Numerical Algorithms manuscript No.
(will be inserted by the editor)

Padua2DM: fast interpolation and cubature at the Padua points in Matlab/Octave

Marco Caliari · Stefano De Marchi · Alvise Sommariva · Marco Vianello

Received: January 20, 2010

Abstract We have implemented in Matlab/Octave two fast algorithms for bivariate Lagrange interpolation at the so-called Padua points on rectangles, and the corresponding versions for algebraic cubature.

Keywords Padua points, fast algorithms, Lagrange interpolation, Fast Fourier Transform, algebraic cubature.

1 The Padua points

In this paper, we discuss an efficient implementation in Matlab/Octave (cf. [11, 10]) of bivariate interpolation and cubature at the so-called Padua points. Such points are the first known example of optimal points for total degree polynomial interpolation in two variables, with a Lebesgue constant increasing like log square of the degree; see [1,2,4,5]. Moreover, the associated algebraic cubature formula has shown a very good behavior, comparable to that of the one-dimensional Clenshaw–Curtis rule, cf. [13].

Denoting by $\mathbb{P}_n^2$ the set of bivariate polynomials of total degree at most n, the $N = (n+1)(n+2)/2 = \dim\left(\mathbb{P}_n^2\right)$ Padua points ($n > 0$) are the set of

M. Caliari
University of Verona
Ca' Vignal 2
Strada Le Grazie, 15
37134 Verona (Italy)
E-mail: {marco.caliari}@univr.it

S. De Marchi, A. Sommariva, and M. Vianello
University of Padua
Via Trieste, 63
35121 Padova (Italy)
E-mail: {demarchi,marcov,alvise}@math.unipd.it

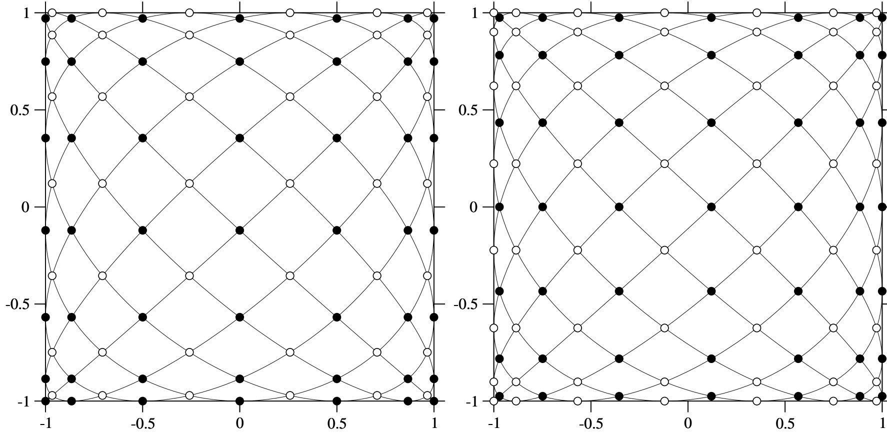

Fig. 1 The Padua points with their generating curve for $n = 12$ (left, 91 points) and $n = 13$ (right, 105 points), also as union of two Chebyshev-like grids: filled bullets $= C^{\mathrm{E}}_{n+1} \times C^{\mathrm{O}}_{n+2}$, open bullets $= C^{\mathrm{O}}_{n+1} \times C^{\mathrm{E}}_{n+2}$.

points

$$\mathrm{Pad}_n = \{\boldsymbol{\xi} = (\xi_1, \xi_2)\} = \left\{\gamma\left(\frac{k\pi}{n(n+1)}\right), \quad k = 0, \ldots, n(n+1)\right\}$$

where $\gamma(t)$ is their "generating curve" (cf. [1])

$$\gamma(t) = (-\cos((n+1)t), -\cos(nt)), \quad t \in [0, \pi] \tag{1}$$

Notice that two of the points are consecutive vertices of the square, $2n - 1$ other points lie on the edges of the square, and the remaining (interior) points are double points corresponding to self-intersections of the generating curve (see Fig. 1).

The Padua points (for n even) were introduced for the first time in [4, formula (9)] (in that formula there is a misprint, $n - 1$ has to be replaced by $n + 1$). Denoting by C_{n+1} the set of the $n + 1$ Chebyshev–Gauss–Lobatto points

$$C_{n+1} = \{z^n_j = \cos((j-1)\pi/n),\ j = 1, \ldots, n+1\}$$

and

$$C^{\mathrm{E}}_{n+1} = \{z^n_j \in C_{n+1},\ j-1 \text{ even}\}$$
$$C^{\mathrm{O}}_{n+1} = \{z^n_j \in C_{n+1},\ j-1 \text{ odd}\}$$

then

$$\mathrm{Pad}_n = (C^{\mathrm{E}}_{n+1} \times C^{\mathrm{O}}_{n+2}) \cup (C^{\mathrm{O}}_{n+1} \times C^{\mathrm{E}}_{n+2}) \subset C_{n+1} \times C_{n+2}$$

which is valid also for n odd.

The fundamental Lagrange polynomials of the Padua points are

$$L_{\boldsymbol{\xi}}(\boldsymbol{x}) = w_{\boldsymbol{\xi}}\left(K_n(\boldsymbol{\xi}, \boldsymbol{x}) - \frac{1}{2}\hat{T}_n(\xi_1)\hat{T}_n(x_1)\right) \tag{2}$$

where $K_n(\boldsymbol{x}, \boldsymbol{y})$, with $\boldsymbol{x} = (x_1, x_2)$ and $\boldsymbol{y} = (y_1, y_2)$, is the reproducing kernel of the space $\mathbb{P}_n^2([-1,1]^2)$ with the inner product

$$\langle f, g\rangle = \frac{1}{\pi^2}\int_{[-1,1]^2} f(x_1,x_2)g(x_1,x_2)\frac{\mathrm{d}x_1}{\sqrt{1-x_1^2}}\frac{\mathrm{d}x_2}{\sqrt{1-x_2^2}} \tag{3}$$

that is

$$K_n(\boldsymbol{x}, \boldsymbol{y}) = \sum_{k=0}^{n}\sum_{j=0}^{k}\hat{T}_j(x_1)\hat{T}_{k-j}(x_2)\hat{T}_j(y_1)\hat{T}_{k-j}(y_2)$$

Here $\hat{T}_j$ denotes the scaled Chebyshev polynomial of degree j, i.e. $\hat{T}_0 = T_0 \equiv 1$, $\hat{T}_p = \sqrt{2}T_p$, $T_p(\cdot) = \cos(p\arccos(\cdot))$, and $\{\hat{T}_j(x_1)\hat{T}_{k-j}(x_2),\ 0 \le j \le k \le n\}$ is the product Chebyshev orthonormal basis corresponding to (3) (cf. [8]). Moreover, the weights $w_{\boldsymbol{\xi}}$ are

$$w_{\boldsymbol{\xi}} = \frac{1}{n(n+1)} \cdot \begin{cases} 1/2 & \text{if } \boldsymbol{\xi} \text{ is a vertex point} \\ 1 & \text{if } \boldsymbol{\xi} \text{ is an edge point} \\ 2 & \text{if } \boldsymbol{\xi} \text{ is an interior point} \end{cases}$$

We notice that the $\{w_{\boldsymbol{\xi}}\}$ are indeed weights of a cubature formula for the product Chebyshev measure in (3), which is exact on "almost all" polynomials in $\mathbb{P}_{2n}^2([-1,1]^2)$, namely on all polynomials orthogonal to $T_{2n}(x_1)$. Such a cubature formula is derived from quadrature along the generating curve and is the key to obtaining the fundamental Lagrange polynomials (2); cf. [1].

Finally, extension to degree $n = 0$ is trivial: if we take as unique Padua point the point $\boldsymbol{\xi} = (-1,-1)$ given for $t = 0$ by the generating curve (1) and the corresponding Lagrange polynomial (2), then it follows that $w_{\boldsymbol{\xi}} = 2$.

2 Fast computation of the interpolation coefficients

The polynomial interpolation formula can be written in the bivariate Chebyshev orthonormal basis as

$$\mathcal{L}_n f(\boldsymbol{x}) = \sum_{\boldsymbol{\xi}\in\mathrm{Pad}_n} f(\boldsymbol{\xi})w_{\boldsymbol{\xi}}\left(K_n(\boldsymbol{\xi}, \boldsymbol{x}) - \frac{1}{2}\hat{T}_n(\xi_1)\hat{T}_n(x_1)\right)$$

$$= \sum_{k=0}^{n}\sum_{j=0}^{k} c_{j,k-j}\hat{T}_j(x_1)\hat{T}_{k-j}(x_2) - \frac{1}{2}\sum_{\boldsymbol{\xi}\in\mathrm{Pad}_n} f(\boldsymbol{\xi})w_{\boldsymbol{\xi}}\hat{T}_n(\xi_1)\hat{T}_0(\xi_2)\hat{T}_n(x_1)\hat{T}_0(x_2)$$

$$= \sum_{k=0}^{n}\sum_{j=0}^{k} c_{j,k-j}\hat{T}_j(x_1)\hat{T}_{k-j}(x_2) - \frac{c_{n,0}}{2}\hat{T}_n(x_1)\hat{T}_0(x_2) \tag{4}$$

where the coefficients are defined as

$$c_{j,k-j} = \sum_{\boldsymbol{\xi}\in\mathrm{Pad}_n} f(\boldsymbol{\xi})w_{\boldsymbol{\xi}}\hat{T}_j(\xi_1)\hat{T}_{k-j}(\xi_2), \quad 0 \le j \le k \le n \tag{5}$$

We can define the $(n+1)\times(n+2)$ matrix computed corresponding to the Chebyshev-like grid $C_{n+1}\times C_{n+2}$ with entries

$$\mathbb{G}(f) = (g_{r,s}) = \begin{cases} w_{\boldsymbol{\eta}} f(\boldsymbol{\eta}) & \text{if } \boldsymbol{\eta} = (z_r^n, z_s^{n+1}) \in \mathrm{Pad}_n \\ 0 & \text{if } \boldsymbol{\eta} = (z_r^n, z_s^{n+1}) \in (C_{n+1}\times C_{n+2}) \setminus \mathrm{Pad}_n \end{cases}$$

In [5] we computed the coefficients (5) by a double matrix-matrix product involving the matrix $\mathbb{G}(f)$. For the sake of completeness, we report again that construction. Given a vector $S=(s_1,\dots,s_m)\in[-1,1]^m$, first we define the rectangular Chebyshev matrix

$$\mathbb{T}(S) = \begin{pmatrix} \hat{T}_0(s_1) & \cdots & \hat{T}_0(s_m) \\ \vdots & \cdots & \vdots \\ \hat{T}_n(s_1) & \cdots & \hat{T}_n(s_m) \end{pmatrix} \in \mathbb{R}^{(n+1)\times m} \tag{6}$$

Then it is easy to check that the coefficients $c_{j,l}$, $0\le j\le n$, $0\le l\le n-j$ are the entries of the upper-left triangular part of the matrix

$$\mathbb{C}(f) = \mathbb{T}(C_{n+1})\,\mathbb{G}(f)\,(\mathbb{T}(C_{n+2}))^t \tag{7}$$

where, with a little abuse of notation, $C_{n+1}=(z_1^n,\dots,z_{n+1}^n)$ is the *vector* of the Chebyshev–Gauss–Lobatto points, too. We shall term (7) MM-old algorithm, cf. [5].

First, we present here a more refined matrix algorithm, by exploiting the fact that the Padua points are union of two Chebyshev subgrids. Indeed, defining the two matrices

$$\mathbb{G}_1(f) = \left(w_{\boldsymbol{\xi}} f(\boldsymbol{\xi}),\, \boldsymbol{\xi} = (z_r^n, z_s^{n+1}) \in C_{n+1}^{\mathrm{E}} \times C_{n+2}^{\mathrm{O}}\right)$$

$$\mathbb{G}_2(f) = \left(w_{\boldsymbol{\xi}} f(\boldsymbol{\xi}),\, \boldsymbol{\xi} = (z_r^n, z_s^{n+1}) \in C_{n+1}^{\mathrm{O}} \times C_{n+2}^{\mathrm{E}}\right)$$

then we can compute the coefficient matrix as

$$\mathbb{C}(f) = \mathbb{T}(C_{n+1}^{\mathrm{E}})\,\mathbb{G}_1(f)\,(\mathbb{T}(C_{n+2}^{\mathrm{O}}))^t + \mathbb{T}(C_{n+1}^{\mathrm{O}})\,\mathbb{G}_2(f)\,(\mathbb{T}(C_{n+2}^{\mathrm{E}}))^t \tag{8}$$

by storing and multiplying matrices of smaller dimension than those in (7). Indeed, in (8) the $\mathbb{T}$ matrices have size about $n\times n/2$ and the $\mathbb{G}$ matrices about $n/2\times n/2$, whereas all matrices in (7) have size about $n\times n$. We term this method MM (Matrix Multiplication) in the numerical tests.

Moreover, we pursue an alternative computational strategy, based on the special structure of the Padua points. Indeed, the coefficients $c_{j,l}$ can be rewritten as

$$c_{j,l} = \sum_{\boldsymbol{\xi}\in\mathrm{Pad}_n} f(\boldsymbol{\xi}) w_{\boldsymbol{\xi}} \hat{T}_j(\xi_1)\hat{T}_l(\xi_2) = \sum_{r=0}^{n}\sum_{s=0}^{n+1} g_{r,s}\hat{T}_j(z_r^n)\hat{T}_l(z_s^{n+1})$$

$$= \beta_{j,l}\sum_{r=0}^{n}\sum_{s=0}^{n+1} g_{r,s}\cos\frac{jr\pi}{n}\cos\frac{ls\pi}{n+1} = \beta_{j,l}\sum_{s=0}^{M-1}\left(\sum_{r=0}^{N-1} g^0_{r,s}\cos\frac{2jr\pi}{N}\right)\cos\frac{2ls\pi}{M}$$

Input: `Gf` $\leftrightarrow \mathbb{G}(f)$

```
% compute the coefficient matrix by a double FFT
Gfhat = real(fft(Gf,2*n)); % 1-dimensional FFT along columns
Gfhat = Gfhat(1:n+1,:);
Gfhathat = real(fft(Gfhat,2*(n+1),2)); % 1-dimensional FFT along rows
Cf = Gfhathat(:,1:n+1);
Cf = 2*Cf;
Cf(1,:) = Cf(1,:)/sqrt(2);
Cf(:,1) = Cf(:,1)/sqrt(2);
% compute the interpolation coefficient matrix
C0f = fliplr(triu(fliplr(Cf))); % extract the upper left triangular part
C0f(n+1,1) = C0f(n+1,1)/2;
```

Output: `C0f` $\leftrightarrow \mathbb{C}_0(f)$

Table 1 Fragment of Matlab/Octave code for the fast computation of the interpolation coefficient matrix.

where $N = 2n$, $M = 2(n+1)$ and

$$\beta_{j,l} = \begin{cases} 1 & j = l = 0 \\ 2 & j \neq 0,\ l \neq 0 \\ \sqrt{2} & \text{otherwise} \end{cases} \qquad g^0_{r,s} = \begin{cases} g_{r,s} & 0 \le r \le n \text{ and } 0 \le s \le n+1 \\ 0 & r > n \text{ or } s > n+1 \end{cases}$$

Then, it is possible to recover the coefficients $c_{j,l}$ by a double Discrete Fourier Transform, namely

$$\begin{aligned} \hat{g}_{j,s} &= \mathrm{Re}\left(\sum_{r=0}^{N-1} g^0_{r,s} \mathrm{e}^{-2\pi \mathrm{i} jr/N}\right), \quad 0 \le j \le n,\ 0 \le s \le M-1 \\ \frac{c_{j,l}}{\beta_{j,l}} = \hat{\hat{g}}_{j,l} &= \mathrm{Re}\left(\sum_{s=0}^{M-1} \hat{g}_{j,s} \mathrm{e}^{-2\pi \mathrm{i} ls/M}\right), \quad 0 \le j \le n,\ 0 \le l \le n-j \end{aligned} \tag{9}$$

The Matlab/Octave code for the computation of the interpolation coefficient matrix by a double Fast Fourier Transform (the FFT-based method) is reported in Table 1. We note that all indexes starting from 0 are shifted by 1 (as required in Matlab/Octave) and that we are using the Matlab subindexing notation to identify submatrices.

According to [5], we call $\mathbb{C}_0(f)$ the interpolation coefficient matrix

$$\mathbb{C}_0(f) = (c'_{j,l}) = \begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & \cdots & c_{0,n} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,n-1} & 0 \\ \vdots & \vdots & \iddots & \iddots & \vdots \\ c_{n-1,0} & c_{n-1,1} & 0 & \cdots & 0 \\ \frac{c_{n,0}}{2} & 0 & \cdots & 0 & 0 \end{pmatrix} \in \mathbb{R}^{(n+1)\times(n+1)} \tag{10}$$

which is essentially the upper-left triangular part of the matrix $\mathbb{C}(f)$, but the modification on the last element of the first column.

Input: `X`$\leftrightarrow \boldsymbol{X}$, `grid`, `C0f`$\leftrightarrow \mathbb{C}_0(f)$

```
% compute the Chebyshev matrices
TX1 = cos([0:n]'*acos(X(:,1)'));
TX2 = cos([0:n]'*acos(X(:,2)'));
TX1(2:n+1,:) = TX1(2:n+1,:)*sqrt(2);
TX2(2:n+1,:) = TX2(2:n+1,:)*sqrt(2);
if (grid == false)
% LnfX corresponds to the set of scattered points [X(:,1),X(:,2)]
  LnfX = sum((TX1'*C0f).*TX2',2);
else
% LnfX corresponds to the grid of points meshgrid(X(:,1),X(:,2))
  LnfX = (TX1'*C0f*TX2)';
end;
```

Output: `LnfX`$\leftrightarrow \mathcal{L}_n f(\boldsymbol{X})$

Table 2 Fragment of Matlab/Octave code for the evaluation of the interpolation formula on a vector or on a grid of target points.

Remark 1 It is worth comparing the computational complexity, say $c(n)$, of the three methods described by formulas (7), (8) and (9) for the computation of the interpolation coefficients. It is easy to see that, asymptotically, $c(n) \sim 4n^3$ for (7), $c(n) \sim 2n^3$ for (8), and $c(n) = \mathcal{O}(n^2 \log n)$ for (9). Nevertheless, as shown in Section 4, the matrix multiplication methods turn out to be competitive with the FFT-based method up to high degrees.

2.1 Evaluation of the interpolant

For a given function f and degree n, the interpolation coefficient matrix $\mathbb{C}_0(f)$ can be computed once and for all. Then, it is easy to see that the polynomial interpolation formula (4) can be evaluated at any $\boldsymbol{x} = (x_1, x_2) \in [-1, 1]^2$ by

$$\mathcal{L}_n f(\boldsymbol{x}) = (\mathbb{T}(x_1))^t \; \mathbb{C}_0(f) \; \mathbb{T}(x_2)$$

It is also possible to evaluate the polynomial interpolation formula on a set $\boldsymbol{X}$ of target points, at the same time. Given the vector X_1 of the first components of a set of target points and the vector X_2 of the corresponding second components, then

$$\mathcal{L}_n f(\boldsymbol{X}) = \operatorname{diag}\left((\mathbb{T}(X_1))^t \, \mathbb{C}_0(f) \mathbb{T}(X_2)\right) \tag{11}$$

The result $\mathcal{L}_n f(\boldsymbol{X})$ is a (column) vector containing the evaluation of the interpolation polynomial at the corresponding target points. There exists a nice way to compute (11) without performing the two whole matrix products, whose Matlab/Octave code is given in Table 2 (`grid == false` branch).

If the target points are a Cartesian grid $\boldsymbol{X} = X_1 \times X_2$, then it is possible to evaluate the polynomial interpolation in a more compact form

$$\mathcal{L}_n f(\boldsymbol{X}) = \left((\mathbb{T}(X_1))^t \, \mathbb{C}_0(f) \mathbb{T}(X_2)\right)^t \tag{12}$$

The result $\mathcal{L}_n f(\boldsymbol{X})$ is a matrix whose i-th row and j-th column contains the evaluation of the interpolation polynomial at the point with first component the j-th element in X_1 and second component the i-th element in X_2. In fact, this is the usual way a Cartesian grid is constructed in Matlab/Octave, via the built-in function `meshgrid` (see [11]). Also instance (12) is implemented in the code (`else` branch in Table 2).

Remark 2 We notice that the interpolation formulas above can be immediately extended to arbitrary rectangles $[a, b] \times [c, d]$, by the standard affine transformation. This possibility is automatically managed by the code.

3 Fast computation of the cubature weights

In a recent paper [13], the interpolatory cubature formula corresponding to the Padua points has been studied. It has been called "nontensorial Clenshaw–Curtis cubature" since it is a bivariate analogue of the classical Clenshaw–Curtis quadrature formula (cf. [6]), in the total-degree polynomial space. From the results of the previous section, we can write

$$\int_{[-1,1]^2} f(\boldsymbol{x})\mathrm{d}\boldsymbol{x} \approx I_n(f) = \int_{[-1,1]^2} \mathcal{L}_n f(\boldsymbol{x})\mathrm{d}\boldsymbol{x} = \sum_{k=0}^{n}\sum_{j=0}^{k} c'_{j,k-j}\, m_{j,k-j}$$

$$= \sum_{j=0}^{n}\sum_{l=0}^{n} c'_{j,l}\, m_{j,l} = \sum_{\substack{j=0 \\ j \text{ even}}}^{n}\ \sum_{\substack{l=0 \\ l \text{ even}}}^{n} c'_{j,l}\, m_{j,l} \tag{13}$$

where the $m_{j,l}$ are the *Chebyshev moments*, i.e.

$$m_{j,l} = \int_{-1}^{1} \hat{T}_j(t)\mathrm{d}t \int_{-1}^{1} \hat{T}_l(t)\mathrm{d}t$$

We have

$$\int_{-1}^{1} \hat{T}_j(t)\mathrm{d}t = \begin{cases} 2 & j = 0 \\ 0 & j \text{ odd} \\ \dfrac{2\sqrt{2}}{1-j^2} & j \text{ even} \end{cases}$$

and then, defining the Chebyshev even-moment matrix

$$\mathbb{M} = (m_{j,l}), \quad 0 \le j, l \le n,\ j \text{ even},\, l \text{ even}$$

the cubature formula (13) can be evaluated by the Matlab/Octave code reported in Table 3, where we have used the fact that only the (even,even) pairs of indexes are active in the summation process.

On the other hand, it is often desiderable to have a cubature formula that involves only the function values at the nodes and the corresponding cubature weights. Again, a simple matrix formulation is available, using the fact that

Input: `C0f` $\leftrightarrow \mathbb{C}_0(f)$

```
% compute the even Chebyshev moments
k = [0:2:n]';
mom = 2*sqrt(2)./(1-k.^2);
mom(1) = 2;
% compute the Chebyshev even-moment matrix
[M1,M2] = meshgrid(mom);
M = M1.*M2;
% compute the moment-based cubature formula
Int = sum(sum(C0f(1:2:n+1,1:2:n+1).*M));
```

Output: `Int` $\leftrightarrow I_n(f)$

Table 3 Fragment of Matlab/Octave code for the evaluation of the cubature formula by moments.

the Padua points are the union of two subgrids of product Chebyshev points. First, observe that

$$I_n(f) = \sum_{j \text{ even}}^{n} \sum_{l \text{ even}}^{n} c'_{j,l}\, m_{j,l} = \sum_{\substack{j=0 \\ j \text{ even}}}^{n} \sum_{\substack{l=0 \\ l \text{ even}}}^{n} c_{j,l}\, m'_{j,l}$$

where $m'_{n,0} = m_{n,0}/2$ for n even, and $m'_{n-1,0} = m_{n-1,0}$ for n odd.

Now, using the formula for the coefficients (5) we can write

$$I_n(f) = \sum_{\boldsymbol{\xi} \in \mathrm{Pad}_n} \lambda_{\boldsymbol{\xi}}\, f(\boldsymbol{\xi})$$

$$= \sum_{\boldsymbol{\xi} \in C^{\mathrm{E}}_{n+1} \times C^{\mathrm{O}}_{n+2}} \lambda_{\boldsymbol{\xi}}\, f(\boldsymbol{\xi}) + \sum_{\boldsymbol{\xi} \in C^{\mathrm{O}}_{n+1} \times C^{\mathrm{E}}_{n+2}} \lambda_{\boldsymbol{\xi}}\, f(\boldsymbol{\xi})$$

where

$$\lambda_{\boldsymbol{\xi}} = w_{\boldsymbol{\xi}} \sum_{\substack{j=0 \\ j \text{ even}}}^{n} \sum_{\substack{l=0 \\ l \text{ even}}}^{n} m'_{j,l}\, \hat{T}_j(\xi_1) \hat{T}_l(\xi_2) \tag{14}$$

It is convenient to define the *modified* Chebyshev even-moment matrix

$$\mathbb{M}_0 = (m'_{j,l}) = \begin{pmatrix} m_{0,0} & m_{0,2} & \cdots & \cdots & m_{0,p_n} \\ m_{2,0} & m_{2,2} & \cdots & m_{2,p_n-2} & 0 \\ \vdots & \vdots & \iddots & \iddots & \vdots \\ m_{p_n-2,0} & m_{p_n-2,2} & 0 & \cdots & 0 \\ m'_{p_n,0} & 0 & \cdots & 0 & 0 \end{pmatrix} \in \mathbb{R}^{([\frac{n}{2}]+1)\times([\frac{n}{2}]+1)}$$

where $p_n = n$ for n even, and $p_n = n-1$ for n odd, the Chebyshev matrix corresponding to even degrees (cf. (6))

$$\mathbb{T}^{\mathrm{E}}(S) = \begin{pmatrix} \hat{T}_0(s_1) & \cdots & \hat{T}_0(s_m) \\ \hat{T}_2(s_1) & \cdots & \hat{T}_2(s_m) \\ \vdots & \cdots & \vdots \\ \hat{T}_{p_n}(s_1) & \cdots & \hat{T}_{p_n}(s_m) \end{pmatrix} \in \mathbb{R}^{([\frac{n}{2}]+1)\times m}$$

Input: `W1`$\leftrightarrow$ $\left(w_{\boldsymbol{\xi}},\, \boldsymbol{\xi} \in C^{\mathrm{E}}_{n+1} \times C^{\mathrm{O}}_{n+2}\right)^t$, `W2`$\leftrightarrow$ $\left(w_{\boldsymbol{\xi}},\, \boldsymbol{\xi} \in C^{\mathrm{O}}_{n+1} \times C^{\mathrm{E}}_{n+2}\right)^t$

```
% compute the Chebyshev matrices on the two subgrids
argn1 = linspace(0,pi,n+1);
argn2 = linspace(0,pi,n+2);
k = [0:2:n]';
l = (n-mod(n,2))/2+1;
TE1 = cos(k*argn1(1:2:n+1));
TE1(2:l,:) = TE1(2:l,:)*sqrt(2);
TO1 = cos(k*argn1(2:2:n+1));
TO1(2:l,:) = TO1(2:l,:)*sqrt(2);
TE2 = cos(k*argn2(1:2:n+2));
TE2(2:l,:) = TE2(2:l,:)*sqrt(2);
TO2 = cos(k*argn2(2:2:n+2));
TO2(2:l,:) = TO2(2:l,:)*sqrt(2);
% compute the modified Chebyshev even-moment matrix
mom = 2*sqrt(2)./(1-k.^2);
mom(1) = 2;
[M1,M2] = meshgrid(mom);
M = M1.*M2;
M0 = fliplr(triu(fliplr(M))); % extract the upper left triangular part
if (mod(n,2) == 0)
  M0(n/2+1,1) = M0(n/2+1,1)/2;
end
% compute the cubature weights on the two subgrids
L1 = W1.*(TE1'*M0*TO2)';
L2 = W2.*(TO1'*M0*TE2)';
```

Output: `L1`$\leftrightarrow$ $\left(\lambda_{\boldsymbol{\xi}},\, \boldsymbol{\xi} \in C^{\mathrm{E}}_{n+1} \times C^{\mathrm{O}}_{n+2}\right)^t$, `L2`$\leftrightarrow$ $\left(\lambda_{\boldsymbol{\xi}},\, \boldsymbol{\xi} \in C^{\mathrm{O}}_{n+1} \times C^{\mathrm{E}}_{n+2}\right)^t$

Table 4 Fragment of Matlab/Octave code for the computation of the cubature weights.

and the matrices of interpolation weights on the subgrids of Padua points,

$$\mathbb{W}_1 = \left(w_{\boldsymbol{\xi}},\, \boldsymbol{\xi} = (z^n_r, z^{n+1}_s) \in C^{\mathrm{E}}_{n+1} \times C^{\mathrm{O}}_{n+2}\right)^t$$
$$\mathbb{W}_2 = \left(w_{\boldsymbol{\xi}},\, \boldsymbol{\xi} = (z^n_r, z^{n+1}_s) \in C^{\mathrm{O}}_{n+1} \times C^{\mathrm{E}}_{n+2}\right)^t$$

It is then easy to show that the cubature weights $\{\lambda_{\boldsymbol{\xi}}\}$ can be computed in the matrix form

$$\mathbb{L}_1 = \left(\lambda_{\boldsymbol{\xi}},\, \boldsymbol{\xi} = (z^n_r, z^{n+1}_s) \in C^{\mathrm{E}}_{n+1} \times C^{\mathrm{O}}_{n+2}\right)^t = \mathbb{W}_1 \cdot \left(\mathbb{T}^{\mathrm{E}}(C^{\mathrm{E}}_{n+1}))^t \mathbb{M}_0 \mathbb{T}^{\mathrm{E}}(C^{\mathrm{O}}_{n+2})\right)^t$$
$$\mathbb{L}_2 = \left(\lambda_{\boldsymbol{\xi}},\, \boldsymbol{\xi} = (z^n_r, z^{n+1}_s) \in C^{\mathrm{O}}_{n+1} \times C^{\mathrm{E}}_{n+2}\right)^t = \mathbb{W}_2 \cdot \left(\mathbb{T}^{\mathrm{E}}(C^{\mathrm{O}}_{n+1}))^t \mathbb{M}_0 \mathbb{T}^{\mathrm{E}}(C^{\mathrm{E}}_{n+2})\right)^t$$

where the dot denotes the Hadamard (or Schur) product (entrywise product). The corresponding Matlab/Octave code is reported in Table 4. The definition of the weights matrices $\mathbb{L}_i$, $i = 1, 2$, makes use of transposes in order to be compatible with the Matlab/Octave meshgrid-like structure of the matrices $\mathbb{W}_i$ (see also (12)).

An alternative approach for the computation of the cubature weights is based on the observation that (14) itself is a double Discrete Fourier Transform, in some sense "dual" with respect to that for the interpolation coefficients (the roles of the points and of the indexes are interchanged). An FFT-based

implementation is then feasible, in analogy to what happens in the univariate case with the Clenshaw–Curtis formula (cf. [15]). The algorithm is quite similar to that in Table 1, and we do not describe it for brevity. A comparison of the computational complexities shows that $c(n) \sim n^3$ for the MM method and $c(n) = \mathcal{O}(n^2 \log n)$ for the FFT-based method.

It is worth recalling that the cubature weights are not all positive, but the negative ones are few and of small size. Indeed, the cubature formula is stable and convergent for every continuous integrand, since

$$\lim_{n\to\infty} \sum_{\boldsymbol{\xi}\in\mathrm{Pad}_n} |\lambda_{\boldsymbol{\xi}}| = 4$$

as it has been proved in [13].

Remark 3 As with interpolation, the code automatically manages cubature over arbitrary rectangles $[a,b] \times [c,d]$.

4 Numerical tests

In this section we present some numerical tests on the accuracy and performance of the various implementations of interpolation and cubature at the Padua points. All the experiments have been made by the Matlab/Octave package Padua2DM (see Section 5), run in Matlab 7.6.0 on an Intel Core2 Duo 2.20GHz processor. Similar results have been obtained with the self-compiled 3.2.3 version of Octave.

n	20	40	60	80	100	300	500	1000
MM-old	0.001	0.002	0.004	0.006	0.010	0.119	0.302	1.624
MM	0.002	0.003	0.003	0.008	0.008	0.101	0.298	1.353
FFT	0.001	0.001	0.001	0.002	0.003	0.034	0.115	0.387

Table 5 CPU time (in seconds, average over 100 runs) for the computation of the interpolation coefficients at a sequence of degrees.

n	20	40	60	80	100	300	500	1000
MM	0.001	0.001	0.001	0.002	0.003	0.027	0.092	0.554
FFT	0.001	0.001	0.002	0.002	0.004	0.028	0.111	0.389

Table 6 CPU time (in seconds, average over 100 runs) for the computation of the cubature weights at a sequence of degrees.

In Tables 5 and 6 we show the CPU times (seconds) for the computation of the interpolation coefficients and cubature weights at a sequence of degrees,

by the MM and the FFT-based algorithms. Concerning interpolation (Table 5) we also give a comparison with a Matlab/Octave implementation of the algorithm in [5], termed MM-old. Despite of the remarkable difference in the theoretical computational complexities, the matrix multiplication methods are comparable or even superior than the FFT-based up to high degrees. This is a well-known phenomenon in the computation of Discrete Fourier Transforms (see, for instance, [3, § 10.5]). Indeed, due to the use of optimized BLAS (Basic Linear Algebra Subprograms) by Matlab/Octave, the matrix multiplication methods are much more competitive than operation counts indicate.

The results suggest that, tendentially, the FFT-based method is preferable for interpolation, whereas the MM method is better for cubature. Indeed, the MM algorithm is more efficient than the FFT-based one in the cubature instance, since the matrices have lower dimension due to restriction to even indexes, and is competitive with the FFT up to very high degrees.

In Figure 2 we report the relative errors of interpolation (top) and cubature (bottom) versus the polynomial degree for the classical Franke test function in $[0,1]^2$ (cf. [9]), namely

$$\begin{aligned} f(x_1,x_2) = & \frac{3}{4}\exp(-((9x_1-2)^2+(9x_2-2)^2)/4) \\ & + \frac{3}{4}\exp(-(9x_1+1)^2/49-(9x_2+1)/10) \\ & + \frac{1}{2}\exp(-((9x_1-7)^2+(9x_2-3)^2)/4) \\ & - \frac{1}{5}\exp(-(9x_1-4)^2-(9x_2-7)^2) \end{aligned} \tag{15}$$

The interpolation errors are in the max norm on a suitable control mesh, normalized to the maximum deviation of the function from its mean. Here a second advantage of the FFT-based method for interpolation appears: it is able to attain close to machine precision, whereas the MM algorithm stagnates around 10^{-13}. On the contrary, the MM algorithm seems more stable for cubature than for interpolation setting, and reaches the machine precision error level. These observations have been confirmed by many other numerical experiments.

In Figures 3 and 4 we show the interpolation and cubature errors versus the number of points (i.e., of function evaluations), for a Gaussian

$$f(x_1,x_2) = \exp(-(x_1^2+x_2^2)), \quad (x_1,x_2) \in [-1,1]^2 \tag{16}$$

and a C^2 function (with third derivatives singular at the origin)

$$f(x_1,x_2) = (x_1^2+x_2^2)^{3/2}, \quad (x_1,x_2) \in [-1,1]^2 \tag{17}$$

Interpolation and cubature at the Padua points are compared with tensorial formulas, and in the case of cubature also with the few known minimal formulas (cf. [12]).

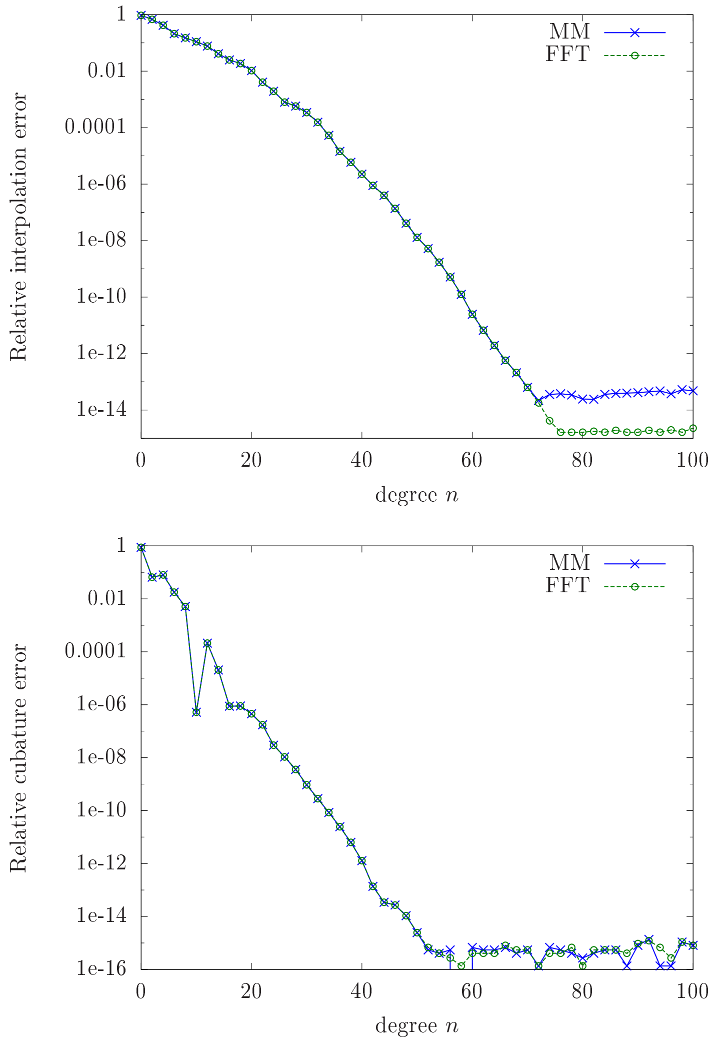

Fig. 2 Relative errors of interpolation (top) and cubature (bottom) versus the interpolation degree for the Franke test function (see (15)) in $[0,1]^2$, by the Matrix Multiplication (MM) and the FFT-based algorithms.

We see two opposite situations. Concerning interpolation, the Padua points perform better than tensor-product Chebyshev–Lobatto points only with regular functions. On the other hand, nontensorial cubature at the Padua points performs always better than tensorial Clenshaw–Curtis cubature (which, as known, uses tensor-product Chebyshev–Lobatto points). However, it is less accurate than tensorial Gauss–Legendre–Lobatto and minimal formulas on analytic entire functions, whereas it appears the best one, even with respect to the minimal formulas, on less regular functions. This phenomenon, confirmed

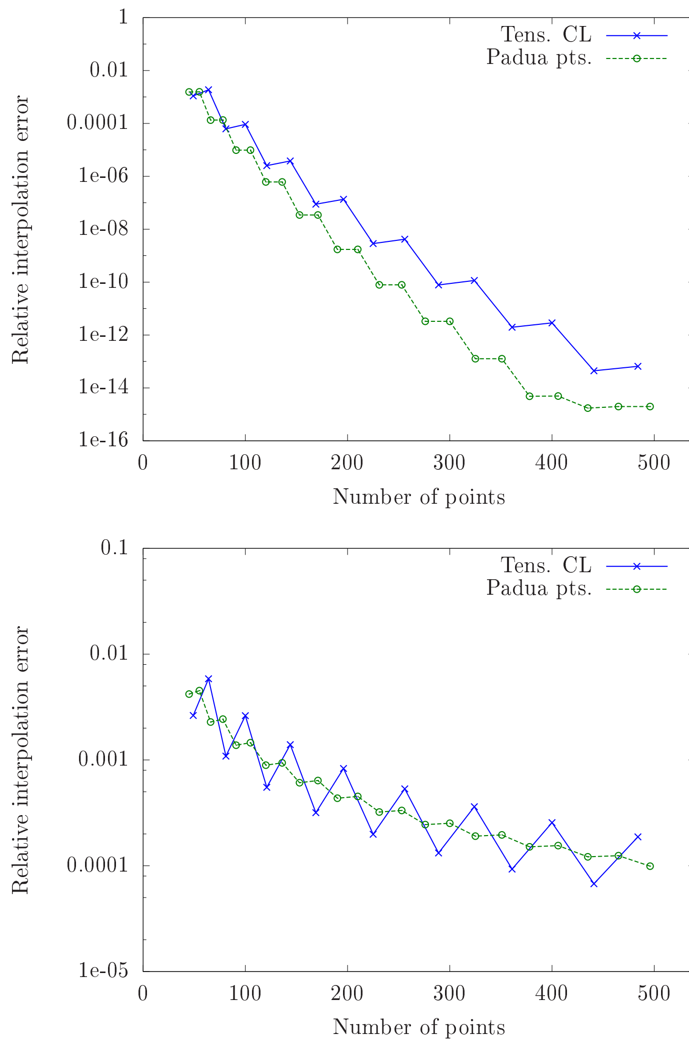

Fig. 3 Relative interpolation errors versus the number of interpolation points for the Gaussian (top, see (16)) and the C^2 function (bottom, see (17)) in $[-1,1]^2$; Tens. CL = Tensorial Chebyshev–Lobatto interpolation.

by many other examples (cf. [13]) and present also in 3d with nontensorial cubature at new sets of Chebyshev hyperinterpolation points (cf. [7]), is quite similar to that studied in the one-dimensional case for the classical Clenshaw–Curtis formula (cf. [14]), but is still theoretically unexplained in the multivariate setting. Nevertheless, numerical cubature at the Padua points seems to provide one of the best algebraic cubature formulas presently known for the square.

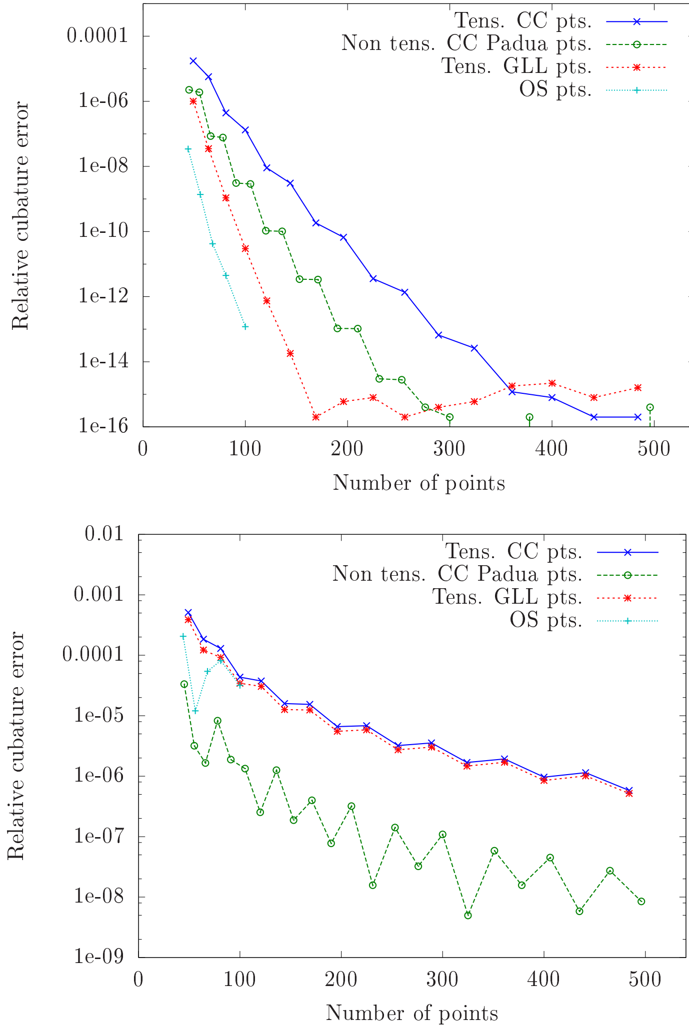

Fig. 4 Relative cubature errors versus the number of cubature points (CC = Clenshaw–Curtis, GLL = Gauss–Legendre–Lobatto, OS = Omelyan–Solovyan) for the Gaussian $f(\boldsymbol{x}) = \exp(-|\boldsymbol{x}|^2)$ (top) and the C^2 function $f(\boldsymbol{x}) = |\boldsymbol{x}|^3$ (bottom); the integration domain is $[-1,1]^2$, the integrals up to machine precision are, respectively: 2.230985141404135 and 2.508723139534059.

5 Code

The software Padua2DM runs both in Matlab and in Octave. It consists of two main functions: `pdint` for interpolation and `pdcub` for cubature on rectangles.

- `pdint`: calls three other functions: `pdpts`, that produces Padua points; `pdcfsFFT` that constructs the interpolation coefficient matrix (10) using

the FFT-based method, and `pdval` that evaluates the interpolant on a set of target points. For completeness we also provide the function `pdcfsMM`, which computes the interpolation coefficient matrix using the Matrix Multiplication (MM) method described in Section 2.

- `pdcub`: calls, as for `pdint`, the function `pdpts`, and `pdwtsMM` which computes the cubature weights using the MM method. For completeness we included also the function `pdwtsFFT` which uses the FFT-based method to compute the weights.

The package provides many demonstration scripts, whose names are `demo_*`, which basically reproduce all numerical experiments presented in the paper. The auxiliary functions `testfunct`, `cubature_square`, `omelyan_solovyan_rule` are used within the demo scripts.

For more details concerning input and output parameters and the usage of single functions, see the corresponding `help`.

The package has been successfully tested on Matlab 6.1.x, 6.5.x, 7.6.x and Octave 3.0.x, 3.1.x, 3.2.x.

The software is available from Netlib (`http://www.netlib.org/numeralgo/`) as `na29` package, as a single .tgz (tar zipped) archive.

References

1. Bos, L., Caliari, M., De Marchi, S., Vianello, M., Xu, Y.: Bivariate Lagrange interpolation at the Padua points: the generating curve approach. J. Approx. Theory 143, 15–25 (2006).
2. Bos, L., De Marchi, S., Vianello, M., Xu, Y.: Bivariate Lagrange interpolation at the Padua points: the ideal theory approach. Numer. Math. 108, 43–57 (2007).
3. Boyd, J. P.: Chebyshev and Fourier Spectral Methods, Second edition, Dover, New York (2001).
4. Caliari, M., De Marchi, S., Vianello, M.: Bivariate polynomial interpolation on the square at new nodal sets. Appl. Math. Comput. 165, 261–274 (2005).
5. Caliari, M., De Marchi, S., Vianello, M.: Algorithm 886: Padua2D: Lagrange Interpolation at Padua Points on Bivariate Domains. ACM Trans. Math. Software 35-3, 21:1–21:11 (2008).
6. Clenshaw, C.W., Curtis, A.R.: A method for numerical integration on an automatic computer. Numer. Math. 2, 197–205 (1960).
7. De Marchi, S., Vianello, M., Xu, Y.: New cubature formulae and hyperinterpolation in three variables. BIT Numerical Mathematics 49(1), 55–73 (2009).
8. Dunkl, C.F. and Xu, Y.: Orthogonal polynomials of several variables, Cambridge University Press, Cambridge, 2001.
9. Franke, R.: A critical comparison of some methods for interpolation of scattered data, Naval Postgraduate School Monterey CA, Tech. Rep. NPS-53-79-003, March 1979.
10. GNU Octave, `http://www.gnu.org/software/octave/`.
11. MATLAB, `http://www.mathworks.com/products/matlab/`.
12. Omelyan, I.P., Solovyan, V.B.: Improved cubature formulae of high degrees of exactness for the square. J. Comput. Appl. Math. 188, 190–204 (2006).
13. Sommariva, A., Vianello, M., Zanovello, R.: Nontensorial Clenshaw–Curtis cubature. Numer. Algorithms 49, 409–427 (2008).
14. Trefethen, L.N.: Is Gauss quadrature better than Clenshaw–Curtis?. SIAM Rev. 50, 67–87 (2008).
15. Waldvogel, J.: Fast construction of the Fejér and Clenshaw–Curtis quadrature rules. BIT Numerical Mathematics 46, 195–202 (2006).