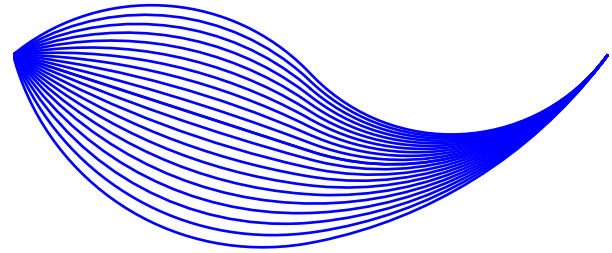


Construction of a Biarc

Algorithm

Given two distinct points, `p1` and `p2`, in the plane and, correspondingly, two nonzero vectors, `u1` and `u2`, there is a one-parameter family of biarcs (i.e., a curve consisting of two arcs with common tangent at their join) starting at `p1` and normal there to `u1` and ending at `p2` and normal there to `u2`. One way to parametrize this family of biarcs is by the normal direction, `v`, at the point `q` at which the two arcs join. With a nonzero `v` chosen, there is then exactly one choice of `q`, hence the entire

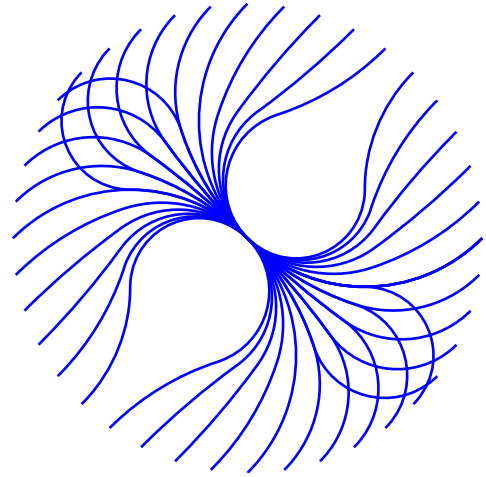
biarc is then determined. In the construction used in `rscvn`, `v` is chosen as the reflection, across the perpendicular to the segment from `p1` to `p2`, of the average of the vectors `u1` and `u2`, -- after both vectors have been so normalized that their length is 1 and that they both point to the right of the segment from `p1` to `p2`. This choice for `v` seems natural in the two standard cases: (i) `u2` is the reflection of `u1` across the perpendicular to the segment from `p1` to `p2`; (ii) `u1` and `u2` are parallel. This choice of `v` is validated by Biarcs as a Function of the Left Normal on page 11-88 which shows the resulting biarcs when `p1`, `p2`, and `u2 = [.809;.588]`are kept fixed and only the normal at `p1` is allowed to vary.



Biarcs as a Function of the Left Normal

But it is impossible to have the interpolating biarc depend continuously at all four data, `p1`, `p2`, `u1`, `u2`. There has to be a discontinuity as the normal directions, `u1` and `u2`, pass through the direction from `p1` to `p2`. This is illustrated in Biarcs as a Function of One Endpoint on page 11-89 which shows the biarcs when one point, `p1 = [0;0]`, and the two

normals, `u1 = [1;1]` and `u2 = [1;-1]`, are held fixed and only the other point, `p2`, moves, on a circle around `p1`.



Biarcs as a Function of One Endpoint

See Also

`rsmak`, `cscvn`

rsmak

Purpose

Put together rational spline for standard geometric shapes

Syntax

```
rs = rsmak(shape,parameters)
```

Description

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

```
rsmak('arc',radius,center,[alpha,beta])
rsmak('circle',radius,center)
rsmak('cone',radius,halfheight)
rsmak('cylinder',radius,height)
rsmak('southcap',radius,center)
rsmak('torus',radius,ratio)
```

with `1` the default value for `radius`, `halfheight` and `height`, and the origin the default for `center`, and the arc running through all the angles from `alpha` to `beta` (default is `[-pi/2,pi/2]`), and the cone, cylinder, and torus centered at the origin with their major circle in the (x,y)-plane, and the minor circle of the torus having radius `radius*ratio`, the default for `ratio` being `1/3`.

From these, one may generate related shapes by affine transformations, with the help of `fncmb(rs,transformation)`.

See `rpmak` for more information on other options.

See Also

`rpmak`

Purpose

Solve almost block-diagonal linear system

Syntax

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

Description

`x = slvblk(blokmat,b)` returns the solution (if any) of the linear system `Ax = b`, with the matrix `A` stored in `blokmat` in the spline almost block-diagonal form. At present, only the command `spcol` provides such a description, of the matrix whose typical entry is the value of some derivative (including the 0th derivative, i.e., the value) of a B-spline at some site. If the linear system is overdetermined (i.e., has more equations than unknowns but is of full rank), then the least-squares solution is returned.

The right side `b` may contain several columns, and is expected to contain as many rows as there are rows in the matrix described by `blokmat`.

`x = slvblk(blockmat,b,w)` returns the vector `x` that minimizes the *weighted* sum $\Sigma_j w(j)((Ax - b)(j))^2$.

Examples

`sp=spmak(knots,slvblk(spcol(knots,k,x,1),y.'))` provides in `sp` the B-form of the spline s of order `k` with knot sequence `knots` that matches the given data `(x,y)`, i.e., for which `s(x)` equals `y`.

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

sorted

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.

`pointer = sorted(meshsites,sites)` is the integer row vector whose `j`-th entry equals the number of entries in `meshsites` that are $\le$ `ssites(j)`, with `ssites` the vector `sort(sites)`. Thus, if both `meshsites` and `sites` are nondecreasing, then

```
meshsites(pointer(j))   sites(j) < meshsites(pointer(j)+1)
```

with the obvious interpretations when

```
pointer(j) < 1    or    length(meshsites) < pointer(j) + 1
```

Specifically, having `pointer(j) < 1` then corresponds to having `sites(j)` strictly to the left of `meshsites(1)`, while having `length(meshsites) < pointer(j)+1` then corresponds to having `sites(j)` at, or to the right of, `meshsites(end)`.

Examples

The statement

```
sorted([1 1 1 2 2 3 3 3],[0:4])
```

will generate the output `0 3 5 8 8`, as will the statement

```
sorted([3 2 1 1 3 2 3 1],[2 3 0 4 1])
```

Algorithm

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

Purpose

Least-squares spline approximation

Syntax

```
spap2(knots,k,x,y)
spap2(l,k,x,y)
sp = spap2(...,x,y,w)
spap2({knorl1,...,knorlm},k,{x1,...,xm},y)
spap2({knorl1,...,knorlm},k,{x1,...,xm},y,w)
```

Description

`spap2(knots,k,x,y)` returns the B-form of the spline *f* of order `k` with the given knot sequence `knots` for which

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

in the weighted mean-square sense, meaning that the sum

$$\sum_j w(j)\,|\,y(:,j) - f\,(x(j))\,|^2$$

is minimized, with default weights equal to 1. The data values `y(:,j)` may be scalars, vectors, matrices, even ND-arrays, and $|z|^2$ stands for the sum of the squares of all the entries of z. Data points with the same site are replaced by their average.

If the sites `x` satisfy the (Schoenberg-Whitney) conditions

$$\begin{aligned} &\text{knots}(j) < x(j) < \text{knots}(j+k) \\ (**) \qquad &\qquad j = 1,\ldots,\text{length}(x) = \text{length}(\text{knots}) - k \end{aligned}$$

then there is a unique spline (of the given order and knot sequence) satisfying (*) exactly. No spline is returned unless (**) is satisfied for some subsequence of `x`.

`spap2(l,k,x,y)` , with `l` a positive integer, returns the B-form of a least-squares spline approximant, but with the knot sequence chosen for you. The knot sequence is obtained by applying `aptknt` to an appropriate subsequence of `x`. The resulting piecewise-polynomial consists of `l` polynomial pieces and has `k-2` continuous derivatives.

If you feel that a different distribution of the interior knots might do a better job, follow this up with

```
sp1 = spap2(newknt(sp),k,x,y));
```

`sp = spap2(...,x,y,w)` lets you specify the weights `w` in the error measure (given above). `w` must be a vector of the same size as `x`, with nonnegative entries. All the weights corresponding to data points with the same site are summed when those data points are replaced by their average.

`spap2({knorl1,...,knorlm},k,{x1,...,xm},y)` provides a least-squares spline approximation to *gridded* data. Here, each `knorli` is either a knot sequence or a positive integer. Further, `k` must be an `m`-vector, and `y` must be an `(r+m)`-dimensional array, with `y(:,i1,...,im)` the datum to be fitted at the `site [x{1}(i1),...,x{m}(im)]`, all `i1`, ..., `im`. However, if the spline is to be scalar-valued, then, in contrast to the univariate case, `y` is permitted to be an `m`-dimensional array, in which case `y(i1,...,im)` is the datum to be fitted at the `site [x{1}(i1),...,x{m}(im)]`, all `i1`, ..., `im`.

`spap2({knorl1,...,knorlm},k,{x1,...,xm},y,w)` also lets you specify the weights. In this `m`-variate case, `w` must be a cell array with `m` entries, with `w{i}` a nonnegative vector of the same size as `xi`, or else `w{i}` must be empty, in which case the default weights are used in the `i`th variable.

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 `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 `l` polynomial pieces, use instead

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

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

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

for a possibly 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 $(B_{j,k}(x_i))$, and `slvblk` solves the linear system (*) in the (weighted) least-squares sense, using a block QR factorization.

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

See Also

`slvblk`, `spapi`, `spcol`

Purpose

Spline interpolation

Syntax

```
spline = spapi(knots,x,y)
spapi(k,x,y)
spapi({knork1,...,knorkm},{x1,...,xm},y)
spapi(...,'noderiv')
```

Description

`spline = spapi(knots,x,y)` returns the spline *f* (if any) of order

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

with knot sequence `knots` for which

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

If some of the entries of `x` are the same, then this is taken in the osculatory sense, i.e., in the sense that $D^{m(j)}f(x(j) = y(:, j)$, with `m(j) :=` #{ `i < j : x(i) = x(j)` }, and $D^m f$ the mth derivative of f. Thus r-fold repetition of a site z in `x` corresponds to the prescribing of value and the first $r - 1$ derivatives of f at z. If you don't want this, call `spapi` with an additional (fourth) argument, in which case, at each data site, the average of all data values with the same data site is matched.

The data values, `y(:,j)`, may be scalars, vectors, matrices, or even ND-arrays.

`spapi(k,x,y)` , with `k` a positive integer, merely specifies the desired spline order, `k`, in which case `aptknt` is used to determine a workable (though not necessarily optimal) knot sequence for the given sites `x`. In other words, the command `spapi(k,x,y)` has the same effect as the more explicit command `spapi(aptknt(x,k),x,y)`.

`spapi({knork1,...,knorkm},{x1,...,xm},y)` returns the B-form of a tensor-product spline interpolant to *gridded* data. Here, each `knorki` is either a knot sequence, or else is a positive integer specifying the polynomial order to be used in the `i`th variable, thus leaving it to `spapi` to provide a corresponding knot sequence for the `i`th variable. Further, `y` must be an `(r+m)`-dimensional array, with `y(:,i1,...,im)` the datum to be fitted at the site `[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.

`spapi(...,'noderiv')` with the string `'noderiv'` as a fourth argument, has the same effect as `spapi(...)` except that data values sharing the same site are interpreted differently. With the fourth argument present, the average of the data values with the same data site is interpolated at such a site. Without it, data values with the same data site are interpreted as values of successive derivatives to be matched at such a site, as described above, in the first paragraph of this Description.

Examples

`spapi([0 0 0 0 1 2 2 2 2],[0 1 1 1 2],[2 0 1 2 -1])` produces the unique cubic spline f on the interval [0..2] with exactly one interior knot, at 1, that satisfies the five conditions

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

These include 3-fold matching at 1, i.e., matching there to prescribed values of the function and its first two derivatives.

Here is an example of osculatory interpolation, to values `y` and slopes `s` at the sites `x` by a quintic spline:

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

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

As a related example, if the function `sin(x)` is to be 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 with the aid of `aptknt`. In fact, if you wanted to interpolate the same data by quintic splines, simply change the `4` to `6`.

As a bivariate example, here is a bivariate interpolant.

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

As an illustration of osculatory interpolation to gridded data, here is complete bicubic interpolation, with the data explicitly derived from the bicubic polynomial $g(u,v) = u^3v^3$, to make it easy for you to see exactly where the slopes and slopes of slopes (i.e., cross derivatives) must be placed in the data values supplied. Since our g is a bicubic polynomial, its interpolant, f, must be g itself. We test this.

```
sites = {[0,1],[0,2]}; coefs = zeros(4,4); coefs(1,1) = 1;
g = ppmak(sites,coefs);
Dxg = fnval(fnder(g,[1,0]),sites);
Dyg = fnval(fnder(g,[0,1]),sites);
Dxyg = fnval(fnder(g,[1,1]),sites);
f = spapi({4,4}, {sites{1}([1,2,1,2]),sites{2}([1,2,1,2])}, ...
          [fnval(g,sites), Dyg ; ...
           Dxg.'         , Dxyg]);
if any( squeeze( fnbrk(fn2fm(f,'pp'), 'c') ) - coefs )
'something went wrong', end
```

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

`csapi`, `spap2`, `spaps`, `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

$$\text{knots}(j) < x(j) < \text{knots}(j+k), \text{ all } j$$

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

Purpose

Smoothing spline

Syntax

```
sp = spaps(x,y,tol)
[sp,values] = spaps(x,y,tol)
[sp,values,rho] = spaps(x,y,tol)
[...] = spaps(x,y,tol,arg1,arg2,...)
[...] = spaps({x1,...,xr},y,tol,...)
```

Description

`sp = spaps(x,y,tol)` returns the B-form of the smoothest function *f* that lies within the given tolerance `tol` of the given data points `(x(j), y(:,j)), j=1:length(x)`. The data values `y(:,j)` may be scalars, vectors, matrices, even ND-arrays. Data points with the same data site are replaced by their weighted average, with its weight the sum of the corresponding weights, and the tolerance `tol` is reduced accordingly.

`[sp,values] = spaps(x,y,tol)` also returns the smoothed values, i.e., `values` is the same as `fnval(sp,x)`.

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

$$E(f) = \sum_{j=1}^{n} w(j) \, | \, (y(:,j) - f(x(j))) \, |^2$$

with the default choice for the weights `w` making $E(f)$ the composite trapezoidal rule approximation to $\int_{\min(x)}^{\max(x)} | \, y - f \, |^2$, and $|z|^2$ denoting the sum of squares of the entries of z.

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

$$F(D^m f) = \int_{\min(x)}^{\max(x)} \lambda(t) \left| D^m f(t) \right|^2 dt$$

where $D^m f$ denotes the `m`th derivative of f. The default value for `m` is `2`, the default value for the roughness measure weight λ is the constant 1, and this makes f a cubic smoothing spline.

When `tol` is nonnegative, then the spline f is determined as the unique minimizer of the expression $\rho E(f) + F(D^m f)$, with the smoothing parameter ρ (optionally returned) so chosen that $E(f)$ equals `tol`. Hence, when `m` is `2`, then, after conversion to ppform, the result should be the same (up to roundoff) as obtained by csaps(x,y,$\rho/(\rho + 1)$). Further, when `tol` is zero, then the "natural" or variational spline interpolant of order $2m$ is returned. For large enough `tol`, the least-squares approximation to the data by polynomials of degree `<m` is returned.

When `tol` is negative, then ρ is taken to be `-tol`.

The default value for the weight function λ in the roughness measure is the constant function 1. But you may choose it to be, more generally, a piecewise constant function, with breaks only at the data sites. Assuming the vector `x` to be strictly increasing, you specify such a piecewise constant λ by inputting `tol` as a vector of the same size as `x`. In that case, `tol(i)` is taken as the constant value of λ on the interval (`x(i-1)` .. `x(i)`), `i=2:length(x)`, while `tol(1)` continues to be used as the specified tolerance.

`[sp,values,rho] = spaps(x,y,tol)` also returns the actual value of ρ used as the third output argument.

`[...] = spaps(x,y,tol,arg1,arg2,...)` lets you specify the weight vector `w` and/or the integer `m`, by supplying them as an `argi`. For this, `w` must be a nonnegative vector of the same size as `x`; `m` must be `1` (for a piecewise linear smoothing spline), or `2` (for the default cubic smoothing spline), or `3` (for a quintic smoothing spline).

If the resulting smoothing spline, sp, is to be evaluated outside its basic interval, it should be replaced by `fnxtr(sp,m)` to ensure that its `m`-th derivative is zero outside that interval.

`[...] = spaps({x1,...,xr},y,tol,...)` returns the B-form of an `r`-variate tensor-product smoothing spline that is roughly within the specified tolerance to the given *gridded*

data. (For *scattered* data, use `tpaps`.) Now `y` is expected to supply the corresponding gridded values, with `size(y)` equal to `[length(x1),...,length(xr)]` in case the function is scalar-valued, and equal to `[d,length(x1),...,length(xr)]` in case the function is `d`-valued. Further, `tol` must be a cell array with `r` entries, with `tol{i}` the tolerance used during the `i`-th step when a univariate (but vector-valued) smoothing spline in the `i`-th variable is being constructed. The optional input for `m` must be an `r`-vector (with entries from the set `{1,2,3}`), and the optional input for `w` must be a cell array of length `r`, with `w{i}` either empty (to indicate that the default choice is wanted) or else a positive vector of the same length as `xi`.

Examples

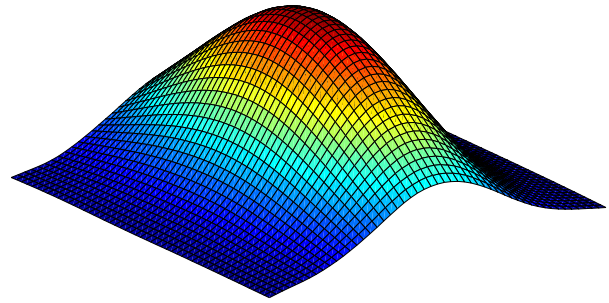
The statements

```
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.

```
x = -2:.2:2; y=-1:.25:1; [xx,yy] = ndgrid(x,y); rand('seed',39);
z = exp(-(xx.^2+yy.^2)) + (rand(size(xx))-.5)/30;
sp = spaps({x,y},z,8/(60^2));  fnplt(sp), axis off
```

produces the figure below, showing a smooth approximant to noisy data from a smooth bivariate function. Note the use of `ndgrid` here; use of `meshgrid` would have led to an error.



Algorithm

Reinsch's approach 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

`csaps`, `spap2`, `spapi`, `tpaps`

References

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

spbrk

Purpose

Part(s) of a B-form or a BBform

Syntax

```
[knots,coefs,n,k,d] = spbrk(sp)
out1 = spbrk(sp,part)
[out1,...,outo] = spbrk(sp, part1,...,parti)
spbrk(sp)
```

Description

`[knots,coefs,n,k,d] = spbrk(sp)` breaks the B-form in `sp` into its parts and returns as many of them as are specified by the output arguments.

`out1 = spbrk(sp,part)` returns the part specified by the string `part` which may be (the beginning character(s) of) one of the following strings: `'knots'` or `'t'`, `'coefs'`, `'number'`, `'order'`, `'dimension'`, `'interval'`, `'breaks'`.

If `part` is the 1-by-2 matrix `[A,B]`, the restriction/extension of the spline in `sp` to the interval with endpoints `A` and `B` is returned, in the same form.

`[out1,...,outo] = spbrk(sp, part1,...,parti)` returns in `outj` the part specified by the string `partj`, `j=1:o`, provided `o<=i`.

`spbrk(sp)` returns nothing, but prints out all the parts.

See Also

, `fnbrk`

Purpose

B-spline collocation matrix

Syntax

```
colmat = spcol(knots,k,tau)
colmat = spcol(knots,k,tau,arg1,arg2,...)
```

Description

`colmat = spcol(knots,k,tau)` returns the matrix, with `length(tau)` rows and `length(knots)-k` columns, whose (i,j)th entry is

$$D^{m(i)}B_j(\text{tau}(i))$$

This is the value at tau(i) of the $m(i)$th derivative of the jth B-spline of order `k` for the knot sequence `knots`. Here, `tau` is a sequence of sites, assumed to be *nondecreasing*, and m = knt2mlt(tau), i.e., $m(i)$ is $\#\{j < i : \text{tau}(j) = \text{tau}(i)\}$, all i.

`colmat = spcol(knots,k,tau,arg1,arg2,...)` also returns that matrix, but gives you the opportunity to specify some aspects.

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

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

Examples

To solve approximately the non-standard second-order ODE

$$D^2y(t) = 5 \cdot (y(t) - \sin(2t))$$

on the interval [0..π], using cubic splines with 10 polynomial pieces, you can use `spcol` in the following way:

```
tau = linspace(0,pi,101); k = 4;
knots = augknt(linspace(0,pi,11),k);
colmat = spcol(knots,k,brk2knt(tau,3));
```

```
coefs = (colmat(3:3:end,:)/5-colmat(1:3:end,:))\(-sin(2*tau).');
sp = spmak(knots,coefs.');
```

You can check how well this spline satisfies the ODE by computing and plotting the residual, $D^2y(t) - 5 \cdot (y(t) - \sin(2t))$, on a fine mesh:

```
t = linspace(0,pi,501);
yt = fnval(sp,t);
D2yt = fnval(fnder(sp,2),t);
plot(t,D2yt - 5*(yt-sin(2*t)))
title(['residual error; to be compared to max(abs(D^2y)) = ',...
         num2str(max(abs(D2yt)))])
```

The statement `spcol([1:6],3,.1+[2:4])` provides the matrix

```
ans =

    0.5900   0.0050        0
    0.4050   0.5900   0.0050
         0   0.4050   0.5900
```

in which the typical row records the values at 2.1, or 3.1, or 4.1, of all B-splines of order 3 for the knot sequence `1:6`. There are three such B-splines. The first one has knots 1,2,3,4, and its values are recorded in the first column. In particular, the last entry in the first column is zero since it gives the value of that B-spline at 4.1, a site to the right of its last knot.

If you add the string `'sl'` as an additional input to `spcol`, then you can ask `bkbrk` to extract detailed information about the block structure of the matrix encoded in the resulting output from `spcol`. Thus, the statement `bkbrk(spcol(1:6,3,.1+2:4,'sl'))` gives:

```
block 1 has 2 row(s)
    0.5900   0.0050        0
    0.4050   0.5900   0.0050
next block is shifted over 1 column(s)
block 2 has 1 row(s)
```

```
    0.4050   0.5900   0.0050
next block is shifted over 2 column(s)
```

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 equal to $\max(m)$, values of all B-splines of that order are generated by recurrence and used to compute the jth derivative at those sites of all B-splines of order `k`.

The resulting rows of B-spline values (each row corresponding to a particular `tau(i)`) are then assembled into the overall (usually rather sparse) matrix.

When the optional argument `'sl'` is present, these rows are instead assembled into a convenient almost block-diagonal form that takes advantage of the fact that, at any site `tau(i)`, at most `k` B-splines of order `k` are nonzero. This fact (together with the natural ordering of the B-splines) implies that the collocation matrix is almost block-diagonal, i.e., has a staircase shape, with the individual blocks or steps of varying height but of uniform width `k`.

The command `slvblk` is designed to take advantage of this storage-saving form available when used, in `spap2`, `spapi`, or `spaps`, to help determine the B-form of a piecewise-polynomial function from interpolation or other approximation conditions.

See Also

`slvblk`, `spap2`, `spapi`

Limitations

The sequence `tau` is assumed to be nondecreasing.

Purpose

Spline curve by uniform subdivision

Syntax

```
spcrv(c,k)
spcrv(c)
spcrv(c,k,maxpnt)
```

Description

`spcrv(c,k)` provides a dense sequence *f*(tt) of points on the uniform B-spline curve *f* of order `k` with B-spline coefficients `c`. Explicitly, this is the curve

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

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]` equals `size(c)`.

`spcrv(c)` chooses the order `k` to be 4.

`spcrv(c,k,maxpnt)` makes sure that at least `maxpnt` points are generated. The default value for the maximum number of sites `tt` to be generated is `100`.

The parameter interval that the site sequence `tt` fills out uniformly is the interval [`k`/2 .. (`n`-`k`/2)].

The output consists of the array *f*(tt).

Examples

The following would show a questionable broken line and its smoothed version:

```
points = [0 0 1 1 0 -1 -1 0 0 ;
          0 0 0 1 2 1 0 -1 -2];
plot(points(1,:),points(2,:),':')
values = spcrv(points,3);
hold on, plot(values(1,:),values(2,:)), hold off
```

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

`fnplt`

splinetool

Purpose

Experiment with some spline approximation methods

Syntax

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

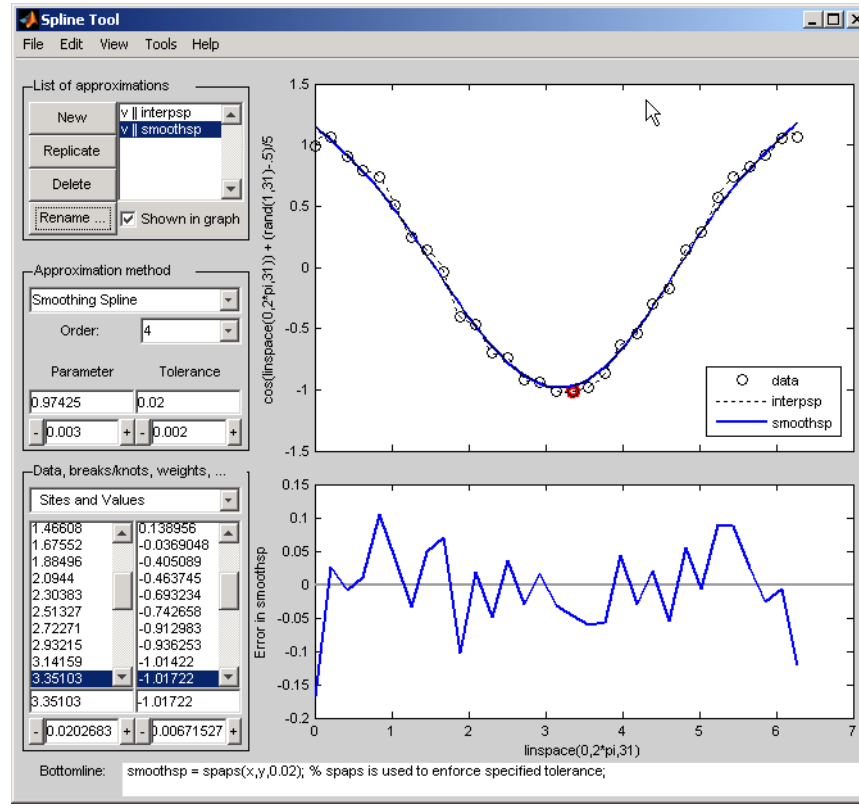
Description

`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 in the following figure comparing cubic spline interpolation with a smoothing spline on sample data created by adding noise to the cosine function.


Spline Tool
File Edit View Tools Help
List of approximations
New
Replicate
Delete
Rename ...
interpsp
smoothsp
Shown in graph
Approximation method
Smoothing Spline
Order:
4
Parameter
Tolerance
0.97425
0.02
0.003
0.002
Data, breaks/knots, weights, ...
Sites and Values
1.46608 0.138956
1.67552 -0.0369048
1.88496 -0.405089
2.0944 -0.463745
2.30383 -0.693234
2.51327 -0.742658
2.72271 -0.912983
2.93215 -0.936253
3.14159 -1.01422
3.35103 -1.01722
0.0202683
0.00671527
cos(linspace(0,2*pi,31)) + (rand(1,31)-.5)/5
data
interpsp
smoothsp
Error in smoothsp
linspace(0,2*pi,31)
Bottomline:
smoothsp = spaps(x,y,0.02); % spaps is used to enforce specified tolerance;

Approximation Methods

The approximation methods and options supported by the GUI are shown below.

Approximation Method	Option
Cubic Interpolating Spline	Adjust the type and values of the end conditions.
Smoothing Spline	Choose between cubic (order 4) and quintic (order 6) splines. Adjust the value of the tolerance and/or smoothing parameter. Adjust the weights in the error and roughness measures.
Least-Squares Approximation	Vary the order from 1 to 14. The default order is 4, which gives cubic approximating splines. Modify the number of polynomial pieces. Add and move knots to improve the fit. Adjust the weights in the error measure.
Spline Interpolation	Vary the order from 2 to 14. The default order is 4, which gives cubic spline interpolants. If the default knots supplied are not satisfactory, you can move them around to vary the fit.

Graphs

You can generate and compare several approximations to the same data. One of the approximations is always marked as "current" using a thicker line width. The following displays are available:

- Data graph. It shows:
  - The data
  - The approximations chosen for display in **List of approximations**
  - The current knot sequence or the current break sequence

- Auxiliary graph (if viewed) for the current approximation. You can invoke this graph by selecting any one of the items in the **View** menu. It shows one of the following:
  - The first derivative
  - The second derivative
  - The error

By default, the error is the difference between the given data values and the value of the approximation at the data sites. In particular, the error is zero (up to round-off) when the approximation is an interpolant. However, if you provide the data values by specifying a function, then the error displayed is the difference between that function and the current approximation. This also happens if you change the y-label of the data graph to the name of a function.

Menu Options

You can annotate and print the graphs with the **File > Print to Figure** menu.

You can export the data and approximations to the workspace for further use or analysis with the **File > Export Data** and **File > Export Spline** menus, respectively.

You can create, with the **File > Generate M-file** menu, a function M-file that you can use to generate, from the original data, any or all graphs currently shown. This M-file also provides you with a written record of the Spline Toolbox commands used to generate the current graph(s).

You can save, with the **Replicate** button, the current approximation before you experiment further. If, at a later time, you click on the approximation so saved, `splinetool` restores everything to the way it was, including the data used in the construction of the saved approximation. This is true even if, since saving this approximation, you have edited the data while working on other approximations.

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

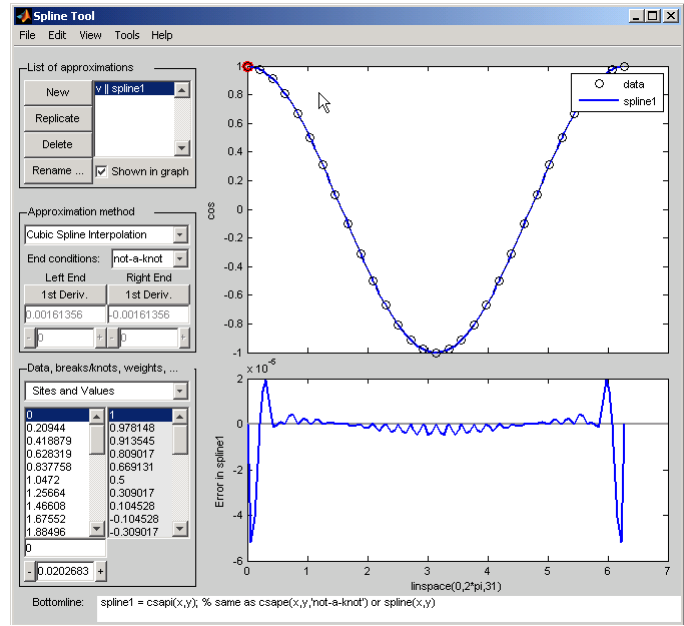
You can toggle the grid or the legend in the graph(s) with the **Tools** menu.

Examples

Exploring End Conditions For Cubic Spline Interpolation

The purpose of this example is to explore the various end conditions available with cubic spline interpolation:

1. Type `splinetool` at the command line.
2. Select **Import your own data** from the initial screen, and accept the default function. You should see the following display.



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

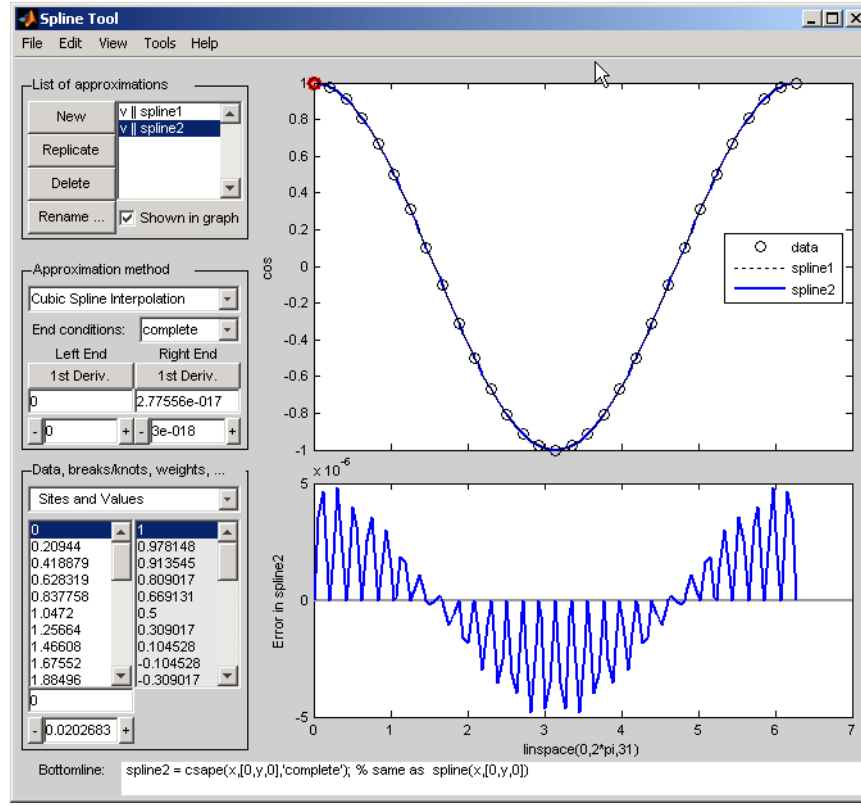
The vector `x` of data sites is `linspace(0,2*pi,31)` and the values are `cos(x)`. This differs from simply providing the vector `y` of values in that the cosine function is explicitly recorded as the underlying function. Therefore, the error shown in the graph is the error in the spline as an approximation to the cosine rather than as an approximation to the given values. Notice the resulting relatively large error, about `5e-5`, near the endpoints.

3 For comparison, follow these steps:

- Click on **New** in the **List of approximations**.
- In **Approximation method**, select **complete** from the list of **End conditions**.

- Since the first derivative of the cosine function is sine, adjust the first-derivative values to their known values of zero at both the left end and the right end.

This procedure results in the display shown below (after the mouse is used to move the Legend further down). Note that the right end slope is zero only up to round-off. **Bottomline** tells you that the toolbox function `csape` was used to create the spline.



Be impressed by the improvement in the error, which is only about `5e-6`.

4 For further comparison, follow these steps:

- Click on **New** in the **List of approximations**.
- In **Approximation method**, select **natural** from the list of **End conditions**.

Note the deterioration of the approximation near the ends, an error of about `2e-3`, which is much worse than with the not-a-knot end conditions.

5 For a final comparison, follow these steps:

- Click on **New** in the **List of approximations**.
- Since we know that the cosine function is periodic, in **Approximation method**, select **periodic** from the list of **End conditions**.

Note the dramatic improvement in the approximation, back to an error of about `5e-6`, particularly compared to the **natural** end conditions.

Estimating the Second Derivative at an Endpoint

This example uses cubic spline interpolation and least-squares approximation to determine an estimate of the initial acceleration for a drag car:

1 Type `splinetool` at the command line or if the GUI is already running, click on **File > Restart**.

2 Choose **Richard Tapia's drag racing data**. These data show the distance traveled by a drag car as a function of time. The message window asks you to estimate the initial acceleration by setting the initial speed to zero. Click on **OK,** or use **Space** or **Enter,** to remove the message window.

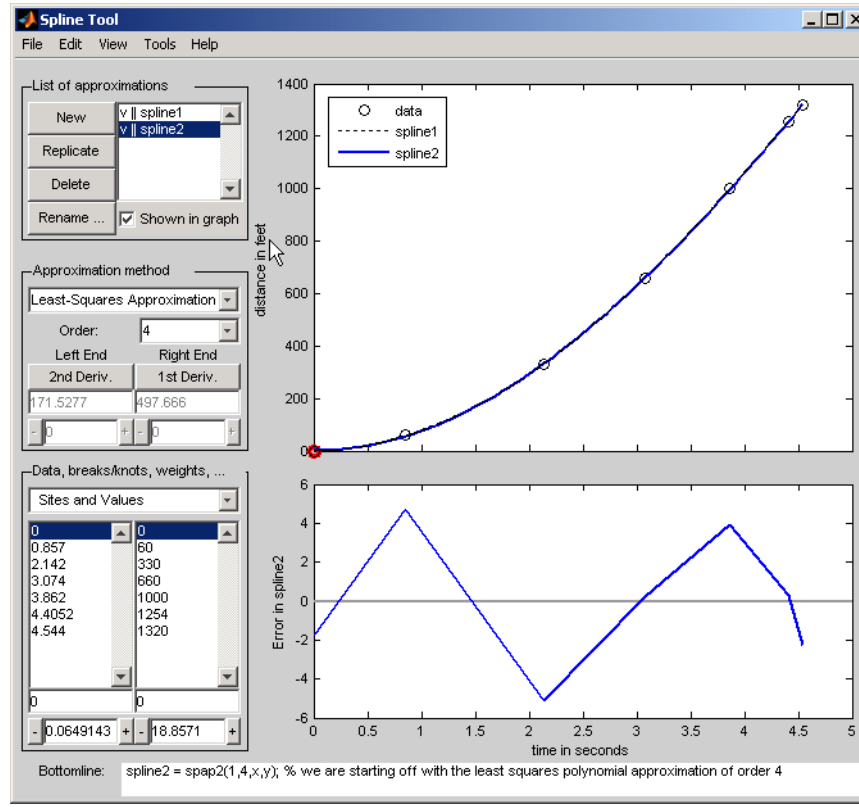
3 In **Approximation method**, select **complete** from the list of **End conditions**.

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

5. Look for the value of the initial acceleration, which is given by the value of the second derivative at the left endpoint. You can toggle between the first derivative and the second derivative at this endpoint by clicking on the **left end** button. The value of the second derivative should be around `187` in the units chosen. Choose **View > Show 2nd Derivative** to see this graphically.

6. For comparison, click on **New**, then choose **Least-Squares Approximation** as the **Approximation method**. With this method, you can no longer specify end conditions. Instead, you may vary the order of the method. Verify that the initial acceleration is close to the cubic interpolation value.

   The results of this procedure are shown below.



Least-Squares Approximation

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

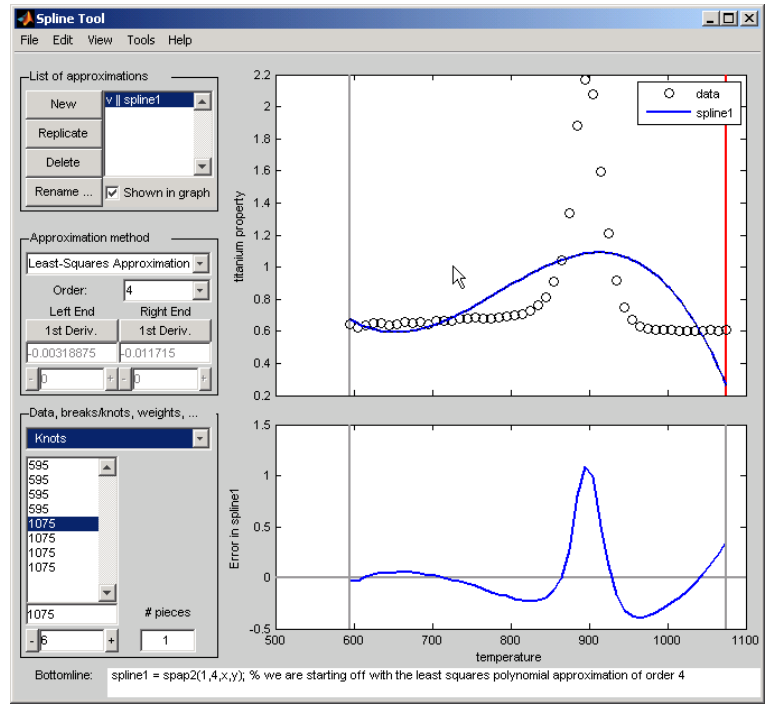
1. Type `splinetool` at the command line or if the GUI is already running, click on **File > Restart**.

2. Choose **Titanium heat data**.

3 Select **Least-Squares Approximation** as the **Approximation method.**

4 Notice how poorly this approximates the data since there are no interior knots. To view the current knots and add new knots, choose **knots** from **Data, breaks/knots, weights**. The knots are now listed in **knots**, and also displayed in the data graph as vertical lines. Notice that there are just the two end knots, each with multiplicity 4.

5 Right-click in the data graph and choose **Add Knot**. This brings up crosshairs for you to move with the mouse. Its precise horizontal location is shown in the edit field below the list of knots. A mouse click places a new knot at the current location of the crosshairs. One possible strategy is to place the additional knot at the place of maximum absolute error, as shown in the auxiliary graph below the data graph.



If you right-click and choose **Replicate Knot**, you will increase the multiplicity of the current knot, which is shown by its repeated occurrence in **Knots**. If you don't like a particular knot, you can delete it. To delete a specific knot, you must first select it in either the list of knots or the data graph, and then right-click in the graph and choose **Delete Knot**.

6 You could also ask for an approximation using six polynomial pieces, which corresponds to five interior knots. To do this, enter `6` as **# pieces** in **Data, breaks/knots, weights**.

7 After you have the five interior knots, try to make the error even smaller by moving the knots. To do this, select the knot you want to move by clicking on its vertical line in the graph, then use the

interface control below **Knots** in **Data, breaks/knots, weights** and observe how the error changes with the movement of the knot. You can also use the edit field to overwrite the current knot location. You could also try **adjust**, which redistributes the current knot sequence.

8 Use **Replicate** in **List of approximations** to save any good knot distribution for later use. Rename the replicated approximation to `lstsqr` using **Rename**. To return to the original approximation, click on its name in **List of approximations**.

Smoothing Spline

This example experiments with smoothing splines:

1 Type `splinetool` at the command line or, if the GUI is already running, click on **File > Restart**.

2 Choose **Titanium heat data**.

3 In **Approximation method**, choose **Smoothing Spline**.

4 Vary **Parameter** between 0 and 1, which changes the approximation from the least-squares straight-line approximation to the "natural" cubic spline interpolant.

5 Vary **Tolerance** between 0 and some large value, even `inf`. The approximation changes from the best possible one, the "natural" cubic spline interpolant, to the least-squares straight-line approximation.

6 As you increase the **Parameter** value or decrease the **Tolerance** value, the error decreases. However, a smaller error corresponds to more roughness, as measured by the size of the second derivative. To see this, choose **View > Show 2nd Derivative** and vary the **Parameter** and **Tolerance** values once again.

7 This step modifies the weights in the error measure to force the approximation to pass through a particular data point.

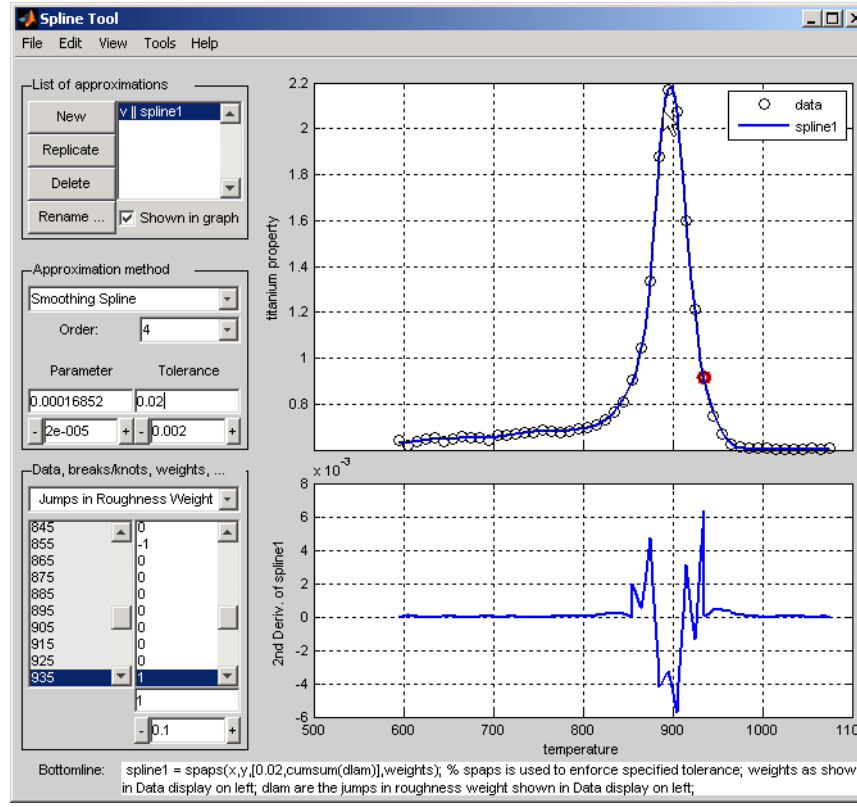
- Set **Tolerance** to `0.2`. Notice that the approximation does not pass through the highest data point. To see the large error at this site, choose **View > Error**.
- To force the smoothing spline to go through this point, choose **Error Weights** from **Data, breaks/knots, weights**.
- Click on the highest data point in the graph and notice its site, which is indicated in **Sites and Values**.
- Use the edit field beneath the list of weights to change the current weight to `1000`. Notice how much closer the smoothing spline now comes to that highest data point, and the decrease in the error at that site. Turn on the grid, by **Tools > Grid**, to locate the error at that site more readily.

8 This step modifies the weights in the roughness measure to permit a more accurate but less smooth approximation in the peak area while insisting on a smoother, hence less accurate, approximation away from the peak area.

- Choose **Jumps in Roughness Weight** from **Data, breaks/knots, weights**.
- Choose **View > Show 2nd Derivative**
- Select any data point to the left of the peak in the data.
- Set the jump at the selected site to `-1` by changing its value in the edit field below it. Since the roughness weight for the very first site interval is 1, you have just set the roughness weight to the right of the highlighted site to 0. Correspondingly, the second derivative has become relatively small to the left of that site.
- Select any data point to the right of the peak in the data.
- Set the jump across the selected site to `1`. Since the roughness weight just to the left of the highlighted site is 0, you have just set the roughness weight to the right of the highlighted site to 1. Correspondingly, the second derivative has become relatively small to the right of that site. The total effect is a very smooth but not very accurate fit away from the peak, while in the peak area,

the spline fit is much better but the second derivative is much larger, as is shown in the auxiliary graph below.

At the sites where there is a jump in the roughness weight, there is a corresponding jump in the second derivative. If you increase the **Parameter** value, the error across the peak area decreases but the second derivative remains quite large, while the opposite holds true away from the peak area.



See Also

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

Purpose

Taylor coefficients from local B-coefficients

Syntax

```
[v,b] = splpp(tx,a)
[v,b] = sprpp(tx,a)
```

Description

These are utility commands of use in the conversion from B-form to ppform (and in certain evaluations), but of no interest to the casual user.

`[v,b] = splpp(tx,a)` provides the matrices `v` and b, both of the same size `[r,k]` as `a`, and related to the input in the following way.

For `i=1:r`, `b(i,:)` are the B-coefficients, with respect to the knot sequence `[tx(i,1:k-1),0,...,0]`, of the polynomial of order `k` on the interval `[tx(i,k-1) .. tx(i,k)]` whose `k` B-spline coefficients, with respect to the knot sequence `tx(i,:)`, are in `a(i,:)`. This is done by repeated knot insertion (of the knot 0). It is assumed that `tx(i,k-1)<0<=tx(i,k)`.

For `i=1:r`, `v(i,:)` are the polynomial coefficients for that polynomial, i.e., `v(i,j)` is the number $D^{k-j}s(0–)/k-j)!$, `j=1:k`, with s having the knots `tx(i,:)` and the B-coefficients `a(i,:)`.

`[v,b] = sprpp(tx,a)` carries out exactly the same job, except that now `b(i,:)` are the B-coefficients for that polynomial with respect to the knot sequence `[0,...,0,tx(i,k: 2*(k-1))]`, and, correspondingly, `v(i,j)` is $D^{k-j}s(0 +)/k-j)!$, `j=1:k`. Also, now it is assumed that `tx(i,k-1)<=0<tx(i,k)`.

Examples

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

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

with s the B-spline with knots -2, -1, 0, 1. This is so because the `l` in `splpp` indicates the limit from the left, and the second argument, `[0 1 0]`, indicates the spline s in question to be

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

i.e., this particular linear combination of the third-order B-splines for the knot sequence ..., -2, -1,0,1,... (Note that the values calculated do not depend on the knots marked ?.) The above statement also provides the sequence `b = 0 1.0000 0.5000` of B-spline coefficients for the polynomial piece of *s* on the interval [-1. .0], and with respect to the knot sequence ?, -2, -1, 0, 0, ?.

In other words, on the interval [-1. .0], the B-spline with knots 2, -1, 0, 1 can be written

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

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

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

with *s* the B-spline with knots ?,-1,0,1. Its polynomial piece on the interval [0..1] is independent of the choice of ?, so we might as well think of ? as -2, i.e., we are dealing with the same B-spline as before. Note that the last two numbers agree with the limits from the left computed above, while the first number does not. This reflects the fact that a quadratic B-spline with simple knots is continuous with continuous first, but discontinuous second, derivative. (It also reflects the fact that the leftmost knot of a B-spline is irrelevant for its right-most polynomial piece.) The sequence `b = 0.5000 0 0` also provided states that, on the interval [0. .1], the B-spline $B(\cdot\,|\,[?,-1,0,1])$ can be written

$$0.5 \times B(\cdot\,|\,[0,0,0,1]) + 0 \times B(\cdot\,|\,[0,0,1,2]) + 0 \times B(\cdot\,|\,[0,1,2,?])$$

Purpose

Put together spline in B-form

Syntax

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

Description

The command `spmak(...)` puts together a spline function in B-form, from minimal information, with the rest inferred from the input. `fnbrk` returns all the parts of the completed description. In this way, the actual data structure used for the storage of this form is easily modified without any effect on the various `fn...` commands that use this construct.

`spmak(knots,coefs)` returns the B-form of the spline specified by the knot information in `knots` and the coefficient information in `coefs`.

The action taken by `spmak` depends on whether the function is univariate or multivariate, as indicated by `knots` being a sequence or a cell array. For the description, let `sizec` be `size(coefs)`.

If `knots` is a sequence (required to be non-decreasing), then the spline is taken to be univariate, and its order `k` is taken to be `length(knots)-sizec(end)`. This means that each 'column' `coefs(:,j)` of `coefs` is taken to be a B-spline coefficient of the spline, hence the spline is taken to be `sizec(1:end-1)`-valued. The basic interval of the B-form is [`knots(1)` .. `knots(end)`].

Knot multiplicity is held to be $\leq$ `k`. This means that the coefficient `coefs(:,j)` is simply ignored in case the corresponding B-spline has only one distinct knot, i.e., in case `knots(j)` equals `knots(j+k)`.

If `knots` is a cell array, of length `m`, then the spline is taken to be `m`-variate, and `coefs` must be an (`r`+`m`)-dimensional array, – except when the spline is to be scalar-valued, in which case, in contrast to the univariate case, `coefs` is permitted to be an `m`-dimensional array, but `sizec` is reset by

```
sizec = [1, sizec]; r = 1;
```

The spline is `sizec(1:r)`-valued. This means the output of the spline is an array with `r` dimensions, e.g., if `sizec(1:2) = [2, 3]` then the output of the spline is a 2-by-3 matrix.

The spline is `sizec(1:r)`-valued, the `i`th entry of the `m`-vector `k` is computed as `length(knots{i}) - sizec(r+i)`, `i=1:m`, and the `i`th entry of the cell array of basic intervals is set to `[knots{i}(1), knots{i}(end)]`.

`spmak(knots,coefs,sizec)` lets you supply the intended size of the array `coefs`. Assuming that `coefs` is correctly sized, this is of concern only in the rare case that `coefs` has one or more trailing singleton dimensions. For, MATLAB suppresses trailing singleton dimensions, hence, without this explicit specification of the intended size of `coefs`, `spmak` would interpret `coefs` incorrectly.

`spmak` prompts you for `knots` and `coefs`.

`sp = spmak(knots,coeffs)` returns the spline `sp`.

Examples

`spmak(1:6,0:2)` constructs a spline function with basic interval [1. .6], with 6 knots and 3 coefficients, hence of order 6 - 3 = 3.

`spmak(t,1)` provides the B-spline $B(\cdot|t)$ in B-form.

The coefficients may be `d`-vectors (e.g., 2-vectors or 3-vectors), in which case the resulting spline is a curve or surface (in R^2 or R^3).

If the intent is to construct a 2-vector-valued bivariate polynomial on the rectangle [–1..1] × [0..1], linear in the first variable and constant in the second, say

```
coefs = zeros([2 2 1]); coefs(:,:,1) = [1 0;0 1];
```

then the straightforward

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

will result in the error message `'There should be no more knots than coefficients'`, because the trailing singleton dimension of

`coefs` will not be perceived by `spmak`, while proper use of that third argument, as in

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

will succeed. Replacing here `[2 2 1]` by `size(coefs)` would not work.

See the demo "Intro to B-form" for other examples.

See Also

`spbrk`

Diagnostics

There will be an error return if the proposed knot sequence fails to be nondecreasing, or if the coefficient array is empty, or if there are not more knots than there are coefficients. If the spline is to be multivariate, then this last diagnostic may be due to trailing singleton dimensions in `coefs`.

spterms

Purpose

Explanation of Spline Toolbox terms

Syntax

```
spterms(term)
expl = spterms(term)
[...,term] = spterms(...)
```

Description

`spterms(term)` provides, in a message box, an explanation of the technical term indicated by the string `term` as used in the Spline Toolbox product and, specifically, in the GUI `splinetool`. Only the first few (but at least two) letters of the desired term need to be specified, and the full term is shown in the title of the message box.

`expl = spterms(term)` returns, in `expl`, the string, or cell array of strings, comprising the explanation of the desired term.

`[...,term] = spterms(...)` also returns, in `term`, the fully spelled-out term actually used.

Examples

`spterms('sp')` gives an explanation of the term ‘spline’, while `spterms('spline i')` explains the terms ‘spline interpolation’.

`help spterms` provides the list of all available terms.

See Also

`splinetool`, “List of Terms” on page A-3 in the Spline Toolbox documentation.

Purpose

Scattered translates collocation matrix

Syntax

```
colmat = stcol(centers,x,type)
colmat = stcol(...,'tr')
```

Description

`colmat = stcol(centers,x,type)` is the matrix whose (`i`,`j`)th entry is

$$\psi_j\left(x(:,i)\right), \quad i = 1:\mathrm{size}(x,2), j = 1:n$$

with the bivariate functions ψ_j and the number `n` depending on the `centers` and the string `type`, as detailed in the description of `stmak`.

`centers` and `x` must be matrices with the same number of rows.

The default for `type` is the string `'tp'`, and for this default, `n` equals `size(centers,2)`, and the functions ψ_j are given by

$$\psi_j(x) = \psi\left(x - \mathrm{centers}(:,j)\right), \quad j = 1:n$$

with ψ the thin-plate spline basis function

$$\psi(x) = |x|^2 \log|x|^2$$

and with $|x|$ denoting the Euclidean norm of the vector x.

Note See `stmak` for a description of other possible values for `type`.

The matrix `colmat` is the coefficient matrix in the linear system

$$\sum_j a_j \psi_j(x(:,i)) = y_i, \quad i = 1:\mathrm{size}(x,2)$$

that the coefficients a_j of the function $f = \Sigma_j a_j \psi_j$ must satisfy in order that f interpolate the value y_i at the site `x(:,i)`, all `i`.

`colmat = stcol(...,'tr')` returns the transpose of the matrix returned by `stcol(...)`.

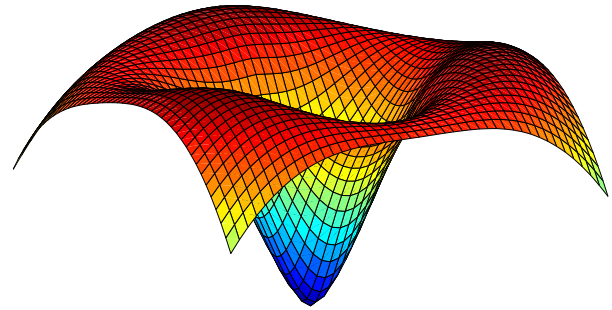
Examples

Example 1. The following evaluates and plots the function

$$f(x) = \psi(x - c_1) + \psi(x - c_2) + \psi(x - c_3) - 3.5\psi(x)$$

on a regular mesh, with ψ the above thin-plate basis function, and with c_1, c_2, c_3 three points on the unit circle; see the figure below.

```
a = [0,2/3*pi,4/3*pi]; centers = [cos(a), 0; sin(a), 0];
[xx,yy] = ndgrid(linspace(-2,2,45));
xy = [xx(:) yy(:)].';
coefs = [1 1 1 -3.5];
zz = reshape( coefs*stcol(centers,xy,'tr') , size(xx));
surf(xx,yy,zz), view([240,15]), axis off
```



Example 2. The following also evaluates, on the same mesh, and plots the length of the gradient of the function in Example 1.

```
zz = reshape( sqrt(...
        ([coefs,0]*stcol(centers,xy,'tp10','tr')).^2 + ...
        ([coefs,0]*stcol(centers,xy,'tr','tp01')).^2),
size(xx));
figure, surf(xx,yy,zz), view([220,-15]), axis off
```

See Also

spcol, stmak

Purpose

Put together function in stform

Syntax

```
stmak(centers,coefs)
st = stmak(centers,x,type)
st = stmak(centers,coefs,type,interv)
```

Description

`stmak(centers,coefs)` returns the stform of the function *f* given by

$$f(x) = \sum_{j=1}^{n} \text{coefs}(:,j) \cdot \psi(x - \text{centers}(:,j))$$

with

$$\psi(x) = |x|^2 \log|x|^2$$

the thin-plate spline basis function, and with $|x|$ denoting the Euclidean norm of the vector *x*.

`centers` and `coefs` must be matrices with the same number of columns.

`st = stmak(centers,x,type)` stores in `st` the stform of the function *f* given by

$$f(x) = \sum_{j=1}^{n} \text{coefs}(:,j) \cdot \psi_j(x)$$

with the ψ_j as indicated by the string `type`, which can be one of the following:

- `'tp00'`, for the thin-plate spline;
- `'tp10'`, for the first derivative of a thin-plate spline wrto its first argument;
- `'tp01'`, for the first derivative of a thin-plate spline wrto its second argument;
- `'tp'`, the default.

Here are the details.

`'tp00'`	$\psi_j(x) = \varphi(\|x - c_j\|^2)$, c_j =`centers(:,j), j=1:n-3` with $\varphi(t) = t\log(t)$ $\psi_{n-2}(x) = x(1)$ $\psi_{n-1}(x) = x(2)$ $\psi_n(x) = 1$
`'tp10'`	$\psi_j(x) = \varphi(\|x - c_j\|^2)$, c_j =`centers(:,j), j=1:n-1` with $\varphi(t) = (D_1 t)(\log t + 1)$, and $D_1 t$ the partial derivative of $t = t(x) = \|x - c_j\|^2$ wrto $x(1)$ $\psi_n(x) = 1$
`'tp01'`	$\psi_j(x) = \varphi(\|x - c_j\|^2)$, c_j =`centers(:,j), j=1:n-1` with $\varphi(t) = (D_2 t)(\log t + 1)$, and $D_2 t$ the partial derivative of $t = t(x) = \|x - c_j\|^2$ wrto $x(2)$ $\psi_n(x) = 1$
`'tp'` (default)	$\psi_j(x) = \varphi(\|x - c_j\|^2)$, c_j =`centers(:,j), j=1:n` with $\varphi(t) = t\log(t)$

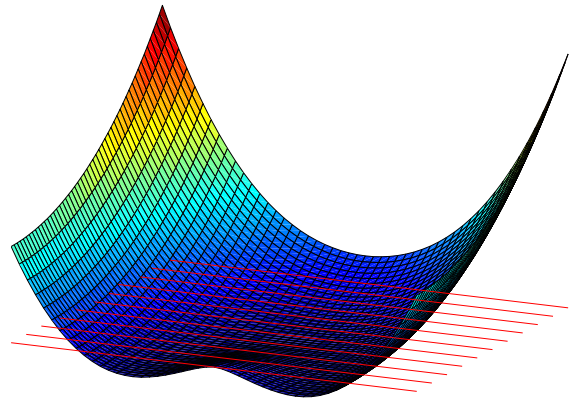
`st = stmak(centers,coefs,type,interv)` also specifies the basic interval for the stform, with `interv{j}` specifying, in the form `[a,b]`, the range of the `j`th variable. The default for `interv` is the smallest such box that contains all the given centers.

Examples

Example 1. The following generates the figure below, of the thin-plate spline basis function, $\psi(x) = |x|^2 \log|x|^2$, but suitably restricted to show that this function is negative near the origin. For this, the extra lines are there to indicate the zero level.

```
inx = [-1.5 1.5]; iny = [0 1.2];
fnplt(stmak([0;0],1),{inx,iny})
hold on, plot(inx,repmat(linspace(iny(1),iny(2),11),2,1),'r')
```

```
view([25,20]),axis off, hold off
```

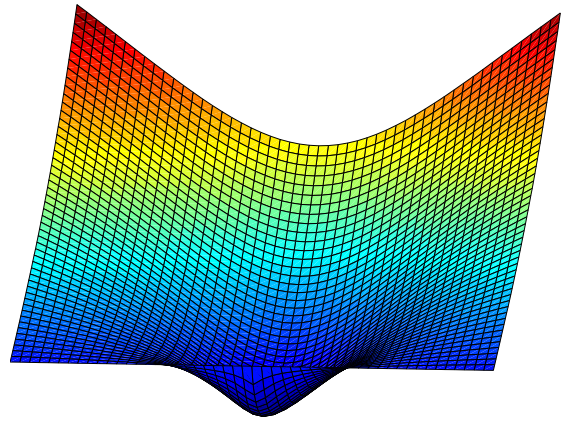


Example 2. We now also generate and plot, on the very same domain, the first partial derivative $D_2\psi$ of the thin-plate spline basis function, with respect to its second argument.

```
inx = [-1.5 1.5]; iny = [0 1.2];
fnplt(stmak([0;0],[1 0],'tp01',{inx,iny}))
view([13,10]),shading flat,axis off
```

Note that, this time, we have explicitly set the basic interval for the stform.

The resulting figure, below, shows a very strong variation near the origin. This reflects the fact that the *second* derivatives of ψ have a logarithmic singularity there.



See Also

stcol

subplus

Purpose

Positive part

Syntax

```
xp = subplus(x)
```

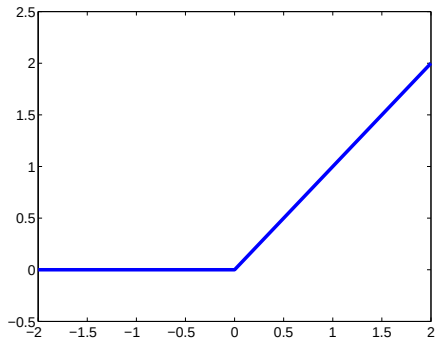
Description

`xp = subplus(x)` returns $(x)_+$, i.e., the positive part of `x`, which is `x` if `x` is nonnegative and `0` if `x` is negative. In other words, `xp` equals `max(x,0)`. If `x` is an array, this operation is applied entry by entry.

Examples

Example 1. Here is a plot of the essential part of the subplus function, as generated by

```
x = -2:2; plot(x,subplus(x),'linew',4), axis([-2,2,-.5,2.5])
```

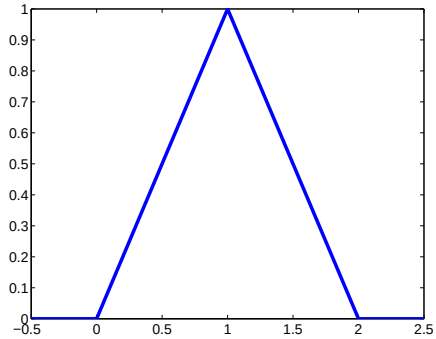


Example 2. The following anonymous function describes the so-called hat function:

```
hat = @(x) subplus(x) - 2*subplus(x-1) + subplus(x-2);
```

i.e., the spline also given by `spmak(0:2,1)`, as the following plot shows.

```
x = -.5:.5:2.5; plot(x,hat(x),'linew',4), set(gca,'Fontsize',16)
```


1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
−0.5
0
0.5
1
1.5
2
2.5

titanium

Purpose

Titanium test data

Syntax

```
[x,y] = titanium
```

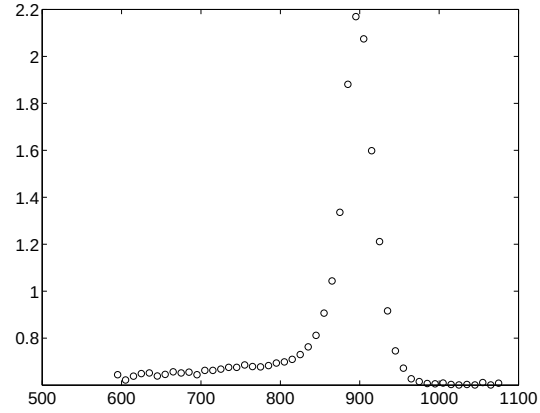
Description

`[x,y] = titanium` returns measurements of a certain property of titanium as a function of temperature. Since their use in , these data have become a standard test for data fitting since they are hard to fit by classical techniques and have a significant amount of noise.

Examples

The plot of the data shown below is generated by the following commands:

```
[x,y] = titanium; plot(x,y,'ok'), set(gca,'Fontsize',16)
```



References

C. de Boor and J. R. Rice, Least squares cubic spline approximation II - Variable knots, CSD TR 21, Comp.Sci., Purdue Univ., April 1968.

Purpose

Thin-plate smoothing spline

Syntax

```
tpaps(x,y)
tpaps(x,y,p)
[...,p] = tpaps(...)
```

Description

`tpaps(x,y)` is the stform of a thin-plate smoothing spline *f* for the given data sites `x(:,j)` and the given data values `y(:,j)`. The `x(:,j)` must be distinct points in the plane, the values can be scalars, vectors, matrices, even ND-arrays, and there must be exactly as many values as there are sites.

The thin-plate smoothing spline *f* is the unique minimizer of the weighted sum

$$pE(f) + (1-p)R(f)$$

with $E(f)$ the error measure

$$E(f) = \sum_j \left| y(:,j) - f\left(x(:,j)\right)\right|^2$$

and $R(f)$ the roughness measure

$$R(f) = \int (\left|D_1 D_1 f\right|^2 + 2\left|D_1 D_2 f\right|^2 + \left|D_2 D_2 f\right|^2)$$

Here, the integral is taken over all of R^2, $|z|^2$ denotes the sum of squares of all the entries of z, and $D_i f$ denotes the partial derivative of f with respect to its ith argument, hence the integrand involves second partial derivatives of f. The smoothing parameter `p` is chosen so that `(1-p)/p` equals the average of the diagonal entries of the matrix `A`, with `A + (1-p)/p*eye(n)` the coefficient matrix of the linear system for the `n` coefficients of the smoothing spline to be determined. This choice of `p` is meant to ensure that we are in between the two extremes, of interpolation (when `p` is close to `1` and the coefficient matrix is essentially `A`) and complete smoothing (when `p` is close to `0` and the

coefficient matrix is essentially a multiple of the identity matrix). This should serve as a good first guess for `p`.

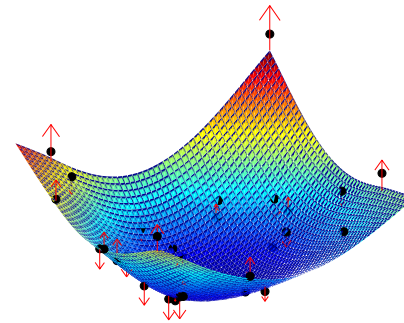
`tpaps(x,y,p)` also inputs the *smoothing parameter*, `p`, a number between 0 and 1. As the smoothing parameter varies from 0 to 1, the smoothing spline varies, from the least-squares approximation to the data by a linear polynomial when `p` is `0`, to the thin-plate spline interpolant to the data when `p` is `1`.

`[...,p] = tpaps(...)` also returns the smoothing parameter actually used.

Examples

Example 1. The following code obtains values of a smooth function at 31 randomly chosen sites, adds some random noise to these values, and then uses `tpaps` to recover the underlying exact smooth values. To illustrate how well `tpaps` does in this case, the code plots, in addition to the smoothing spline, the exact values (as black balls) as well as each arrow leading from a smoothed value to the corresponding noisy value.

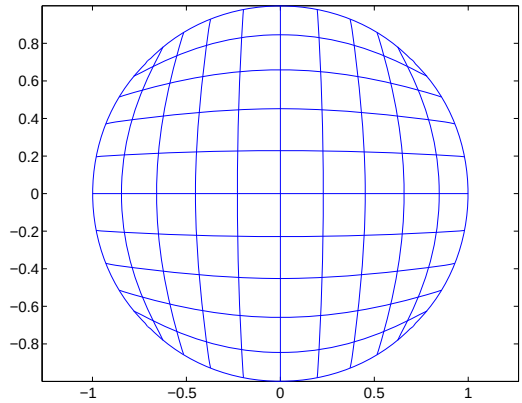
```
rand('seed',23); nxy = 31;
xy = 2*(rand(2,nxy)-.5); vals = sum(xy.^2);
noisyvals = vals + (rand(size(vals))-.5)/5;
st = tpaps(xy,noisyvals); fnplt(st), hold on
avals = fnval(st,xy);
plot3(xy(1,:),xy(2,:),vals,'wo','markerfacecolor','k')
quiver3(xy(1,:),xy(2,:),avals,zeros(1,nxy),zeros(1,nxy), ...
         noisyvals-avals,'r'), hold off
```



Example 2. The following code uses an interpolating thin-plate spline to vector-valued data values to construct a map, from the plane to the plane, that carries the unit square $\{x : |x(j)| \le 1, j = 1{:}2\}$ approximately onto the unit disk $\{x : x(1)^2 + x(2)^2 \le 1\}$, as shown by the picture generated.

```
n = 64; t = linspace(0,2*pi,n+1); t(end) = [];
values = [cos(t); sin(t)];
centers = values./repmat(max(abs(values)),2,1);
st = tpaps(centers, values, 1);
fnplt(st), axis equal
```

Note the choice of `1` for the smoothing parameter here, to obtain interpolation.



Limitations

The determination of the smoothing spline involves the solution of a linear system with as many unknowns as there are data points. Since the matrix of this linear system is full, the solving can take a long time even if, as is the case here, an iterative scheme is used when there are more than 728 data points. The convergence speed of that iteration is strongly influenced by `p`, and is slower the larger `p` is. So, for large problems, use interpolation, i.e., `p` equal to 1, only if you can afford the time.

See Also

`csaps`, `spaps`

A

Glossary

The Glossary consists of these sections:

Introduction

This glossary provides brief definitions of the basic mathematical terms and notation 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 because 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, you 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

Because MATLAB uses the notation `[a,b]` to indicate a matrix with the two columns, `a` and `b`, this guide uses the notation [a .. b] to indicate the closed interval with endpoints `a` and `b`. This guide does 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 $\Re^d$. 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. In the Spline Toolbox, vectors are column vectors.

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 $\Re^d$, 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 $\Re$, that, for $i = 1,...,d$, associates to i the real number $x(i)$.

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

There are scalar-valued, vector-valued, matrix-valued, and *ND*-valued splines. Scalar-valued functions have the real numbers $\Re$ (or, more generally, the complex numbers) as their target, while d-vector-valued functions have $\Re^d$ as their target; if, more generally, $\boldsymbol{d}$ is a vector of positive integers, then $\boldsymbol{d}$-valued functions have the $\boldsymbol{d}$-dimensional real arrays as their target. Spline Toolbox can deal with univariate and multivariate functions. The former have some real interval, or, perhaps, all of $\Re$ as their domain, while m-variate functions have some subset, or perhaps all, of $\Re^m$ 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 $g_1(x_1)g_2(x_2)...g_m(x_m)$ 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 the MATLAB 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

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

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 $\boldsymbol{d}$-vector. Correspondingly, the coefficient list now becomes a matrix of size `[d,k]`.

Multivariate polynomials appear in this toolbox mainly 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 $[k_1,...,k_m]$ say, and its value at some m-vector x is, correspondingly, given by

$$\sum_{i_1=1}^{k_1} \cdots \sum_{i_m=1}^{k_m} a(i_1,...,i_m)\big(x(i_1)-c(i_1)\big)^{k_1-i_1}\cdots\big(x(i_m)-c(i_m)\big)^{k_m-i_m},$$

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 $\xi_1 < ... < \xi_{l+1}$, 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*):

$$B_{j,k}(x) = B\left(x \mid t_j,...,t_{j+k}\right) = \left(t_{j+k} - t_j\right)\left[t_j,...,t_{j+k}\right](x - \cdot)_+^{k-1}.$$

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

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

But, instead of trying to understand the above formula for the B-spline, look at the reference pages for the GUI `bspligui` for some of the basic properties of the B-spline, and use that GUI to gain some firsthand experience with this intriguing function. Its most important property for the purposes of this toolbox is also the reason Schoenberg used the letter B in its name:

Every space of (univariate) piecewise-polynomials of a given order has a Basis consisting of B*-splines (hence the "B" in B-spline).*

Splines

Consider the set

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

of all (scalar-valued) piecewise-polynomials of order k with breaks $\xi_1 < ... < \xi_{l+1}$ that, for $i = 2...l$, may have a jump across ξ_i in its μ_ith derivative but have no jump there in any lower order derivative. This set is a linear space, in the sense that any scalar multiple of a function in S is again in S, as is the sum of any two functions in S.

Accordingly, S contains a basis (in fact, infinitely many bases), that is, a sequence $f_1,...,f_n$ so that every f in S can be written *uniquely* in the form

$$f(x) = \sum_{j=1}^{n} f_j(x) a_j,$$

for suitable coefficients a_j. The number n appearing here is the *dimension* of the linear space S. The coefficients a_j are often referred to as the *coordinates* of f with respect to this basis.

In particular, according to the Curry-Schoenberg Theorem, our space S has a basis consisting of B-splines, namely the sequence of all

B-splines of the form $B(\cdot \mid t_j,...,t_{j+k})$, $j = 1...n$, with the knot sequence t obtained from the break sequence ξ and the sequence μ by the following conditions:

- Have both ξ_1 and ξ_{l+1} occur in t exactly k times
- For each $i = 2{:}l$, have ξ_i occur in t exactly $k - \mu_i$ times
- Make sure the sequence is nondecreasing and only contains elements from ξ

Note the correspondence between the multiplicity of a knot and the smoothness of the spline across that knot. In particular, at a simple knot, that is a knot that appears exactly once in the knot sequence, only the $(k - 1)$st derivative may be discontinuous.

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. Because $s = wr$, Leibniz' rule tells us that

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

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

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

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

Thin-plate splines

A bivariate thin-plate spline is of the form

$$f(x) = \sum_{j=1}^{n-3} \varphi\left(\left|x - c_j\right|^2\right) a_j + x(1)a_{n-2} + x(2)a_{n-1} + a_n,$$

with $\varphi(t) = t\log t$ a univariate function, and $\|y\|$ denoting the Euclidean length of the vector y. The sites c_j are called the *centers*, and the radially symmetric function $\psi(x) := \varphi(|x|^2)$ is called the *basis function*, of this particular stform.

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 because, for arbitrary n *data points* (x_i,y_i) with distinct data sites, there is exactly one polynomial of order $n - 1$ that matches these data. Explicitly, choose the f_j in the above "model" to be

$$f_j(x) = \prod_{i \neq j} (x - x_i),$$

which is an $n - 1$ degree polynomial for each j. $f_j(x_i) = 0$ for every $i \neq j$, but $f_j(x_j) \neq 0$ as long as the x_i are all distinct. Set $a_j = y_j/f_j(x_j)$ so that

$$f(x_j) = f_j(x_j)a_j = y_j \text{ for all } j.$$

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 $t_1 \le t_2 \le \ldots \le t_{n+k}$. For this choice, there is the following important theorem.

Schoenberg-Whitney Theorem

Let $x_1 < x_2 < ... < x_n$. 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

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

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

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

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

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

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,

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

with λ a nonnegative weight function that is usually just the constant function 1, and $D^m f$ the mth 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

$$\rho E(f) + F\left(D^m f\right) \quad \text{or} \quad \mathsf{p}E(f) + (1-\mathsf{p})F\left(D^m f\right),$$

for some choice of ρ or of p, 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 $\rho \geq 0$ or $\mathsf{p} \in [0..1]$ used, the more closely f matches the given data, and the larger is its mth derivative.

For data values y_i at sites c_i in the *plane*, one uses instead the error measure and roughness measure

$$E(f) = \sum_i |y_i - f(c_i)|^2, \quad F\left(D^2 f\right) = \int \left(|D_{11} f|^2 + 2|D_{12} f|^2 + |D_{22} f|^2\right),$$

and, correspondingly, the minimizer of the sum $\rho E(f) + F(D^2 f)$ is not a polynomial spline, but is a thin-plate spline.

Note that the unique minimizer of $\rho E(f) + F(D^2 f)$ for given $0 < \rho < \infty$ is also the unique minimizer of $\mathsf{p}E(f) + (1-\mathsf{p})F(D^2 f)$ for $\mathsf{p} = \rho/(1+\rho) \in (0 .. 1)$ and *vice versa*.

2D, 3D, ND

Terms such as "a 2D problem" or "a 3D problem" are not used in this toolbox, because they are not well defined. For example a 2D problem could be any one of the following:

- Points on some curve, where you must construct a spline curve, i.e., a vector-valued spline function of one variable.
- Points on the graph of some function, where you must construct a scalar-valued spline function of one variable.

- Data sites in the plane, where you must construct a bivariate scalar-valued spline function.

A “3D problem” is similarly ambiguous. It could involve a curve, a surface, a function of three variables, ... Better to classify problems by the domain and target of the function(s) to be constructed.

Almost all the spline construction commands in this toolbox can deal with ND-valued data, meaning that the data values are ND-arrays. If `d` is the size of such an array, then the resulting spline is called `d`-valued.