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

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

**Abstract** We present a numerical code in Matlab/Octave that implements fast versions of the Lagrange interpolation formula, and of the corresponding algebraic cubature formula, at the so-called Padua points in rectangles.

**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 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 [1–4]. Moreover, the associated algebraic cubature formula has shown a very good behavior, comparable to that of the one-dimensional Clenshaw–Curtis rule, cf. [9].

The $N = (n + 1)(n + 2)/2 = \dim\left(\mathbb{P}_n^2\right)$ Padua points corresponding to degree $n > 0$ are the set of points

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

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

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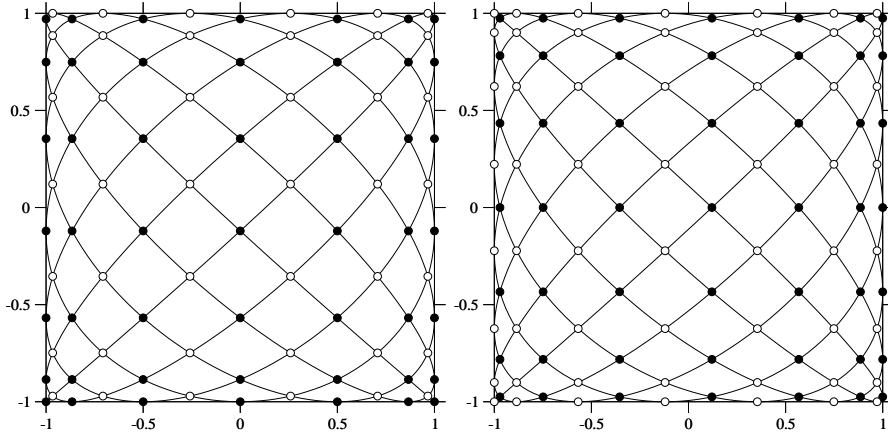E-mail: {marcov,alvise}@math.unipd.it

**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 $= C_{n+1}^{\mathrm{E}} \times C_{n+2}^{\mathrm{O}}$, open bullets $= C_{n+1}^{\mathrm{O}} \times C_{n+2}^{\mathrm{E}}$.

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 [3, 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

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

and

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

then

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

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 $\mathbb{P}_n^2([-1, 1]^2)$ equipped with the inner product

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

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 normalized Chebyshev polynomial of degree $j$, i.e. $\hat{T}_0 = T_0 \equiv 1$, $\hat{T}_p = \sqrt{2}T_p$, $T_p(\cdot) = \cos(p\arccos(\cdot))$. 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, which is exact on "almost all" polynomials in $\mathbb{P}^2_{2n}([-1,1]^2)$, namely on all polynomials orthogonal to $T_{2n}(x_1)$. Such a cubature formula stems 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

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

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

$$= \sum_{k=0}^{n} \sum_{j=0}^{k} c_{j,k-j}\hat{T}_j(x_1)\hat{T}_{k-j}(x_2) - \frac{c_{n,0}}{2}\hat{T}_n(x_1)\hat{T}_0(x_2) \tag{3}$$

where the coefficients are defined as

$$c_{j,k-j} = \sum_{\boldsymbol{\xi} \in \text{Pad}_n} f(\boldsymbol{\xi})w_{\boldsymbol{\xi}}\hat{T}_j(\xi_1)\hat{T}_{k-j}(\xi_2), \quad 0 \le j \le k \le n \tag{4}$$

and can be computed once and for all. 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

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

In [4] we computed the coefficients (4) 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 $S = (s_1, \ldots, s_m) \in [-1, 1]^m$, first we define the rectangular Chebyshev matrix

$$\mathbb{T}(S) = \begin{pmatrix} \hat{T}_0(s_1) & \cdots & \hat{T}_0(s_m) \\ \vdots & \cdots & \vdots \\ \hat{T}_n(s_1) & \cdots & \hat{T}_n(s_m) \end{pmatrix} \in \mathbb{R}^{(n+1) \times m} \tag{5}$$

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

$$\mathbb{C}(f) = \mathbb{T}(C_{n+1}) \, \mathbb{G}(f) \, (\mathbb{T}(C_{n+2}))^t \tag{6}$$

where, with a little abuse of notation, $C_{n+1} = (z_1^n, \ldots, z_{n+1}^n)$ is the *vector* of the Chebyshev–Gauss–Lobatto points, too.

A slightly more refined algorithm can be obtained, 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

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

by multiplying matrices of smaller dimension than those in (6). We term this approach MM (Matrix Multiplication) in the numerical tests.

In this paper, 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

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

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

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

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

Input: `Gf` ↔ $\mathbb{G}(f)$

```
Gfhat = real(fft(Gf,2*n));
Gfhat = Gfhat(1:n+1,:);
Gfhathat = real(fft(Gfhat,2*(n+1),2));
COf = Gfhathat(:,1:n+1);
COf = 2*COf;
COf(1,:) = COf(1,:)/sqrt(2);
COf(:,1) = COf(:,1)/sqrt(2);
COf = fliplr(triu(fliplr(COf)));
COf(n+1,1) = COf(n+1,1)/2;
```

Output: `COf` ↔ $\mathbb{C}_0(f)$

**Table 1** Matlab code for the fast computation of the coefficient matrix.

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

$$
\hat{g}_{j,s} = \text{REAL}\left(\sum_{r=0}^{N-1} g_{r,s}^0 e^{-2\pi i j r/N}\right), \quad 0 \le j \le n,\ 0 \le s \le M-1
$$

$$
\frac{c_{j,l}}{\beta_{j,l}} = \hat{\hat{g}}_{j,l} = \text{REAL}\left(\sum_{s=0}^{M-1} \hat{g}_{j,s} e^{-2\pi i l s/M}\right), \quad 0 \le j \le n,\ 0 \le l \le n-j
$$

(7)

According to [4], we call $\mathbb{C}_0(f)$ the interpolation coefficients matrix

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

(8)

which is essentially the upper-left triangular part of the matrix $\mathbb{C}(f)$ in (6), but the modification on the last element of the first column. The Matlab code for the computation of the coefficient matrix by a double Fast Fourier Transform is reported in Table 1. We just notice that all the indexes starting from 0 are shifted by 1 (as required in Matlab).

2.1 Evaluation of the interpolant

It is easy to see that the polynomial interpolation formula (3) 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

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

```
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 == true)
  LnfX = (TX1'*COf*TX2)';
else
  LnfX = sum((TX1'*COf).*TX2',2);
end;
```

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

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

components, then

$$\mathcal{L}_n f(\boldsymbol{X}) = \operatorname{diag}\left(\left(\mathbb{T}(X_1)\right)^t \mathbb{C}_0(f) \mathbb{T}(X_2)\right) \tag{9}$$

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 (9) without performing the two whole matrix products, whose Matlab code is given in Table 2.

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

$$\mathcal{L}_n f(\boldsymbol{X}) = \left(\left(\mathbb{T}(X_1)\right)^t \mathbb{C}_0(f) \mathbb{T}(X_2)\right)^t \tag{10}$$

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, via the built-in function `meshgrid` (see [7]).

## 3 Fast computation of the cubature weights

In a recent paper [9], the interpolatory cubature formula corresponding to the Padua points has been studied. It has been termed "nontensorial Clenshaw–Curtis cubature" since it is a bivariate analogous of the classical Clenshaw–Curtis quadrature formula (cf. [5]), in the total-degree polynomial space. From the results of the previous section, we can write

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

$$= \sum_{j=0}^{n}\sum_{l=0}^{n} c'_{j,l}\, m_{j,l} = \sum_{j\,\mathrm{even}}^{n}\sum_{l\,\mathrm{even}}^{n} c'_{j,l}\, m_{j,l} \tag{11}$$

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

```
k = [0:2:n];
mom = 2*sqrt(2)./(1-k.^2);
mom(1) = 2;
[M1,M2] = meshgrid(mom);
M = M1.*M2;
COfM = COf(1:2:n+1,1:2:n+1).*M;
Int = sum(sum(COfM));
```

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

**Table 3** Matlab code for the evaluation of the cubature formula by moments.

where the *moments* $m_{j,l}$ are

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

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, the cubature formula (11) can be evaluated by the Matlab 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 the Padua points are the union of two subgrids of product Chebyshev points. First, observe that

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

with

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

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

Now, using the formula for the coefficients (4) we can write

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

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

where

$$\lambda_{\boldsymbol{\xi}} = w_{\boldsymbol{\xi}} \sum_{j\,\text{even}}^{n} \sum_{l\,\text{even}}^{n} m'_{j,l}\, \hat{T}_j(\xi_1)\hat{T}_l(\xi_2) \tag{12}$$

Defining the Chebyshev matrix corresponding to even degrees (cf. (5))

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

and the matrices of interpolation weights on the subgrids of Padua points, $\mathbb{W}_1 = \left(w_{\boldsymbol{\xi}},\, \boldsymbol{\xi} \in C_{n+1}^{\mathrm{E}} \times C_{n+2}^{\mathrm{O}}\right)^t$, $\mathbb{W}_2 = \left(w_{\boldsymbol{\xi}},\, \boldsymbol{\xi} \in C_{n+1}^{\mathrm{O}} \times C_{n+2}^{\mathrm{E}}\right)^t$ it is then easy to show that the cubature weights $\{\lambda_{\boldsymbol{\xi}}\}$ can be computed in the matrix form

$$\mathbb{L}_1 = \left(\lambda_{\boldsymbol{\xi}},\, \boldsymbol{\xi} \in C_{n+1}^{\mathrm{E}} \times C_{n+2}^{\mathrm{O}}\right)^t = \mathbb{W}_1.\left(\mathbb{T}^{\mathrm{E}}(C_{n+1}^{\mathrm{E}}))^t \mathbb{M}_0 \mathbb{T}^{\mathrm{E}}(C_{n+2}^{\mathrm{O}})\right)^t$$

$$\mathbb{L}_2 = \left(\lambda_{\boldsymbol{\xi}},\, \boldsymbol{\xi} \in C_{n+1}^{\mathrm{O}} \times C_{n+2}^{\mathrm{E}}\right)^t = \mathbb{W}_2.\left(\mathbb{T}^{\mathrm{E}}(C_{n+1}^{\mathrm{O}}))^t \mathbb{M}_0 \mathbb{T}^{\mathrm{E}}(C_{n+2}^{\mathrm{E}})\right)^t$$

where the dot means that the final product is made componentwise. The corresponding Matlab 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 meshgrid-like structure of the matrices $\mathbb{W}_i$ (see also (10)).

An alternative approach for the computation of the cubature weights is based on the observation that (12) itself is a bivariate 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. [11]). The algorithm is quite similar to that in Table 1, and we do not describe it for brevity.

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 [9].

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

```
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);
mom = 2*sqrt(2)./(1-k.^2);
mom(1) = 2;
[M1,M2] = meshgrid(mom);
M = M1.*M2;
M0 = fliplr(triu(fliplr(M)));
if (mod(n,2) == 0)
  M0(n/2+1,1) = M0(n/2+1,1)/2;
end
L1 = W1.*(TE1'*M0*TO2)';
L2 = W2.*(TO1'*M0*TE2)';
```

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

**Table 4** Matlab code for the computation of the cubature weights.

## 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.

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

| $n$ | 20 | 40 | 60 | 80 | 100 | 200 | 300 | 400 | 500 |
|-----|------|------|------|------|------|------|------|------|------|
| MM | 0.003 | 0.001 | 0.003 | 0.004 | 0.006 | 0.022 | 0.065 | 0.142 | 0.206 |
| FFT | 0.002 | 0.002 | 0.002 | 0.002 | 0.006 | 0.029 | 0.055 | 0.088 | 0.137 |

**Table 5** CPU time (in seconds) for the computation of the interpolation coefficients at a sequence of degrees.

| $n$ | 20 | 40 | 60 | 80 | 100 | 200 | 300 | 400 | 500 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| MM | 0.004 | 0.000 | 0.001 | 0.002 | 0.003 | 0.010 | 0.025 | 0.043 | 0.071 |
| FFT | 0.005 | 0.001 | 0.003 | 0.003 | 0.005 | 0.025 | 0.048 | 0.090 | 0.142 |

**Table 6** CPU time (in seconds) for the computation of the cubature weights at a sequence of degrees.

In Figure 2 we report the relative errors of interpolation (top) and cubature (bottom) for the classical Franke test function in $[0,1]^2$ (versus the degree). 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 approach for interpolation appears: it is able to arrive close to machine precision, whereas the MM algorithm stagnates around $10^{-13}$. On the contrary, the MM algorithm seems more stable in the cubature than in the 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 and a $C^2$ function (with third derivatives singular at the origin). 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. [8]).

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 by many other examples (cf. [9]) and present also in 3d with nontensorial cubature at new sets of Chebyshev hyperinterpolation points (cf. [6]), is quite similar univariate to that studied in the one-dimensional case for the classical Clenshaw–Curtis formula (cf. [10]), 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.

## 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 coefficients matrix (8) using the FFT approach, and `pdval` that evaluates the interpolant on a set of target points. For completeness we also provide the function `pdcfsMM`, which computes

the coefficients matrix using the Matrix Multiplication (MM) approach described in Section 2.

- `pdcub`: calls, as for `pdint`, the function `pdpts`, and `pdwtsMM` which computes the cubature weights using the MM approach. For completeness we included also the function `pdwtsFFT` which uses the FFT approach 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, `funct`, `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 all package has been successfully tested on Matlab 6.1.x, 6.5.x, 7.6.x and Octave 2.1.x, 3.0.x, 3.1.x.

## 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. Caliari, M., De Marchi, S., Vianello, M.: Bivariate polynomial interpolation on the square at new nodal sets. Appl. Math. Comput. 165, 261–274 (2005).
4. Caliari, M., De Marchi, S., Vianello, M.: Algorithm 886: Padua2D: Lagrange Interpolation at Padua Points on Bivariate Domains. ACM Trans. Math. Software 35-3 (2008).
5. Clenshaw, C.W., Curtis, A.R.: A method for numerical integration on an automatic computer. Numer. Math. 2, 197–205 (1960).
6. De Marchi, S., Vianello, M., Xu, Y.: New cubature formulae and hyperinterpolation in three variables. BIT Numerical Mathematics 49(1), 55–73 (2009)
7. The MathWorks: MATLAB documentation set. http://www.mathworks.com (2009).
8. 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).
9. Sommariva, A., Vianello, M., Zanovello, R.: Nontensorial Clenshaw–Curtis cubature. Numer. Algorithms 49, 409–427 (2008).
10. Trefethen, L.N.: Is Gauss quadrature better than Clenshaw–Curtis?. SIAM Rev. 50, 67–87 (2008).
11. Waldvogel, J.: Fast construction of the Fejér and Clenshaw–Curtis quadrature rules. BIT Numerical Mathematics 46, 195–202 (2006).
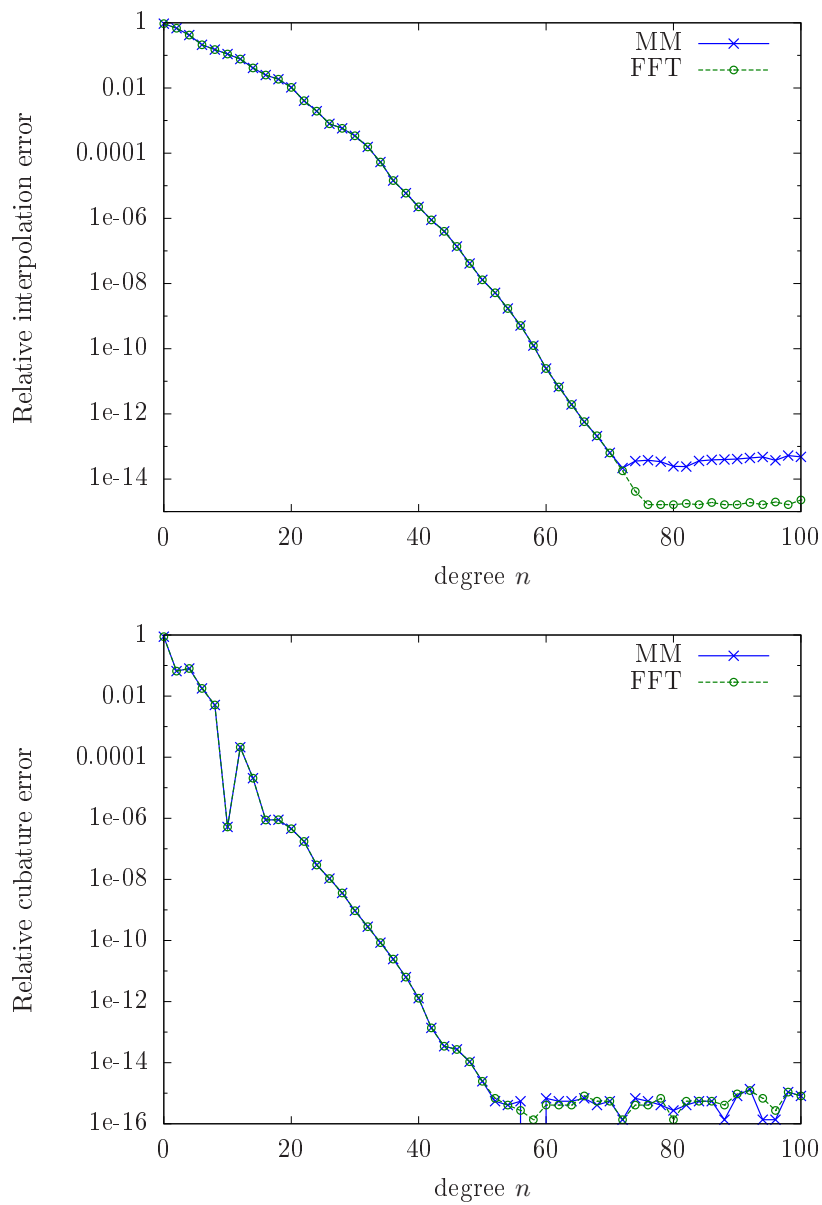
**Fig. 2** Relative errors of interpolation (top) and cubature (bottom) versus the interpolation degree for the Franke test function in $[0, 1]^2$, by the Matrix Multiplication (MM) and the FFT-based algorithms.
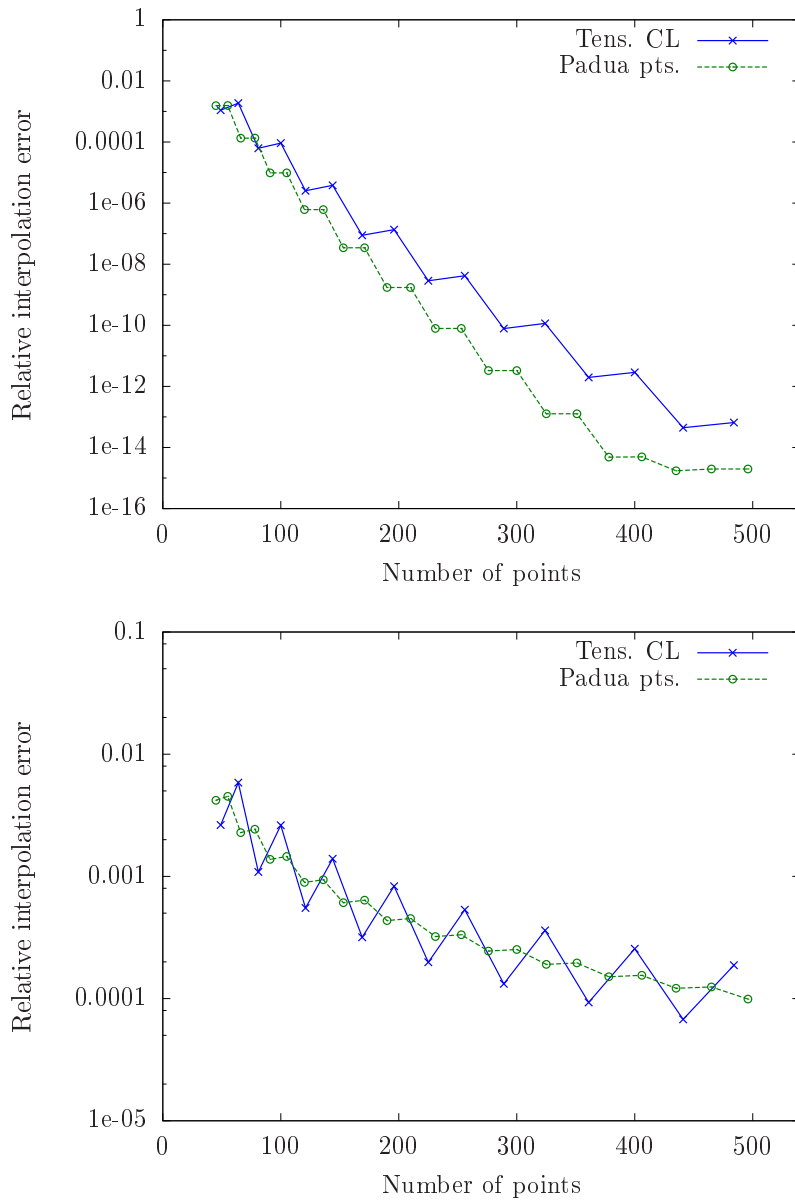
**Fig. 3** Relative interpolation errors versus the number of interpolation points for the Gaussian $f(\boldsymbol{x}) = \exp\left(-|\boldsymbol{x}|^2\right)$ (top) and the $C^2$ function $f(\boldsymbol{x}) = |\boldsymbol{x}|^3$ (bottom) in $[-1, 1]^2$; Tens. CL = Tensorial Chebyshev–Lobatto interpolation.
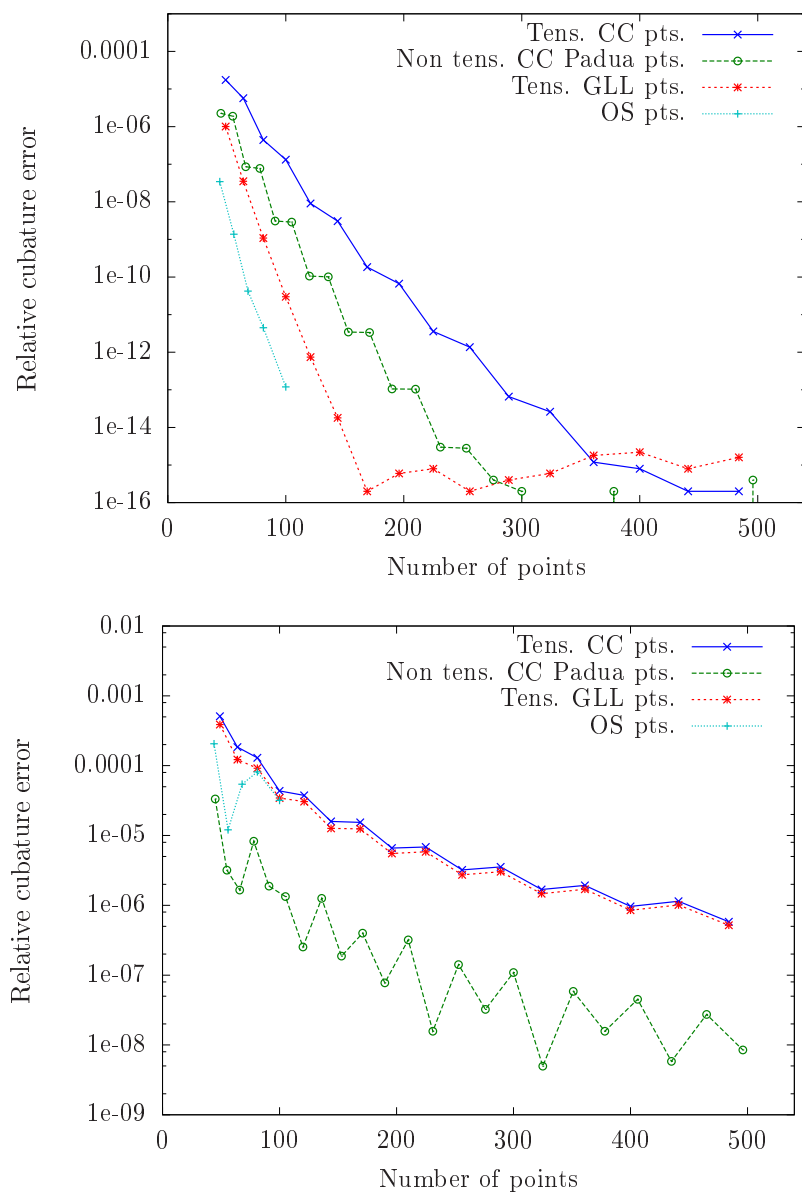
**Fig. 4** Relative cubature errors versus the number of cubature points (CC = Clenshaw–Curtis, GLL = Gauss–Legendre–Lobatto, OS = Omelyan–Solovyan) for the Gaussian $f(\boldsymbol{x}) = \exp(-|\boldsymbol{x}|^2)$ (top) and the $C^2$ function $f(\boldsymbol{x}) = |\boldsymbol{x}|^3$ (bottom); the integration domain is $[-1, 1]^2$, the integrals up to machine precision are, respectively: 2.230985141404135 and 2.508723139534059.