Hyper2d: A numerical code for hyperinterpolation on rectangles

Marco Caliari [a], Marco Vianello [a,*], Stefano De Marchi [b], Roberto Montagna [b]

[a] *Department of Pure and Applied Mathematics, University of Padova, Italy*
[b] *Department of Computer Science, University of Verona, Italy*

Abstract

Hyperinterpolation at Morrow–Patterson–Xu cubature points for the product Chebyshev measure provides a simple and powerful polynomial approximation method on rectangles. Here, we present an accurate and efficient Matlab/Octave implementation of the hyperinterpolation formula, accompanied by several numerical tests.
© 2006 Elsevier Inc. All rights reserved.

Keywords: Bivariate hyperinterpolation; Minimal cubature; Morrow–Patterson–Xu points; Lebesgue constant

1. Introduction

Hyperinterpolation of multivariate continuous functions on compact subsets or manifolds, originally introduced by Sloan in [15], is a discretized orthogonal projection on polynomial subspaces, which provides an approximation method more general (in some sense) than interpolation. Its main success up to now, has been given by the application to polynomial approximation on the sphere; see, e.g. [10,13].

Indeed, in order to become an efficient approximation tool in the uniform norm, hyperinterpolation needs a "good" cubature formula (i.e., positive weights and high algebraic degree of exactness), together with "slow" increase of the Lebesgue constant (the operator norm). The importance of these basic features can be understood by summarizing briefly the structure of hyperinterpolation.

Let $\Omega \subset \mathbb{R}^d$ be a compact subset (or lower dimensional manifold), and μ a positive measure such that $\mu(\Omega) = 1$ (i.e., a normalized positive and finite measure on Ω). For every function $f \in C(\Omega)$ the μ-orthogonal projection of f on $\Pi_n^d(\Omega)$ (the subspace of d-variate polynomials of degree $\leqslant n$ restricted to Ω) can be written as

$$S_n f(\boldsymbol{x}) = \int_\Omega K_n(\boldsymbol{x}, \boldsymbol{y}) f(\boldsymbol{y}) \mathrm{d}\mu(\boldsymbol{y}) \quad \text{with } S_n p = p \quad \text{for } p \in \Pi_n^d(\Omega), \tag{1}$$

---

* Corresponding author.
 E-mail address: marcov@math.unipd.it (M. Vianello).

where $\boldsymbol{x} = (x_1, x_2, \ldots, x_d)$, $\boldsymbol{y} = (y_1, y_2, \ldots, y_d)$, and the so-called reproducing kernel K_n is defined by

$$K_n(\boldsymbol{x}, \boldsymbol{y}) = \sum_{k=0}^{n} \sum_{|\alpha|=k} P_\alpha(\boldsymbol{x}) P_\alpha(\boldsymbol{y}), \quad \alpha = (\alpha_1, \alpha_2, \ldots, \alpha_d) \tag{2}$$

the set of polynomials $\{P_\alpha, |\alpha| = \alpha_1 + \cdots + \alpha_d = k, 0 \leqslant k \leqslant n\}$ being any μ-orthonormal basis of $\Pi_n^d(\Omega)$, with P_α of total degree $|\alpha|$; cf. [8, Section 3.5].

Now, given a cubature formula for μ with $N = N(n)$ nodes $\xi \in \Xi \subset \Omega$, $\xi = (\xi_1, \xi_2, \ldots, \xi_d)$, and positive weights $\{w_\xi\}$, which is exact for polynomials of degree $\leqslant 2n$,

$$\int_\Omega p(\boldsymbol{x}) \mathrm{d}\mu = \sum_{\xi \in \Xi} w_\xi p(\xi), \quad \forall p \in \Pi_{2n}^d(\Omega), \tag{3}$$

we obtain from (1) the polynomial approximation of degree n

$$f(\boldsymbol{x}) \approx L_n f(\boldsymbol{x}) = \sum_{\xi \in \Xi} w_\xi K_n(\boldsymbol{x}, \xi) f(\xi). \tag{4}$$

It is known that necessarily $N \geqslant \dim(\Pi_n^d)(\Omega)$, and that (4) is a polynomial interpolation at Ξ whenever the equality holds; cf. [15,10].

The hyperinterpolation error in the uniform norm, due to exactness on $\Pi_{2n}^d(\Omega)$, can be easily estimated as

$$\|f - L_n f\|_\infty \leqslant (1 + \Lambda_n) E_n(f), \quad \Lambda_n = \|L_n\| = \max_{\boldsymbol{x} \in \Omega} \left\{ \sum_{\xi \in \Xi} w_\xi |K_n(\boldsymbol{x}, \xi)| \right\}, \tag{5}$$

where Λ_n is the operator norm of $L_n : (C(\Omega), \|\cdot\|_\infty) \to (\Pi_n^d(\Omega), \|\cdot\|_\infty)$, usually termed the "Lebesgue constant" in the interpolation framework.

The aim of this paper is to provide an efficient implementation of hyperinterpolation in dimension $d = 2$ on rectangles, based on cubature at MPX (Morrow–Patterson–Xu) points [16,6]. In Section 2, we discuss a Matlab-like implementation of hyperinterpolation at MPX points. The corresponding Matlab/Octave functions are displayed and described in Section 3. In Section 4 we state a conjecture on the asymptotics of the Lebesgue constant, based on a wide set of numerical experiments. Finally, we provide the numerical results corresponding to hyperinterpolation of several test functions.

2. Hyperinterpolation at Morrow–Patterson–Xu (MPX) points

In the paper [16], Xu introduced a set of Chebyshev-like points in the square $\Omega = [-1, 1]^2$, which generate a (near) minimal degree cubature for the normalized product Chebyshev measure,

$$\mathrm{d}\mu = \frac{1}{\pi^2} \frac{\mathrm{d}x_1 \, \mathrm{d}x_2}{\sqrt{1 - x_1^2} \sqrt{1 - x_2^2}}, \quad \Omega = [-1, 1]^2. \tag{6}$$

For even degrees, such points and the corresponding minimal cubature formula were originally proposed by Morrow and Patterson in [12]. In addition, Xu proved that these points are also suitable for constructive polynomial interpolation, in a polynomial subspace $\mathscr{V}_n$, $\Pi_{n-1}^2 \subset \mathscr{V}_n \subset \Pi_n^2$. Xu-like interpolation, recently studied thoroughly in [2–4], turned out to be a good approximation method in the uniform norm. In particular, its Lebesgue constant is $\mathcal{O}(\log^2 n)$, n being the degree, i.e. the polynomial approximation is "near-optimal" (cf. [5]).

Hyperinterpolation at the MPX points, even though is not interpolant, shares the same good computational features of Xu-like interpolation, as it has been recently shown in [6]. In particular, hyperinterpolation (of degree n) and interpolation (of degree $n + 1$) at the same set of MPX points exhibit very close errors. Here we describe an efficient Matlab-like implementation of the hyperinterpolation formula on rectangles.

Consider the $n + 2$ Chebyshev–Lobatto points on the interval $[-1, 1]$

$$z_k = z_{k,n+1} = \cos \frac{k\pi}{n+1}, \quad k = 0, \ldots, n+1. \tag{7}$$

The MPX points on the square Ω for cubature with exactness degree $2n + 1$, are defined as the two dimensional Chebyshev-like set

$$\Xi = A \cup B, \quad \mathrm{card}(\Xi) = N,$$

where

- Case n odd, $n = 2m - 1$

$$\begin{aligned} A_{\mathrm{odd}} &= \{(z_{2i}, z_{2j+1}),\ 0 \leqslant i \leqslant m,\ 0 \leqslant j \leqslant m - 1\}, \\ B_{\mathrm{odd}} &= \{(z_{2i+1}, z_{2j}),\ 0 \leqslant i \leqslant m - 1,\ 0 \leqslant j \leqslant m\} \end{aligned} \tag{8}$$

with $N = (n + 1)(n + 3)/2$. These points generate a minimal cubature formula, that is

$$\int_{\Omega} p(\boldsymbol{x}) \mathrm{d}\mu = \sum_{\xi \in \Xi} w_\xi p(\xi), \quad \forall p \in \Pi_{2n+1}^2, \tag{9}$$

where the weights are simply $w_\xi = (n + 1)^{-2}$ for $\xi \in \Xi \cap \partial\Omega$ (boundary points), $w_\xi = 2(n + 1)^{-2}$ for $\xi \in \Xi \cap \overset{\circ}{\Omega}$ (interior points); cf. [12,16].

- Case n even, $n = 2m$

$$\begin{aligned} A_{\mathrm{even}} &= \{(z_{2i}, z_{2j}),\ 0 \leqslant i \leqslant m,\ 0 \leqslant j \leqslant m\}, \\ B_{\mathrm{even}} &= \{(z_{2i+1}, z_{2j+1}),\ 0 \leqslant i \leqslant m,\ 0 \leqslant j \leqslant m\} \end{aligned} \tag{10}$$

with $N = (n + 2)^2/2$. The weights for the corresponding near minimal cubature formula are $w_\xi = (n + 1)^{-2}/2$ for $\xi = (1, 1)$ and $\xi = (-1, -1)$ (corner points), $w_\xi = (n + 1)^{-2}$ for the other boundary points and $w_\xi = 2(n + 1)^{-2}$ for the interior points.

Hence, in view of (3) we can construct the hyperinterpolation formula (4), which is not interpolant, since in both cases

$$N > v = \dim(\Pi_n^2) = \frac{(n + 1)(n + 2)}{2}. \tag{11}$$

The polynomial approximation (4) can be rewritten as

$$L_n f(\boldsymbol{x}) = \sum_{k=0}^{n} \sum_{|\alpha|=k} c_\alpha P_\alpha(\boldsymbol{x}), \quad c_\alpha = \sum_{\xi \in \Xi} w_\xi f(\xi) P_\alpha(\xi), \tag{12}$$

where the coefficients c_α can be computed once and for all. Now, take the μ-orthonormal basis

$$\{P_\alpha(\boldsymbol{x}) = T_j(x_1) T_{k-j}(x_2),\ \alpha = (j, k - j),\ 0 \leqslant j \leqslant k \leqslant n\}, \tag{13}$$

where T_j is the normalized Chebyshev polynomial of degree j (that is $T_0(\cdot) = 1$, $T_j(\cdot) = \sqrt{2}\cos(j\arccos(\cdot))$). In order to implement efficiently the hyperinterpolation formula (12) in Matlab/Octave, it has to be rewritten in a matrix formulation, avoiding iteration loops. Consider the matrices

$$D(\Xi, f) = \mathrm{diag}([w_\xi f(\xi), \xi = (\xi_1, \xi_2) \in \Xi]) \in \mathbb{R}^{N \times N}, \tag{14}$$

$$T^{(i)}(\Xi) = \begin{bmatrix} \cdots & T_0(\xi_i) & \cdots \\ \cdots & T_1(\xi_i) & \cdots \\ \vdots & \vdots & \vdots \\ \cdots & T_n(\xi_i) & \cdots \end{bmatrix} \in \mathbb{R}^{(n+1) \times N}, \quad i = 1, 2, \tag{15}$$

where the underbrace is $\xi \in \Xi$.

and

$$B_0(\varXi,f) = \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & \cdots & b_{1,n+1} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & 0 & \cdots & 0 \\ b_{n+1,1} & 0 & \cdots & 0 & 0 \end{bmatrix} \in \mathbb{R}^{(n+1)\times(n+1)}, \tag{16}$$

which is the upper-left triangular part of

$$(b_{i,j}) = B(\varXi,f) = T^{(1)}(\varXi)D(\varXi,f)(T^{(2)}(\varXi))', \tag{17}$$

where the $'$ symbol denotes the transposition, that is the coefficients $\{c_\alpha\}$ in (12). Then, (12) becomes

$$L_n f(\boldsymbol{x}) = [\, T_0(x_1) \quad T_1(x_1) \quad \cdots \quad T_n(x_1)\,]B_0(\varXi,f)\begin{bmatrix} T_0(x_2) \\ T_1(x_2) \\ \vdots \\ T_n(x_2) \end{bmatrix}. \tag{18}$$

Given a set $X \subset \varOmega$ of target points with cardinality M, we compute

$$T^{(i)}(X) = \begin{bmatrix} \cdots & T_0(x_i) & \cdots \\ \cdots & T_1(x_i) & \cdots \\ \vdots & \vdots & \vdots \\ \cdots & T_n(x_i) & \cdots \\ & \underbrace{}_{\boldsymbol{x}\in X} & \end{bmatrix} \in \mathbb{R}^{(n+1)\times M}, \quad i = 1, 2, \tag{19}$$

and then

$$L_n f(X) = \Big[\underbrace{\cdots \quad L_n f(\boldsymbol{x}) \quad \cdots}_{\boldsymbol{x}\in X}\Big] = \mathrm{diag}((T^{(1)}(X))'B_0(\varXi,f)T^{(2)}(X))'. \tag{20}$$

Notice that the meaning of the keyword "diag" is different in (14) and (20) (as it is in Matlab/Octave): in (14) it represents a diagonal matrix with the specified diagonal, whereas in (20) it represents the diagonal (as a column vector) of the specified matrix.

Clearly, we can immediately extend the hyperinterpolation formula to a function f defined on a generic rectangle $[a,b]\times[c,d]$, via the affine mapping

$$\boldsymbol{\sigma} : [-1,1]^2 \to [a,b]\times[c,d], \quad \sigma_1(t_1,t_2) = \frac{b-a}{2}t_1 + \frac{b+a}{2}, \quad \sigma_2(t_1,t_2) = \frac{d-c}{2}t_2 + \frac{d+c}{2}. \tag{21}$$

Indeed, for a set of target points $X \subset [a,b]\times[c,d]$, we have simply

$$L_n f(X) = \mathrm{diag}((T^{(1)}(\boldsymbol{\sigma}^{-1}(X)))'B_0(\varXi, f\circ\boldsymbol{\sigma})T^{(2)}(\boldsymbol{\sigma}^{-1}(X)))'. \tag{22}$$

Remark 1. The representation (12) of the hyperinterpolation polynomial is particularly suitable for a Matlab-like implementation as (14)–(22), since it allows to easily avoid bottlenecks like recurrences and iterations loops, via predefined matrix functions. Moreover, for evaluation at a large number of points, it compares favourably with other implementations. First, we observe that a simple analysis of the hyperinterpolation algorithm gives the following complexity estimates for construction (excluding evaluation of the function f at the MPX points), and evaluation at M target points:

- construction: cost of (15) + cost of (17) $\approx 2c_T nN + 2vN$ flops,
- evaluation: cost of (19) + cost of (20) $\approx 2c_T nM + 2vM$ flops,

where N is the number of MPX cubature points, $v = \dim(\Pi_n^2)$, and c_T denotes the average evaluation cost of a single Chebyshev polynomial via its trigonometric representation. Notice that $N \approx n^2/2 \approx v$, already for moderate values of the degree n (cf. (8), (10) and (11)).

The hyperinterpolation polynomial can also be computed via (4), by using the compact trigonometric formula for the reproducing kernel obtained by Xu (cf. [16]) and adopted in the Fortran implementation of [6]. In practice, such a formula is severely ill-conditioned and has to be stabilized, as shown in [2]. After stabilization, for degrees up to the hundreds its final pointwise evaluation complexity (excluding evaluation of f at the MPX points) is of the order of $24c_{\sin}N$ flops, that is linear in the number of MPX points. Here $c_{\sin}$ denotes the average cost of the sine function. Thus the implementation (14)–(22) in terms of flops is more convenient than the stabilized Xu formula on a large number of evaluation points, say $M \gg N$, since $2v \ll 24c_{\sin}N$. This happens in many applications, like quality plotting or data compression (see, e.g. [3]).

It is also worth noticing, however, that in practice, due to internal Matlab/Octave optimizations of matrix operations, for $M \gg N$ the bulk is given by the computation of (19), and thus the CPU times turn out to increase linearly instead of quadratically in the degree (see the numerical tables in the last section). In these cases, our present implementation is still more convenient than that based on the stabilized Xu formula for the reproducing kernel, since $2c_T n \ll 24c_{\sin}N$ already for relatively small values of n.

Remark 2. The implementation (14)–(22) could be easily extended to the construction and evaluation of the Lagrange interpolation polynomial at the MPX points. The interpolation formula, however, involves two sums like (4), one with K_n and another with K_{n+1}, see [16]. Even if optimized, the resulting algorithm is more expensive than that for hyperinterpolation. Since hyperinterpolation errors are very close to interpolation errors (see [6]), the former should be preferred as an approximation tool whenever the interpolation property is not a strict requirement.

Remark 3. We discuss here the construction of a practical a posteriori error estimate, that could be useful in several applications of hyperinterpolation. Going back to the meaning of hyperinterpolation as a discretized truncated Fourier series (μ-orthogonal projection on Π_n^2), i.e. to the fact that $L_n f(x) \approx S_n f(x)$, since $\{c_\alpha\}$ are the Fourier coefficients discretized by cubature at MPX points, we can write the following chain of estimates

$$\|f - L_n f\|_\infty \approx \|f - S_n f\|_\infty \leqslant 2 \sum_{k=n-2}^{\infty} \sum_{|\alpha|=k} \left| \int_\Omega P_\alpha(\boldsymbol{y}) f(\boldsymbol{y}) \mathrm{d}\mu(\boldsymbol{y}) \right|$$

$$\approx 2 \sum_{k=n-2}^{n} \sum_{|\alpha|=k} \left| \int_\Omega P_\alpha(\boldsymbol{y}) f(\boldsymbol{y}) \mathrm{d}\mu(\boldsymbol{y}) \right| \approx 2 \sum_{k=n-2}^{n} \sum_{|\alpha|=k} |c_\alpha|, \tag{23}$$

where the bound $|P_\alpha(\boldsymbol{x})| \leqslant 2$ has been used (cf. (13)). The passage from the first to the second row in (23) is somehow empirical, but reminiscent of popular error estimates for one-dimensional Chebyshev series, based on the last two or three coefficients (cf., e.g. [1]). In fact, here we use just the coefficients corresponding to the last three values of k, namely $k = n - 2, n - 1, n$. The practical behavior of (23) has been satisfactory in almost all our numerical tests; see the last section.

3. Matlab/Octave code

Here, we report the Matlab/Octave code of the main functions for the hyperinterpolation on rectangles. A Matlab/Octave interface based on this code can be downloaded from [7].

The function hypcoeffs (Table 1) builds the matrix $B_0(\Xi, f \circ \boldsymbol{\sigma})$ in (22), via the mapping $\boldsymbol{\sigma}$ in (21).

The function MPXpts (Table 2) called in hypcoeffs provides the MPX points and weights for cubature of exactness degree $2n - 1$ (corresponding to the subspace $\mathscr{V}_n$ of n-degree polynomials, cf. [16]) without using

Table 1
Function hypcoeffs

```
function [B0]=hypcoeffs(n,a,b,c,d)
[xi1,xi2,wxi]=MPXpts(n + 1)
Txi1=T(n,xi1);
Txi2=T(n,xi2);
fxi=f(((b − a)*xi1+(b + a))/2,((d − c)*xi2+(d + c))/2)));
B0=Txi1.*repmat(wxi.*fxi,n + 1,1)*Txi2';
B0=fliplr(triu(fliplr(B0)));
```

Table 2
Function MPXpts

```
  function [xi1,xi2,wxi]=MPXpts(n)
  if (mod(n,2)==0)
    m=n/2;
    xi1=repmat(z(2:2:n),1,m + 1);
    wxi=[ones(1,m) repmat(2*ones(1,m),1,m-1) ones(1,m)];
    xi1=[xi1 reshape(repmat(z(1:2:n + 1),m,1),1,m*(m + 1))];
    wxi=[wxi ones(1,m) 2*ones(1,m*(m-1)) ones(1,m)];
    xi2=xi1([m*(m + 1)+1:2*m*(m + 1),1:m*(m + 1)]);
  else
    m=(n-1)/2;
    xi1=repmat(z(1:2:n),1,m + 1);
    wxi=[0.5 ones(1,m) repmat([1 2*ones(1,m)],1,m)];
    xi1=[xi1 reshape(repmat(z(n + 1:−2:2),m + 1,1),1,(m + 1)^2)];
    wxi=[wxi 0.5 ones(1,m) repmat([1 2*ones(1,m)],1,m)];
    xi2=−xi1([(m + 1)^2 + 1:2*(m + 1)^2,1:(m + 1)^2]);
  end
  wxi=wxi/n^2;
```

iteration loops, via the Matlab/Octave functions repmat and reshape. Clearly, in order to hyperinterpolate at degree n, it has to be called with the input argument set to $n + 1$, since exactness degree at least $2n$ is needed.

The function hypval (Table 3) computes the vector $L_n f(X)$ in (22) via the inverse mapping σ^{-1}. Notice that it computes the diagonal of the matrix specified in (22) without performing the two whole matrix products.

The function T (Table 4), called by hypcoeffs and hypval, computes the normalized Chebyshev polynomials arrays $T^{(i)}$ in (15). Notice that, due to roundoff errors, the input s of T, when called by hypval, could lie out of $[-1, 1]$, and in these cases is set to the nearest endpoint.

Table 3
Function hypval

```
function [Lnfx]=hypval(B0,n,a,b,c,d,x1,x2)
Tx1=T(n,(2*x1−(b + a))/(b − a));
Tx2=T(n,(2*x2−(d + c))/(d − c));
Lnfx=sum((Tx1'*B0).*Tx2',2)';
```

Table 4
Function T

```
function t=T(n,s)
t=cos([0:n]'*acos(max(min(s,1),−1)));
t(2:n + 1,:)=sqrt(2)*t(2:n + 1,:);
```

4. Numerical tests

Hyperinterpolation of degree n at the MPX points possesses two important features, that make it a good approximation tool in the uniform norm, for functions that can be sampled without restrictions on rectangles.

The first is that its Lebesgue constant increases very slowly, as that of near-optimal interpolation points on the square (cf. [4,5]). Indeed, as proved in [6], it can be rigorously bounded by

$$\Lambda_n \leqslant 8A_n^2 + 5A_n + 3, \quad A_n = \frac{2}{\pi}\log(n+1) + 5. \tag{24}$$

However, the factor 8 in (24) is an overestimate. Indeed, a wide set of numerical experiments on the maximization of the Lebesgue function up to degree $n = 1000$ (not reported for brevity), lead to the following:

• **conjecture:** $\Lambda_n \leqslant B_n \sim c\left(\frac{2}{\pi}\log n\right)^2$, with $c < \frac{3}{2}$.
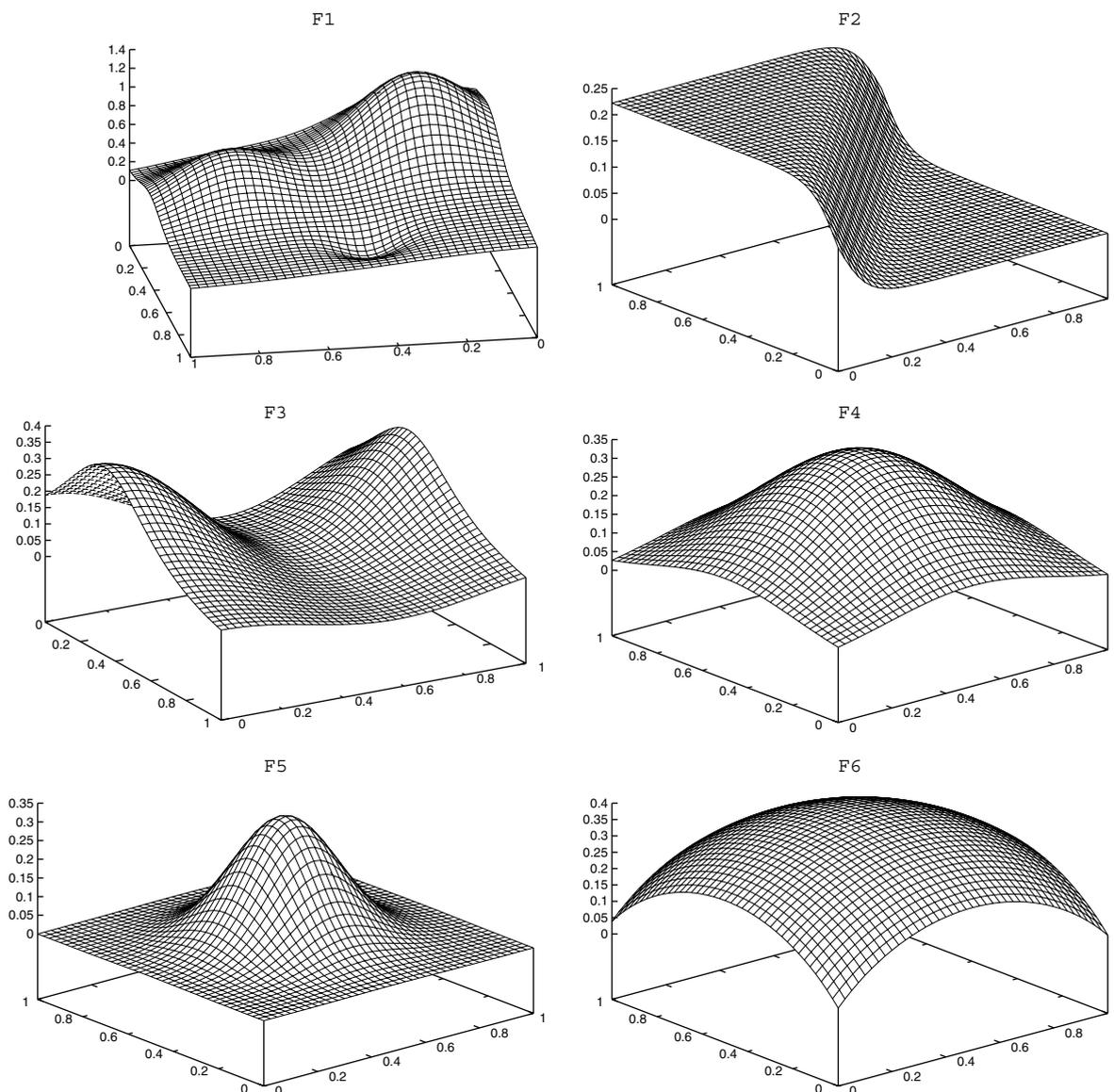

Fig. 1. Franke's test functions.

In addition, with the implementation (14)–(20) the average pointwise complexity over a large number of target points, say $M \gg N$ points, is of the order of $2v \approx n^2$ flops (see Remark 1), that is linear in the dimension of the polynomial space and quadratic in the degree.

In this section we show the hyperinterpolation errors in the max-norm normalized to the max deviation of the function from its mean, at a sequence of degrees, $n = 10, 20, \ldots, 60$, on a well-known test functions suite by Franke–Renka (cf. [9,14]). These 10 functions, termed F1, ..., F10, are plotted in Figs. 1 and 2 below. The corresponding "true" errors, reported in Tables 5 and 6, have been computed on a 100×100 uniform control grid. In the tables we report also (in parenthesis) the a posteriori empirical error estimate given by the last term of (23), normalized as above.

The last four functions, proposed by Renka in [14], are considered more challenging for the testing of interpolation methods at scattered points, due to their multiple features and abrupt transitions. Here, we can see in Tables 5 and 6 that only F10, and much less severely F2, are really "difficult" for hyperinterpolation at MPX points. In all the other cases the approximation behavior of the hyperinterpolation polynomial is quite satisfactory. With the smoothest functions, like F4 and F6, the error stabilizes rapidly around machine precision. It is interesting to observe that with the oscillating function $F7(x_1, x_2) = 2\cos(10x_1)\sin(10x_2) + \sin(10x_1 x_2)$, the error starts decaying rapidly as soon as the degree n allows to recover the oscillations. On the other hand, the troubles with F10 are natural, since it has a gradient discontinuity in the center of the square, whereas the MPXpoints cluster at the boundary.

As for the empirical error estimates, we can see that they tend to overestimate in almost all the cases, except for F2 and F10, where they underestimate the true errors. The worst overestimate arises with F4 for $n = 20$ (estimate/error ≈ 158), whereas the worst underestimate concerns the less smooth test function F10 for $n = 60$ (estimate/error ≈ 0.18). In general, we can consider the behavior of the (normalized) a posteriori estimate (23) satisfactory enough, even though this topic needs further investigations.
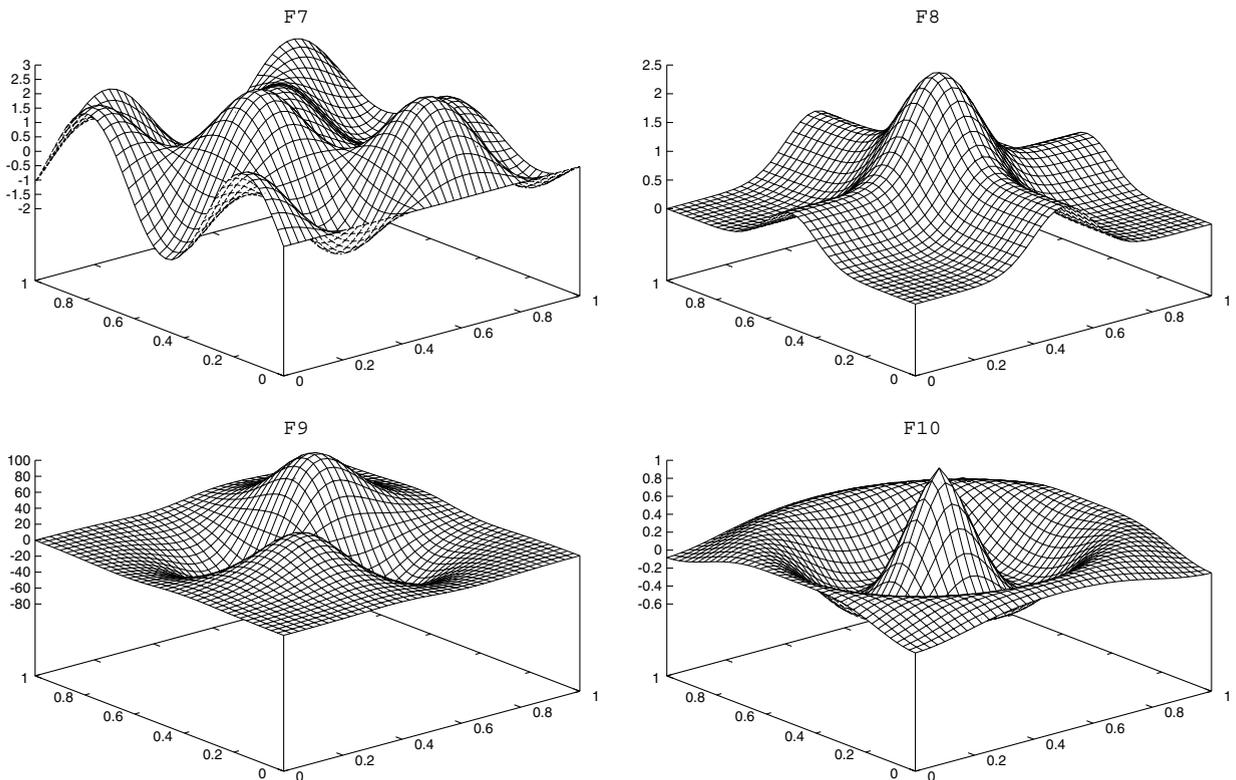

Fig. 2. Renka's additional test functions.

Table 5
"True" and estimated (in parenthesis) hyperinterpolation errors for the Franke's test functions in Fig. 1, in the max-norm normalized to the max deviation of each function from its mean

	n					
	10	20	30	40	50	60
	N					
	72	242	512	882	1352	1922
F1	7.3E−2	4.4E−3	1.6E−4	1.2E−6	8.6E−9	2.4E−11
	(1.5E−1)	(1.5E−2)	(5.3E−4)	(9.0E−6)	(5.8E−8)	(1.7E−10)
F2	2.9E−1	6.3E−2	1.2E−2	2.1E−3	3.9E−4	6.6E−5
	(1.4E−1)	(2.1E−2)	(3.3E−3)	(5.7E−4)	(1.0E−4)	(1.7E−5)
F3	3.7E−3	5.7E−6	1.0E−8	1.6E−11	4.0E−14	3.3E−14
	(4.3E−2)	(6.7E−5)	(1.0E−7)	(1.8E−10)	(2.9E−13)	(7.7E−15)
F4	2.1E−4	4.0E−10	1.0E−14	1.1E−14	1.0E−14	1.5E−14
	(1.0E−2)	(6.3E−8)	(2.8E−14)	(5.7E−15)	(6.7E−15)	(3.7E−15)
F5	3.7E−2	5.3E−5	9.7E−9	4.0E−13	7.3E−15	9.0E−15
	(2.3E−1)	(8.0E−4)	(2.6E−7)	(1.7E−11)	(2.7E−15)	(2.0E−16)
F6	2.1E−5	8.0E−9	4.0E−12	4.0E−15	5.1E−15	5.9E−15
	(3.3E−4)	(8.6E−8)	(4.0E−11)	(2.4E−14)	(3.3E−15)	(1.6E−15)

Table 6
As in Table 4 for the additional test functions in Fig. 2

	n = 10	n = 20	n = 30	n = 40	n = 50	n = 60
F7	2.1E−1	4.0E−6	3.3E−13	9.0E−15	1.9E−14	1.4E−14
	(7.3E−1)	(1.6E−4)	(2.6E−11)	(7.0E−15)	(6.0E−15)	(7.0E−15)
F8	1.2E−1	2.3E−3	1.7E−5	4.3E−8	4.0E−11	2.1E−14
	(2.4E−1)	(1.1E−2)	(1.3E−4)	(4.8E−7)	(6.1E−10)	(3.0E−13)
F9	3.3E−1	4.6E−3	2.0E−5	5.3E−8	8.9E−11	1.1E−13
	(9.8E−1)	(3.7E−2)	(2.5E−4)	(7.0E−7)	(1.1E−9)	(1.6E−12)
F10	5.5E−1	1.2E−1	6.4E−2	4.0E−2	2.8E−2	2.0E−2
	(8.7E−1)	(7.0E−2)	(1.9E−2)	(9.1E−3)	(5.4E−3)	(3.7E−3)

We stress that hyperinterpolation at MPX points is very stable. Indeed, we could hyperinterpolate at much higher degrees without drawbacks. For example, we can take $n = 300$ ($N = 45{,}602$ MPX points), obtaining an error of 3.6×10^{-12} for the test function F2.

Finally, in Table 7 we report the CPU times for construction and evaluation of the hyperinterpolation polynomial at the $M = 10{,}000$ target points belonging to the control grid. The tests have been performed with Matlab 6.5 on a AMD Athlon 2800+ processor machine. As expected from the complexity analysis in Remark 1 since $M \gg N$, the evaluation time is clearly dominant. It is worth noticing that the increase of the evaluation (and of the total) CPU time is linear in the degree, and not quadratic as expected from the flops estimates. This can be ascribed to the fact that, due to internal Matlab/Octave optimizations of matrix operations (see, e.g. [11]), the dominant execution time is given by the computation of the Chebyshev polynomials arrays at the target points in (19).

Table 7
CPU times (s) for construction (excluding evaluation of f) and evaluation of the hyperinterpolation polynomial at $M = 10{,}000$ target points; cf. Remark 1

	n = 10	n = 20	n = 30	n = 40	n = 50	n = 60
Constr. time	0.0015	0.0095	0.011	0.03	0.06	0.1
Eval. time	0.17	0.29	0.44	0.59	0.75	0.89

Acknowledgements

References

[1] Z. Battles, L.N. Trefethen, An extension of MATLAB to continuous functions and operators, SIAM J. Sci. Comput. 25 (2004) 1743–1770.
[2] L. Bos, M. Caliari, S. De Marchi, M. Vianello, A numerical study of the Xu polynomial interpolation formula, Computing 76 (2005) 311–324.
[3] L. Bos, M. Caliari, S. De Marchi, M. Vianello, Bivariate interpolation at Xu points: results, extensions and applications, Electron. Trans. Numer. Anal. 25 (2006) 1–16.
[4] L. Bos, S. De Marchi, M. Vianello, On the Lebesgue constant for the Xu interpolation formula, J. Approx. Theory (in press) (available online 17 April 2006).
[5] M. Caliari, S. De Marchi, M. Vianello, Bivariate polynomial interpolation on the square at new nodal sets, Appl. Math. Comput. 165 (2005) 261–274.
[6] M. Caliari, S. De Marchi, M. Vianello, Hyperinterpolation on the square, J. Comput. Appl. Math. (in press) (preprint available at www.math.unipd.it/~marcov/publications.html).
[7] M. Caliari, S. De Marchi, R. Montagna, M. Vianello, Hyper2d: a Matlab/Octave interface for hyperinterpolation on rectangles, downloadable from www.math.unipd.it/~marcov/software.html.
[8] C.F. Dunkl, Y. Xu, Orthogonal Polynomials of Several Variables, Encyclopedia of Mathematics and its Applications, vol. 81, Cambridge University Press, Cambridge, 2001.
[9] R. Franke, Scattered data interpolation: tests of some methods, Math. Comput. 38 (1982) 181–200.
[10] K. Hesse, I.H. Sloan, Hyperinterpolation on the sphere, UNSW School of Mathematics, preprint AMR05/23, 2005.
[11] C. Moler, MATLAB incorporates LAPACK. Increasing the speed and capabilities of matrix computation, MATLAB News & Notes – Winter 2000.
[12] C.R. Morrow, T.N.L. Patterson, Construction of algebraic cubature rules using polynomial ideal theory, SIAM J. Numer. Anal. 15 (1978) 953–976.
[13] M. Reimer, Multivariate Polynomial Approximation, International Series of Numerical Mathematics, vol. 144, Birkhäuser, Basel, 2003.
[14] R.J. Renka, Algorithm 792: Accuracy tests of ACM algorithms for interpolation of scattered data in the plane, ACM Trans. Math. Software 25 (1999) 79–93.
[15] I.H. Sloan, Polynomial interpolation and hyperinterpolation over general regions, J. Approx. Theory 83 (1995) 238–254.
[16] Y. Xu, Lagrange interpolation on Chebyshev points of two variables, J. Approx. Theory 87 (1996) 220–238.