J Sci Comput
DOI 10.1007/s10915-011-9535-x

Efficient Implementation of Smoothness-Increasing Accuracy-Conserving (SIAC) Filters for Discontinuous Galerkin Solutions

Hanieh Mirzaee · Jennifer K. Ryan · Robert M. Kirby

Received: 12 May 2011 / Revised: 1 August 2011 / Accepted: 25 August 2011

Abstract The discontinuous Galerkin (DG) methods provide a high-order extension of the finite volume method in much the same way as high-order or spectral/hp elements extend standard finite elements. However, lack of inter-element continuity is often contrary to the smoothness assumptions upon which many post-processing algorithms such as those used in visualization are based. Smoothness-increasing accuracy-conserving (SIAC) filters were proposed as a means of ameliorating the challenges introduced by the lack of regularity at element interfaces by eliminating the discontinuity between elements in a way that is consistent with the DG methodology; in particular, high-order accuracy is preserved and in many cases increased. The goal of this paper is to explicitly define the steps to efficient computation of this filtering technique as applied to both structured triangular and quadrilateral meshes. Furthermore, as the SIAC filter is a good candidate for parallelization, we provide, for the first time, results that confirm anticipated performance scaling when parallelized on a shared-memory multi-processor machine.

Keywords High-order methods · Discontinuous Galerkin · SIAC filtering · Accuracy enhancement

H. Mirzaee · R.M. Kirby
School of Computing, University of Utah, Salt Lake City, Utah, USA

H. Mirzaee
e-mail: mirzaee@cs.utah.edu

R.M. Kirby
e-mail: kirby@cs.utah.edu

J.K. Ryan (✉)
Delft Institute of Applied Mathematics, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: J.K.Ryan@tudelft.nl

Published online: 22 September 2011

1 Introduction

The discontinuous Galerkin (DG) methods provide a high-order extension of the finite volume method in much the same way as high-order or spectral/hp elements [17, 27] extend standard finite elements. The DG methodology allows for a dual path to convergence through both elemental h and polynomial p refinement, making it highly desirable for computational problems which require resolution fidelity. In the overview of the development of the discontinuous Galerkin method, Cockburn et al. [4] trace the developments of DG and provide a succinct discussion of the merits of this extension to finite volumes.

The primary mathematical advantage of DG is that unlike classic continuous Galerkin FEM that seeks approximations which are piecewise continuous, the DG methodology only requires functions which are L_2 integrable. Much like FEM, DG uses the variational form; however, instead of constraining the solution to being continuous across element interfaces, the DG method merely requires weak constraints on the fluxes between elements. This feature provides a discretization flexibility that is difficult to match with conventional continuous Galerkin methods.

Lack of inter-element continuity, however, is often contrary to the smoothness assumptions upon which many post-processing algorithms such as those used in visualization are based. A class of post-processing techniques was introduced in [7, 22] as a means of gaining increased accuracy from DG solutions through the exploitation of the superior convergence rates of DG in the negative-order norm; these filters have as a secondary consequence that they increase the smoothness of the output solution. Building upon these concepts, in [25, 28] smoothness-increasing accuracy-conserving (SIAC) filters were proposed as a means of ameliorating the challenges introduced by the lack of regularity at element interfaces while at the same time maintaining accuracy constraints that are consistent with the verification process used in the original simulation. In essence, in the application domain, one seeks to increase smoothness *without destroying* (i.e. by maintaining) the order of accuracy of the original input DG solution.

The basic operation performed to gain the smoothness and accuracy benefits is convolution of the DG solution against a judiciously constructed B-spline based kernel. The goal of this paper is to explicitly define the steps to efficient computation of the post-processor applied to different mesh structures. In addition, we explain how well the inexact post-processor (see [18, 19]) performs computationally comparing to the exact scheme. Furthermore, as the SIAC filter is a good candidate for parallelization, we provide, for the first time, results that confirm anticipated performance scaling when parallelized on a shared-memory multi-processor machine. We further note that in the following sections the terms *filter* and *post-processor* are used interchangeably.

We begin by reviewing the basics of the discontinuous Galerkin method. In Sect. 3 we provide an overview of the SIAC filter. We continue this section by explaining how to construct the convolution kernel in Sect. 3.1. The implementation of the SIAC filter will be discussed in Sect. 3.2 in one-dimension. Moving on to higher dimensions, we provide implementation details for quadrilateral, triangular and hexahedral mesh structures in Sects. 3.2.1, 3.2.2 and 3.2.3 respectively. We continue by explaining how to modify the exact post-processor in order to achieve the inexact scheme in Sect. 4. In Sect. 5 we provide performance analysis as well as the parallel implementation of the post-processor. Finally, Sect. 6 concludes the paper.

2 The Discontinuous Galerkin Method

In this paper, we focus our attention on simulation results that arise as solutions of the linear hyperbolic equation

$$\begin{aligned} \mathbf{u}_t + \nabla \cdot (a(\mathbf{x}, t)\mathbf{u}) = 0, \quad \mathbf{x} \in \Omega \times [0, T], \\ u(\mathbf{x}, 0) = u_o(\mathbf{x}), \quad \mathbf{x} \in \Omega, \end{aligned} \tag{1}$$

where $\mathbf{x} \in \Omega$ and $t \in \mathbb{R}$. We also assume smooth initial conditions are given along with periodic boundary conditions. The DG formulation for this equation has been well-studied in the series of papers [2, 3, 5, 9–12]. Here we present a brief introduction.

We use the weak form of (1) to derive our discontinuous Galerkin approximation. That is, we multiply by a test function $\mathbf{v}$ to obtain

$$\frac{d}{dt}\int_\Omega \mathbf{u}(\mathbf{x}, t)\mathbf{v}\, d\mathbf{x} + \int_\Gamma (a(\mathbf{x}, t)\mathbf{u}) \cdot \hat{n}\mathbf{v}\, d\Gamma - \int_\Omega (a(\mathbf{x}, t)\mathbf{u}) \cdot \nabla \mathbf{v}\, d\mathbf{x} = 0, \tag{2}$$

where $\hat{n}$ denotes the unit outward normal to the boundary and Γ the boundary of our domain, Ω.

We can now define our discontinuous Galerkin approximation to (1) using (2). We begin by defining a suitable tessellation of the domain Ω, $\mathcal{T}(\Omega) = \tilde{\Omega}$. We note that the current form of the post-processor requires using a rectangular domain in two-dimensions. However, it was also computationally extended to structured triangulations in [18]. Secondly, we define an approximation space, $\mathbf{V}_h$, consisting of piecewise polynomials of degree less than or equal to k on each element of our mesh. Our discontinuous Galerkin approximation will then be of order $k + 1$. Using the variational formulation and taking our test function $\mathbf{v}_h$ from our approximation space we obtain

$$\frac{d}{dt}\int_{\tilde{\Omega}} \mathbf{u}(\mathbf{x}, t)\mathbf{v}_h\, d\mathbf{x} + \sum_{e\in\partial\tilde{\Omega}} \int_e (a(\mathbf{x}, t)\mathbf{u}) \cdot \hat{n}_{e,\tilde{\Omega}}\mathbf{v}_h\, d\Gamma - \int_{\tilde{\Omega}} (a(\mathbf{x}, t)\mathbf{u}) \cdot \nabla \mathbf{v}_h\, d\mathbf{x} = 0, \tag{3}$$

where $\hat{n}_{e,\tilde{\Omega}}$ denotes the outward unit normal to edge e. We then obtain the numerical scheme

$$\frac{d}{dt}\int_{\tilde{\Omega}} \mathbf{u}_h(\mathbf{x}, t)\mathbf{v}_h\, d\mathbf{x} + \sum_{e\in\partial\tilde{\Omega}} \int_e \mathbf{h}\left(\mathbf{u}_h(\mathbf{x}^-, t), \mathbf{u}_h(\mathbf{x}^+, t)\right)\mathbf{v}_h\, d\Gamma - \int_{\tilde{\Omega}} (a(\mathbf{x}, t)\mathbf{u}_h) \cdot \nabla \mathbf{v}_h\, d\mathbf{x} = 0 \tag{4}$$

for all test functions $\mathbf{v}_h \in \mathbf{V}_h$, where $\mathbf{h}(\mathbf{u}(\mathbf{x}^-, t), \mathbf{u}(\mathbf{x}^+, t))$ is a consistent two-point monotone Lipschitz flux as in [2] and $\mathbf{u}_h$ is the DG approximations of degree k.

We note that this scheme and its implementation have been well studied. Therefore we concentrate on the details of efficient computational implementation of the post-processor itself.

3 Smoothness-Increasing Accuracy-Conserving Filter

Smoothness-Increasing Accuracy-Conserving (SIAC) filters were first introduced as a class of post-processors for the discontinuous Galerkin method in [6, 7]. The filtering technique was extended to a broader set of applications such as being used for filtering within streamline visualization algorithms in [8, 13, 21, 22, 25, 28]. Given a sufficiently smooth exact

solution, the rate of convergence of a DG approximation of degree k is $k+1$ (see [2, 20]). However, after convolving the approximation against a filter constructed from a linear combination of B-splines of order $k+1$, we can improve the order of convergence to $2k+1$. Additionally, we filter out the oscillations in the error and restore the $\mathcal{C}^{k-1}$-continuity at the element interfaces. Here we present a brief introduction to this post-processing technique. For a more detailed discussion of the mathematics of the post-processor see [1, 7, 13, 22].

The post-processor itself is simply the discontinuous Galerkin solution at the final time T, convolved against a linear combination of B-splines. That is, in one-dimension,

$$u^\star(x) = \frac{1}{h}\int_{-\infty}^{\infty} K^{r+1,k+1}\left(\frac{y-x}{h}\right)u_h(y)dy, \tag{5}$$

where $u^\star$ is the post-processed solution, h is the characteristic length and

$$K^{r+1,k+1}(x) = \sum_{\gamma=0}^{r} c_\gamma^{r+1,k+1}\psi^{(k+1)}(x-x_\gamma), \tag{6}$$

is the convolution kernel. $\psi^{(k+1)}$ is the B-spline of order $k+1$ and $c_\gamma^{r+1,k+1}$ represent the kernel coefficients. x_γ represent the positions of the kernel nodes and are defined later in this section along with $c_\gamma^{r+1,k+1}$. The superscript $r+1, k+1$ typically represent the number of kernel nodes as well as the B-spline order. In the following discussions we shall drop this superscript for the sake of a less cluttered explanation.

We wish to point out that a more local kernel is used in the interior of the domain, which has a *symmetric* form. For the symmetric kernel we use $r+1 = 2k+1$ B-splines. However, near boundaries and shocks the symmetric kernel can not be used if there are non-periodic boundary conditions as the symmetric kernel requires an equal amount of information from both sides of the point that is being post-processed. In this case, we apply a *one-sided* (or *position-dependent*) form of the kernel, proposed by Slingerland et al. [24], which requires $r+1 = 4k+1$ B-splines. Following [24], we combine the kernels in the interior and at the boundaries through a convex combination:

$$u_h^\star(x) = \theta(x)(u_h^\star(x))_{r=2k} + (1-\theta(x))(u_h^\star(x))_{r=4k}. \tag{7}$$

In this formulation, $\theta \in \mathcal{C}^{k-1}$ and is equal to one in the interior of the domain where the symmetric kernel is used.

In this paper, we concentrate on the implementation issues and strategies that are useful for a numerical practitioner to apply the post-processor effectively. Therefore, in the sections that follow, we investigate how the post-processed solution given in (7) can efficiently be implemented in multi-dimensions.

3.1 Construction of the Kernel

We remind the reader that the convolution kernel used in the SIAC filter is formulated as

$$K(x) = \sum_{\gamma=0}^{r} c_\gamma \psi^{(k+1)}(x-x_\gamma), \tag{8}$$

where

$$x_\gamma = -\frac{r}{2} + \gamma + \lambda(x), \quad \gamma = 0, \ldots, r \tag{9}$$

represent the positions of the kernel nodes. As in [24], $\lambda(x)$ is defined as a shift function that depends upon the evaluation point, x, and is given by

$$\lambda(x) = \begin{cases} \min\{0, -\frac{r+k+1}{2} + \frac{x-x_L}{h}\}, & x \in [x_L, \frac{x_L+x_R}{2}), \\ \max\{0, \frac{r+k+1}{2} + \frac{x-x_R}{h}\}, & x \in [\frac{x_L+x_R}{2}, x_R], \end{cases} \tag{10}$$

where x_L and x_R denote the left and right boundaries of the domain. We note that for $\lambda(x) = 0$ and $r = 2k$, we produce the symmetric kernel used in the interior of the domain. In order to construct the kernel we define the B-splines, $\psi^{(k+1)}$, and then demonstrate both how to implement this in the construction of the kernel, which includes calculating the coefficients used in the kernel.

The convolution kernel used in the SIAC filter is a linear combination of B-splines. The B-spline is of order $k+1$, and for efficient computation is obtained using the recursion relations

$$\begin{aligned} \psi^{(1)}(x) &= \chi_{[-1/2,1/2]}, \\ \psi^{(k+1)}(x) &= \frac{1}{k}\left(\left(\frac{k+1}{2} + x\right)\psi^{(k)}\left(x + \frac{1}{2}\right) + \left(\frac{k+1}{2} - x\right)\psi^{(k)}\left(x - \frac{1}{2}\right)\right), \end{aligned} \tag{11}$$

where $\chi_{[-1/2,1/2]}$ is the characteristic function defined over $[-1/2, 1/2]$.

A B-spline of order $k+1$ is a piecewise polynomial of degree k over each individual interval separated by the B-spline knots. Using (11), one can also calculate the polynomial coefficients as fractions, *a priori*, store them in a matrix and then use some polynomial evaluation scheme such as Horner's method to evaluate the B-spline at some arbitrary point. As an example, for $k = 2$ the B-spline has the form

$$\psi^{(3)}(x) = \begin{cases} \frac{1}{2}x^2 + \frac{3}{2}x + \frac{9}{8}, & x \in [-\frac{3}{2}, -\frac{1}{2}), \\ -x^2 + \frac{3}{4}, & x \in [-\frac{1}{2}, \frac{1}{2}), \\ \frac{1}{2}x^2 - \frac{3}{2}x + \frac{9}{8}, & x \in [\frac{1}{2}, \frac{3}{2}], \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

We should also note that from the aforementioned definitions, it is obvious that the B-splines have compact support, meaning that a B-spline $\psi^{(k+1)}(x)$ of degree k with knots $x_0 < \cdots < x_{k+1}$ is zero outside of $[x_0, x_{k+1}]$, where $x_0 = -\frac{(k+1)}{2}$ and $x_{k+1} = \frac{(k+1)}{2}$. This leads to a more efficient scheme for B-spline evaluations since the points outside this interval simply result in a zero value. Algorithm 1 depicts the pseudo-code for evaluating the B-spline at a particular point x. We further note that B-splines have been well-studied, and we refer the interested reader to [14, 16, 23] for a more thorough discussion.

Algorithm 1 Evaluating $\psi^{(k+1)}(x)$

```
if x in [x_0, x_{k+1}] then
    Find the interval of x, call it [x_j, x_{j+1}]
    ψ^{(k+1)}(x) = value of the corresponding polynomial over [x_j, x_{j+1}] (as in (11)) at x
else
    ψ^{(k+1)}(x) = 0
end if
```

Now that the B-splines are defined, they can be used to construct the convolution kernel in the SIAC filters. However, the kernel coefficients c_γ remain to be defined.

One of the important properties of the kernel as mentioned in [22], is that it reproduces polynomials up to a certain degree, r, which equals $2k$ for the symmetric kernel and $4k$ for the one-sided kernel. This means that the convolution of the kernel with a polynomial of degree less than or equal to r is equal to that polynomial itself. In addition, it guarantees that the accuracy of the DG approximation is not destroyed by the convolution. Using the monomials we obtain the following linear system for the kernel coefficients:

$$\sum_{\gamma=0}^{r} c_\gamma \int_{\mathbb{R}} \psi^{(k+1)}(y)(y-x-x_\gamma)^m dy = x^m, \quad m = 0, 1, \ldots, r. \tag{13}$$

To calculate the integral in (13), we use Gaussian quadrature with $\frac{k+m}{2}+1$ quadrature points [17]. As an example for $k = 1$ and the symmetric kernel, (13) gives

$$\begin{bmatrix} 1 & 1 & 1 \\ x+1 & x & x-1 \\ x^2+2x+\frac{7}{6} & x^2+\frac{1}{6} & x^2-2x+\frac{7}{6} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix}. \tag{14}$$

Equation (14) must hold for all x, we simply set $x = 0$ and obtain the coefficients

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{12} \\ \frac{7}{6} \\ -\frac{1}{12} \end{bmatrix}. \tag{15}$$

The linear system in (13) is a non-singular system. Hence, the existence and uniqueness of the kernel coefficients is guaranteed (see [7] for a proof). Linear algebra routines provided by the LAPACK library can be used for solving the system (visit www.netlib.org/lapack/). Algorithm 2 provides the pseudo-code for constructing the matrix of the linear system mentioned in (13). Note that for calculating the integral in (13) (Line 12 in Algorithm 2), the integration region $\mathbb{R}$ actually reduces to $[-(k+1)/2, (k+1)/2]$, and that we need to divide this region into subintervals that respect the continuity breaks in the B-splines. This is required for the integral to be evaluated exactly to machine precision using Gaussian quadrature.

As we mentioned earlier in this section, for the one-sided kernel the position of the kernel nodes depend on the evaluation point x through a continuous shift function $\lambda(x)$ (see (10)). This means that the kernel coefficients change depending on the current evaluation point. Therefore, contrary to the symmetric kernel case, the kernel coefficients for the one-sided kernel need to be re-calculated for each evaluation point.

Having defined the kernel, we continue by demonstrating how to implement the post-processor operator by evaluating the integral in (5).

3.2 Evaluation of the Convolution Operator

Traditionally, SIAC filters are implemented as small matrix-vector multiplications [22]. That is, considering a fixed number of evaluation points per element, a number of coefficient matrices are produced. These are computed one time and stored for future use. The post-processing is then implemented in a simple manner via these small matrix-vector multiplications of the pre-stored coefficient matrices and the coefficients of the numerical solution.

Algorithm 2 Constructing the B-spline coefficient matrix

$\lambda = \lambda(x)$ (Given by (10))
$rowSize = r + 1$
$colSize = r + 1$
$LinMatrix[rowSize][colSize]$
$bsplineKnots = [-\frac{k+1}{2}, -\frac{k+1}{2} + 1, \ldots, \frac{k+1}{2}]$
for $row = 0$ to $rowSize$ **do**
  for $col = 0$ to $colSize$ **do**
    $LinMatrix[row][col] = 0$
    $\gamma = col$
    $x_\gamma = -\frac{r}{2} + \gamma + \lambda$
    {Evaluate the integral in (13)}
    for $i = 0$ to $size(bsplineKnots) - 1$ **do**
      $\Omega = [bsplineKnots[i], bsplineKnots[i + 1]]$
      $x =$ map the Gaussian quadrature points obtained over $[-1, 1]$ to Ω
      $LinMatrix[row][col] += \int_\Omega \psi^{(k+1)}(x)(x + x_\gamma)^{row} dx$ using Gaussian quadrature
    end for
  end for
end for

However, as this approach is not suitable for the more general case of unstructured meshes or for the one-sided kernel, we discuss in this section how the integral in Equation (5) can be evaluated directly.

We begin by introducing the notion of a *standard region* (sometimes referred to as the reference element). In order to evaluate a DG approximation at an arbitrary point or to compute an integral using Gaussian quadrature, we often first need to map the points to a standard region (see [17]). In this section we introduce the 1D standard element, Ω_{st} such that

$$\Omega_{st} = \{\xi | -1 \leq \xi \leq 1\}. \tag{16}$$

Therefore, to evaluate our DG approximation $u_h(x)$ in (5) at a point x defined on the interval $I_i = [x_a, x_b]$, which we refer to as the *local region*, we have

$$u_h(x) = \sum_{l=0}^{k} u_i^{(\ell)} \phi^\ell(\mu^{-1}(x)), \tag{17}$$

where $u_i^{(\ell)}$ are the local polynomial modes on I_i resulting from a discontinuous Galerkin approximation, ϕ^ℓ are the polynomial basis functions of degree ℓ defined over the standard region and $\mu(\xi)$ is the affine mapping from the standard to local region given by

$$\mu(\xi) = x_a \frac{1 - \xi}{2} + x_b \frac{1 + \xi}{2}. \tag{18}$$

To evaluate the post-processed solution at an isolated point $x \in I_i$ by directly evaluating the integral in the convolution operator, we have,

$$u^\star(x) = \frac{1}{h} \int_{-\infty}^{\infty} K\left(\frac{y - x}{h}\right) u_h(y) dy = \frac{1}{h} \sum_{I_{i+j} \in Supp\{K\}} \int_{I_{i+j}} K\left(\frac{y - x}{h}\right) u_h(y) dy, \tag{19}$$

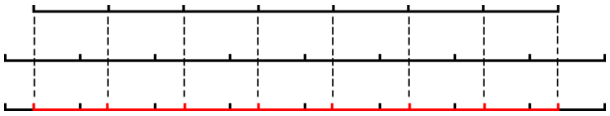

Fig. 1 A possible kernel-mesh overlap in one dimension. *Upper line* represents the kernel, middle line depicts a DG mesh and the *lower line* (*dashed red*) represents the integration mesh

where the second equation is due to the compact support property of the kernel. In order to evaluate the integral in (19) exactly, we need to divide the interval I_{i+j} to subintervals over which there is no break in regularity in the integrand. We then use Gaussian quadrature with sufficient quadrature points to evaluate the integration. Figure 1 shows how the integration regions are constructed from the intersection of the kernel knots and the DG element interfaces. As the figure demonstrates, in the final integration mesh (red line) each DG element is divided into two subintervals so that there is no break in continuity. The integration will then be carried out over these subintervals.

Formulating this we have

$$\begin{aligned} u^\star(x) &= \frac{1}{h}\int_{-\infty}^{\infty} K\left(\frac{y-x}{h}\right) u_h(y)dy, \\ &= \frac{1}{h}\sum_{I_{i+j}} \int_{I_{i+j}} K\left(\frac{y-x}{h}\right) u_h(y)dy, \\ &= \frac{1}{h}\sum_{I_{i+j}} \Bigg[\int_{-1}^{1} K\left(\frac{\mu_{s_1}(\xi)-x}{h}\right) u_h(\mu_{s_1}(\xi))|J_1|d\xi \\ &\quad + \int_{-1}^{1} K\left(\frac{\mu_{s_2}(\xi)-x}{h}\right) u_h(\mu_{s_2}(\xi))|J_2|d\xi\Bigg], \end{aligned} \tag{20}$$

where s_1 and s_2 are the two aforementioned subintervals within each DG element, i.e., $s_1 \cup s_2 = I_{i+j}$. In addition, $\mu_{s_1}(\xi)$ and $\mu_{s_2}(\xi)$ represent the mappings from the standard to local regions s_1 and s_2 and $|J_1|$ and $|J_2|$ are the Jacobians of these mappings. Moreover, the number of quadrature points should be enough to integrate polynomials of degree at least $2k$. In addition, similar to (17), we evaluate the DG approximation in (20) as

$$u_h(x) = \sum_{l=0}^{k} u_{I_{i+j}}^{(\ell)} \phi^{\ell}(\mu_{I_{i+j}}^{-1}(x)), \tag{21}$$

where in this case $x = \mu_{s_{1,2}}(\xi)$ and belongs to element I_{i+j}.

We note that the support of the kernel is given by

$$\left[K_a = x + h\left(-\frac{(r+\ell)}{2} - \lambda(x)\right), K_b = x + h\left(\frac{(r+\ell)}{2} - \lambda(x)\right)\right], \tag{22}$$

where x is the evaluation point, $\ell = k+1$, and $\lambda(x)$ is the shift function in (10). Consequently, the position of the *kernel breaks* are given by,

$$K_a + h, K_a + 2h, \ldots, K_b. \tag{23}$$

It is clear from (22), that the one-sided kernel ($r = 4k$) has a larger support than the symmetric one ($r = 2k$). Moreover, in Sect. 3, we mentioned that we combine the symmetric and one-sided kernels through a coefficient function θ. We do so in order to have a smooth transition between these two kernels. We note that if one is only concerned about accuracy and not smoothness, this step can be avoided. One choice of the coefficient function θ is

$$\theta(x) = \begin{cases} 0, & x \in [x_L, a_1), \\ p(x), & x \in [a_1, a_2], \\ 1, & x \in (a_2, b_2), \\ q(x), & x \in [b_2, b_1], \\ 0, & x \in (b_1, x_R], \end{cases} \tag{24}$$

where,

$$a_1 = x_L + \frac{3k+1}{2}h, \qquad a_2 = x_L + \left(\frac{3k+1}{2} + 2\right)h,$$
$$b_1 = x_R - \frac{3k+1}{2}h, \qquad b_2 = x_R - \left(\frac{3k+1}{2} + 2\right)h,$$

and where p and q are polynomials of degree $2k+1$. We further require that $p(a_1) = 0$, $p(a_2) = 1$, and $\frac{d^n p}{dx^n}(a_1) = \frac{d^n p}{dx^n}(a_1) = 0$ for all $n = 1, \ldots, k$. A similar definition holds for q. We remark that other choices for θ may also work in practice.

From (24), we understand that we only need to calculate the convex combination when θ assumes a value other than 0 or 1, namely, when $x \in [a_1, a_2]$ or $x \in [b_2, b_1]$. In these *transition regions*, a smooth transition happens between the one-sided kernel and the symmetric kernel over two mesh elements.

Algorithm 3 provides a pseudo-code for implementing the convolution operator in a one-dimensional field. We again emphasize, depending on the evaluation point, we either calculate the post-processed solution using the one-sided kernel, the symmetric kernel or both when x is in a transition region. In line 8, it is stated that the subintervals are the result of the kernel and DG mesh intersection, which is a geometric problem. This is not addressed in this

Algorithm 3 1D-Convolution

1: **for** each evaluation point x **do**
2: $I_i =$ the element to which x belongs
3: $h =$ the size of the element I_i
4: **if** one-sided **then**
5: Calculate the one-sided kernel coefficients
6: **end if**
7: {Find the integration subintervals}
8: $S =$ kernel and mesh intersection
9: {The following for loop implements the third line in (20)}
10: **for** each s in S **do**
11: $intg+ =$ Evaluate $\int_{-1}^{1} K\left(\frac{\mu_s(\xi)-x}{h}\right) u_h(\mu_s(\xi))|J|d\xi$
12: **end for**
13: $u^\star(x) = intg/h$
14: **end for**

paper for complex mesh structures. In one dimension, this problem is fairly straightforward as it is the result of the sorted merge of the kernel breaks and the mesh element interfaces that are (partially) covered by the support of the kernel (note that kernel breaks and element interfaces are already sorted lists). For a uniform mesh, the footprint of the kernel can be found in constant computational time. For a non-uniform mesh structure, the footprint can be found in $O(\log(N))$, N being the number of elements in one direction (or total number of elements in 1D). Moreover, for uniform meshes, h represents the uniform mesh spacing. For non-uniform meshes, we simply consider h as being the size of the element (length of element in 1D and length of element sides in 2D). For a more thorough discussion on post-processing over non-uniform mesh structures we refer the reader to [13].

We further note that the post-processed polynomial is of degree at least $2k+1$. Therefore, if we want to post-process a DG approximation of degree k over the entire field so that a transformation to a modal representation is feasible, we need to evaluate the post-processor at $2k+2$ collocating points per element. Moreover, in our initial DG approximation space, we have $N \times (k+1)$ degrees of freedom (N being the number of elements in the field), whereas in the post-processed solution space, there are $N \times (2k+2) - N \times (k-1)$ degrees of freedom. The first term is due to higher order polynomials and the second term is due to what is removed (constrained) due to continuity.

In the following sections we explain how this implementation of the post-processor extends to two and three dimensions, considering the common DG element types, triangles and quadrilaterals in 2D and hexahedra in 3D.

3.2.1 Quadrilateral Mesh Implementation

In two dimensions the convolution kernel is a tensor product of the one dimensional kernels

$$\begin{aligned}\bar{K}(x,y) &= \sum_{\gamma_1=0}^{r_1}\sum_{\gamma_2=0}^{r_2} c_{\gamma_1} c_{\gamma_2} \psi^{(k+1)}(x - x_{\gamma_1})\psi^{(k+1)}(y - x_{\gamma_2}) \\ &= K(x) \times K(y),\end{aligned} \tag{25}$$

where x_{γ_1} and x_{γ_2} are the position of the kernel nodes in the x_1- and x_2-directions and the two-dimensional coordinate system is denoted with (x_1, x_2). Furthermore, $r_1,\ r_2$ are either $2k$ or $4k$, depending on whether we use a symmetric or one-sided kernel in each direction.

The two-dimensional convolution over quadrilateral mesh structures is therefore,

$$\begin{aligned}u^\star(x,y) &= \frac{1}{h_1 h_2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} K\left(\frac{x_1 - x}{h_1}\right) K\left(\frac{x_2 - y}{h_2}\right) u_{h_1,h_2}(x_1,x_2)dx_1dx_2, \\ &= \frac{1}{h_1 h_2}\sum_{I_{i+d_1,j+d_2} \in Supp\{\tilde{K}\}} \iint_{I_{i+d_1,j+d_2}} K\left(\frac{x_1 - x}{h_1}\right) \\ &\quad \times K\left(\frac{x_2 - y}{h_2}\right) u_{h_1,h_2}(x_1,x_2)dx_1dx_2\end{aligned} \tag{26}$$

which evaluates the post-processed solution at $(x, y) \in I_{i,j}$. Notice that the scaling of the kernel does not require $h_1 = h_2$.

Once again to evaluate the integral in (26) exactly to machine precision, we need to divide the integration region $I_{i+d_1,j+d_2}$ which has a quadrilateral shape to subregions over which there is no break in regularity.

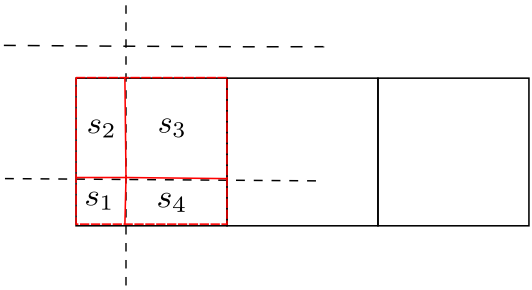

Fig. 2 A possible kernel-mesh overlap in two dimensions. *Dashed lines* represent the kernel patch and *solid line* represent the DG mesh. *Red lines* depict possible integration regions over one element

Figure 2 demonstrates a possible kernel-mesh intersection over a quadrilateral mesh. As shown in the figure, the post-processing kernel can be viewed as a two-dimensional patch which is the immediate result of the tensor product of the one-dimensional kernels on each direction.

For the example in Fig. 2, we can see that it is necessary to break down the DG element to four subelements in order to evaluate the integration in (26) exactly, i.e.,

$$\begin{aligned}
&\int\int_{I_{i+d_1,j+d_2}} K\left(\frac{x_1-x}{h_1}\right)K\left(\frac{x_2-y}{h_2}\right)u_{h_1,h_2}(x_1,x_2)dx_1dx_2\\
&=\sum_{n=0}^{3}\int\int_{s_n} K\left(\frac{x_1-x}{h_1}\right)K\left(\frac{x_2-y}{h_2}\right)u_{h_1,h_2}(x_1,x_2)dx_1dx_2\\
&=\sum_{n=0}^{3}\int_{-1}^{1} K\left(\frac{\mu_{s_{n_2}}(\xi_2)-y}{h_2}\right)|J_2|\\
&\quad\times\left(\int_{-1}^{1} K\left(\frac{\mu_{s_{n_1}}(\xi_1)-x}{h_1}\right)u_{h_1,h_2}(\mu_{s_{n_1}}(\xi_1),\mu_{s_{n_2}}(\xi_2))|J_1|d\xi_1\right)d\xi_2, \qquad (27)
\end{aligned}$$

where s_n indicates the integration region resulting from the kernel-mesh intersection. Furthermore, the third equation is obtained using the tensor product property of the two-dimensional kernel. This allows us to write the two-dimensional integration as a product of one-dimensional integrations. $\mu_{s_{n_1}}(\xi_1)$ and $\mu_{s_{n_2}}(\xi_2)$ are used to denote the one-dimensional mappings from the standard element to the local regions, in this case are the sides of the subelement s_n in the x_1 and x_2 directions. $|J_1|$ and $|J_2|$ are the Jacobians of these transformations. In addition, the number of quadrature points required for integration should be chosen to exactly integrate polynomials of degree $2k$ in each direction.

Another point that we would like to mention here, is the evaluation of our DG approximation, $u_{h_1,h_2}(x,y)$. To evaluate the DG approximation at an arbitrary point$(x,y)\in I_{i,j}$, we have

$$u_{h_1,h_2}(x,y)=\sum_{p=0}^{k}\sum_{q=0}^{k}u_{I_{i,j}}^{(pq)}\phi^{(pq)}(\xi_1,\xi_2), \qquad (28)$$

where ξ_1 and ξ_2 are obtained using the appropriate one-dimensional inverse mappings from local to standard regions (see Sect. 3.2).

If we define our two-dimensional basis function $\phi^{(pq)}(\xi_1, \xi_2)$ as the tensor product of one-dimensional basis functions (see [17])

$$\phi^{pq}(\xi_1, \xi_2) = \psi_p^a(\xi_1)\psi_q^b(\xi_2), \tag{29}$$

with ψ_p^a and ψ_q^b being the modified or orthogonal basis functions as in [17], then we can evaluate the DG approximation in (28) as

$$u_{h_1,h_2}(x, y) = \sum_{p=0}^{k} \psi_p^a(\xi_1) \sum_{q=0}^{k} u_{I_{i,j}}^{(pq)} \psi_q^b(\xi_2). \tag{30}$$

This is the so-called *sum-factorization* technique introduced in [17]. Using this approach the number of operations needed to evaluate the DG approximation at $O(k^2)$ quadrature points formed by the tensor product of one-dimensional points, reduces from $O(k^4)$ to $O(k^3)$. This aids the numerical practitioner in saving on the overall computational cost of the post-processor. Algorithm 4 provides a pseudo-code for the two-dimensional convolution over quadrilateral mesh structures. Lines 12 and 13 are implemented the same way as the one-dimensional case explained in the previous section. Consequently, S_1 and S_2 are integration sets that represent the one-dimensional integration regions on each direction. This means that the tensor product of these two sets produces the two-dimensional integration regions as shown in Fig. 2. In addition, in lines 3 and 4, if we are dealing with a uniform mesh, $h_1 = h_2$ and is equal to the uniform mesh spacing on each direction. Otherwise as we mentioned in the previous section, we choose the length of the element size on each direction for h_1 and h_2, respectively (Fig. 2).

Algorithm 4 2D quadrilateral mesh convolution

1: **for** each evaluation point (x, y) **do**
2: $I_{i,j}$ = the element to which (x, y) belongs
3: h_1 = length of $I_{i,j}$ side in direction x_1
4: h_2 = length of $I_{i,j}$ side in direction x_2
5: **if** one-sided in x_1-direction **then**
6: Calculate the one-sided kernel coefficients for x_1-direction
7: **end if**
8: **if** one-sided in x_2-direction **then**
9: Calculate the one-sided kernel coefficients for x_2-direction
10: **end if**
11: {Find the 1D integration subintervals on each direction}
12: S_1 = kernel and mesh intersection in direction x_1
13: S_2 = kernel and mesh intersection in direction x_2
14: **for** s_1 in S_1 **do**
15: **for** s_2 in S_2 **do**
16: $intg+=$ Evaluate the outer integral in (27)
17: **end for**
18: **end for**
19: $u^\star(x, y) = intg/(h_1 h_2)$
20: **end for**

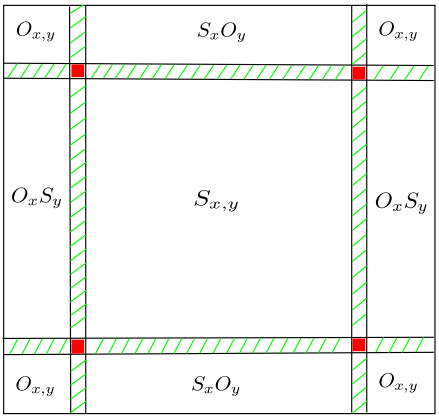

Fig. 3 Transition regions in a two-dimensional field. S indicates the symmetric kernel and O is the one-sided kernel

Similar to the one-dimensional case explained in the previous section, in order to have a smooth transition between the one-sided and the symmetric kernel, we use a convex combination. In two dimensions, the convex combination has the following form

$$u^\star(x,y) = \theta(x)\theta(y)u^\star_{S_{x,y}}(x,y) + (1-\theta(x))\theta(y)u^\star_{O_x,S_y}(x,y) + \theta(x)(1-\theta(y))u^\star_{S_x,O_y}(x,y) + (1-\theta(x))(1-\theta(y))u^\star_{O_{x,y}}(x,y), \quad (31)$$

where $u^\star_{S_{x,y}}(x,y)$ indicates the post-processed value at (x,y) using the symmetric kernel in both directions, and $u^\star_{O_{x,y}}(x,y)$ is the post-processed value using the one-sided kernel in both directions. However, it is not necessary to evaluate all four post-processed values mentioned in (31) whenever we are in a transition region (see (24) for the definition of a transition region). Usually, we only evaluate two post-processed values. Figure 3 depicts the transition regions for a two-dimensional field. In the striped green regions we only evaluate two post-processed values whereas in the red regions we need to calculate four post-processed values for a smooth transition.

3.2.2 Structured Triangular Mesh Implementation

In this section we discuss post-processing over structured triangular meshes. For this, we simply take the quadrilateral mesh implementation and apply the same kernel for structured triangular meshes. This means that we still use the kernel definition given in (25) and evaluate the integral in (26). However, now this is over a triangular region. The accuracy-enhancement capabilities of the SIAC filter over structured triangular regions has been thoroughly discussed in [18]. In general we are able to observe the $2k+1$ convergence order for post-processing over structured triangular grids. Here, we further explain the details of the implementation.

Figure 4 depicts a possible kernel-mesh intersection for a structured triangular mesh. As was done in the quadrilateral mesh case, we note that it is necessary to divide the element into subregions that respect both the element interfaces and kernel breaks, in order to perform the integrations exactly to machine precision. Moreover, we choose to further divide these subregions into triangles as shown in red in Fig. 4. Therefore, the post-processed solution at $(x,y) \in I_{i,j}$ becomes

$$u^\star(x,y) = \frac{1}{h_1h_2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} K\left(\frac{x_1-x}{h_1}\right)K\left(\frac{x_2-y}{h_2}\right)u_{h_1,h_2}(x_1,x_2)dx_1dx_2,$$

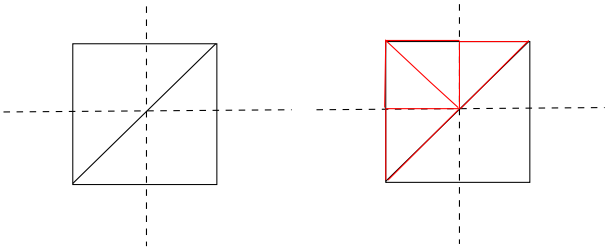

Fig. 4 A possible kernel-mesh overlap in two dimensions over a structured triangular mesh (*left*). *Dashed lines* represent the kernel patch and *solid lines* represent the DG mesh. In the right image, *red lines* depict possible integration regions over the upper element

$$= \frac{1}{h_1 h_2} \sum_{I_{i+d_1,j+d_2} \in Supp\{\tilde{K}\}} \left[\int\int_{U(I_{i+d_1,j+d_2})} K\left(\frac{x_1 - x}{h_1}\right) \times K\left(\frac{x_2 - y}{h_2}\right) u_{h_1,h_2}(x_1, x_2) dx_1 dx_2 + \int\int_{L(I_{i+d_1,j+d_2})} K\left(\frac{x_1 - x}{h_1}\right) K\left(\frac{x_2 - y}{h_2}\right) u_{h_1,h_2}(x_1, x_2) dx_1 dx_2 \right]. \quad (32)$$

In (32), we have simply modified (26) in Sect. 3.2.1 by dividing the integration over a quadrilateral element into two triangular elements $U(I_{i+d_1,j+d_2})$ and $L(I_{i+d_1,j+d_2})$. We refer to the quadrilateral element in this case a *super-element*, a term used by Mirzaee et al. in [18] to indicate the quadrilateral combination of the two diagonally aligned triangles in the DG structured mesh.

We consider one of the integrals in (32), and explain how this integration can be evaluated exactly to machine precision. After triangulating the integration regions as shown in Fig. 4, we arrive at

$$\begin{aligned} &\int\int_{U(I_{i+d_1,j+d_2})} K\left(\frac{x_1 - x}{h_1}\right) K\left(\frac{x_2 - y}{h_2}\right) u_{h_1,h_2}(x_1, x_2) dx_1 dx_2 \\ &= \sum_{n=0}^{3} \int\int_{\tau_n} K\left(\frac{x_1 - x}{h_1}\right) K\left(\frac{x_2 - y}{h_2}\right) u_{h_1,h_2}(x_1, x_2) dx_1 dx_2 \\ &= \int_{-1}^{1} \int_{-1}^{-\xi_2} K\left(\frac{\mu_1(\xi_1, \xi_2) - x}{h_1}\right) \\ &\quad \times K\left(\frac{\mu_2(\xi_1, \xi_2) - y}{h_2}\right) u_{h_1,h_2}(\mu_1(\xi_1, \xi_2), \mu_2(\xi_1, \xi_2)) |J_\xi| d\xi_1 d\xi_2, \end{aligned} \quad (33)$$

where τ_n is the triangular subelement in $U(I_{i+d_1,j+d_2})$, μ_1 and μ_2 are the appropriate mappings from the standard to local triangular region which are defined later in the section and $|J_\xi| = |\frac{\partial(x_1,x_2)}{\partial(\xi_1,\xi_2)}|$.

In (33), the third equation is derived by mapping the standard triangular element defined as

$$T_{st} = \{(\xi_1, \xi_2) | -1 \le \xi_1, \xi_2; \xi_1 + \xi_2 \le 0\} \quad (34)$$

to the local region τ_n. We also note that in order to have the triangular expansion be as efficient as the quadrilateral one, we want to be able to define the two dimensional basis functions used to evaluate our DG approximation, in terms of a tensor product of one-

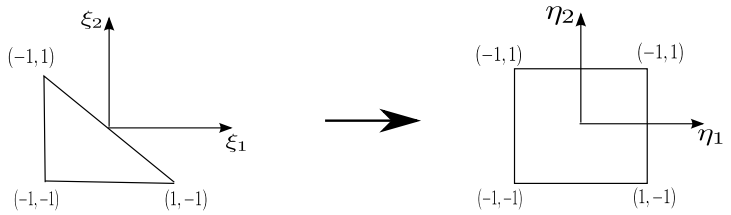

Fig. 5 Triangle to rectangle transformation

dimensional basis functions. Consequently, we define a mapping from the Cartesian coordinate system to the so-called *collapsed coordinate system* such that

$$\eta_1 = 2\frac{1+\xi_1}{1-\xi_2} - 1, \qquad \eta_2 = \xi_2, \tag{35}$$

with the inverse transformation

$$\xi_1 = \frac{(1+\eta_1)(1-\eta_2)}{2} - 1, \qquad \xi_2 = \eta_2. \tag{36}$$

These new local coordinates (η_1, η_2) define the standard triangular region by

$$T_{st} = \{(\eta_1, \eta_2) | -1 \leq \eta_1, \eta_2 \leq 1\}. \tag{37}$$

The transformation in (35) can be interpreted as a mapping from the triangular region to a rectangular one as seen in Fig. 5. This transformation is also known as the *Duffy transformation* (see [15]). Furthermore, for more information regarding tensorial basis functions and the collapsed coordinate system we refer the interested reader to [17].

Using the collapsed coordinate system, the integral in (33) now becomes

$$\begin{aligned}
&\int_{-1}^{1}\int_{-1}^{-\xi_2} K\left(\frac{\mu_1(\xi_1,\xi_2)-x}{h_1}\right) K\left(\frac{\mu_2(\xi_1,\xi_2)-y}{h_2}\right) u_{h_1,h_2}(\mu_1(\xi_1,\xi_2),\mu_2(\xi_1,\xi_2))|J_\xi| d\xi_1 d\xi_2 \\
&\quad = \int_{-1}^{1}\int_{-1}^{1} K\left(\frac{\mu_1^e(\eta_1,\eta_2)-x}{h_1}\right) \\
&\qquad \times K\left(\frac{\mu_2^e(\eta_1,\eta_2)-y}{h_2}\right) u_{h_1,h_2}(\mu_1^e(\eta_1,\eta_2),\mu_2^e(\eta_1,\eta_2))|J_\xi||J_\eta| d\eta_1 d\eta_2
\end{aligned} \tag{38}$$

where $|J_\eta| = |\frac{\partial(\xi_1,\xi_2)}{\partial(\eta_1,\eta_2)}|$.

Equation (38) results in the value of the two-dimensional integral over the triangular region τ_n. We note that since the kernel is a function of both variables in the standard as well as the collapsed coordinate systems, we can not separate the two-dimensional integration in terms of one-dimensional integrations as we did in (27). Consequently, the number of quadrature points required for integration should be enough to integrate polynomials of degree $3k$ exactly. Furthermore, if we denote the vertices of τ_n as x_i^A, x_i^B and x_i^C then we have

$$x_i = \mu_i(\xi_1,\xi_2) = x_i^A\frac{-\xi_2-\xi_1}{2} + x_i^B\frac{1+\xi_1}{2} + x_i^C\frac{1+\xi_2}{2}, \quad i = 1,2. \tag{39}$$

Substituting ξ_1 and ξ_2 using (36) we arrive at

$$x_i = \mu_i^e(\eta_1, \eta_2) = x_i^A \frac{1-\eta_1}{2}\frac{1-\eta_2}{2} + x_i^B \frac{1+\eta_1}{2}\frac{1-\eta_2}{2} + x_i^C \frac{1+\eta_2}{2}, \quad i = 1, 2. \tag{40}$$

We can also use a similar procedure to (28) to evaluate the DG approximation at $(x, y) \in U(I_{i,j})$. That is,

$$u_{h_1,h_2}(x, y) = \sum_{p=0}^{k}\sum_{q=0}^{k-p} u_{U(I_{i,j})}^{pq} \phi^{pq}(\xi_1, \xi_2), \tag{41}$$

with the difference that the basis functions are given by

$$\phi^{pq}(\xi_1, \xi_2) = \psi_p^a(\eta_1)\psi_{pq}^b(\eta_2), \tag{42}$$

with ψ_p^a and ψ_{pq}^b being the orthogonal or modified basis functions for triangular elements defined in [17]. In (41), $U(I_{i,j})$ represents the DG triangular element that contains (x, y) and η_1 and η_2 are obtained by first applying an inverse mapping that maps $U(I_{i,j})$ to the standard triangular region given in (34), and then using (35). The use of the sum-factorization technique mentioned in the previous section, is not beneficial here, since the quadrature points obtained as a result of several mappings do not necessarily follow a tensor-product form.

Algorithm 5 provides a pseudo-code for implementing the convolution operator over structured triangular meshes. As we mentioned earlier, for implementation purposes, we consider the kernel in two dimensions as a patch or a two-dimensional matrix of squares. Therefore, to find the intersection region of a triangle with the kernel, we simply find the intersection of the triangle with these squares (line 16 and 22 in Algorithm 5). For this we have used the *Sutherland-Hodgman* polygon clipping algorithm from computer graphics (see [26]). Moreover, similar ideas discussed in the previous section, apply to the scaling parameters h_1 and h_2 for the non-uniform mesh structure. For general unstructured grids these parameters as well as the kernel definition need to be modified properly to gain optimal error convergence. However, the general implementation scheme for unstructured triangular grids will be similar to that of the structured ones. Once we identify the elements covered by the kernel support, we solve a series of geometric intersection problems using our clipping algorithm to recognize the integration regions (similar to Fig. 4). As the unstructured mesh will be more complex, we are likely to get more integration regions comparing to the structured mesh. Investigation of the SIAC filter for unstructured triangular meshes is the subject of an ongoing research

We further note that in this paper, when choosing a distribution of points for integration, we prefer the *Lobatto*-type quadrature. Particularly, for triangular regions, we choose *Gauss-Lobatto-Legendre* (GLL) points on the x_1-direction and *Gauss-Radau-Legendre* (GRL) on the x_2-direction. GRL points absorb the Jacobian of the collapsed coordinate transformation and do not include the singularity at the collapsed vertex. For more information on these types of quadrature points, we refer the reader to [17].

We conclude this section by adding that the one-sided kernel explained in previous sections and the convex combination in (31) also apply to the triangular meshes.

Algorithm 5 2D triangular mesh convolution

for each evaluation point (x, y) **do**
  $I_{i,j}$ = the super-element to which (x, y) belongs
  h_1 = size of $I_{i,j}$ in direction x_1
  h_2 = size of $I_{i,j}$ in direction x_2
  if one-sided in x_1-direction **then**
    Calculate the one-sided kernel coefficients for x_1-direction
  end if
  if one-sided in x_2-direction **then**
    Calculate the one-sided kernel coefficients for x_2-direction
  end if
  {This simply gives the super-elements (partially) covered by the 2D kernel}
  kFootPrint = the footprint of the 2D kernel on the DG mesh
  for each super-element I in *kFootPrint* **do**
    {lower triangle}
    $L(I)$ = lower triangle
    intgRegions = intersection of $L(I)$ with each square in the 2D kernel patch
    for each triangle τ in *intgRegions* **do**
      intg $+=$ Result of the integral in (38)
    end for
    {upper triangle}
    $U(I)$ = upper triangle
    intgRegions = intersection of $U(I)$ with each square in the 2D kernel patch
    for each triangle τ in *intgRegions* **do**
      intg $+=$ Result of the integral in (38)
    end for
  end for
  $u^\star(x, y) = intg/(h_1 h_2)$
end for

3.2.3 Hexahedral Mesh Implementation

Similar to the two-dimensional case, the convolution kernel in three dimensions, can also be formed by performing the tensor product of one-dimensional kernels. That is,

$$\begin{aligned}\hat{K}(x, y, z) &= \sum_{\gamma_1=0}^{r_1}\sum_{\gamma_2=0}^{r_2}\sum_{\gamma_3=0}^{r_3} c_{\gamma_1} c_{\gamma_2} c_{\gamma_3} \psi(x-\gamma_1)\psi(y-\gamma_2)\psi(z-\gamma_3), \\ &= K(x) \times K(y) \times K(z),\end{aligned} \tag{43}$$

where x_{γ_1}, x_{γ_2}, and x_{γ_3} are the position of the kernel nodes in x_1-, x_2- and x_3-directions and we have denoted the three-dimensional coordinate system with (x_1, x_2, x_3). Furthermore, r_d, $d = 1, 2, 3$ is either $2k$ or $4k$, depending on whether we use a symmetric or one-sided kernel in the x_d-direction.

From (43), it is clear that the three-dimensional post-processor over hexahedral meshes will be a natural extension of the two-dimensional quadrilateral post-processor given in Sect. 3.2.1 and therefore, we will not provide further detail for this type of the post-processing.

In general, the three-dimensional post-processor can also be applied to other types of mesh elements in three dimensions such as tetrahedra, prisms and pyramids. However, the process of finding the geometric intersections of the mesh with the convolution kernel will be significantly more complex. In addition, as we mentioned earlier, kernel modification is required to make the post-processor suitable for general unstructured grids.

4 Post-Processing Using Inexact Integration

In Sect. 3.2 it was noted that in order to evaluate the integral involved in the convolution exactly to machine precision one needs to respect the breaks in continuity in the integrand. We explained how four integration regions can be formed for quadrilateral and triangular elements as a result of kernel-mesh intersection. However, the integration regions can be much greater than this when we consider totally unstructured grids. Numerical integration is quite costly and therefore in this section we use the idea of inexact integration to overcome this issue by ignoring the breaks in regularity of the kernel—so called kernel breaks—in the integration regions. This was demonstrated to be effective in [18, 19]. The idea is that at the DG element interfaces there is a lack of continuity, whereas at the kernel breaks there exist $\mathcal{C}^{k-1}$-continuity. Gaussian quadrature can overcome the higher-order continuity of the kernel breaks, but not the weak continuity of the DG breaks. Here we discuss the implementation details for the triangular mesh given in [18].

If we choose to implement the SIAC filter such that we only respect the DG element breaks, the integral in (33) becomes

$$\begin{aligned}
&\int\int_{U(I_{i+d_1,j+d_2})} K\left(\frac{x_1-x}{h_1}\right)K\left(\frac{x_2-y}{h_2}\right)u_{h_1,h_2}(x_1,x_2)dx_1dx_2\\
&\quad=\int_{-1}^{1}\int_{-1}^{-\xi_2} K\left(\frac{\mu_1(\xi_1,\xi_2)-x}{h_1}\right)\\
&\qquad\times K\left(\frac{\mu_2(\xi_1,\xi_2)-y}{h_2}\right)u_{h_1,h_2}(\mu_1(\xi_1,\xi_2),\mu_2(\xi_1,\xi_2))|J_\xi|d\xi_1d\xi_2\\
&\quad=\int_{-1}^{1}\int_{-1}^{1} K\left(\frac{\mu_1^e(\eta_1,\eta_2)-x}{h_1}\right)\\
&\qquad\times K\left(\frac{\mu_2^e(\eta_1,\eta_2)-y}{h_2}\right)u_{h_1,h_2}(\mu_1^e(\eta_1,\eta_2),\mu_2^e(\eta_1,\eta_2))|J_\xi||J_\eta|d\eta_1d\eta_2, \qquad (44)
\end{aligned}$$

where $\mu_i(\xi_1,\xi_2)$ and $\mu_i^e(\eta_1,\eta_2)$, $i=1,2$, are defined in (39) and (40) with x_i^A, x_i^B and x_i^C being the vertices of the DG triangular element $U(I_{i+r_1,j+r_2})$. Moreover, $|J_\xi|$ and $|J_\eta|$ are the Jacobian of these transformations.

For structured DG triangular meshes, it is possible to have up to seven integration regions when considering the kernel-mesh intersection. However, the reader should notice that these integration regions can increase significantly when dealing with unstructured meshes. The inexact scheme ignores these subregions and only considers the DG element itself. Therefore the integration regions are reduced to two triangular regions per a super-element.

Algorithm 6 provides a pseudo-code for the inexact post-processor over triangular meshes. Notice that in lines 16 and 19 we only perform one integration, which is over the DG element itself. However, we emphasize that in order to compensate for the lack of regularity in the integrand, one might need to use more quadrature points than what is required

Algorithm 6 2D inexact convolution for triangular meshes

for each evaluation point (x, y) **do**
  $I_{i,j}$ = the super-element to which (x, y) belongs
  h_1 = size of $I_{i,j}$ in direction x_1
  h_2 = size of $I_{i,j}$ in direction x_2
  if one-sided in x_1-direction **then**
    Calculate the one-sided kernel coefficients for x_1-direction
  end if
  if one-sided in x_2-direction **then**
    Calculate the one-sided kernel coefficients for x_2-direction
  end if
  {This simply gives the super-elements (partially) covered by the 2D kernel}
  kFootPrint = the footprint of the 2D kernel on the DG mesh
  for each super-element I in *kFootPrint* **do**
    {lower triangle}
    $L(I)$ = lower triangle
    intg += Result of the integral in (44) over $L(I)$
    {upper triangle}
    $U(I)$ = upper triangle
    intg += Result of the integral in (44) over $U(I)$
  end for
  $u^\star(x, y) = intg/(h_1 h_2)$
end for

with the exact scheme to gain the desired accuracy. Moreover, as the integration regions (DG elements) and hence the quadrature points are known prior to post-processing for this case, the values of the DG approximation at the quadrature points (x, y), i.e. $u_{h_1,h_2}(x, y)$, can be precalculated over the entire domain and reused many times during post-processing. We further note that the one-sided kernel ideas discussed in the previous sections also apply to the inexact scheme.

4.1 A Discussion on the Computational Complexity

In this section we discuss the complexity of computational cost for post-processing using exact quadrature and inexact quadrature.

As mentioned in Sect. 3.2, finding the intersection of the kernel mesh with the DG mesh is a geometric intersection problem that in general can be quite complex in two and three dimensions. The complexity increases due to the elements being further tessellated into sub-elements over which numerical integrations are performed. In our analysis thus far, we have only considered uniform triangular meshes where at most one kernel break per direction lies within a super-element. However, in the case of totally unstructured triangular meshes the number of breaks can be up to several breaks within an element which will result in more integration regions and therefore more numerical quadratures. Here, we consider the cost of the convolution operator, assuming that the integration subintervals have already been found.

The number of elements that are covered by the kernel support is dependent on the extent (width) of the convolution kernel, which is a function of the polynomial order per element. Therefore, if we denote the polynomial order as k and assume that all elements have the

same polynomial order in both directions, the number of elements that need to be considered for every evaluation point will be $O(k^2)$ in two dimensions. Furthermore, for each of these elements, depending on the number of integration regions within each element, a series of numerical quadratures must be performed. After transforming to the collapsed coordinate system, we evaluate integrals as shown in (38). Gaussian quadrature in two dimensions will be performed in $O(k^2)$ operations. However, we need to evaluate the kernel as well as the DG approximation at each of the quadrature points used in the integration. The DG approximation can be calculated at $O(k^2)$ quadrature points in $O(k^3)$ floating point operations using the so-called *sum-factorization* technique when possible, and in $O(k^4)$ operations otherwise. Kernel evaluation can also be performed in $O(k^3)$ operations in the case of quadrilateral elements or in $Q(k^4)$ for triangular elements. Hence, the overall cost of performing one numerical integration will be $O(k^4)$. Consequently, from (33), the cost of numerical quadrature on a single triangular element will be $O(Mk^4)$, with M being the number of integration regions within the element. When performing the exact post-processor scheme, $M \geq 1$ and is at most seven per triangular element in the case of a uniform mesh. However, this upper bound increases in the case of totally unstructured grids and will play a significant role in the overall performance of the algorithm, especially, when post-processing the entire DG field such that a transform to the modal representation is feasible. In that case, the total computational cost is $O(MNk^8)$, with N being the total number of elements in the field. On the other hand, the value of M is always one when performing the inexact approach. However, we note that depending on the accuracy requirements from post-processing when using the inexact scheme, more quadrature points may be needed when performing numerical integrations. Therefore, computationally, there might be a breakthrough after which the inexact post-processor will not perform better (if not worse) than the exact scheme. We discuss this more in the next section.

We also add that whenever the use of the position-dependent kernel is necessary, the kernel coefficients should be recalculated for the current evaluation point. In Sect. 3.1, we explained how to compute these coefficients by means of solving a linear system. Forming the matrix of the linear system costs $O(k^4)$ floating point operations and solving it is generally accomplished in $O(k^3)$ operations.

5 Performance Analysis of the Post-Processor

In this section we provide performance results for post-processing DG fields using the implementation strategies in Sect. 3.2. The post-processor is a good candidate for parallelization because when filtering an entire computational field, evaluating the post-processed value at one quadrature point is independent of the other. Therefore, having access to a multi-processor machine, we can have separate threads evaluate the post-processed value at different points without any communications among them. Using *OpenMP*, only a few compiler directives are required to parallelize the execution of the post-processor and gain proper scaling in the performance on a multi-processor shared-memory machine.

We note that although we are only considering the performance of the SIAC filter, we provide the performance when applied to a discontinuous Galerkin solution. For our results, we consider the traditional second order wave equation,

$$\eta_{tt} - \eta_{xx} - \eta_{yy} = 0, \qquad (x, y) \in (0, 1) \times (0, 1), \qquad T = 6.28. \tag{45}$$

We rewrite (45) as a system of first-order linear equations,

$$\begin{aligned} \eta_t + u_x + v_y &= 0 \\ u_t + \eta_x &= 0 \\ v_t + \eta_y &= 0, \end{aligned} \tag{46}$$

with initial conditions

$$\begin{aligned} \eta(x, y, 0) &= 0.01 \times (\sin(2\pi x) + \sin(2\pi y)) \\ u(x, y, 0) &= 0.01 \times (\sin(2\pi x)) \\ v(x, y, 0) &= 0.01 \times (\sin(2\pi y)), \end{aligned} \tag{47}$$

and 2π periodic boundary conditions in both directions. We apply the post-processor to the solutions of this DG problem for the η variable, after one period in time over triangular mesh structures. The numerical behavior of the SIAC filter for (45) is examined in [18]. From there, we observe the expected $2k+1$ order accuracy for filtering over structured triangular meshes. Here, we provide a thorough performance analysis of the parallelization.

Algorithm 7 depicts the condensed version of Algorithm 5 presented in Sect. 3.2.2. The OpenMP directives in lines 1 and 3 are used to parallelize the execution of the post-processor. As displayed in Algorithm 7, there exist three principle nested for loops in the code, and we choose to parallelize the outer most one to minimize the overhead due to initiation of OpenMP directives.

The performance results for post-processing an entire triangular DG field provided in this section consider six evaluation points per element. Results are provided for both the uniform and the smoothly-varying triangular meshes shown in Fig. 6, first post-processing using only the symmetric kernel, and then post-processing using the position-dependent kernel that uses a convex combination of filter types. Finally, we provide a performance comparison between the filtering approaches using exact and inexact integration. Note that

Algorithm 7 Parallel-2D-tri-post-processor

```
1:  ♯ pragma omp parallel
2:  {
3:  ♯ pragma omp for schedule(static/dynamic)
4:  for each evaluation point (x, y) do
5:     for each super-element I in kFootPrint do
6:        {lower triangle}
7:        for each triangle in intgRegions do
8:           intg += Result of the integral in (38)
9:        end for
10:       {upper triangle}
11:       for each triangle in intgRegions do
12:          intg += Result of the integral in (38)
13:       end for
14:    end for
15:    {Lines 5–14 will be repeated here if convex combination is needed.}
16:    u*(x, y) = intg/(h1 h2)
17: end for
18: }
```

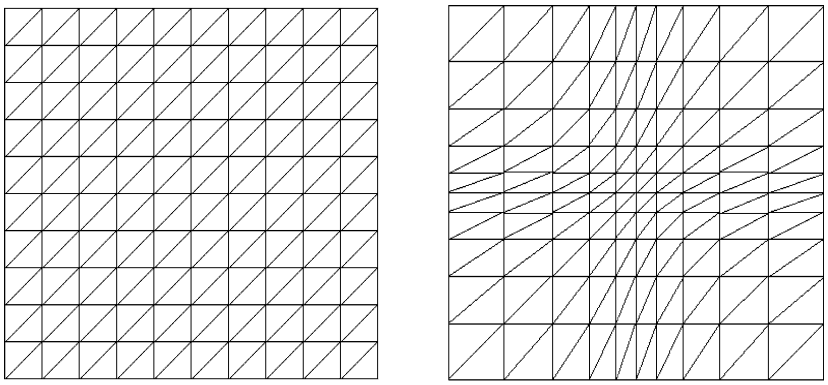

Fig. 6 Examples of the (**a**) uniform and the (**b**) smoothly-varying triangular meshes used in calculations

the timing results have been gathered on a SGI multi-processor machine with 2.67 GHz CPUs, using up to 16 threads.

5.1 Symmetric Kernel

The timing results for the symmetric kernel post-processing are given in Table 1 (uniform mesh) and Table 2 (smoothly-varying mesh). We note that the workload for the uniform mesh is statically assigned to each thread, as each thread performs an equal amount of work. However, for the smoothly-varying mesh, we have used dynamic scheduling to simulate an equal workload for each thread. Figures 7 and 8 demonstrate the performance scaling plots for the uniform and smoothly-varying meshes respectively. The scaling results have been calculated from the following,

$$scaling = \frac{T_{serial}}{T_{parallel}} \tag{48}$$

where T_{serial} represents the serial execution time and $T_{parallel}$ is the parallel execution time. Ideally, this parameter should result in the number of threads used in the parallel execution. We see that as we increase the order of the polynomial, the scaling approaches the theoretically desired scaling.

5.2 Position-Dependent Kernel

Applying the position-dependent SIAC filter allows for post-processing up to boundaries and allows for non-periodic boundary conditions. This position-dependent kernel uses a convex combination of kernels (see (31)) to allow for a smooth transition between the one-sided kernel and the symmetric kernel. The timing results for this position-dependent kernel are given in Table 3 for the uniform mesh and in Table 4 for the smoothly-varying mesh. We remind the reader that the difficulty in implementing the kernel near the boundary is that in each iteration of the OpenMP for loop, we need to re-calculate the kernel coefficients for the one-sided kernel. Moreover, if we are in a transition region, we need to calculate the convex combination of different kernel types. Consequently, the timing results are larger than the results for the symmetric kernel. In addition, the one-sided kernel has a wider support, which

Table 1 Timing results in seconds for symmetric kernel post-processing over the entire domain for the uniform triangular mesh considering $\mathbb{P}^2$, $\mathbb{P}^3$ and $\mathbb{P}^4$ polynomials. *th* represents the number of threads used in the parallel execution

Mesh	$th = 1$	$th = 2$	$th = 4$	$th = 8$	$th = 16$
$\mathbb{P}^2$					
$20^2 \times 2$	8.68	4.39	2.19	1.13	0.63
$40^2 \times 2$	34.74	17.70	8.85	4.48	2.47
$80^2 \times 2$	137.85	68.92	34.61	17.68	9.53
$\mathbb{P}^3$					
$20^2 \times 2$	39.76	19.98	10.00	5.02	2.64
$40^2 \times 2$	159.68	79.68	40.04	20.22	10.47
$80^2 \times 2$	632.34	316.43	158.77	80.45	40.39
$\mathbb{P}^4$					
$20^2 \times 2$	154.76	77.05	38.69	19.35	9.81
$40^2 \times 2$	617.32	310.56	155.65	78.27	39.43
$80^2 \times 2$	2455.01	1238.38	622.12	311.82	157.08

Table 2 Timing results in seconds for symmetric kernel post-processing over the entire domain for the smoothly-varying triangular mesh considering $\mathbb{P}^2$, $\mathbb{P}^3$ and $\mathbb{P}^4$ polynomials. *th* represents the number of threads used in the parallel execution

Mesh	$th = 1$	$th = 2$	$th = 4$	$th = 8$	$th = 16$
$\mathbb{P}^2$					
$20^2 \times 2$	13.74	6.95	3.46	1.78	0.94
$40^2 \times 2$	52.77	26.51	13.28	6.80	3.51
$80^2 \times 2$	208.10	104.59	52.40	26.84	13.92
$\mathbb{P}^3$					
$20^2 \times 2$	59.39	29.77	14.90	7.59	3.84
$40^2 \times 2$	223.59	112.04	56.14	28.57	14.64
$80^2 \times 2$	853.65	426.84	213.43	106.72	53.37
$\mathbb{P}^4$					
$20^2 \times 2$	202.85	101.81	51.06	25.72	12.98
$40^2 \times 2$	765.83	384.75	192.09	96.83	48.71
$80^2 \times 2$	2960.00	1483.83	742.29	372.28	188.58

results in more floating point operations near the boundaries. Dynamic scheduling has been used in this case.

Figures 9 and 10, provide the performance scaling plots when the position-dependent kernel is used. We again notice that as we increase the order of the polynomial, the scaling gets closer to the theoretical scaling. We further add that the behavior of the position-dependent kernel over the triangular mesh structure when using higher order polynomials ($\mathbb{P}^4$ in this case) has not yet been fully investigated and is the subject of an ongoing research. Moreover, for the coarsest smoothly-varying mesh ($N = 20^2 \times 2$), the width of the one-sided kernel close to the boundaries exceeds the width of the domain. We do not consider this as a valid scenario, therefore, the timing results have not been provided for this case.

We note here that Algorithm 7 can also be used for single-point post-processing. Where the super-elements over which the integrations are performed (line 5) will be divided among the threads. Similarly, we would expect to gain proper performance scaling when using multiple threads.

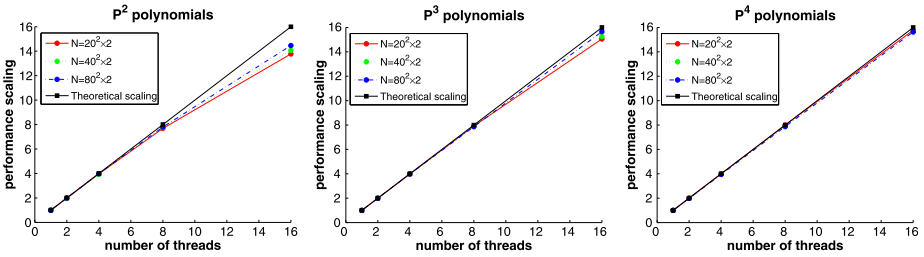

Fig. 7 Symmetric kernel post-processor performance scaling for the uniform triangular mesh. N represents the number of elements in the field

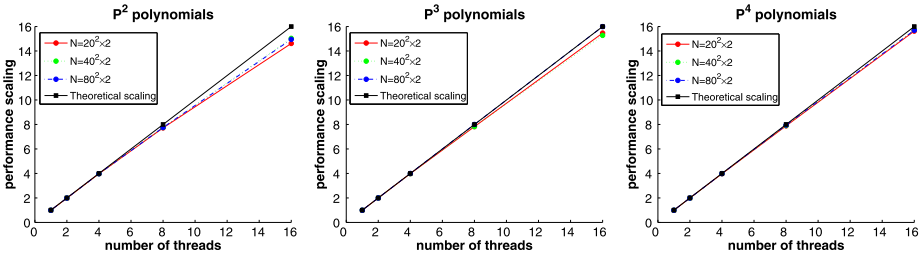

Fig. 8 Symmetric kernel post-processor performance scaling for the smoothly-varying triangular mesh. N represents the number of elements in the field

Table 3 Timing results in seconds for post-processing the entire domain for the uniform triangular mesh considering $\mathbb{P}^2$ and $\mathbb{P}^3$ polynomials. A position-dependent filter has been used to allow for post-processing near the boundaries. *th* represents the number of threads used in the parallel execution

Mesh	$th = 1$	$th = 2$	$th = 4$	$th = 8$	$th = 16$
$\mathbb{P}^2$					
$20^2 \times 2$	16.74	8.39	4.20	2.12	1.10
$40^2 \times 2$	51.45	25.74	12.89	6.48	3.36
$80^2 \times 2$	170.82	85.22	42.69	21.50	11.14
$\mathbb{P}^3$					
$20^2 \times 2$	83.92	41.94	21.00	10.54	5.33
$40^2 \times 2$	251.86	126.02	63.00	31.63	15.93
$80^2 \times 2$	819.21	409.33	204.92	102.79	52.00

5.3 Inexact quadrature

In Sect. 4, we discussed one effective strategy for overcoming the computational cost of post-processing discontinuous Galerkin approximations, through the use of inexact quadrature. In the inexact post-processing scheme, we try to overcome the lack of regularity in the integrand by increasing the number of quadrature points required for integration.

Table 5 presents the timing results for post-processing the solutions to the DG problem in (45) using inexact quadrature and the symmetric kernel, over the uniform and smoothly-varying triangular meshes. We have measured two different timings for these results. As we increase the number of quadrature points, t_1 represents the first execution time after which the post-processed solution has an L^2-error value better than the initial DG approximation. t_2 represents the execution time to get similar error values to the exact post-processing scheme.

Table 4 Timing results in seconds for post-processing the entire domain for the smoothly-varying triangular mesh considering $\mathbb{P}^2$ and $\mathbb{P}^3$ polynomials. A position-dependent kernel has been used to allow for post-processing near the boundaries. *th* represents the number of threads used in the parallel execution

Mesh	$th = 1$	$th = 2$	$th = 4$	$th = 8$	$th = 16$
$\mathbb{P}^2$					
$40^2 \times 2$	156.11	77.95	39.11	19.59	9.98
$80^2 \times 2$	514.80	257.47	128.96	64.65	32.74
$\mathbb{P}^3$					
$40^2 \times 2$	963.90	481.94	241.20	120.79	60.84
$80^2 \times 2$	3112.99	1556.49	777.81	388.71	194.97

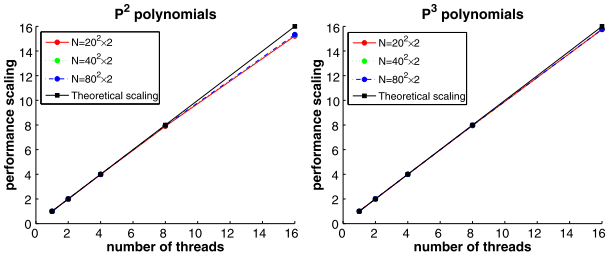

Fig. 9 Post-processor performance scaling for the uniform triangular mesh when the position-dependent kernel is used to allow for post-processing near the boundaries. *N* represents the number of elements in the field

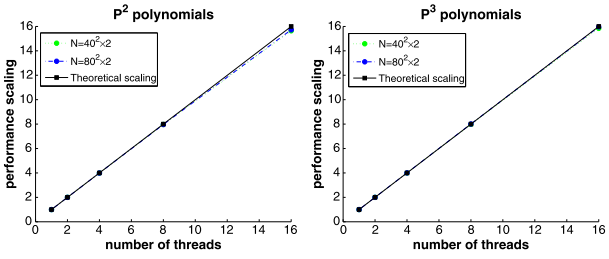

Fig. 10 Post-processor performance scaling for the smoothly-varying triangular mesh when using a position-dependent SIAC filter. *N* represents the number of elements in the field

From Table 5 it is obvious that the inexact scheme is more efficient when post-processing over a uniform mesh. For the smoothly-varying mesh, the inexact scheme is still a good choice for the coarser mesh structure. However, we notice that t_2 increases substantially compared to the exact post-processing execution time for the finer mesh. This is due to the increased number of quadrature points in the inexact scheme which in turn makes the evaluation of integrations an expensive task.

Let us look more closely at the results obtained for the smoothly-varying mesh. For this type of mesh, finding the integration regions (solving the geometric intersection problem mentioned in the earlier sections) does not take more than 5% of the overall computational time when performing the exact post-processing. Therefore, the major computational bottleneck is due to the several numerical quadratures evaluated per element. From our analysis of the smoothly varying mesh case, there are at most fifteen integration regions that can occur within an element as the result of kernel-mesh overlap. In addition, for quartic polynomials we need fifty-six quadrature points per integration region to evaluate the integrals over the triangular mesh exactly to machine precision. This means that we process at most $15 \times 56 = 840$ quadrature points per element when applying the exact post-processor. However, for the inexact case and $N = 40^2 \times 2$ mesh elements, we need at least 3306 quadrature points per element to get similar results to the exact scheme, which renders the inexact post-

Table 5 Serial execution timing results in seconds for inexact post-processing over the entire domain, using only the symmetric kernel, for the uniform and smoothly-varying triangular meshes, considering $\mathbb{P}^2$, $\mathbb{P}^3$ and $\mathbb{P}^4$ polynomials. t_1 represents the time where the inexact scheme yields results better than the DG initial approximation. t_2 represents the computational time spent to produce results similar to the exact post-processor. t_{exact} is the exact post-processing execution time

Mesh	t_1	t_2	t_{exact}
$\mathbb{P}^2$			
$20^2 \times 2$	2.26	3.15	8.68
$40^2 \times 2$	8.79	22.13	34.74
$\mathbb{P}^3$			
$20^2 \times 2$	10.01	10.01	39.76
$40^2 \times 2$	40.20	40.20	159.68
$\mathbb{P}^4$			
$20^2 \times 2$	30.54	30.54	154.76
$40^2 \times 2$	123.27	123.27	617.32

Uniform mesh

Mesh	t_1	t_2	t_{exact}
$\mathbb{P}^2$			
$20^2 \times 2$	4.80	34.54	13.74
$40^2 \times 2$	50.06	1696.98	52.77
$\mathbb{P}^3$			
$20^2 \times 2$	19.75	74.16	59.39
$40^2 \times 2$	297.41	3200	223.59
$\mathbb{P}^4$			
$20^2 \times 2$	71.97	224.00	202.85
$40^2 \times 2$	800.00	9300.00	765.83

Smoothly-varying mesh

Table 6 Serial execution timing results in seconds for inexact post-processing over the entire domain, using the position-dependent kernel near the boundaries, for the uniform triangular mesh, considering $\mathbb{P}^2$, $\mathbb{P}^3$ and $\mathbb{P}^4$ polynomials. t_1 represents the time where the inexact scheme yields results better the DG initial approximation. t_2 represents the computational time spent to produce results similar to the exact post-processor. t_{exact} is the exact post-processor execution time

Mesh	t_1	t_2	t_{exact}
$\mathbb{P}^2$			
$20^2 \times 2$	7.78	15.39	16.74
$40^2 \times 2$	24.71	49.62	51.45
$\mathbb{P}^3$			
$20^2 \times 2$	40.88	40.88	83.92
$40^2 \times 2$	126.52	126.52	251.86

Uniform mesh

processor inefficient in this case. Therefore, for fine mesh structures, unless the cost of finding the geometric intersections accounts for the majority of the computational complexity, the inexact post-processor performs worse than the exact post-processor in terms of computational cost. Nonetheless, post-processing that uses inexact integration is still a good choice for uniform meshes or non-uniform meshes with coarser mesh structures. Similar conclusions can be drawn for inexact post-processing when using the position-dependent kernel near the boundaries. Table 6, provides the timing results when using the position-dependent kernel. The inexact post-processor is not efficient for the smoothly-varying mesh structure and the position-dependent kernel, and therefore we neglect to provide timing results for this case.

6 Summary and Conclusions

This paper presents the explicit steps a numerical practitioner should use in order to implement the Smoothness-Increasing Accuracy-Conserving (SIAC) filters in an efficient manner. We consider quadrilateral and triangular element shapes in two-dimensions and hexahedra in three-dimensions for both uniform and smoothly-varying mesh structures. We address the computational tasks performed when post-processing discontinuous Galerkin (DG) fields. As the conventional way of post-processing through the use of matrix-vector multiplications has limited applicability, we provide a more general scheme to calculate the post-processed value, by directly evaluating the convolution operator of the post-processor. We demonstrate that when evaluating the convolution operator exactly to machine precision, we need to respect the breaks in continuity over the integration regions which are due to the element interfaces and kernel breaks. Consequently, the number of numerical quadratures can increase significantly when dealing with general mesh structures. In an attempt to overcome the cost of several numerical quadratures we discuss how using inexact quadrature in the post-processing step can be applied by ignoring the breaks in continuity of the convolution kernel, when performing the integrations. Furthermore, we provide results for the first time, that demonstrate the efficiency of the post-processor when parallelized on a shared-memory multi-processor machine, considering both the symmetric and the one-sided implementation of the post-processor. Moreover, we compare the performance of the inexact post-processor with the exact approach. In general these points can be drawn from our analysis:

- Calculating the post-processed values at a set of quadrature points within the computational field are independent of each other. Therefore, we expect to get close to perfect scaling by simply parallelizing this task on a multi-processor machine. This is indeed demonstrated in Figs. 7 through 10 in Sect. 5.
- In terms of computational time, the post-processor using inexact quadrature performs noticeably faster than the exact approach when implemented for a uniform mesh structure.
- For the smoothly-varying mesh, the post-processor using inexact quadrature is not beneficial for finer mesh structures, as a prohibitive number of quadrature points is needed for the integrations before we obtain results similar to those using exact quadrature.

As the post-processor using inexact quadrature performs more efficiently than the exact scheme when applied to uniform meshes, it can be used when performing local post-processing (not over the entire computational field).

Acknowledgements The first and third authors are sponsored in part by the Air Force Office of Scientific Research (AFOSR), Computational Mathematics Program (Program Manager: Dr. Fariba Fahroo), under grant number FA9550-08-1-0156. The second author is sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-09-1-3055. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Bramble, J., Schatz, A.: Higher order local accuracy by averaging in the finite element method. Math. Comput. **31**, 94–111 (1977)

2. Cockburn, B.: Discontinuous Galerkin methods for convection-dominated problems. In: High-Order Methods for Computational Physics. Lecture Notes in Computational Science and Engineering, vol. 9. Springer, Berlin (1999)
3. Cockburn, B., Hou, S., Shu, C.-W.: The Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. IV. The multidimensional case. Math. Comput. **54**, 545–581 (1990)
4. Cockburn, B., Karniadakis, G.E., Shu, C.-W.: Discontinuous Galerkin Methods: Theory, Computation and Applications. Springer, Berlin, (2000)
5. Cockburn, B., Lin, S.-Y., Shu, C.-W.: TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. III. One dimensional systems. J. Comput. Phys. **84**, 90–113 (1989)
6. Cockburn, B., Luskin, M., Shu, C.-W., Süli, E.: Post-processing of Galerkin methods for hyperbolic problems. In: Proceedings of the International Symposium on Discontinuous Galerkin Methods. Springer, Berlin (1999)
7. Cockburn, B., Luskin, M., Shu, C.-W., Süli, E.: Enhanced accuracy by post-processing for finite element methods for hyperbolic equations. Math. Comput. **72**, 577–606 (2003)
8. Cockburn, B., Ryan, J.K.: Local derivative post-processing for discontinuous Galerkin methods. J. Comput. Phys. **228**, 8642–8664 (2009)
9. Cockburn, B., Shu, C.-W.: TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. II. General framework. Math. Comput. **52**, 411–435 (1989)
10. Cockburn, B., Shu, C.-W.: The Runge-Kutta local projection P^1-discontinuous-Galerkin finite element method for scalar conservation laws. Modél. Math. Anal. Numér. **25**, 337–361 (1991)
11. Cockburn, B., Shu, C.-W.: The Runge-Kutta discontinuous Galerkin method for conservation laws. V. Multidimensional systems. J. Comput. Phys. **141**, 199–224 (1998)
12. Cockburn, B., Shu, C.-W.: Runge-Kutta discontinuous Galerkin methods for convection-dominated problems. J. Sci. Comput. **16**, 173–261 (2001)
13. Curtis, S., Kirby, R.M., Ryan, J.K., Shu, C.-W.: Post-processing for the discontinuous Galerkin method over non-uniform meshes. SIAM J. Sci. Comput. **30**(1), 272–289 (2007)
14. de Boor, C.: A Practical Guide to Splines. Springer, New York (2001)
15. Duffy, M.G.: Quadrature over a pyramid or cube of integrands with a singularity at a vertex. SIAM J. Numer. Anal. **19**(6), 1260–1262 (1982)
16. Schoenberg, I.J.: Cardinal Spline Interpolation. Conference Board of the Mathematical Sciences Regional Conference Series, vol. 12. Society for Industrial Mathematics, Philadelphia (1987)
17. Karniadakis, G.E., Sherwin, S.J.: Spectral/hp element methods for CFD, 2nd edn. Oxford University Press, London (2005)
18. Mirzaee, H., Ji, L., Ryan, J.K., Kirby, R.M.: Smoothness-increasing accuracy-conserving (SIAC) post-processing for discontinuous Galerkin solutions over structured triangular meshes. SIAM J. Num. Anal. (2011, in press)
19. Mirzaee, H., Ryan, J.K., Kirby, R.M.: Quantification of errors introduced in the numerical approximation and implementation of smoothness-increasing accuracy-conserving (SIAC) filtering of discontinuous Galerkin (DG) fields. J. Sci. Comput. **45**, 447–470 (2010)
20. Ritcher, G.: An optimal-order error estimate for the discontinuous Galerkin method. Math. Comput. **50**, 75–88 (1988)
21. Ryan, J.K., Shu, C.-W.: On a one-sided post-processing technique for the discontinuous Galerkin methods. Methods Appl. Anal. **10**, 295–307 (2003)
22. Ryan, J.K., Shu, C.-W., Atkins, H.: Extension of a post-processing technique for the discontinuous Galerkin method for hyperbolic equations with application to an aeroacoustic problem. SIAM J. Sci. Comput. **26**, 821–843 (2005)
23. Schumaker, L.: Spline Functions: Basic Theory. Wiley, New York (1981)
24. Slingerland, P.V., Ryan, J.K., Vuik, C.: Position-dependent smoothness-increasing accuracy-conserving (SIAC) filtering for improving discontinuous Galerkin solutions. SIAM J. Sci. Comput. **33**, 802–825 (2011)
25. Steffen, M., Curtis, S., Kirby, R.M., Ryan, J.K.: Investigation of smoothness enhancing accuracy-conserving filters for improving streamline integration through discontinuous fields. IEEE Transactions on Visualization and Computer Graphics **14**(3), 680–692 (2008)
26. Sutherland, I.E., Hodgman, G.W.: Reentrant polygon clipping. Commun. ACM **17**(1), 32–42 (1974)
27. Szabó, B.A., Babuska, I.: Finite Element Analysis. Wiley, New York (1991)
28. Walfisch, D., Ryan, J.K., Kirby, R.M., Haimes, R.: One-sided smoothness-increasing accuracy-conserving filtering for enhanced streamline integration through discontinuous fields. J. Sci. Comput. **38**, 164–184 (2009)