Spline Toolbox

For Use with MATLAB®

Carl de Boor

Computation

Visualization

Programming

User's Guide

Version 2

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc.
24 Prime Park Way
Natick, MA 01760-1500 Mail

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

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

Spline Toolbox User's Guide

Printing History: March 1990 First printing
November 1992 Second printing
January 1998 Third Printing
January 1999 Revised for Version 2.0.1 (Release 11) (Online only)

Contents

1 Tutorial

2 Reference

Index

1

Tutorial

Tutorial

This toolbox contains MATLAB versions of the essential programs of the B-spline package (extended to handle also spline *curves*) as described in *A Practical Guide to Splines*, (Applied Math. Sciences Vol. 27, Springer Verlag, New York (1978), xxiv + 392p), hereafter referred to as *PGS*. The toolbox makes it possible to create and work with piecewise polynomial functions and curves in the convenient interactive environment that MATLAB provides.

The typical use envisioned for this toolbox involves the construction and subsequent use of a piecewise polynomial (pp) 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 pp 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 curve 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, a spline or piecewise polynomial function is a structure, and you usually do not need to know just how the structure might describe that function nor do you need to know the details (such as breaks or knots or coefficients). At present, there are two forms possible, the B-form (and a variant, the BBform) and the ppform, but many of the M-files accept both. For example, `fnval(f,x)` returns the values at `x` of the function described by the structure `f`, regardless of what form is used in that description.

The toolbox supports two forms for the representation of pp 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 pp function is to be evaluated extensively. There is an M-file for the conversion from one to the other.

The two forms, the B-form and the ppform, are exactly the B-representation and the pp representation used in *PGS*, with one exception (which corrects what I now consider to be a flaw in the setup in *PGS*): A pp function is still always taken to be continuous from the right, *except* at the right endpoint of its *basic interval*, at which it is taken to be continuous from the left. In conjunction with the capability of changing the basic interval (with the aid of `fnbrk`), this makes it possible to compute limits from the left without the artifice of *small perturbations* of the knots or breaks involved.

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. The Spline Toolbox takes advantage of the bandedness in two ways. Some M-files (e.g., `csapi`) make use of MATLAB's sparse matrix facilities, while others (e.g., `spapi`) make explicit use of the fact that these systems are, more precisely, almost block-diagonal, and the toolbox is equipped with M-files for the generation and solution of such systems, within the space needed to store its nontrivial parts.

Bivariate (or even multivariate) pp functions can be constructed as tensor products of the univariate functions provided here. Other multivariate forms may be added at a later time.

There are various demos, all accessible via the M-file `spdemos`, and you are strongly urged to have a look at some of them before attempting to use this toolbox, or even before reading on.

Some Simple Examples

Here are some simple ways to make use of the M-files 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 statement

```
spdemos
```

Check the reference pages if you have specific questions about the use of the M-files mentioned. Check subsequent sections of this tutorial if you have specific questions about the terminology used; a look into the index may help.

Suppose you want to interpolate to given values y(*i*) at x(*i*), *i* = 1, …, n, by some spline, and assume, for simplicity, that the sequence x is strictly increasing. The simplest spline interpolant is provided, and displayed, by the statements

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

It is the cubic spline interpolant with breaks at the data abscissae x(*i*) and with the so-called *not-a-knot* end conditions.

Actually, the broken line interpolant is simpler, and it can be obtained, and evaluated at some point sequence xx, by

```
pl = spmak(augknt(x,2),y); values = fnval(pl,xx);
```

If a *periodic* spline interpolant is wanted, it is supplied by the statement

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

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

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

provides the cubic spline interpolant with breaks at the x(*i*) and with its slope at x(1) equal to 3, and its second derivative at x(n) equal to -4.

If you want to interpolate at points other than the breaks and/or by splines other than cubic splines with simple knots, then you would use the statement

```
sp = spapi(knots,x,y);
```

in which knots specifies the spline to be used. Roughly speaking, the entries of the nondecreasing sequence knots are the breaks of the spline, and multiplicities (repeats) in this sequence specify the smoothness of the spline across such a break. There will be an interpolating spline if and only if x

satisfies the *Schoenberg-Whitney conditions* with respect to knots, i.e., if and only if

knots(*i*)+ ≤ x(*i*) ≤ knots(*i*+k)-, *i*=1,...n

with k := length(knots) - n, and equality permitted only if the knot x(*i*) in question has multiplicity k, in which case the + or - following the knot indicates that the limit from the right, respectively, left is being matched at that knot. Further, the polynomial pieces of the resulting spline will be of degree < k. For example, if x is uniformly spaced, then, for any positive integer k no bigger than n, the following will work:

```
sp = spapi(augknt(linspace(x(1),x(n),n-k+2),k),x,y);
```

As another example, the toolbox provides the so-called *optimal* spline interpolant to arbitrary data, by the statement

```
oi = spapi(augknt([x(1),optknt(x,k),x(n)],k),x,y);
```

in which the needed knot sequence is chosen in an optimal fashion.

Osculatory interpolation is also easily provided. For example, if, in addition to 1-row matrices x and y, you are also given the 1-row matrix s (of the same length n), then

```
xx = reshape([x;x],1,2*n); yy = reshape([y;s],1,2*n);
ch = spapi(augknt(x,4,2), xx, yy);
```

gives the cubic Hermite interpolant to the data, i.e., the $C^{(1)}$ piecewise cubic function with breaks at the x(*i*), taking on the value y(*i*) and slope s(*i*) at x(*i*), all *i*. The same interpolant can also be obtained by

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

since spapi will reorder the data points according to their abscissae.

If the data are noisy, you might prefer to use the cubic *smoothing spline* instead. It is provided by the statement

```
cs = csaps(x,y,p);
```

This requires you to supply some number p ∈ [0..1] in order to specify the amount of smoothing you want. At one extreme, p = 0, you'll get maximal smoothing, i.e., you'll get the least-squares straight-line fit to the data. At the other extreme, p = 1, you'll get no smoothing at all, i.e., you'll get the cubic

spline interpolant with the so-called *natural* end conditions. Since choice of this smoothing parameter `p` can be tricky, you might prefer to use

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

which provides the smoothest natural cubic spline which fits the data to within `tol`.

If a least-squares approximant is wanted instead, it is provided by the statement

```
sp = spap2(knots,k,x,y);
```

in which both the knot sequence `knots` and the order `k` of the spline must be provided. Further, there is such a least-squares spline fit only if some subsequence of `x` satisfies the Schoenberg-Whitney conditions with respect to `knots`. Incidentally, except for some input-checking, the statement

```
spap2(knots,k,x,y);
```

is short-hand for the statement

```
spmak(knots,slvblk(spcol(knots,k,x,1),y.').');
```

in which the spline is constructed from the knots and from the B-spline coefficients obtained (via `slvblk`) as the least-squares solution of the appropriate linear system, with the matrix of that system supplied by `spcol`.

If `f` is one of these splines `cs`, `pl`, `oi`, `ch`, `sp` so constructed, then it can be displayed by the statement

```
fnplt(f)
```

Its value at `a` is given by the statement

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

Its second derivative is constructed by the statement

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

or by the statement

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

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

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

The toolbox supports *vector-valued* splines. For example, if you want a spline curve through given planar points (x(*i*), y(*i*)), *i* = 1, ..., n, then the statements

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

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

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

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

```
fnplt(spapi(augknt(t,4,2),t,reshape(z,2,2*n)))
```

Vector-valued splines are also used in the approximation to *gridded data*, in any number of variables, using *tensor-product* splines. The same spline-construction M-files are called, only the form of the input differs. For example, if x is an m-vector and y is an n-vector and z is an array of size [m,n], then

```
cs = csapi({x,y},z);
```

provides 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); 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'});
fnplt(sph), axis equal, axis off
```

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

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

Both plots are shown below.

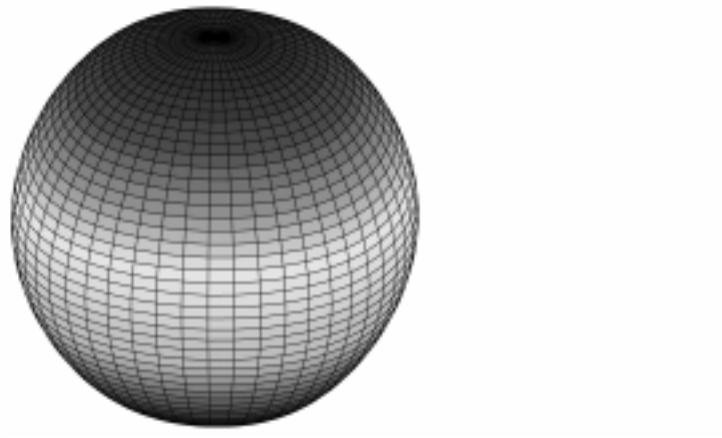

Figure 1-1 A Sphere Made by a 3D-Valued Bivariate Tensor Product Spline

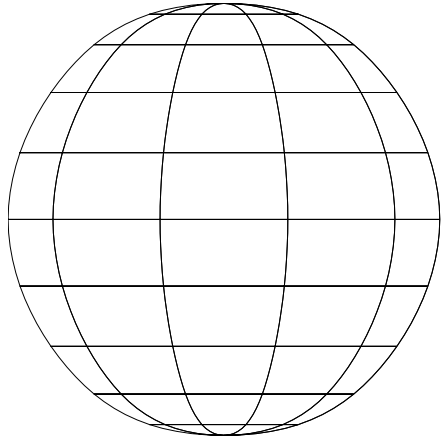

Figure 1-2 A Planar Projection of The Above Spline Sphere

Splines: An Overview

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

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

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

There are two commonly used ways to represent a spline, the ppform and the B-form. The *ppform* of a spline provides a description in terms of its *breaks* $\xi_1, \ldots, \xi_{l+1}$ and the *local polynomial coefficients* c_{ji} of its pieces

$$p_j(x) = \sum_{i=1}^{k} (x-\xi_j)^{k-i} / (k-i)! c_{ji}$$

It is convenient for the evaluation and other uses of a spline. The *B-form* has become the standard way to represent a spline during its construction, since the B-form makes it easy to build in smoothness requirements across breaks. The B-form describes a spline as a linear combination

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

of B-splines. Here, $B_{j,k} := B(\cdot \mid t_j,...,t_{j+k})$ 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 pp 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

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

The multiplicity of the knots governs the smoothness: 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. Since each B-spline has only small support, the linear system for the B-spline coefficients of the spline to be determined, by interpolation or best approximation, or even as the approximate solution of some differential equation, is *banded*, hence easily solvable. 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 points $x_1 < \cdots < x_n$ uniquely by a spline of order k with knot sequence $t_1, \ldots, t_{n+k}$ if and only if $B_{j,k}(x_j) \neq 0$ for all j (Schoenberg-Whitney Conditions). Computations with B-splines are facilitated by stable *recurrence relations*

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

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

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

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

The above *constructive* approach is not the only avenue to splines. In the *variational* approach, a spline is obtained as a *best interpolant*, e.g., as the function with smallest kth derivative among all those matching prescribed function values at certain points. As it turns out, among the many such splines

available, only those that are piecewise polynomial (or, perhaps, piecewise exponential) functions have found much use. Of particular practical interest is the *smoothing spline* $s = s_\lambda$ which, for given data (x_i,y_i) with $x_i \in [a..b]$, all i, and given corresponding positive weights w_i, and for given *smoothing parameter* λ, minimizes

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

over all functions f with m derivatives. It turns out that the smoothing spline s is a spline of order $2m$ with a break at every data point. The art of using the smoothing spline consists in choosing λ so that s contains as much of the information, and as little of the supposed noise, in the data as possible.

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

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

This spline is of order k in x, of order l in y, and of order m in z. Similarly, the ppform of a tensor-product spline is specified by break sequences in each of the variables and, for each (hyper-)rectangle thereby specified, a coefficient array. Further, as in the univariate case, the coefficients may be vectors, typically 2-vectors or 3-vectors, making it possible to represent, e.g., (certain) surfaces in 3-space.

The ppform

A (univariate) *piecewise polynomial*, or *pp*, *f* is characterized by its *break sequence* `breaks` and the *coefficient array* `coefs` of the local power form (see (*) below) of its polynomial pieces. The coefficients may be vectors (usually 2-vectors or 3-vectors) in which case *f* is a curve (in 2-space or 3-space). To comply with the standard treatment of vectors, any coefficient and any value of a spline curve is always written as a 1-*column* matrix. Since MATLAB used to only admit two-dimensional arrays, this useful generalization from functions to curves used to introduce complications. To avoid these mostly notational complications, the present discussion deals only with the case when the coefficients are scalars. Still, this is a good point to stress the fact that, in this toolbox, each coefficient or value of a spline curve is written as a 1-*column* matrix.

The break sequence is assumed to be strictly increasing,

`breaks(1) < breaks(2) <` $\cdots$ `< breaks(l+1)`

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

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

The items `breaks`, `coefs`, `l`, and `k`, make up the *ppform* of *f*. The *basic interval* of this form is [`breaks(1)`..`breaks(l+1)`].

In these terms, the precise description of the pp *f* is

(*) $f(t)$ = `polyval(coefs(j,:),` t `- breaks(j))`

for `breaks(j)` $\le t <$ `breaks(j+1)`.

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

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

For *t* not in `[breaks(1)..breaks(l+1))`, *f*(*t*) is defined by extending the first, respectively last, polynomial piece. For example,
f(t) = `polyval(coefs(1,:),`*t*`-breaks(1))` in case *t* < `breaks(1)`.

A pp is usually constructed in some M-file, through a process of interpolation or approximation, or conversion from some other form (e.g., from a spline), and is output as a one-row vector. But it is also possible to make one up from scratch, using the statement

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

For example, we might say `pp=ppmak(-5:-1,-22:-11)`; thus supplying the uniform break sequence `-5:-1` and the coefficient sequence `-22:-11`. Since this break sequence has 5 entries, hence 4 break intervals, while the coefficient sequence has 12 entries, we have, in effect specified a pp of order 3 (= 12/4). The command

```
fnbrk(pp)
```

prints out all the constituent parts of this pp, 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.

Here are some operations you can perform on a pp.

`v=fnval(pp,x)` evaluates,

`dpp=fnder(pp)` differentiates,

`dirpp=fndir(pp,dir)` differentiates in the direction `dir`,

`ipp=fnint(pp)` integrates,

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

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

`fnplt(pp)` plots pp on its basic interval,

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

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

For example, here is a plot of the particular pp we just made up. First, the basic plot:

```
x = [-55:-5]/10);
plot(x, fnval(pp,x),'-.')
```

Then add to the plot the breaklines:

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

Finally add to that plot the plot of its third piece:

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

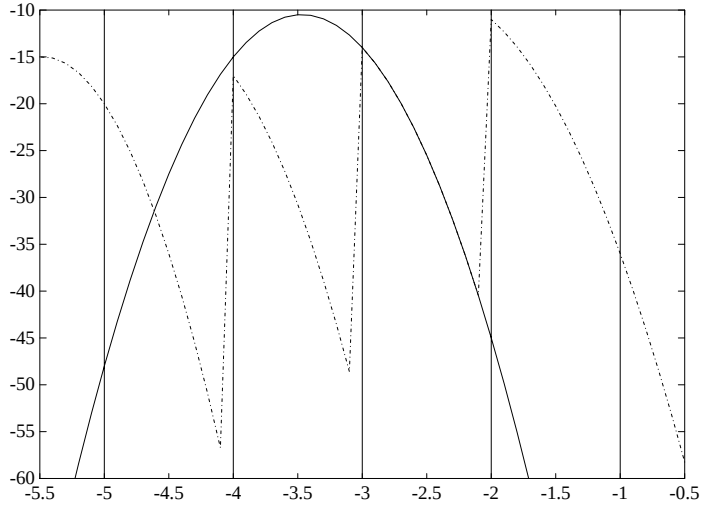

Figure 1-3 A pp Function, its Breaks, and the Polynomial Giving its Third Piece

Here is the final picture. It shows the pp dash-dotted 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 pp at a break is its limit from the *right*, and that the value of the pp outside its basic interval is obtained by extending its leftmost, respectively its rightmost, polynomial piece.

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

The B-form

A (univariate) spline *f* is specified by its (nondecreasing) knot sequence `t` and by its B-spline coefficient sequence `a`. The coefficients may be `d`-vectors (usually 2-vectors or 3-vectors and required to be written as 1-column matrices) in which case *f* is a curve (in 2-space or 3-space) and the coefficients are called the *control points* for the curve.

Roughly speaking, such a spline is pp 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*.) These four items, `t`, `a`, `n`, and `k`, make up the B-form of the spline *f*.

This means, explicitly, that

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

with $B_{i,k} = B(\cdot \mid \mathrm{t}(i{:}i+\mathrm{k}))$ the *i*th B-spline of order `k` for the given knot sequence `t`, i.e., the B-spline with knots `t(`*i*`)`, …, `t(`*i* + `k)`. The basic interval of this form is `[t(1)..t(n+k)]`. Note that a spline is zero outside its basic interval while, after conversion to ppform (via `fn2fm`), this is (usually) not the case since, outside its basic interval, a pp is defined by extension of its first or last polynomial piece.

The building blocks for (the B-form of) a spline are the B-splines. The next page shows a picture of such a B-spline, 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, as generated by the command

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

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 pp of order k with breaks at the points $t(i), \ldots, t(i+k)$. These knots may coincide, and the precise *multiplicity* governs the smoothness with which the two polynomial pieces join there. The rule is:

knot multiplicity + condition multiplicity = order

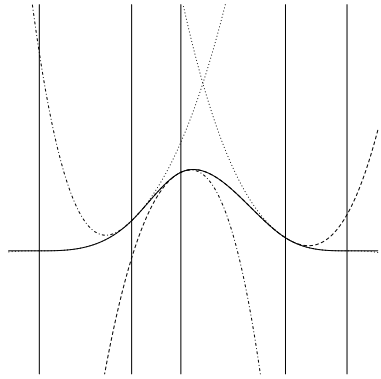

Figure 1-4 A B-Spline of Order 4, and the Four Cubic Polynomials (distinguished by different dash-patterns) From Which it is Made

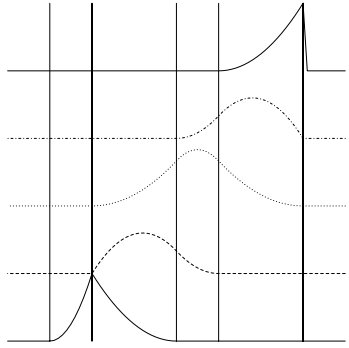

Figure 1-5 All Third-order B-Splines for a Certain Knot Sequence With Various Knot Multiplicities

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.

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

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

Then, the appropriate knot sequence t should contain the break ξ_i exactly $k - \mu_i$ times, $i = 2, \ldots, l$. In addition, it should contain the two endpoints, a and b, of the basic interval exactly k times. (This last requirement can be relaxed, but has become standard.) With this choice, there is exactly one way to write each spline s with the properties described as a linear combination 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];
```

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

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

and this is provided by the statement

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

The preceding page shows a picture of all the third-order B-splines for a certain mystery knot sequence `t`. 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. 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. The picture is generated by the following MATLAB statements (which use the

M-file spcol from this package to generate the function values of all these B-splines at a fine net x).

```
x=[-10:70]/10;
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), hold off
```

Further illustrated examples can be found by running spalldm2.

The short-hand

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

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 (needlessly) appropriated by the CAGD community to mean what is called here a *spline in B-form*, with the unhappy result that, in any discussion between mathematicians/ approximation theorists and people in Graphics, one now always has to check in what sense the term is being used.

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

For example, we might say

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

thus supplying the uniform knot sequence 1:10 and the coefficient sequence 3:8. Since there are 10 knots and 6 coefficients, the order must be 4(= 10 - 6), i.e., we get a cubic spline. The command

```
fnbrk(sp)
```

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

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

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

But the point of the spline toolbox is that there shouldn't be any need for you to look up these details. You simply use `sp` as an argument to M-files that evaluate, differentiate, integrate, convert, or plot the spline whose description is contained in `sp`.

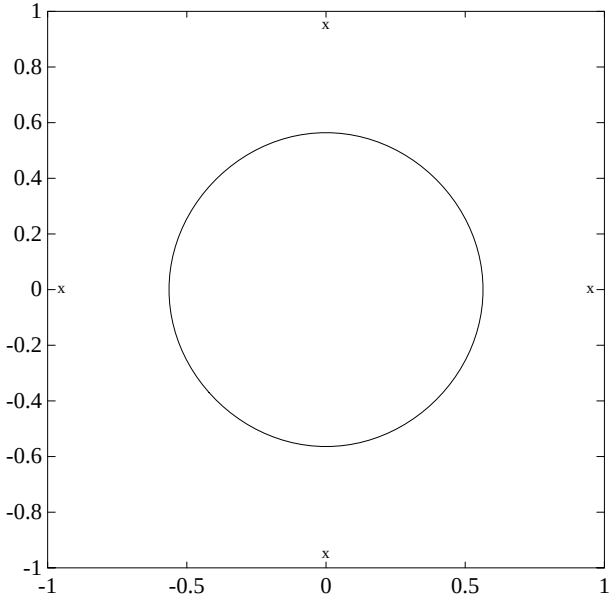

Figure 1-6 Spline Approximation to a Circle; Control Points are Marked x.

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 above plot, generated by a subsequent

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

shows. (Insertion of additional control points $(\pm.95,\pm.95)/\sqrt{1.9}$ would make a visually perfect circle.)

Here are more details. The spline curve generated has the form $\Sigma_{j=1}^{8} B_{j,5}\,a(:,j)$, with `[-4:8]` the (uniform) knot sequence, and with its control points $a(:,j)$ the sequence $(0, \alpha)$, $(-\alpha, 0)$, $(0, -\alpha)$, $(\alpha, 0)$, $(0, \alpha)$ $(-\alpha, 0)$, $(0, -\alpha)$, $(\alpha, 0)$ with $\alpha = .95$. Only the curve part between the parameter values 0 and 4 is actually plotted.

The following M-files are available for spline work: There is `spmak` and `fnpbrk` to make up a spline and take it apart again. There is `fn2fm` to convert from B-form to ppform. You can evaluate, differentiate, integrate, plot, or refine a spline with the aid of `fnval`, `fnder`, `fndir`, `fnint`, `fnplt, and fnrfn`.

There are four M-files for generating knot sequences: `augknt` for providing boundary knots; `aveknt` for providing knot averages (as good interpolation points); `newknt` for redistribution of knots to suit a given approximand better; and `optknt` for providing *optimal* knots for interpolation at given points.

For display of a spline curve with (two-dimensional) coefficient sequence and a uniform knot sequence, there is `spcrv` (which relies on no other M-file in this toolbox, hence is handy for independent graphics work).

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 point 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`, an M-file for solving linear systems with almost-block-diagonal coefficient matrix.

`spcol` and `slvblk` are used in `spapi` for the construction of an interpolating spline, i.e., a spline that matches given data exactly. They are also used (in

spap2) to construct a spline that best approximates data in the least-squares sense. The use of spcol and slvblk in solving differential equations is illustrated in a subsequent example. The smoothing spline, of order 2, 4, or 6, is obtained in spaps using MATLAB's sparse matrix facility.

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 B-form-related M-files in this toolbox are service routines, i.e., routines not likely to be of direct interest to the casual user. These include sprpp and splpp, of use in converting from B-form to ppform; and sorted, of use in relating data points to mesh intervals.

Tensor Product Splines

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

For simplicity, the discussion to follow deals just with bivariate splines.

The tensor-product idea is very simple: If f is a function of x and g is a function of y, then their tensor-product

$p(x,y) := f(x)\, g(y)$

is a function of x and y, a bivariate function. More generally, with $s = (s_1, ..., s_{m+h})$ and $t = (t_1, ..., t_{n+k})$ knot sequences and $(a_{ij} : i = 1, ..., m; j = 1, ..., n)$ a corresponding coefficient array, we obtain a bivariate spline as

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

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

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

constructs this form. Further, `fnplt`, `fnval`, `fnder`, `fndir`, `fnrfn`, `fn2fm` can be used to plot, evaluate, differentiate and integrate, refine, and convert this form. You are most likely to construct such a form by looking for an interpolant or approximant to gridded data. E.g., 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(i)) for i=1:M, j=1:N)`, by the command:

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

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

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

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

Figure 1-7 A Moebius Band Made by Vector-Valued Bivariate Spline Interpolation

Example: A Nonlinear ODE

The following sample can be run via `difeqdem`.

We consider the nonlinear singularly perturbed problem

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

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

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

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

Since cubics are of order 4, we have

```
k = 4;
```

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

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

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

This implies that we have

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

= 10 degrees of freedom. We collocate at two points per polynomial piece, i.e., at 8 points altogether. This, together with the two side conditions, gives us 10 conditions, which matches the 10 degrees of freedom.

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

```
gauss=[-1; 1]/(sqrt(3)*2);
```

From this, we obtain the whole collection of collocation points by

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

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

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

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

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

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

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

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

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

with

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

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

We use `spcol` to supply the matrix

```
 colmat =
spcol(knots,k,reshape([points([111],:),1,(length(points)));
```

From this, we get the collocation matrix by combining the row triple of `colmat` for x using the weights $w_0(x)$, $w_1(x)$, $w_2(x)$ to get the row for x of the actual

matrix. For this, we need a current approximation *y*. Initially, we get it by interpolating some reasonable initial guess from our pp space at the `points`. We use the parabola $()^2 - 1$ (i.e., the function $x \mapsto x^2-1$) that satisfies the end conditions as the initial guess, and pick the matrix from the full matrix `colmat`. Here it is, in several (cautious) steps:

```
intmat = colmat([2 1+[1:8]*3,1+9*3],:);
coefs = intmat\[0 colpnts.*colpnts-1 0].';
y = spmak(knots,coefs.');
```

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

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

The resulting printout of the errors

```
maxdif =
0.2067
maxdif =
0.0121
maxdif =
3.9515e-005
maxdif =
4.4322e-010
```

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

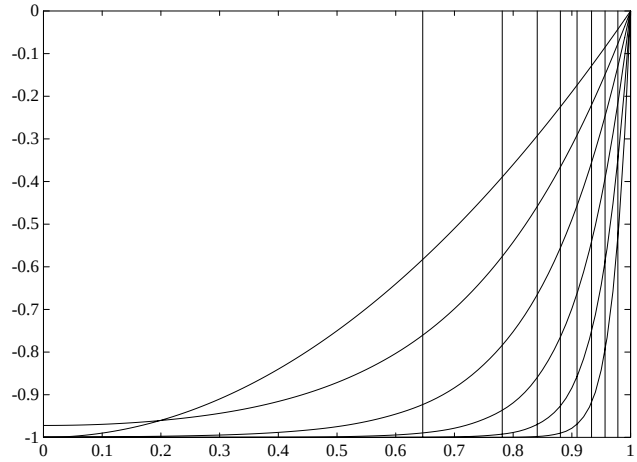

Figure 1-8 Solutions of a Nonlinear ODE With Increasingly Strong Boundary Layer

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

We use newknt to construct an appropriate (finer) mesh from the current approximation:

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

From the new break sequence, we generate the new collocation point sequence:

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

We use spcol to supply the matrix

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

and use our current approximate solution z as the initial guess:

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

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

```
tolerance=1.e-9;
while 1
  vtau=fnval(y,colpnts);
```

etc.

Repeated passes through this process generate a sequence of solutions, for ε = 1/10, 1/30, 1/90, 1/270, 1/810. The resulting solutions, ever flatter at 0 and ever steeper at 1, are shown in the plot above. The plot also shows the final break sequence (as a sequence of vertical bars).

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

Example: Construction of the Chebyshev Spline

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

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

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

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

```
k = 4;
```

and use the break sequence

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

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

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

Note the quadruple knot at each end. Since `k = 4`, this makes `[0..8] = [breaks(1)..breaks(1p1)]` the interval $[t_k..t_{n+1}]$ of interest, with `n = length(t)-k` the dimension of the resulting spline space $S_{k,t}$. The same knot sequence would have been supplied by

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

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

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

recommended as good interpolation point choices. These are supplied by

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

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

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

Here is the resulting picture:

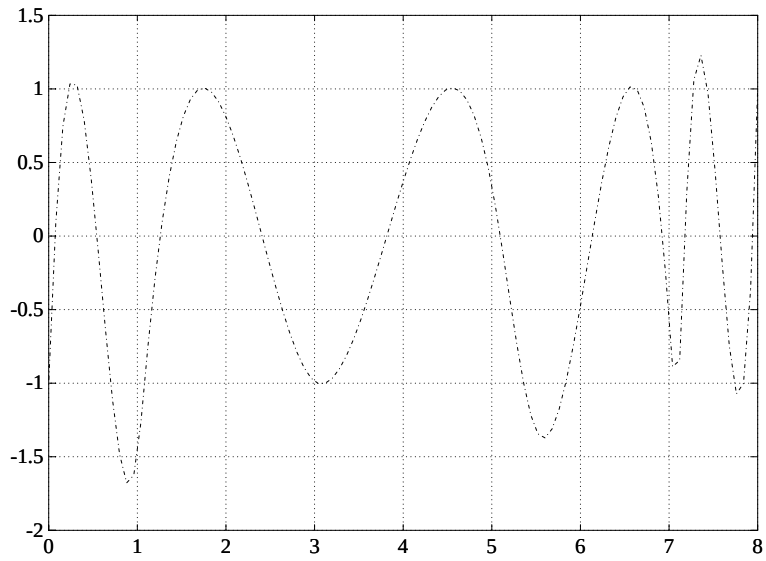

Figure 1-9 First Approximation to a Chebyshev Spline

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

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

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

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

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

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

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

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

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

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

We refine this estimate for the zeros of *Dc* by two steps of the secant method, taking `tau` and this `guess` as our first approximations. First, we evaluate *Dc* at both point sets:

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

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

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

The check

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

shows the improvement.

Now we take these points as our new tau:

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

and check the extrema values of our current approximation there:

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

The difference

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

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

We construct a new spline corresponding to our new choice of tau and plot it on top of the old:

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

Here is the resulting picture:

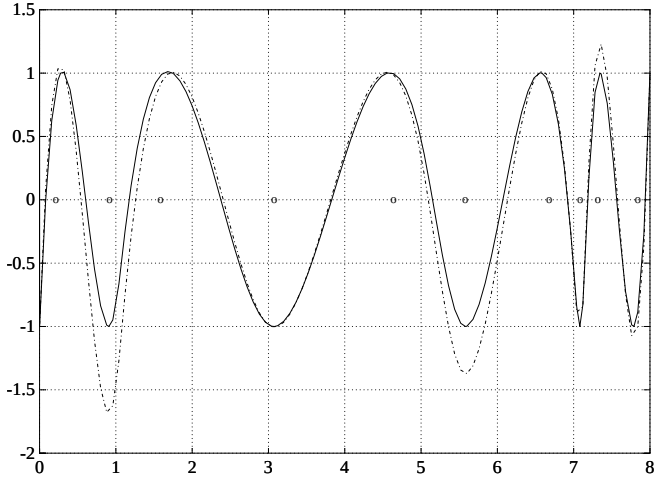

Figure 1-10 A More Nearly Level Spline

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

Example: Approximation by Tensor Product Splines

Since the toolbox can handle splines with *vector* coefficients, it is easy to implement interpolation or approximation to gridded data by tensor product splines, as the following illustration, run by `tspdem`, is meant to show.

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

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

```
x = sort([[0:10]/10,.03 .07, .93 .97]);
y = sort([[0:6]/6,.03 .07, .93 .97]);
[xx,yy] = ndgrid(x,y); z = franke(xx,yy);
```

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

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

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

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

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

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

In particular, the statements

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

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

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

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

Note the use of `vals.'`, in the `mesh` command, needed because of MATLAB's matrix-oriented view when plotting an array. This can be a serious problem in bivariate approximation since there it is customary to think of `z(`*i*`,`*j*`)` as the function value at the point (`x(`*i*`), y(`*j*`)`), while MATLAB thinks of `z(`*i*`,`*j*`)` as the function value at the point (`x(`*j*`), y(`*i*`)`).

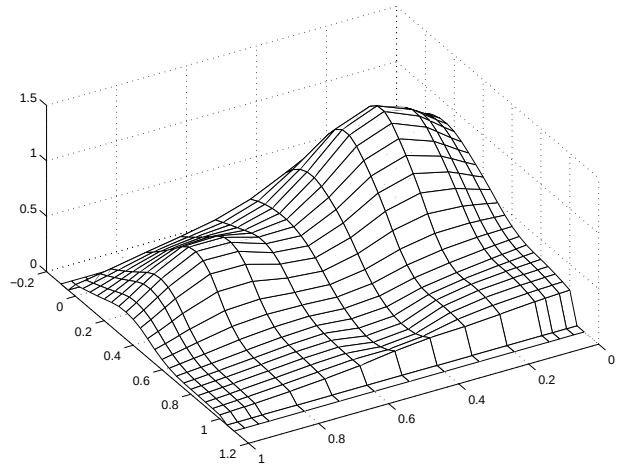

Figure 1-11 A Family of Smooth Curves Pretending to be a Surface

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

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

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

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

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

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

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

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

Note that `spap2(knots,k,x,fx)` expects `fx(:,j)` to be the datum at `x(j)`, i.e., expects each *column* of `fx` to be a function value. Since we wanted to fit the datum `coefsy(:,`r`)` at `x(`r`)`, all r, we had to present `spap2` with the *transpose* of `coefsy`.

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

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

It provides the *bivariate* spline approximation

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

to the original data

`(x(`i`),y(`j`))` $\mapsto$ `z(x(`i`),y(`j`))`, i=1, ..., `Nx`; j=1, ...`Ny`

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

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

you can do the following:

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

See Figure 1-12 for the result.

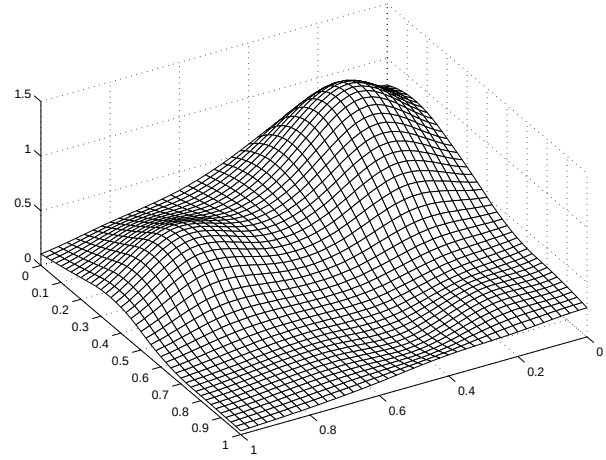

Figure 1-12 Spline Approximation to Franke's Function

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

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

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

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

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

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

```
errors = z - spcol(knotsx,kx,x)*coefs*spcol(knotsy,ky,y).';
disp( max(max(abs(errors)))/max(max(abs(z))) )
0.0539
```

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

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

to fit a data array of size

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

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

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

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

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

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

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

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

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

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

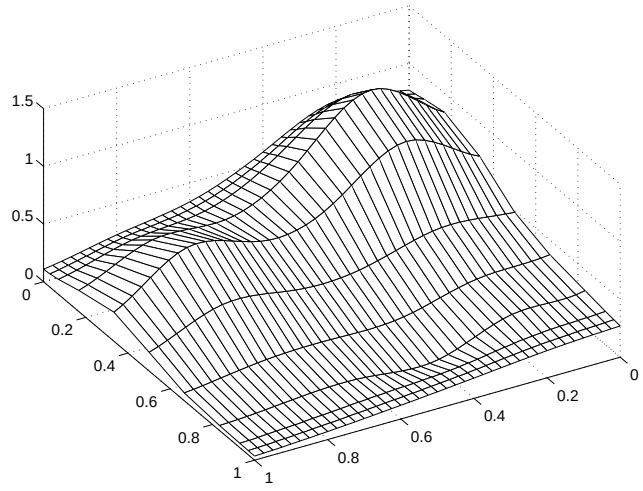

Figure 1-13 Another Family of Smooth Curves Pretending to be a Surface

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

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

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

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

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

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

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

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

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

```
disp( max(max(abs(coefs - coefsb))) )
1.4433e-15
```

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

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

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

In particular, the statements

```
sp = spap2(knotsy,ky,y,z);
coefsy =fnpbrk(sp,'c');
```

provide us with the matrix `coefsy` that satisfies

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

The subsequent computations

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

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

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

In the second, alternative calculation, we first computed

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

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

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

then provided

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

Consequently, `coefsb` = `coefs`.

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

If, for example, the given data over a *three*-dimensional grid is contained in some 3-dimensional array `v` of size `[Nx,Ny,Nz]` (with `v(i,j,k)` containing the value *f*(`x(i)`, `y(j)`, `z(k)`)), then we would start off with

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

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

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

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

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

2

Reference

This chapter contains detailed descriptions of the main commands in the Spline Toolbox. It begins with a listing of entries grouped by subject area and continues with the reference entries in alphabetical order.

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

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

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

The naming convention used is in part the result of discussions with Jörg Peters.

Operators	
`fnval`	evaluation of a function
`fnbrk`	name and part(s) of a form
`fncmb`	arithmetic with functions
`fn2fm`	convert to specified form
`fnder`	derivative of a function
`fnint`	integral of a function
`fnjmp`	jumps, i.e., $f(x+) - f(x-)$
`fnplt`	display a function
`fnrfn`	Insert additional points into the partition for *f*

Cubic Splines	
`csapi`	cubic spline interpolation
`csape`	cubic spline interpolation with end conditions
`csaps`	cubic smoothing spline
`cscvn`	interpolating cubic spline curve
`getcurve`	interactive creation of a cubic spline curve

Splines in PPForm	
`pplst`	some help
`ppmak`	make a pp
`ppual`	evaluate function in ppform, possibly continuous from the left

Splines in B-Form	
splst	some help
spmak	make a spline
spcrv	generate a spline curve
spapi	spline interpolation
spap2	spline approximation
spaps	smoothing spline
spcol	collocation matrix

Work With Breaks and Knots	
augknt	augment a break or knot sequence
aveknt	provide knot averages
brk2knt	breaks with multiplicities into knots
knt2brk	From knots to breaks and their multiplicities
knt2mlt	knot multiplicities
sorted	locate one sequence within another
newknt	provide better(?) break sequence
optknt	provide 'optimal' knots

Customized Linear Equation Solver	
slvblk	solve almost block diagonal system
bkbrk	take apart almost block diagonal matrix

Demonstrations	
spdemos	list of demonstrations
splexmpl	some simple examples
ppalldm2	introduction to ppform
spalldm2	introduction to B-form
bspline	display a B-spline
bsplidem	some B-splines
csapidem	cubic spline interpolation
csapsdem	cubic smoothing spline
pckkntdm	knot choices
spcrvdem	spline curve construction
difeqdem	a singularly perturbed ODE
chebdem	an equi-oscillating spline
tspdem	tensor products

Utilities	
franke	Franke's bivariate test function.
subplus	positive part
titanium	titanium heat data
splpp	convert left of 0 from B to pp
sprpp	convert right of 0 from B to pp

augknt

Purpose

Augment a knot sequence

Syntax

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

Description

`augknt` returns a nondecreasing and augmented knot sequence that has the first and last knot with exact multiplicity `k`. (This may actually shorten the knot sequence.) Also returns the number `addl` of knots added on the left. (This number may be negative.)

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

Examples

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

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

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

Purpose

Provide knot averages

Syntax

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

Description

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

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

recommended as good interpolation point 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].

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

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

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

For sp the B-form of a scalar-valued univariate spline function, with t=fnbrk(sp,'knots') and k=fnbrk(sp,'order'), the points (tstar(*i*),a(*i*)) with tstar=aveknt(t,k) constitute the vertices of the spline's *control polygon*.

See Also

optknt

bkbrk

Purpose

Get the details of an almost block-diagonal matrix

Syntax

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

Description

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

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

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

See Also

`slvblk, spcol`

References

[1] C. de Boor and R. Weiss, "SOLVEBLOK: A package for solving almost block diagonal linear systems", *ACM Trans. Mathem. Software 6* (1980), 80 – 87.

Purpose

Generate a knots sequence from a break sequence with multiplicities

Syntax

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

Description

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

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

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

Examples

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

See Also

`knt2mlt`, `knt2brk`, `augknt`

bspline

Purpose

Display a B-spline and its polynomial pieces

Syntax

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

Description

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

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

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

Examples

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

See the demo `bsplidem` for typical uses of this M-file.

See Also

`subplot`, `bsplidem`

Purpose

Cubic spline interpolation with end conditions

Syntax

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

Description

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

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

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

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

Available conditions are:

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

with e = a(b) the first (last) data point in case j = 1 (j = 2), and (in the Lagrange condition) p the cubic polynomial that interpolates to the given data at e and the three points nearest e.

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

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

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

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

This M-file is a much expanded version of the Fortran routine `CUBSPL` in *PGS*.

Examples

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

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

```
x = 0:4; y=-2:2; s2 = 1/sqrt(2);
 clear v
 v(3,:,:) = [0 1 s2 0 -s2 -1 0].'*[1 1 1 1 1];
 v(2,:,:) = [1 0 s2 1 s2 0 -1].'*[0 1 0 -1 0];
 v(1,:,:) = [1 0 s2 1 s2 0 -1].'*[1 0 -1 0 1];
 sph = csape({x,y},v,{'clamped','periodic'});
 values = fnval(sph,{0:.1:4,-2:.1:2});

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

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

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

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

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

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

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

```
pp1 = csape(x,y);
dp1 = fnder(fnbrk(pp1,1));
pp0 = csape(x,zeros(size(y)),[1,0],[1,0]);
dp0 = fnder(fnbrk(pp0,1));
e = x(1);
lam1 = a*fnval(dp1,e) + b*fnval(fnder(dp1),e);
lam0 = a*fnval(dp0,e) + b*fnval(fnder(dp0),e);
pp = fncmb(pp0,(c-lam1)/lam0,pp1);
```

Algorithm

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

See Also

`csapi`, `spline`

Cautionary Note

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

Purpose

Cubic spline interpolation

Syntax

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

Description

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

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

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

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

This M-file 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 ordinates and can handle gridded data.

Examples

See the demo `csapidem` for various examples.

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

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

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

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

```
x =.0001+[-4:.2:4]; y = -3:.2:3;
[yy,xx] = meshgrid(y,x); r = pi*sqrt(xx.^2+yy.^2); z = sin(r)./r;
bcs = csapi( {x,y}, z ); fnplt( bcs ), axis([-5 5 -5 5 -.5 1])
```

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

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

Algorithm

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

The not-a-knot end condition is used, thus forcing the first and second polynomial piece of the interpolant to coincide, as well as the second-to-last and the last polynomial piece.

See Also

`csape`, `spline`, `tspdem`

Cautionary Note

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

Purpose

Cubic smoothing spline

Syntax

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

Description

The cubic smoothing spline *s* to the given data `x,y` is constructed, for the specified *smoothing parameter* `p` $\in$ `[0..1]` and the optionally specified weight `w`. The smoothing spline minimizes

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

with `w=ones(size(x))` the default value for `w`. For `p` = 0, *s* is the least-squares straight line fit to the data, while, on the other extreme, i.e., for `p` = 1, *s* is the variational, or *natural* cubic spline interpolant. As `p` moves from 0 to 1, the smoothing spline changes from one extreme to the other. The interesting range of `p` is often near $1/(1+h^3/6)$, with *h* the average spacing of the data abscissae. 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 $p = 1/(1 + h^3/.6)$.

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

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

It is in general difficult to choose the parameter `p` without experimentation. For that reason, use of `spaps` is encouraged since there `p` is chosen so as to produce the smoothest spline within a specified tolerance of the data.

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

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

or

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

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

Algorithm

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

See Also

spaps, csape, spap2

Cautionary Note

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

Purpose

Generate an interpolating parametric cubic spline curve.

Syntax

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

Description

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

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

If the first and last point coincide (and there are no other repeated points), then a periodic cubic spline curve is constructed. However, double points result in corners.

Examples

The following provides the plot of a questionable curve through some points (marked as circles):

```
points=[0 1 1 0 -1 -1 0 0 ;
          0 0 1 2 1 0 -1 -2];
fnplt(cscvn(points)); hold on,
plot(points(1,:),points(2,:),'o'), hold off
```

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

```
fnplt(cscvn([0 .8 .9 0 0 -.9 -.8 0; .5 1 0 -1 -1 0 1 .5]))
 axis equal
```

Algorithm

The break sequence `t` is determined as

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

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

See Also

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

References

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

fn2fm

Purpose

Convert from one form to another

Syntax

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

Description

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

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

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

If `form` is `'B-'` (and `F` is in ppform), then the actual smoothness of the function in `f` across each of its interior breaks has to be guessed. This is done by looking, for each interior break, for the first derivative whose jump across that break is not *small* compared to the size of that derivative nearby. The default tolerance used in this is 1.e-12. But the user can assist by supplying a tolerance (strictly between 0 and 1) in the optional argument `sconds`.

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

Examples

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

As another example,

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

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

Algorithm

For a multivariate (tensor-product) function, univariate algorithms are applied in each variable.

For the conversion from B-form (or BBform) to ppform, the utility M-file `sprpp` is used to convert the B-form of all polynomial pieces to their local power form, using repeated knot insertion at the left endpoint.

The conversion from B-form to BBform is accomplished by inserting each knot enough times to increase its multiplicity to the order of the spline.

The conversion from ppform to B-form makes use of the dual functionals discussed in the section “Splines: An Overview” in Chapter 1. Without further information, such a conversion has to ascertain the actual smoothness across each interior break of the function in `f`.

See Also

`ppalldm2`, `spalldm2`, `spmak`, `ppmak`

Cautionary Note

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

fnbrk

Purpose

Name or a part of a form

Syntax

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

Description

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

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

Exactly the same is returned in case `f` is in BBform.

If the form in `f` is a ppform, then the possible choices for `part` are: `'breaks'`, `'coefs'`, `'pieces'` or `'l'`, `'order'`, `'dim'`ension, and `'interval'` (returning the break sequence, the local polynomial coefficients, the number of polynomial pieces, the polynomial order, the (vector) dimension of the coefficients, and the basic interval, respectively). In addition, in this case, `part` can also be a 1-by-2 matrix specifying an interval, in which case the output is the ppform of the restriction/extension of the function in `f` to that interval. Finally, `part` can also be a positive integer, `j`, in which case the output is the ppform of the *j*th polynomial piece of the pp function in `f`.

If the function in `f` is multivariate, then the corresponding multivariate parts are returned. This means, e.g., that knots and breaks are cell arrays, the coefficient array is, in general, higher than 2-dimensional, and order, number and pieces are vectors.

If no output is specified, then there should be only one input argument and, in that case, nothing is returned, but a description of the various parts of the form is printed on the screen instead.

Examples

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

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

provides the (pointwise) sum of those two functions.

If pp contains the ppform of a bivariate spline with at least 4 polynomial pieces in the first variable, then ppp=fnbrk(pp,{4,[-1 1]}) gives the spline that agrees with the spline in pp on the rectangle [b4 .. b5] x [-1 .. 1] , where b4, b5 are the 4th and 5th point in the break sequence for the first variable.

See Also

ppmak, spmak, ppalldm2, spalldm2

fncmb

Purpose

Arithmetic with function(s)

Syntax

```
fn = fncmb(function,matrix)
fn = fncmb(function,function)
fn = fncmb(function,matrix,function)
fn = fncmb(function,matrix,function,matrix)
fn = fncmb(function,'op',function)
```

Description

The intent is to make it easy to carry out the standard linear operations of scaling and adding within a spline space. More than that, a matrix may be applied to a vector-valued function, and even two (univariate) functions in different forms may be added or multiplied pointwise.

Examples

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

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

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

As a simple example, if the function *f* in `f` happens to be scalar-valued, then `f3=fncmb(f,[1;2;3])` contains the description of the function whose value at *x* is the 3-vector $(f(x),2f(x),3f(x))$. Note that, by the convention throughout this toolbox, the subsequent statement `fnval(f3, x)` returns a 1-*column*-matrix. As another simple example, if `f` describes a surface in 3-space, i.e., the function in `f` is 3-vector-valued bivariate, then `f2 = fncmb(f,[1 0 0;0 0 1]);` describes the projection of that surface to the *(x,z)*-plane. As another example, if `t` is a knot sequence of length `n+k` and `a` is a matrix with `n` columns, then `fncmb(spmak(t,eye(n,n)),a)` is the same as `spmak(t,a)`.

Finally, `fncmb(spmak([0:4],1),'+',ppmak([-1 5],[1 -1]))` is the pp with breaks `-1:5` that, on the interval [0 .. 4], agrees with the function $x \mapsto B(x \mid 0,1,2,3,4) + x$ (but has no active break at 0 or 1, hence differs from this function outside the interval `[0 .. 4]`), while `fncmb(spmak([0:4],1),'-',0)` has the same effect as `fn2fm(spmak([0:4],1),'pp')`.

Algorithm

The coefficients are extracted (via `fnbrk`) and operated on by the specified matrix (and, possibly, added), then recombined with the rest of the function description (via `ppmak` or `spmak`). If there are two functions input, then they must be of the same type (see Limitations, below) *except* for the following:

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

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

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

Limitations

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

Further, if two functions are involved, then they must be of the same type. This means that they must either both be in B-form or both be in ppform, and, moreover, have the same knots or breaks, the same order, and the same target. The only exception to this is the command of the form `fncmb(function,'op',function)`.

Cautionary Note

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

fnder

Purpose

Differentiate a function

Syntax

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

Description

`fnder(f,dorder)` is the description of the `dorder`th derivative of the function whose description is contained in `f`. The default value of `dorder` is 1. For negative `dorder`, the particular `|dorder|`-th indefinite integral is returned that vanishes `|dorder|`-fold at the left endpoint of the basic interval.

The output is of the same form as the input, i.e., they are both ppforms or both B-forms.

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

Examples

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

If `f` is in ppform, then, up to rounding errors, `f` and `fnder(fnint(f))` are the same, unless the function described by `f` has jump discontinuities.

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

As an illustration of this last fact, since `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, the subsequent statement `spdi = fnint(fnder(sp))` gives a spline with 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 `spalldm2` and `ppalldm2` for examples.

Algorithm

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

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

See Also

`fnint`, `fnval`, `fnplt`, `ppalldm2`, `spalldm2`

Purpose

Integrate a function

Syntax

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

Description

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

The output is of the same type as the input, i.e., they are both ppforms or both B-forms.

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

Examples

The statement `fnval(fnint(f),[a b])*[-1;1]` provides the definite integral over the interval `[a .. b]` of the function described by `f`.

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

If `f` is in ppform, then, up to rounding errors, `f` and `fnder(fnint(f))` are the same, unless the function described by `f` has jump discontinuities.

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

As an illustration of this last fact, since `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, the subsequent statement `spdi = fnint(fnder(sp))` gives a spline with 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 `spalldm2` and `ppalldm2` for examples.

Algorithm

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

See Also

`fnder`, `fnval`, `fnplt`, `ppalldm2`, `spalldm2`

fnjmp

Purpose

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

Syntax

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

Description

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

Examples

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

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

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

returns instead the vector `[0 -24 24 -24 0]`, consistent with the fact that, in contrast to the B-form, a spline in ppform does not have a discontinuity in any of its derivatives at the endpoints of its basic interval. Note that `fnjmp(fnder(spmak(x,1),3),-x)` returns the vector `[12,0,0,0,12]` since `-x` differs from `x` by roundoff, hence the third derivative of the B-spline provided by `spmak(x,1)` does not have a jump across `-x(2)`,`-x(3)`, and `-x(4)`.

See Also

`ppalldm2`, `spalldm2`

Purpose

Plot a function

Syntax

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

Description

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

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

if *f* is bivariate, the following will be plotted: if *f* is scalar-valued, the graph of if *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 2 variables, then the bivariate function, obtained by choosing the midpoint of the basic interval in each of the variables other than the first two, is plotted.

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

Algorithm

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

The bivariate function *f* described by `f` is evaluated on a 51-by-51 uniform grid if *f* is scalar-valued or *d*-vector-valued with *d*>2 and the result plotted by `surf`. In the contrary case, *f* is evaluated along the meshlines of a 11-by-11 grid, and the resulting planar curves are plotted.

See Also

`fnder`, `fnint`, `fnval`

Cautionary Note

The basic interval for *f* in B-form is the interval containing *all* the knots. This means that, e.g., *f* is sure to vanish at the endpoints of the basic interval unless the first and the last knot are both of full multiplicity *k*, with *k* the order of the spline *f*. Failure to have such full multiplicity is particularly annoying when *f* is a spline curve, since the plot of that curve as produced by `fnplt` is then bound to start and finish at the origin, regardless of what the curve might otherwise do.

Further, since B-splines are zero outside their support, any function in B-form is zero outside the basic interval of its form. This is very much in contrast to a function in ppform whose values outside the basic interval of the form are given by the extension of its leftmost, respectively rightmost, polynomial piece.

Purpose

Insert additional points into the partition for a form

Syntax

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

Description

The form in `f` is refined by the insertion of the entries of `addpts` into the knot sequence or break sequence of the form. This is of use when the sum of two or more functions of different forms is wanted or when the number of degrees of freedom in the form is to be increased to make fine local changes possible. The precise action depends on the form in `f`.

If the form in `f` is a B-form or BBform, then the entries of `addpts` are inserted into the existing knot sequence, subject only to the restriction that the multiplicity of no knot exceed the order of the spline, and 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 only to the restriction that the break sequence be strictly increasing, and the equivalent ppform with this refined break sequence for the function in `f` is returned.

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

Examples

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

Algorithm

The standard *knot insertion* algorithm is used for the calculation of the B-form coefficients for the refined knot sequence, while Horner's method is used for the calculation of the local polynomial coefficients at the additional breaks in the refined break sequence.

See Also

`fncmb`, `spmak`, `ppmak`

fnval

Purpose

Evaluate a function

Syntax

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

Description

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

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

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

Finally, if `x` is a scalar and `f` is in ppform, then `fnval(fnbrk(f,x+[-1 0]),x)` returns *f*(x-).

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

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. Nested multiplication or the B-spline recurrence (see, e.g., [*PGS*; X.(3)]) is then used in lockstep for the evaluation, depending on whether `f` is in ppform or in B-form. Evaluation of a multivariate function takes full advantage of the tensor product structure.

See Also

`ppmak`, `spmak`, `fnbrk`

Purpose

Interactive creation of a cubic (spline) curve

Syntax

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

Description

`getcurve` displays a gridded window and asks for input. As the user clicks on points in the gridded window, the broken line connecting these points is displayed. When the user is done, (indicated by clicking outside the gridded window), a cubic spline curve, `spcv`, through the point sequence, `xy`, is computed (via `cscvn`) and drawn. The point sequence and, optionally, the spline curve are output.

If the last point is *close* to the initial point, a closed curve is drawn. Clicking twice (or more times) in succession at a point permits the curve to have a corner at that point.

See Also

`cscvn`

knt2brk, knt2mlt

Purpose

Breaks and multiplicities from a knot sequence

Syntax

```
[breaks, mults] = knt2brk(knots)
[m, sortedt] = knt2mlt(t)
```

Description

The idea is to extract the distinct elements from a sequence, as well as their multiplicities in that sequence, with *multiplicity* taken in two slightly different senses.

The statement `knt2brk(knots)` returns the distinct elements in `knots`, and in increasing order. The optional second output argument provides the multiplicity with which each distinct element occurs in `knots`. In particular, the two outputs, `breaks` and `mults`, are of the same length, and `knt2brk` is complementary to `brk2knt` in that, for any knot sequence `knots`, the two commands `[xi,mlts] = knt2brk(knots); knots1 = brk2knt(xi,mlts);` give `knots1` equal to `knots`.

The statement `m = knt2mlt(t)` returns a vector of the same length as `t`, with `m(i)` counting the number of entries to the left of the ith entry in `sortedt = sort(t)` that are equal to that entry. This kind of multiplicity vector is needed in `spapi` or `spcol` where such multiplicity is taken to specify which particular derivatives are to be matched at the points 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 $D^{m(i)}s(\mathtt{t}(i)) = \mathtt{z}(i)$, all i. The optional second argument returns `sort(t)`.

Neither `knt2brk` nor `knt2mlt` are 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`

Purpose

Improve break sequence distribution

Syntax

```
newbreaks = newknt(pp,newl)
[newbreaks,distfn] = newknt(pp,newl)
```

Description

`newknt` returns the break sequence that cuts the basic interval of `pp` into `newl` pieces in such a way that a certain piecewise linear monotone function (whose ppform is returned in `distfn` if requested) related to the high derivative of `pp` is equidistributed.

The intent is to choose a break sequence suitable to the fine approximation of a function *f* whose rough approximation in `pp` is assumed to contain enough information about *f* to make this feasible.

Examples

See the last part of the demo `difeqdem` for an illustration.

Algorithm

This is the Fortran routine `NEWNOT` in *PGS*. With *k* the order of the pp 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 pp *f* in such a way that

$$\int_{\text{newbreaks(i)}}^{\text{newbreaks(i+1)}} |D^k f|^{1/k} = \text{const}\ .$$

See Also

`difeqdem`

optknt

Purpose

Provide knot sequence optimal for interpolation

Syntax

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

Description

`t= optknt(tau,k)` provides the knot sequence `t` that is *best* for interpolation from $S_{k,t}$ at the point sequence `tau`. Here, *best* or *optimal* is used in the sense of [3] and [2], and this means the following: For any *recovery scheme* R that provides an interpolant Rg that matches a given g at the points `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 points `t(k+1)`, ..., `t(n)` and nowhere else satisfies

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

Gaffney/Powell called this interpolation scheme *optimal* since it provides the *center* function in the band formed by all interpolants to the given data that, in addition, have their kth derivative between M and $-M$ (for large M).

Examples

See the last part of the demo `spapidem` for an illustration.

Algorithm

This is the Fortran routine `SPLOPT` in *PGS*. It is based on an algorithm described in [1], for the construction of that sign function h mentioned in (**3**) above. It is essentially Newton's method for the solution of the resulting nonlinear system of equations, with `aveknt(tau,k)` providing the first guess for `t(k+1)`, ...,`t(n)`.

See Also

`spapidem`, `aveknt`

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.

pplst, splst

Purpose List available M-files for work with splines in ppform and in B-form

Syntax

```
pplst
splst
```

Description Provides a list of available M-files for work with pp functions in ppform or B-form.

See Also `ppalldm2`, `spalldm2`

Purpose

Put together a spline in ppform

Syntax

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

Description

`ppmak` puts together a pp function in ppform, from minimal information, with the rest inferred from the input. `fnbrk` returns the parts of the completed description. In this way, the actual data structure used for the storage of this form is easily modified without any effect on the various M-files using the construct.

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

The action taken by `ppmak` depends on whether the function is univariate or multivariate, as indicated by `breaks` being a sequence or a cell-array.

If `breaks` is a sequence (which must be nondecreasing, with its first entry different from its last), then the function is assumed to be univariate, and the various parts of its ppform are determined as follows:

1. The number `l` of polynomial pieces is determined as `l=length(breaks)-1`, and the basic interval is, correspondingly, the interval `[breaks(1) .. breaks(l+1)]`.
2. The order `k` and the dimension `d` of the function's target are inferred as follows:
   a. If the dimension `d` is not given explicitly, then `coefs(:,i*k+j)` is assumed to contain the `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 the dimension `d` is obtained from `[d,kl] = size(coefs)` and the order `k` of the pp is obtained as `k = fix(kl/l)`.
   b. If `d` is explicitly specified, then `coefs(i*d+j,:)` is assumed to contain the `j`th components of the coefficient vector for the `(i+1)`st polynomial piece. This corresponds to the format used internally, while the earlier format seems easier to handle when specifying such a pp explicitly. In particular, it is the format in which `fnbrk` returns the coefficient array, hence `d` must be explicitly specified when the input `coefs` is the result of a call to `fnbrk`.

If breaks is a cell array, of length m, then the function is assumed to be m-variate (tensor product), and the various parts of its ppform are determined from the input as follows.

1. The m-*vector* l has length(breaks{i})-1 as its *i*th entry and, correspondingly, the m-cell array of its basic intervals has the interval [breaks{i}(1).. breaks{i}(end)] as its *i*th entry.
2. The dimension d of the function's target and the m-vector *k* of (coordinate-wise polynomial) orders of its pieces are obtained directly from the size of coefs, and any third input argument is ignored.
   a. If coefs is an m-dimensional array, then the function is taken to be scalar-valued, i.e., d = 1, and the m-vector k is computed as size(coefs)./l. Then coefs is reshaped: coefs = reshape(coefs,[1,size(coefs)]).
   b. If coefs is an (m+1)-dimensional array, then d is taken to be size(coefs,1), and the *i*th entry of k is computed as size(coefs,i+1)/l(i), i=1:m.

The coefficient array is internally treated as an equivalent array of size [d,l(1),k(1),l(2),k(2),...,l(m),k(m)], with its (:,i(1),r(1),i(2),r(2),...,i(m),r(m)) entry the coefficient of

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

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

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

Examples

ppmak([0:2],[1:6]) constructs a pp function with basic interval [0..2] and consisting of two pieces of order 3, with the sole interior break 1. The resulting function is scalar, i.e., the dimension d of its target is 1. The function happens to be continuous at that break since the first piece is $x \mapsto x^2 + 2x + 3$, while the second piece is $x \mapsto 4(x-1)^2 + 5(x-1) + 6$.

When the function is univariate and the dimension d is not explicitly specified, then it is taken to be the row number of coefs. The column number should be an integer multiple of the number l of pieces specified by breaks. For example, the statement ppmak([0:2],[1:3;4:6]) leads to an error, since the break

sequence `[0:2]` indicates two polynomial pieces, hence an even number of columns are expected in the coefficient matrix. The modified statement `ppmak([0:1],[1:3;4:6])` specifies the parabolic curve $x \mapsto (1,4)x^2 + (2,5)x + (3,6)$. In particular, the dimension `d` of its target is 2. The differently modified statement `ppmak([0:2],[1:4;5:8])` also specifies a planar (i.e., `d = 2`) curve, but this one is piecewise linear; its first polynomial piece is $\mathbf{x} \mapsto (1,5)x + (2,6)$.

Explicit specification of the dimension `d` leads, in the univariate case, to a different interpretation of the entries of `coefs`. Now the column number indicates the polynomial order of the pieces, and the row number should equal `d` times the number of pieces. Thus, the statement `ppmak([0:2],[1:4;5:8],2)` is in error, while the statement `ppmak([0:2],[1:4;5:8],1)` specifies a scalar piecewise cubic whose first piece is $x \mapsto x^3 + 2x^2 + 3x + 4$.

See `ppalldm2` for other examples.

See Also

`fnbrk`, `ppalldm2`

slvblk

Purpose

Solve an almost block-diagonal linear system

Syntax

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

Description

`slvblk(blokmat,b)` returns the solution (if any) of the linear system `Ax = b`, with the matrix `A` stored in `blokmat` in the spline almost block diagonal form. At present, only the M-file `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 point.

If the system is overdetermined (i.e., has more equations than unknowns but is of full rank), then the least-squares solution is returned. In this case, the optional third argument, `w`, may be supplied to ensure that the solution minimized the *weighted* minimized the *weighted* $l_2\Sigma$

$\Sigma jw(j)*((A*x-b)(j))^2$

The right side `b` may contain several columns, and is expected to contain as many rows as there are rows in the matrix described by `blokmat`.

Examples

`sp=spmak(knots,slvblk(spcol(knots,k,x,1),y.'))` provides in `sp` the B-form of the spline `s` of order `k` with knot sequence `knots` that matches the given data `(x,y)`, i.e., satisfies `s(x)=y`.

Algorithm

The M-file `bkbrk` is used to obtain the essential parts of the coefficient matrix described by `blokmat` (in one of two available forms).

A QR factorization is made of each diagonal block, after it was augmented by the equations not dealt with when factoring the preceding block. The resulting factorization is then used to solve the linear system by backsubstitution.

See Also

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

Purpose

Locate points with respect to meshpoints

Syntax

```
index = sorted(meshpoints,points)
```

Description

Various M-files in this toolbox need to determine the index j for which a given x lies in the interval $[t_j \,..\, t_{j+1}]$, with (t_i) a given nondecreasing sequence, e.g., a knot sequence. This job is done by `sorted` in the following fashion.

The vector `index=sorted(meshpoints,points)` is the integer sequence for which, for all `j`, `index(j)` equals the number of entries in `meshpoints` that are ≤ `spoints(j)`, with `spoints=sort(points)`. Thus, if both `meshpoints` and `points` are nondecreasing, then

```
meshpoints(index(j)) ≤ points(j) < meshpoints(index(j)+1)
```

with obvious interpretations when

```
index(j) < 1 or length(meshpoints) < index(j) + 1
```

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([meshpoints(:).',points(:).'])` is used.

spap2

Purpose

Least-squares approximation by splines

Syntax

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

Description

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

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

in the weighted mean-square sense, with default weights equal to 1. If the abscissae `x` satisfy the (Schoenberg-Whitney) conditions

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

then there is a unique spline (of the given order and knot sequence) satisfying (`*`) exactly. No spline is returned unless (`**`) is satisfied for some subsequence of `x`.

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

Examples

```
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` is a strictly increasing sequence, with entries in `(a..b)`.

As another example, `spap2(augknt(x([1 end]),2), 2, x, y);` provides the least-squares straight-line fit to data `x`,`y` (assuming that `x(1)`< `x(i)` < `x(end)`), while

```
w = ones(size(x)); w([1 end])=100;
spap2(augknt(x([1 end]),2),2, x,y,w);
```

forces that fit to come very close to the first and last data point.

Algorithm

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

Gridded data is fitted tensor-product fashion one variable at a time, taking advantage of the fact that a univariate weighted least-squares fit depends linearly on the values being fitted.

See Also

`spapi`, `slvblk`, `spcol`

spapi

Purpose

Spline interpolation

Syntax

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

Description

Returns the spline f (if any) of order

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

with knot sequence `knots` for which

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

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

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

Examples

`spapi([0 0 0 0 1 2 2 2 2],[0 1 1 1 2],[2 0 1 2 -1])`produces the unique cubic spline f on [0..2] with exactly one interior knot, at 1, that satisfies the five conditions

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

Since the given ordinates are re-ordered in unison with the given abscissae if the latter are not already in decreasing order, it is, e.g., possible to carry out interpolation to values y and slopes s at the increasing point sequence x by a quintic spline by the command

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

with `ddy0` and `ddy1` values for the second derivative.

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

See Also

spaps, spap2, csapi, spline

Limitations

The given (univariate) knots and abscissae must satisfy the S*choenberg-Whitney conditions* for the interpolant to be defined. Assuming the abscissa sequence x to be nondecreasing, this means that we must have

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

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

Cautionary Note

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

spaps

Purpose

Smoothing spline

Syntax

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

Description

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

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

$$F(D^{\mathrm{m}}f) \quad = \quad \int_{\mathrm{x(1)}}^{\mathrm{x(n)}} (D^{\mathrm{m}}f)^2$$

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

$$E(f) \; = \; \sum_{j\,=\,1}^{\mathrm{n}} \mathrm{w}(j)(\mathrm{y}(j) - f(\mathrm{x}(j)))^2$$

The default value for `m` is `2`, leading to the cubic smoothing spline. However, the choices `m=1` and `m=3`, for the linear, respectively the quintic, smoothing spline are available, too, by setting one of the optional inputs `argi` equal to `1` or `3`. Further, the default value for the weight vector `w` makes $E(f)$ the composite trapezoidal rule approximation to $\int_{\mathrm{x(1)}}^{\mathrm{x(n)}} (y-f)^2$ but the weight vector may also be supplied as one of the optional inputs `argi` (as a positive vector of the same length as `x`).

If `x` is not increasing, then both `x` and `y` (as well as `w` if given) will be re-ordered in unison to make `x` increasing. After that, `x` must be strictly increasing.

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

It is also possible to obtain a smoothing spline for *gridded data*. When `x` is a cell array of length `r`, then `y` is expected to supply the corresponding gridded

values, with `size(y)` equal to `[length(x{1}),...,length(x{r})]` in case the function is scalar-valued, and equal to `[d,length(x{1}),...,length(x{r})]` in case the function is `d`-vector-valued. Further, the optional input for `m` must be an `r`-vector (with entries from the set `{1,2,3}`), and the optional argument for `w` must be a cell array of length `r`, with `w{i}` either empty (to indicate that the default choice is wanted) or else a positive vector of the same length as `x{i}`.

Examples

The statements

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

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

Algorithm

Reinsch's approach [1] is used (including his clever way of choosing the equation for the optimal smoothing parameter in such a way that a good initial guess is available and Newton's method is guaranteed to converge and to converge fast).

See Also

`spap2`, `spapi`, `csaps`

References

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

spcol

Purpose

Generate the B-spline collocation matrix

Syntax

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

Description

`spcol` constructs the matrix

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

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

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

If one of the optional arguments is a string with the same first two letters as in `'slvblk'`, the matrix is returned in the almost block-diagonal format (specialized for splines) required by `slvblk` (and understood by `bkbrk`).

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

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

Examples

The statement `spcol([1:6],3,.1+[2:4])` provides the matrix

```
ans =

    0.5900   0.0050        0
    0.4050   0.5900   0.0050
         0   0.4050   0.5900
```

in which the typical row records the values at 2.1, or 3.1, or 4.1, of all B-splines of order 3 for the knot sequence `[1:6]`. There are three such B-splines. The first one has knots 1,2,3,4, and its values are recorded in the first column. In particular, the last entry in the first column is zero since it gives the value of that B-spline at 4.1, a point to the right of its last knot.

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

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

```
block 1 has 2 row(s)
    0.5900   0.0050        0
    0.4050   0.5900   0.0050
next block is shifted over 1 column(s)
block 2 has 1 row(s)
    0.4050   0.5900   0.0050
next block is shifted over 2 column(s)
```

Algorithm

This is the most complex M-file in this toolbox since it has to deal with various ordering and blocking issues. The recurrence relations are used to generate, in lockstep, 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) points at which derivative values are required. These are the points `tau(i)` with m(`i`) > 0. For these, and for every order `k` -j, $j = j_0, j_0$-1, ...,0, with j_0 :=`max`(m), values of all B-splines of that order are generated by recurrence and used to compute the jth derivative at those points 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 point `tau(i)`, at most `k` B-splines of order `k` are nonzero. This fact (together with the natural ordering of the B-splines) implies that the collocation matrix has a staircase shape, with the individual blocks or steps of varying height but of uniform width `k`.

The M-file `slvblk` is designed to take advantage of this storage-saving form available when determining the B-form of a pp function from interpolation or other approximation conditions.

See Also

`slvblk`, `spapi`, `spap2`

Limitations

The sequence `tau` is assumed to be nondecreasing.

spcrv

Purpose

Generate a spline curve

Syntax

```
spcrv(c)
curve = spcrv(c,k,maxpnt)
```

Description

`spcrv(c,k)` provides a dense sequence f(`tt`) of points on the uniform B-spline curve f of order `k` with B-spline coefficients `c`. Explicitly, this is the curve

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

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

The default value for `k` is 4. The default value for the maximum number of points `tt` to be generated is 100.

The parameter interval that the point 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` points. There are situations where use of `fnplt` would be more efficient.

See Also

`spcrvdem`, `fnplt`

Purpose

Run some demos

Syntax

```
spdemos
```

Description

A list of available demonstration programs is offered for execution. Here is the list:

`splexmpl`	some simple examples
`bsplidem`	some B-splines
`csapidem`	cubic spline interpolation
`csapsdem`	cubic smoothing spline
`ppalldm2`	introduction to ppform
`spalldm2`	introduction to B-form
`pckkntdm`	choice of knots
`spcrvdem`	spline curve construction
`difeqdem`	a singularly perturbed ODE
`chebdem`	an equi-oscillating spline
`tspdem`	tensor product

Some of these demonstrations make use of the truncated power function

$$x \mapsto (x)_+ := \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

provided by `subplus`, and of the titanium heat data provided by `titanium`. The Chebyshev spline demo uses `chebloop`, the nonlinear ODE demo uses `difeqite` and `difeqset`, and the tensor product demo uses the following test function provided by `franke`.

$$\begin{aligned} x \mapsto\ & (3/4)\exp(-\|9x-(2,2)\|^2/4) \\ & +(3/4)\exp(-(9x(1)+1)^2/49-(9x(2)+1)/10) \\ & +(1/2)\exp(-\|9x-(7,3)\|^2/4) \\ & -(1/5)\exp(-(9x(1)-4)^2-(9x(2)-7)^2) \end{aligned}$$

splpp, sprpp

Purpose

Convert locally from B-form to ppform

Syntax

```
[v,b] = splpp(tx,a)
[v,b] = sprpp(tx,a)
```

Description

These are utility M-files of use in the conversion from B-form to ppform (and in certain evaluations), but of no interest to the casual user.

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

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

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

Examples

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

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

with *s* the B-spline with knots -2, -1, 0, 1. This is so because the `l` in `splpp` indicates the limit from the left, and the second argument, `[0 1 0]`, indicates that the spline *s* in question be

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

i.e., this particular linear combination of the third-order B-splines for the knot sequence ..., -2, -1,0,1,... (Note that the values calculated do not depend on the knots marked ?.) The above statement also provides the sequence `b = 0 1.0000 0.5000` of B-spline coefficients for the polynomial piece of *s* on the interval [-1. .0], and with respect to the knot sequence ?, -2, -1, 0, 0, ?.

In other words, on the interval [-1. .0], the B-spline with knots 2, -1, 0, 1 can be written

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

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

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

with s the B-spline with knots ?,-1,0,1. Its polynomial piece on the interval [0. .1] is independent of the choice of ?, so we might as well think of ? as -2, i.e., we are dealing with the same B-spline as before. Note that the last two numbers agree with the limits from the left computed above, while the first number does not. This reflects the fact that a quadratic B-spline with simple knots is continuous with continuous first, but discontinuous second, derivative. (It also reflects the fact that the left-most knot of a B-spline is irrelevant for its right-most polynomial piece.) The sequence `b = 0.5000 0 0` also provided states that, on the interval [0. .1], the B-spline $B(\cdot\,|\,[?, -1,0,1])$ can be written

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

Cautionary Note

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

spmak

Purpose

Put together a spline in B-form

Syntax

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

Description

`spmak` puts together a spline function in B-form, from minimal information, with the rest inferred from the input. `fnbrk` returns all the parts of the completed description. In this way, the actual data structure used for the storage of this form is easily modified without any effect on the various M-files using the construct.

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

The coefficients may be `d`-vectors (e.g., 2-vectors or 3-vectors), in which case the resulting spline is a curve or surface (in 2-space or 3-space).

The action taken by `spmak` depends on whether the function is univariate or multivariate, as indicated by `knots` being a sequence or a cell-array.

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

Knot multiplicity is held to be ≤ `k`, with the coefficients corresponding to a B-spline with trivial support ignored.

If `knots` is a cell array, of length `m`, then the spline is taken to be `m`-variate, and `coefs` must be an `(m+1)`-dimensional array, – except when the spline is to be scalar-valued, in which case, in contrast to the univariate case, `coefs` is permitted to be an `m`-dimensional array, but this array is immediately reshaped by

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

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

Examples

`spmak([1:6],[0:2])` constructs a spline function with basic interval [1. .6], with 6 knots and 3 coefficients, hence of order 6 - 3 = 3. `spmak(t,1)` provides the B-spline $B(\cdot \mid t)$ in ppform. See `spalldm2` for other examples.

See Also

`spbrk`, `spalldm2`

Diagnostics

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

Limitations

It appears that, in MATLAB v.5.2, the size of a multidimensional array created by the statement `reshape(1,[1,1,1,...,1])` will be reported as `[1 1]`. This means that the B-form created by the statement `sp = spmak(knots,1)`, with `knots` a cell array of length `m`>2, will not be interpreted correctly, by `fnval` and other commands, as the tensor-product B-spline for the given knots.

Index

J

K

L

M

N

O

P

Q

R

S

T

U

V