On the evaluation of box splines

Carl de Boor
University of Wisconsin–Madison

The first (and for some still the only) multivariate B-spline is what today one would call the simplex spline, since it is derived from a simplex, and in distinction to other polyhedral splines, such as the cone spline and the box spline. The simplex spline was first talked about in 1976. However, it was only after Micchelli [Micchelli, 1980] established recurrence relations for them that the topic of simplex splines and other multivariate B-splines really took off. Their cousins, the box splines, were thought particularly attractive because their recurrence relations turned out to be very simple indeed. It was, therefore, a shock to me when, in the process of doing my bit on the book [de Boor, Höllig, Riemenschneider, 1993], I found that it was nontrivial to make effective use of these recurrence relations. It is one purpose of this note to relate my difficulties and how I tried to deal with them.

This is not the place to give an introduction to box spline theory, nor to review the relevant literature. Rather, the reader is urged to consult [de Boor, Höllig, Riemenschneider, 1993] for missing details and the proper literature references (as well as for many illustrations, two of which are reproduced below).

1 Box splines defined

The s-variate box spline $M_\Xi := M(\cdot|\Xi)$ is defined in terms of its direction matrix $\Xi \in \mathrm{IR}^{s\times n}$, $\Xi(:,j) \in \mathrm{IR}^s\backslash 0$, $j = 1,\ldots,n$, as the distribution or linear functional given by the rule

$$M_\Xi : C(\mathrm{IR}^s) \to \mathrm{IR} : \varphi \mapsto \langle M_\Xi, \varphi\rangle := \int_{\square} \varphi(\Xi t)dt.$$

Here,

$$\square := [0\,..\,1)^n$$

is the half-open unit cube or 'box' in IR^n.

The distribution M_Ξ is always representable as an L_∞-function on $\operatorname{ran}\Xi$. In particular, if Ξ is of full rank, *i.e.*, $\operatorname{ran}\Xi = \mathrm{IR}^s$, then M_Ξ is (represented by) an $L_\infty(\mathrm{IR}^s)$-function, and this function is piecewise polynomial (=: **pp**), of exact degree $n-s$, with support the compact, convex set $\Xi\square$. The **breakplanes** of this pp function are given by certain translates of the hyperplanes spanned by $s-1$ independent columns of Ξ. Particularly simple examples include the **Courant element**

$$\mathbf{C} := M_{\left[\begin{smallmatrix}1&0&1\\0&1&1\end{smallmatrix}\right]},$$

associated with the direction matrix $\left[\begin{smallmatrix}1&0&1\\0&1&1\end{smallmatrix}\right]$, and the **Zwart-Powell element**

$$\mathbf{ZP} := M_{\left[\begin{smallmatrix}1&0&1&-1\\0&1&1&1\end{smallmatrix}\right]},$$

associated with the direction matrix $\left[\begin{smallmatrix}1 & 0 & 1 & -1\\ 0 & 1 & 1 & 1\end{smallmatrix}\right]$. See the figure below for a picture of the support of these two box splines.

The above definition of the box spline is referred to as the **analytic** definition, to distinguish it from the following (equivalent) **inductive** definition which some prefer because of its intuitive appeal and because it often eases proofs. Here, one starts either with

$$M_\Xi = \chi_{\Xi\,\square}/|\det\,\Xi|, \qquad \Xi \in \mathbb{R}^{s\times s}$$

or with

$$\langle M_{[\,]}, \varphi\rangle := \varphi(0), \qquad \forall\varphi \in C(\mathbb{R}^s)$$

and builds up from there via

$$M_{\Xi\cup\zeta} = \int_0^1 M_\Xi(\cdot - t\zeta)dt.$$

For example, with

$$\mathbf{B} := M_{\left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right]} = \chi_{[0..1)^2}$$

the characteristic function of the unit square in $\mathbb{R}^2$, one obtains from it the Courant element by

$$\mathbf{C} = \int_0^1 \mathbf{B}(\cdot - t\zeta)dt, \qquad \zeta := (1,1),$$

and, from this, the ZP element by

$$\mathbf{ZP} = \int_0^1 \mathbf{C}(\cdot - t\xi)dt, \qquad \xi := (-1,1),$$

as illustrated in the following figure.

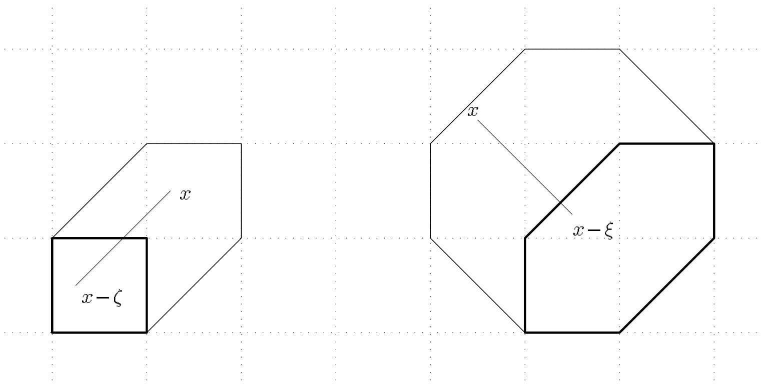

Figure. Courant and ZP by convolution

2 Recurrence relations

Here is the formal statement of the recurrence relations for box splines.

Proposition. *If the box splines* $M_{\Xi\backslash\xi}$, $\xi \in \Xi$, *are continuous at* $x = \Xi t = \sum_{\xi\in\Xi} \xi t_\xi$, *then*

$$(n-s)M_\Xi(x) \;=\; \sum_{\xi\in\Xi} t_\xi M_{\Xi\backslash\xi}(x) \;+\; (1-t_\xi)M_{\Xi\backslash\xi}(x-\xi).$$

The following figure, from [de Boor, Höllig, Riemenschneider, 1993], indicates the various box splines of degree 1 and 0 which, in this way, participate in the evaluation of the ZP element at any point in the indicated triangle in its support.

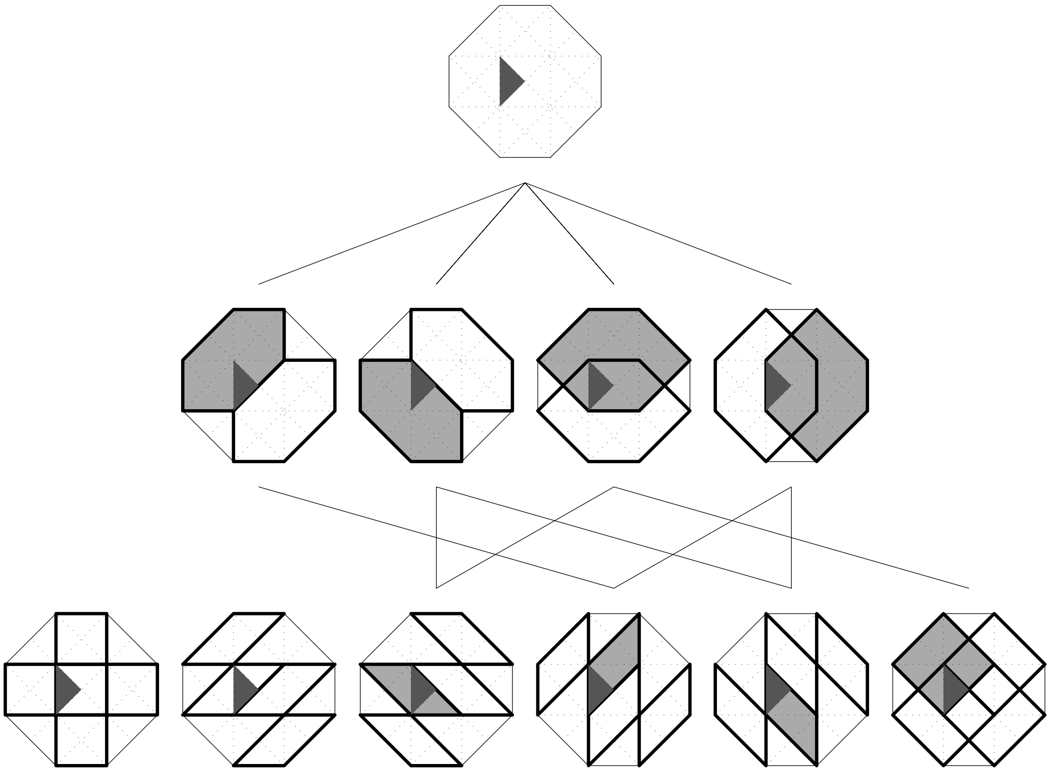

Figure. The ZP element obtained by recurrence

Here are the issues, perhaps minor, which one has to deal with if one wants to convert the above formal statement into a program for the evaluation of $M_\Xi(x)$ for given Ξ and x. For definiteness, I only consider a program written in `MATLAB`. This is attractive since, in this way, one can ignore various computing details. Also, `MATLAB` permits recursion, hence makes it possible to write the program in the recursive spirit of the above formal statement. Finally, `MATLAB` invites use of 'parallel processing', *e.g.*, the simultaneous evaluation of M_Ξ at a sequence of points, hence a `MATLAB` program can be used to advantage when writing programs for machines with parallel architecture. For all `MATLAB`-related detail, the reader is referred to the `MATLAB` documentation, *e.g.*, to [Mathworks, 1989].

A *first issue* arises because there are many t for which $\Xi t = x$. How is one to select a particular t?

One might try for a t with as many zero entries as possible, as that would minimize the number of evaluations of lower-degree box splines needed. However, since both t_ξ and $(1-t_\xi)$ appear as weights in the formula, we would, even with such a choice, have to evaluate each of the $M_{\Xi\backslash\xi}$ in any case at least once. Therefore, since we intend to set up the routine to evaluate simultaneously at a set of points, there is no real savings associated with having many zero entries in t.

It seems more important, if only for numerical stability, to make certain that none of the entries of t is unduly large. Ideally, both t_ξ and $(1 - t_\xi)$ ought to be nonnegative for all ξ. However, it seems sufficient to make certain that t be 'small'.

For this, we compute t as the ℓ_2-smallest solution to $\Xi? = x$, using the QR factorization of Ξ readily supplied by `MATLAB`.

A *second issue* concerns the case not dealt with in the proposition, namely, what if $M_{\Xi\backslash\xi}$ fails to be continuous at x? This issue is closely related to the question of just when to stop the recurrence, *i.e.*, which M_Ξ to evaluate directly, without further recurrence.

Offhand, since the recurrence relation only holds under the assumption that all $M_{\Xi\backslash\xi}$ be continuous at x, one would have to stop as soon as this condition is not met. However, since even high-degree box splines may be discontinuous because of high multiplicities in their direction set, this would make it impossible to write a general-purpose program.

One remedy proposed has been to define $M_\Xi(x)$ as some average value in case M_Ξ is discontinuous at x, and, in this way, to extend the validity of the recurrence relations to all $x \in \mathbb{R}^s$. However, in spite of some effort, I was unable to make this work even in the simplest possible case in which the direction set is in general position, hence M_Ξ has discontinuities only when $n = s$.

At the same time, all box splines are pp, hence have well-defined limiting values even at points of discontinuity. Thus, the restriction to the case of having all $M_{\Xi\backslash\xi}$ continuous at x is only there to avoid a certain discussion needed when this condition is not met. This discussion concerns just how to choose and enforce a particular direction along which this limiting value is to be taken. (In the univariate case, it has become customary to make all splines continuous from the right, except, perhaps, at the right-most point of the interval of interest.) Since any discontinuity of M_Ξ necessarily lies on one of its breakplanes, hence the normal to such a hyperplane is perpendicular to $s-1$ independent columns of Ξ, one must choose this direction so as not to fall into the span of any $s-1$ independent columns of Ξ. Since this only excludes the vectors in the union of certain finitely many hyperplanes, such a choice is not hard to make. Thus, to resolve this issue, we agree to enlarge the argument list to our `MATLAB` function to include some properly chosen fixed direction, to be used as indication of the desired limiting direction whenever, in a terminating call to the function, a decision has to be made on which side of a hyperplane of discontinuity the current argument is supposed to lie. That done, we are now safe to use the recurrence down to the level at which M_Ξ is a (multiple of a) characteristic function, *i.e.*,

$$M_\Xi = \chi_{\Xi\square}/|\det \Xi|, \qquad \Xi \in \mathbb{R}^{s\times s}.$$

For, its value at x is trivial to determine: In this case, Ξ is square, hence (assuming that Ξ is of full rank) the equation $\Xi? = x$ has a unique solution, t, and x lies in $\Xi\square$ exactly when $t \in \square$. Thus, if $t_\xi \in (0\,..\,1)$ for all $\xi \in \Xi$, then the value is $1/|\det \Xi|$. If some $t_\xi \notin [0\,..\,1]$, then the value is 0. Finally, if some t_ξ equals 0 or 1, then x lies on one of the breakplanes of M_Ξ, and the decision of whether or not M_Ξ vanishes at x is made with the aid of the given fixed direction: if the direction, when attached to x, points into $\Xi\square$, then $M_\Xi(x) = 1/|\det \Xi|$, otherwise $M_\Xi(x) = 0$.

This raises the *final issue*, of what to do in case Ξ is not of full rank, as is certain to happen in case of repeated directions. It is also not clear, offhand, how to decide that one is in this case., *i.e.*, how to determine rank Ξ. However, having decided to use the QR factorization of Ξ to solve $\Xi? = x$, that factorization is now handy for settling whether or not Ξ is of full rank. If it is not, then we know that M_Ξ, as a linear functional or distribution, is supported only on a certain hyperplane, hence, as a function on $\mathbb{R}^s$, it is zero. Thus, we would set up our `MATLAB` function to return the value 0 in this case, even if Ξ is not yet square.

3 A failure

I won't list here the `MATLAB` function written along the lines of the preceding discussion. For, even for such simple box splines as the Courant and the ZP elements, it produced terrible output, as the following figure indicates.

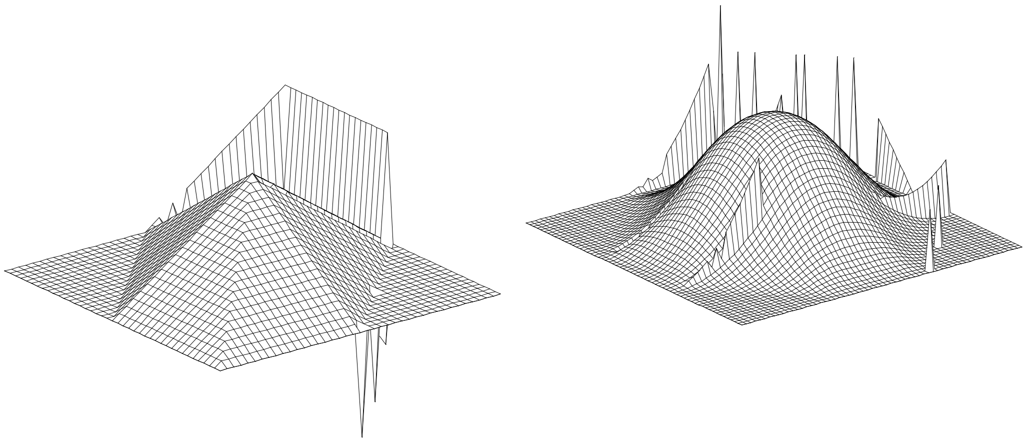

Figure. Faulty values for Courant and ZP obtained with the best intentions.

What went wrong? Simply put: The evaluation ultimately involves step functions, and there is the

Sad Fact. *Step functions are not computable.*

To recall, a computable function has the property that, given more and more digits of the argument, we can, perhaps with increasing effort, compute more and more digits of the value of the function at that argument. In particular, any computable function must be continuous.

The problem turns out to be a major issue in all of geometric modeling, as such simple decisions as, whether or not a given point is inside or outside a given body, or whether or not two bodies intersect, can, strictly speaking, not be made in finite-precision arithmetic.

This conclusion is a bit shocking to someone working with splines, since splines employ step functions in their very definition. Why has this not been recognized as an issue before? While this problem arose already when the recurrence relations for the simplex splines were first programmed, it was not noticed in the univariate setting, since the only way to settle it worked in that setting very nicely. The only way to deal with the above Sad Fact is by fiat. This means that, for an argument 'near' the boundary of the set whose characteristic function is to be evaluated, one uses some arbitrary rule, such as continuity from a certain direction.

Now, that is exactly what we already tried. However, the technical difficulty with such arbitrary rules is to make them consistent. In the case of the box spline recurrence, the decision on which side of a breakplane the given argument lies is being made independently more than once. For example, with reference to the figure showing the recurrence used to evaluate the ZP element, assume that the argument lies on the lower left vertex of the triangle singled out. Then, for the particular choice of t, there are four step function evaluations involved and, in each of them, such a decision has to be made, and made independently, given the very nature of a recurrence.

There is no such difficulty in the univariate case precisely because the decision in which knot interval a given argument lies is made once, and only then does the evaluation commence. In fact, the standard algorithm does not really evaluate a spline. Rather (as is pointed out, *e.g.*, in [de Boor 1978: p. 136]), it provides, for given argument, the value at that argument of the polynomial pointed to by the choice of index. The standard choice of the index only makes certain that the polynomial so selected is the one with which the given spline agrees on the knot interval which contains the given argument.

The analogous procedure in the present case would require one to decide at the outset on which side of any relevant hyperplane the given x lies. However, this is much harder to do than in the univariate case, since it is nontrivial to locate a given x with respect to all relevant hyperplanes generated by $s-1$ columns from Ξ. In fact, which hyperplanes become relevant for the given x becomes obvious only when step functions are to be evaluated.

For this reason, the `MATLAB` M-file `bxrec` (listed in the Appendix) which realizes the recurrence relations employs the following compromise: It returns, for the given arguments, not only a corresponding list of values, but also an indication of which arguments (if any) were found to lie on some breakplane. These are identified by the fact that a call to `bxrec` so labeled them, hence, ultimately, by the fact that, during evaluation of one of the box splines of degree 0, the corresponding argument was found to lie, within a certain 'small' tolerance, on the boundary of the corresponding parallelepiped. Now, the `MATLAB` M-file `bxval` (also listed in the Appendix), which initiates the first call to `bxrec`, simply makes a new attempt at evaluation at each argument x so labeled, using, instead of x, an argument perturbed by a slightly less 'small' tolerance, thereby identifying unambiguously just which of the possibly many polynomial pieces one actually wishes to evaluate 'at' x.

One could have avoided the return of two pieces of information from `bxrec` by simply returning the value `NaN` (not-a-number) whenever the value of a characteristic function is in doubt. However, this leads to repeated testing for the same information.

For the record, here is a figure computed with the aid of these `MATLAB` M-files. It shows the level lines for the ZP element so computed, as that is a much more sensitive indicator of trouble than a plot of the ZP element itself.

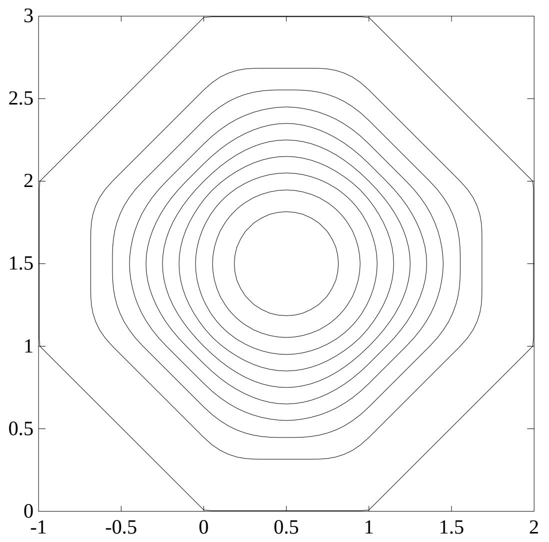

Figure. Contour lines for the correctly evaluated ZP element.

4 Computational cost

It becomes quickly apparent that these `MATLAB` functions are only useful for rather simple box splines, since the computational cost of using the recurrence grows very fast with $n - s$. Precisely, the number of recurrent calls needed for evaluation at one point is

$$C(n, s) := 2n * (2(n-1)) * (2(n-2)) * \cdots * (2(s+1)) = 2^{n-s} n!/s!.$$

E.g., for $s = 2$:

n:	2	3	4	5	6	7	8	9
$C(s, n)$:	1	6	48	480	5,760	80,640	1,290,240	23,224,320

Even if one evaluates 'in parallel', as does `bxrec`, the number of required calls still is

$$C_p(n, s) := n * (n-1) * (n-2) * \cdots * (s+1) = n!/s!,$$

giving, for $s = 2$:

n:	2	3	4	5	6	7	8	9
$C_p(s, n)$:	1	3	12	60	360	2,520	20,160	181,440

5 Using subdivision instead

The cost of using recurrences makes it interesting to consider alternatives. The most widely used is based on the idea of subdivision, as first described in [Cohen, Lyche, Riesenfeld, 1984] and [Dahmen, Micchelli, 1984].

The basic idea is quite geometric. Since the unit box $\square$ can be subdivided into $(1/h)^n$ boxes of sidelength h, the box spline

$$M := M_\Xi$$

is a certain linear combination of scaled and shifted versions of itself. This gives the so-called **refinement equation**

$$M(x) \;=\; \sum_{j\in h\mathbb{Z}^s} M((x-j)/h)\, m^h(j),$$

and the mesh function m^h so defined (on the scaled mesh $h\mathbb{Z}^s$) is the corresponding **mask**.

The mask is nonzero only for those $j \in h\mathbb{Z}^s$ for which the corresponding scaled and shifted box spline, *i.e.*, the function $M((\cdot - j)/h)$, has its support entirely in the support of M, *i.e.*, in $\Xi\square$. Further, the mask converges to M as $h \to 0$, in the sense that

$$M(hc+j) \;\sim\; m^h(j), \qquad j \in h\mathbb{Z}^s,$$

with

$$c := c_\Xi \;:=\; \Xi\,(1,1,\ldots,1)/2$$

the center of the support of $M = M_\Xi$, and with the error usually of order h^2, uniformly in j. Moreover,

$$m^h \;=\; b^h_\xi * b^h_\eta * \cdots * \delta/h^s,$$

with the star indicating (discrete) convolution, with $\xi, \eta, \ldots$ the directions in $\Xi \in \mathbb{Z}^{s\times n}$, and, *e.g.*,

$$b^h_\xi * a \;:=\; h \sum_{k=0}^{1/h\,-1} a(\cdot - kh\xi),$$

and δ the Kronecker Delta, *i.e.*, $\delta(x) = 0$ for all x, except that $\delta(0) = 1$. Finally, if $f = \sum_{j\in\mathbb{Z}^s} M(\cdot - j)a(j)$, then also

$$f = \sum_{j\in h\mathbb{Z}^s} M((\cdot - j)/h)a^h(j),$$

with

$$a^h \;=\; m^h * a : j \mapsto \sum_{k\in\mathbb{Z}^s} m^h(j-k)a(k),$$

hence

$$f(hc+j) \;\sim\; a^h(j), \qquad j \in h\mathbb{Z}^s.$$

Thus the spline function f is being approximated, usually to within $O(h^2)$, by the multilinear interpolant of the mesh function a^h.

To my surprise, I found it nontrivial to implement the discrete convolutions involved here even in the bivariate case. The technical difficulty stems from the desire to make efficient use of the fact that the various masks involved have finite support. Thus, to save others some time perhaps, the Appendix also contains a listing of five `MATLAB` M-files for carrying out bivariate discrete convolution, with some of these specifically designed to construct discrete box splines.

The M-file `convol` computes the convolution product of two arbitrary masks. Its specialization, `condir`, convolves a given mask with one of the special masks b^h_ξ, taking advantage of the fact that the latter have very simple structure. `condir` is used in `msxima` to compute m^h for given Ξ and $\mathtt{nh} := 1/h \in \mathbb{N}$.

Finally, `msmak` and `msbrk` are used to make the other M-files independent of just exactly how the information about a mask is stored.

For accuracy's sake, the M-files work, in effect, with integers only and are in any case constrained to work with matrices, i.e., mesh-functions indexed by $j \in \mathbb{Z}^2$, while m^h is indexed by $j \in h\mathbb{Z}^2$. Also, as already mentioned, the M-files try to record only the nontrivial part of the masks. Thus, m^h is represented in the form

$$m^h(hj) \quad = \quad \mathtt{mask}(j{+}\mathtt{z})\, h^{n-s},$$

with `z` so chosen that the matrix `mask` has as few rows and columns as possible, yet still provides the entire nontrivial part of m^h.

Thus, if `m1 = msxima([1 0; 0 1], nh)` and `m2 = msxima([1 -1; 1 1], nh)`, then `convol(m1, m2)` should give the same result as `msxima([1 0 1 -1; 0 1 1 1], nh)`.

As a more elaborate example, the following `MATLAB` statements will generate the nontrivial values of $m^{1/8}$ for the ZP element.

```
xi = [1 0 1 -1; 0 1 1 1];
[s,n] = size(xi);
nh = 8;
[z,rc,mask] = msbrk(msxima(xi,nh));
% Recall that m^h(j) corresponds to the value of M at hc+j,  for
%  j  in  h Z^s , with  h := 1/nh, and
c = xi*ones(n,1)/2;
% Also, the array  mask  describes   m^h  in the sense that
%               m^h(h*j)  =  mask(j+z)/(nh^n h^s) ,
% with  z  the ‘center’ also supplied by  msbrk .  Thus,
%        mask(j+z) h^(n-s)  =  m^h(h j)  approx.   M(h (c+j))
% or
%              mask(j) h^(n-s)  approx.   M(h(c-z+j))
% So, we’ll evaluate the box spline at this mesh, i.e., at the
% following pointset, and compare.

[nx,ny] = size(mask);
xx = ((c(1)-z(1)+[1:nx])/nh)';  yy = (c(2)-z(2)+[1:ny])/nh;
xxx = xx(:,ones(1,ny));         yyy = yy(ones(nx,1),:);
values = reshape(bxval(xi, [xxx(:),yyy(:)]'), nx,ny);
differ = values - mask/nh^(n-s);

max(max(abs(differ))) - ((1/nh)/2)^2,  mesh(abs(differ))
```

I do not bother to plot the resulting approximations (*i.e.*, the entries of the matrix `mask` h^{n-s}) since they agree with the exact values (recorded in `values`) to better than graphical accuracy. Rather, here is a plot of the absolute error in these values (recorded in `differ`).

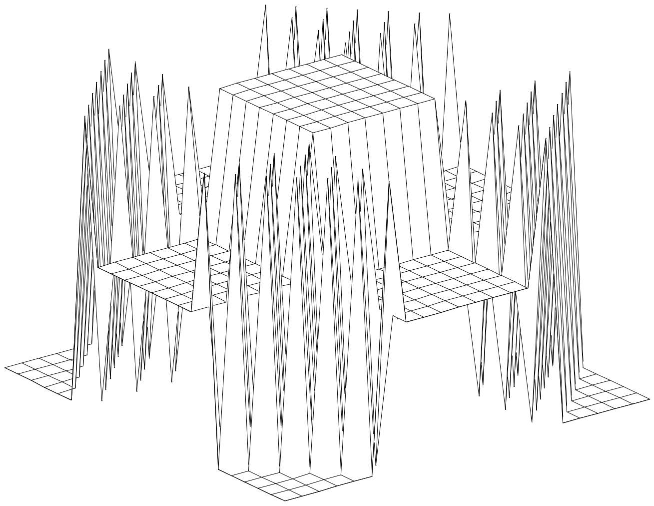

Figure. The absolute error in the values for the ZP element obtained by subdivision, with $h = 1/8$.

Perhaps surprisingly, the error is quite systematic, its absolute maximum equal to $(h/2)^2$ (plus noise). One concludes that extrapolation to the limit (as already recommended in [Dahmen 1987]) should be very effective.

6 Other approaches

While the subdivision approach is the most popular, the use of FFT and its inverse, as is advocated in [Jetter, Stöckler, 1991], might be even faster. Both of these approaches are efficient because, by computing values at a fine grid, the cost per function value can be kept low. Hence, they are of interest only if one needs function values at all the points of a rectangular grid. In the case of subdivision, the grid, as we saw, is even tied to the lattice generated by the columns of Ξ.

Rong-Qing Jia, Sherman Riemenschneider and Dennis Wong have found that, for the box splines on the three- and four-direction mesh, it is possible to reduce significantly the cost of using the recurrence relation, by making explicit use of the details of such very simple direction sets.

For the very same box splines, one finds in [Chui, 1988] and [Lai 1992] means for the determination of their BB-net, *i.e.*, the Bernstein–Bézier form of their polynomial pieces. It seems tempting to develop a general program, based on the inductive definition of the box spline, which uses discrete convolution to build up the BB-net of a box spline starting from the trivially obtainable BB-nets for a piecewise constant box spline. Of course, such a program would have to be restricted to box splines with triangular, or, more generally simplicial, polynomial pieces.

In [Dæhlen, 1989], box splines are evaluated by using simplex splines of one dimension lower. In particular, for bivariate box splines, this brings to bear the standard algorithms for the evaluation of univariate B-splines.

Finally, [Cavaretta, Dahmen, Micchelli, 1991] (and others) propose to use the refinement equation for box splines (or, for that matter, for any function satisfying a refinement equation) to compute the values of the box spline on the mesh $\mathbb{Z}^s/2^k$ from those on the (coarser) mesh $\mathbb{Z}^s/2^{k-1}$. This is possible since, for any $x \in \mathbb{Z}^s/2^k$, all the points $(x-j)/h$ appearing in the refinement equation

$$M(x) \;=\; \sum_{j\in h\mathbb{Z}^s} M((x-j)/h)\, m^h(j)$$

with $h = 1/2$ are in the coarser mesh $\mathbb{Z}^s/2^{k-1}$. Indeed, for such x and h and any $j \in h\mathbb{Z}^s$,

$$(x-j)/h = 2x - 2j \in 2\mathbb{Z}^s/2^k - 2\mathbb{Z}^s/2 = \mathbb{Z}^s/2^{k-1},$$

at least for positive k. This only requires some initial calculation of the values of M on $\mathbb{Z}^s$. For this, [Cavaretta, Dahmen, Micchelli, 1991: p. 18] recommend forming from the refinement equation the following system of equations

$$M(\alpha) \;=\; \sum_{\beta\in\mathbb{Z}^s} M(2\alpha-\beta)\, m^h(\beta/2), \quad \alpha \in \mathbb{Z}^s$$

and solving it for the vector $M_{|\mathbb{Z}^s}$. Since M has compact support, and the support is even known, this is easily written as an ordinary matrix eigenvalue problem, by restricting α, β to some set K, the only requirement being that K contain the support of $M_{|\mathbb{Z}^s}$. Moreover, the eigenvalue involved, 1, can be shown to be simple, hence the corresponding eigenvector is uniquely determined up to scaling. Thus, replacing any one of the equations in this system by a normalizing equation, such as

$$1 = \sum_{\beta\in K} M(\beta),$$

gives a linear system whose unique solution contains the nontrivial part of $M_{|\mathbb{Z}^s}$.

Actually, the same argument could have been made for $M_{|\mathbb{Z}^s/2^k}$ for any $k \in \mathbb{N}$, with only the normalizing constant changing, from 1 to 2^k, and, of course, the system now reading

$$M(\alpha) \;=\; \sum_{\beta\in\mathbb{Z}^s/2^k} M(2\alpha-\beta)\, m^h(2^{k-1}\beta), \quad \alpha \in \mathbb{Z}^s/2^k.$$

However, the linear system to be solved is full, hence even for $k = 0$ there are limits imposed by machine size. E.g., with $M_{r,s,t}$ the typical box spline for the three-direction mesh (the numbers r, s, and t specifying the multiplicities of the three directions in Ξ) and a straightfoward `MATLAB` program, `PC-Matlab` can handle in this way $M_{4,4,4}$ but cannot handle $M_{5,5,5}$, and a DEC 5000 trying for $M_{15,15,15}$ in `Pro-Matlab`, ran out of memory while in the process of assembling the matrix of the linear system. For comparison, the programs developed by Jia, Riemenschneider and Wong (mentioned earlier) can obtain all the grid values of $M_{20,20,20}$ in 15 seconds (albeit in `C` rather than `MATLAB`, and on an HP workstation).

As is made clear in [Dahmen, Micchelli, 1993], the same approach provides fast calculations even for integrals of products of derivatives of box splines, although it requires more work to end up with the desired vector belonging to a simple eigenvalue.

Acknowledgment

The work reported on was supported by the National Science Foundation under Grant No. DMS-9000053 and by the United States Army under Contract No. DAAL03-90-G-0090.

References

C. de Boor, 1978: *A Practical Guide to Splines*. Springer Verlag, New York.

C. de Boor, K. Höllig, S. D. Riemenschneider, 1993: *Box Splines*. Springer-Verlag, Berlin.

A. S. Cavaretta, W. Dahmen, C. A. Micchelli, 1991: Stationary Subdivision. *Mem. Amer. Math. Soc.* **93** No. 453.

C. K. Chui, 1988: *Multivariate Splines*. CBMS-NSF Reg. Conf. Series in Appl. Math., vol. 54, SIAM, Philadelphia.

E. Cohen, T. Lyche, R. Riesenfeld, 1984: Discrete box splines and refinement algorithms. *Comput. Aided Geom. Design* **1** 131–148.

M. Dæhlen, 1989: On the evaluation of box-splines. in *Mathematical Methods in Computer Aided Geometric Design*, T. Lyche and L. Schumaker, editors, Academic Press (New York), 167–179.

W. Dahmen, 1987: Subdivision algorithms – recent results, some extensions and further developments. in *Algorithms for the Approximation of Functions and Data*, J. C. Mason and M. G. Cox, editors, Oxford Univ. Press (Oxford), 21–49.

W. Dahmen, C. A. Micchelli, 1984: Subdivision algorithms for the generation of box-spline surfaces. *Comput. Aided Geom. Design* **1** 115–129.

W. Dahmen, C. A. Micchelli, 1993: Using the refinement equation for evaluating integrals of wavelets. *SIAM J. Numer. Anal.* **xx** xxx–xxx.

K. Jetter, J. Stöckler, 1991: Algorithms for cardinal interpolation using box splines and radial basis functions. *Numer. Math.* **60** 97–114.

M.-J. Lai, 1992: Fortran subroutines for B-nets of box splines on three- and four-directional meshes. *Numer. Algorithms* **2** xxx–xxx.

MathWorks, 1989: *MATLAB User's Guide*. MathWorks Inc., South Natick MA.

C. A. Micchelli, 1980: A constructive approach to Kergin interpolation in $\mathbb{R}^k$: multivariate B-splines and Lagrange interpolation. *Rocky Mountain J. Math.* **10** 485–497.

Appendix: Listing of M-files

Here, for the record, is a listing of the various M-files referred to earlier in this note. Someone interested in making use of these files would be better off obtaining them by anonymous ftp from the machine `stolp.cs.wisc.edu`, where they are available in the encoded `tar` file `box-spline.mfiles`. The procedure is standard: use the command `ftp stolp.cs.wisc.edu` (or, equivalently, the command `ftp 128.105.2.127`), give your name as `anonymous`, use your login name as the password, then issue the command `get box-spline.mfiles message`. This puts a file called `message` into your current directory, and its first few lines will tell you how to get the individual M-files from there. All this presupposes that you are on a machine running `unix`.

```
function values = bxval(xi,xx)
%
%        values = bxval(xi, xx)
%
% returns the values of the box spline with directions  xi  at the
% points  xx(:,j), j=1,2,...

% C de Boor:  23 jun 92

[dx,ignored] = size(xi); [d,ignored] = size(xx);
if (dx ~= d),
  error('directions and points are of different dimensions.'), end

perturb = max(max(abs(xi)))*1.e-10;  % <<<  note use of tolerance
[values, undef] = bxrec(xi,xx);
while ~isempty(undef), %  perturb any point on the mesh, then retry
  [vu, uu] = ...
   bxrec(xi,xx(:,undef)+(rand(d,1)*perturb)*ones(1,length(undef)));
  values(undef) = vu;
  undef = undef(uu);
end


function [values,undef] = bxrec(xi,x)
%
%        [values, undef] = bxrec(xi, x)
%
% recursive m-file for computing (however expensively)  M_{xi}(x).
% Note that  x(:,undef)  have been found to lie `on' the mesh for
% M_{xi}, hence need to be perturbed by the calling program.  For
% this reason, it is better to use  bxval  which calls on  bxrec
% and perturbs the argument if need be until it gets it `off'
% the mesh.
%    The action could be speeded up somewhat by recognizing
% multiplicities explicitly.
```

```
% C de Boor:  4 jul 92/ 12 aug 92

[s,n] = size(xi); [ignored,nx] = size(x);
values = zeros(1,nx); undef = [];
  % zero values will be returned unless  xi  is of full rank.

% Compute the QR factorization for  xi  as a means of telling
% whether or not  xi  is of full rank.  Since the factorization is
% needed for this, it also comes in handy for determining a
% reasonable solution of  xi?  = x .
[q,r,e] = qr(xi); ad = abs(diag(r));
                    % q is unitary, r is upper triangular,
                    % with absol. decreasing diagonal elements,
                    % and  e  is a permutation matrix.
if ad(1) < (1.e+10)*ad(s),% If  xi  is of full rank,
 t = (r*e')\(q'*x);% compute t as the smallest solution of  xi?=x.
                         % Further,
 if (s==n),  % if xi is square, return the characteristic function
         % of  xi(\boxx) , divided by abs(det(xi)) , retaining in
         %  undef  those  j  for which  x(:,j)  is on the mesh.
  undef = find(min([abs(t);abs(1-t)])<1.e-12);
  ok = 1:nx; ok(undef)=[];
  values(ok) = (0<=min(t(:,ok))&max(t(:,ok))<1)/prod(ad);

 else,  % use the recurrence relations, but only for the  x(:,j)
    %  in the smallest axiparallel cube containing supp M_{xi}, ...
  g = find(max(x-sum(max(xi',zeros(n,s)))'*ones(1,nx))<=0& ...
            min(x-sum(min(xi',zeros(n,s)))'*ones(1,nx))>=0);
  lg = length(g);
  j=1; xicut = xi(:,2:n); %  xicut  contains all directions but the
                        % one currently left out.
  while lg>0, % compute and add the jth term of the recurrence:
   [vj, uj] = bxrec(xicut,[x(:,g),x(:,g)-xi(:,j)*ones(1,lg)]);
   values(g) = ...
        values(g) + t(j,g).*vj(1:lg) + (1-t(j,g)).*vj(lg+[1:lg]);
   if ~isempty(uj), % remove undefineds from further consideration
      indic = zeros(1,lg); [1:lg,1:lg];
      indic(ans(uj)) = ones(1,length(uj)); uj = find(indic==1);
      undef = [undef, g(uj)]; g(uj) = []; lg = length(g);
   end
   if (j == n), break, end
   xicut(:,j) = xi(:,j);  % increment  j  and update  xicut .
   j = j+1;
  end
  values = values/(n-s);
 end
end
```

```
function m = convol(m1,m2)
%
%       m = convol(m1, m2)
%
%  returns the discrete convolution of the two masks contained in
%   m1 , m2  (as encoded by  msmak.m  and decoded by  msbrk.m ).
%  If one or the other of these functions is itself a convolution
%  product, it is more efficient to apply the sequence of factors.

%  C de Boor:  11 oct 90/ 25 jun 92

%  if  mi = maski(.+zi) , and  m = m1*m2 = sum_j m1(.-j)m2(j) ,
%  with  *  here denoting convolution, then
%     mask(.+z) = sum_j mask1(.-j+z1)mask2(j+z2)
%  or
%     mask(.+z-z1-z2) = sum_j mask1(.-j)mask2(j)
%  showing that the mask of the convolution product is built up by
%  adding, for each  j  in  supp2 := supp(mask2), the matrix
%   mask1 mask2(j)  to the area  supp1+j +z-z1-z2  of  mask .
%  It is assumed below that, in fact, each  mask  is an ordinary
%  matrix,  i.e., indexed from 1 to ...  .
%  While the center  z  is arbitrary, the required support of  mask
%  is minimized if we choose  z  as is done below.

%  Make the second mask the smaller one:
if (length(m1) < length(m2)), junk = m1; m1 = m2; m2 = junk; end
oo = [1;1];
[z1,rc1,mask1] = msbrk(m1); supp1 = [oo rc1];
[z2,rc2,mask2] = msbrk(m2); supp2 = [oo rc2];
% compute support and center of convolved mask:
supp = supp1+supp2-(z1+z2)*ones(1,2);
z = oo-supp(:,1); supp = supp + z*ones(1,2); rc = supp(:,2);
% compute convolved mask:
mask = zeros(rc(1),rc(2)); shsupp1 = supp1+(z-z1-z2)*ones(1,2);
rangex = shsupp1(1,1):shsupp1(1,2);
rangey = shsupp1(2,1):shsupp1(2,2);
for i=1:rc2(1);
for j=1:rc2(2);
   mask(rangex+i,rangey+j) = ...
                       mask(rangex+i,rangey+j) + mask1*mask2(i,j);
   end; end
m = msmak(z,mask);


function m  =  condir(m1,dir)
%
%       m  =  condir(m1, dir)
%
%  this specialization of  convol.m  returns the discrete convol-
```

```
% ution of the mask contained in  m1  with the one-direction
% mask specified in  dir =  [x; y]*[0:(n-1)] .

% C de Boor:  12 oct 90/ 25 jun 92

oo = [1; 1];
[z1,rc1,mask1] = msbrk(m1); supp1 = [oo rc1];
[jj,n] = size(dir); dor = dir+ones(jj,n); z2 = [1; 1];
if (dir(1,n)<0),
  z2(1) = 1-dir(1,n); dor(1,:)  = dor(1,:)-dir(1,n)*ones(1,n); end
if (dir(2,n)<0),
  z2(2) = 1-dir(2,n); dor(2,:)  = dor(2,:)-dir(2,n)*ones(1,n); end
supp2 = [oo abs(dir(:,n))+oo];
% compute support and center of convolved mask:
supp = supp1+supp2-(z1+z2)*ones(1,2);
z = oo-supp(:,1); supp= supp + z*ones(1,2); rc = supp(:,2);
% compute convolved mask:
mask= zeros(rc(1),rc(2)); shsupp1 = supp1+(z-z1-z2)*ones(1,2);
rangex = shsupp1(1,1):shsupp1(1,2);
rangey = shsupp1(2,1):shsupp1(2,2);
for i = 1:n;
   mask(rangex+dor(1,i),rangey+dor(2,i))= ...
                   mask(rangex+dor(1,i),rangey+dor(2,i)) + mask1;
   end
m = msmak(z,mask);


function mask = msxima(Xi,nh)
%
%        mask = msxima(Xi, nh)
%
% returns the (unscaled) mask associated with the directions  Xi
% when subdividing each direction into  nh  pieces.  The properly
% scaled mask is obtained from this by division by  nh^(n-2) .

% C de Boor:  12 oct 90

mask = msmak([1;1],[1]);
[s,n] = size(Xi);
if (s~=2), error('msxima now only works for bivariate Xi .'),end

for j=1:n
   mask = condir(mask,Xi(:,j)*[0:(nh-1)]);
   end


function m = msmak(z,mask)
%
%        m = msmak(z, mask)
```

```
%
% returns in  m  the mesh-function contained in  mask  with center
%  z  (to be decoded by  msbrk.m ).    mask  is understood to
%  contain the nontrivial part of the two-dimensional mesh-function
%              ZZ^2 --> RR: j |--> mask(j+z) .
% E.g.,  msmak([1;1], 1)  provides the delta-mask.  (See  convol.m
% for more detail.)    Note that this requires one to look at the
% matrix  mask  sideways.  Also, since  mask  is a matrix, indexed
% in the standard way, i.e., with support equal to  [1:r]x[1:c] ,
% the center  z  must necessarily lie at, or to the south-west
% (i.e., left and below) of the lower left corner of the smallest
% rectangle containing the support of the mesh-function, with  mask
% smallest if  z is equal to that corner.  E.g., the delta-function
% is also provided by
%                msmak([3;2], [0 0 0 0; 0 0 0 0; 0 1 0 0])

% C de Boor:  10 oct 90/ 25 jun 92

m = [z(:); size(mask)'; mask(:)];


function [z,rc,mask] = msbrk(m,print)
%
%        [z, rc, mask] = msbrk(m [,print])
%
%  returns the details of the mesh-function contained in  m , i.e.,
% of the map
%              ZZ^2 --> RR : j |--> mask(j+z)
%  with  size(mask) =:  rc .
%    If a second argument is present, the details are printed out.

%  C de Boor:  10 oct 90

l = length(m); z = m(1:2); rc = m(3:4);
if (l-4 ~= rc(1)*rc(2)),
   error('The input does not seem to contain a mask.'), end
mask = reshape(m(5:l),rc(1),rc(2));
if (nargin > 1),  z, rc, mask, end
```