A second-order realizable scheme for moment advection on unstructured grids

Alberto Passalacqua, Frédérique Laurent, Rodney Fox

A second-order realizable scheme for moment advection on unstructured grids

Alberto Passalacqua[a, b, d *], Frédérique Laurent[b, c], Rodney O. Fox[d]

[a]*Department of Mechanical Engineering, Iowa State University, Black Engineering Building, Ames, IA 50011, USA*

[b]*Laboratoire EM2C, CNRS, CentraleSupélec, 3 rue Joliot Curie, Bâtiment Eiffel, 91190 Gif-sur-Yvette, France*

[c]*Fédération de Mathématiques de l'Ecole Centrale Paris, FR CNRS 3487, France*

[d]*Department of Chemical and Biological Engineering, Iowa State University, Sweeney Hall, Ames, IA, 50011-2230, USA*

Abstract

The second-order realizable ζ moment advection scheme developed in Laurent and Nguyen, (2017) is extended to the case of unstructured grids with cells of arbitrary shape. The necessary modifications to the scheme and the conditions under which the scheme ensures the realizability of the advected moment set are presented. The implementation of the scheme in the OpenFOAM® CFD toolbox is verified comparing the results obtained in one-dimensional test cases involving moment sets well inside the moment space, and at the boundary of the moment space. Results obtained with the proposed scheme are compared to the corresponding analytical solution. The scheme is then tested considering two-dimensional cases of pure moment advection with an imposed irrotational velocity field. First, a quadrilateral grid is considered to determine the order of the scheme and compare it to the results reported in Laurent and Nguyen (2017) with the same grid resolution. Then, the accuracy of the scheme on two-dimensional triangular grids is determined.

Keywords: Moment advection, realizability, unstructured grid, advection, schemes, moment methods, quadrature method of moments

1. Introduction

Moment methods are a class of mathematical methods used to determine approximate solutions to

* Corresponding author
Email address: albertop@iastate.edu (A. Passalacqua)

problems involving the evolution of a distribution function. Notable examples of equations describing this evolution are the population balance equation [2,3], the Boltzmann [4,5], the Boltzmann-Enskog [2], the Fokker-Planck and Kolmogorov [6–8] equations, in addition to several others. These equations can be written in compact form as

$$\frac{\partial f}{\partial t} + \nabla_{\mathbf{x}} \cdot (f\mathbf{v}) + \mathcal{S} = 0 \tag{1.1}$$

where $f(t, \mathbf{x}, \boldsymbol{\xi})$ is the distribution, $\mathbf{x}$ is the position vector, $\boldsymbol{\xi}$ is the vector of internal coordinates specific to the problem (e.g. size, composition, velocity, charge…), $\mathbf{v}$ is the velocity and $\mathcal{S}$ contains other terms depending on the evolution equation for f under consideration.

Moment methods [9–11] consist in deriving partial differential equations (PDEs) for the spatio-temporal evolution of a finite vector of moments of the distribution function characteristic of the problem under consideration, in order to reduce the dimensionality of the problem and, consequently, make it computationally more treatable. This is achieved by applying the definition of moment of f

$$m_{ijk\ldots} = \int_{\Omega} \xi_1^i \xi_2^j \xi_3^k \ldots f t, \mathbf{x}, \boldsymbol{\xi}) \mathrm{d}\boldsymbol{\xi} \tag{1.2}$$

to both sides of Eq. (1.1), which leads to conservation equations for the quantities $m_{ijk\ldots}$, which are the moments about the origin of f. The PDEs obtained in this way are then discretized on the domain of interest and solved numerically, typically using the finite-volume method [12,13]. The literature reports many example applications where moment methods were used [9,10,14–43], including applications to aerosols [11,17,18], gas-particle flows [37,44,45], gas-liquid flows [22,22,39,46], combustion [27,43,47], sprays [24,32,35,48] and radiation transport [14], to mention a few. In this work we focus on the transport of moments of a univariate distribution, where $\boldsymbol{\xi}$ contains only one scalar positive quantity, as it happens in the solution of population balance equations describing the evolution of the particle size in a particle population.

A key difficulty in formulating numerical methods for the solution of PDEs describing the evolution of a moment vector is the discretization of the advection term. As illustrated by Wright [49], this difficulty consists in preserving the moment realizability[2], which is systematically compromised if classical discretization schemes, different from first-order upwind, are used to discretize the advection

[2] A moment vector is said realizable on a support if a positive measure exists on the same support whose moments are the same as those in the considered moment vector. It is then in the moment space.

term [49]. The first-order upwind scheme, while used in most of the applications of moment methods found in the literature in order to avoid compromising moment realizability, is too dissipative, and requires extremely refined grid resolutions to achieve satisfactory results, seriously compromising the feasibility of simulations in applications involving large-scale domains or large gradients of the transported moment vector. Wright [49] examined several solutions to the problem of moment corruption due to advection, including the adoption of augmented schemes for vector transport, the transport of surrogate quantities related to the moment vector that needs to be advected, of quadrature weights and abscissae associated to the moment set in certain quadrature-based moment methods [50], and moment correction algorithms [49,51]. This latter approach consists in replacing the compromised moment vector with a valid one, obtained either by removing negative second-order differences [3] or enacting an optimization procedure to identify a moment vector that maximized $\ln\ m_k$ [51].

Kah et al. [32] developed a second-order realizable advection scheme for moment vectors of distribution with compact support, suitable for structured grids, in cases with moment vectors in the interior of the moment space. In their approach, instead of performing a direct spatial reconstruction of the moment vector of interest, the corresponding vector of canonical moments is considered. The boundedness of the canonical moments, which are necessarily positive scalars defined over $]0, 1[$ is leveraged to formulate a limiter which ensures not only the boundedness of the numerical solution, but also the realizability of the vector of transported moments under a condition on the integration time step. The resulting scheme was applied together with adaptive mesh refinement in [52].

Vikas et al. [53] proposed a quasi-high-order realizable advection scheme in the context of quadrature-based moment methods. In their scheme, which was formulated on unstructured grids, the advection term is computed as a function of the quadrature approximation of the NDF [3,11], using a MUSCL-type limited scheme for the reconstruction of the quadrature weights and a first-order upwind scheme for quadrature abscissae. The scheme ensures moment realizability if a condition on the integration time-step is satisfied. However, the formal order of accuracy of the scheme is limited by the first-order reconstruction of the quadrature abscissae. Vikas et al. [54] proposed a realizable scheme for diffusion problems, always in the context of quadrature-based moment methods.

Alldredge and Schneider [55] formulated a discontinuous Galerkin scheme in the context of entropy-based moment closures, coupling it with a strong stability preserving Runge-Kutta method for time integration. Since this approach relies on entropy-based closures, it is suitable for problems with moment vectors in the interior of the moment space but does not deal with the case of moment

vectors are the boundary of the moment space.

Laurent and Nguyen [1] developed a second-order realizable scheme by considering the reconstruction of positive quantities related to the set of transported moments in what they have called ζ scheme. This numerical scheme, which requires a CFL-like condition to ensure the realizability of the moment vector, was successfully applied to the transport of moment vectors of a regular NDF and of a bimodal NDF, with moments possibly at the boundary of the moment space. In both cases the scheme has shown the capability of preserving the realizability of the advected moment vector, while maintaining its accuracy.

The nature of the ζ scheme and, in particular, the version of this scheme called ζ simplified scheme by Laurent and Nguyen [1], makes it an ideal candidate for its extension to unstructured grids with cells of arbitrary shapes because the scheme relies on a traditional MUSCL reconstruction, and only requires a local additional limitation to be applied to the reconstructed quantities on cell faces. The extension of this scheme to unstructured grids and its implementation into the OpenFOAM framework [56] are the topics of this article, the remainder of which is organized as follows: in Sec. 2 the problem of moment transport is introduced. The ζ simplified scheme for hexahedral structured grids of Laurent and Nguyen [1] is summarized in Sec. 3. Its generalization to unstructured grids with cells of arbitrary shapes is discussed in Sec. 4. Finally, the same one- and two-dimensional test cases used by Laurent and Nguyen [1] are used in Sec. 5 to verify the implementation of the numerical scheme, using hexahedral uniform grids. Then, the two-dimensional cases are repeated using a triangular grid, and the order of accuracy of the ζ simplified scheme on unstructured grids is assessed.

2. Moment transport and moment advection

The focus of the present work is on the pure advection problem of a moment vector $\mathbf{m}_\mathrm{N} = (m_0, \dots, m_{\mathrm{N}-1})$ associated to a measure with support over $\mathbb{R}^+$, with a known velocity field $\mathbf{U}$, according to the set of equations

$$\frac{\partial m_k}{\partial t} + \nabla_\mathbf{x} \cdot (m_k \mathbf{U}) = 0, \forall m_k \in \mathbf{m}_\mathrm{N}. \tag{2.1}$$

The solution of Eq. (2.1) is non-trivial because the numerical schemes used to discretize the time derivative and the divergence term need to guarantee the moment vector $\mathbf{m}_\mathrm{N}$ remains realizable. This condition can be expressed through a set of non-linear relationships the components of $\mathbf{m}_\mathrm{N}$ need to

satisfy. These conditions are represented, for moments associated to a measure with support on $\mathbb{R}^+$, through the Hankel determinants [57,58]:

$$\underline{H}_{2p} = \begin{vmatrix} m_0 & \dots & m_p \\ \vdots & \ddots & \vdots \\ m_p & \dots & m_{2p} \end{vmatrix}, \qquad \underline{H}_{2p+1} = \begin{vmatrix} m_1 & \dots & m_{p+1} \\ \vdots & \ddots & \vdots \\ m_{p+1} & \dots & m_{2p+1} \end{vmatrix}, \tag{2.2}$$

with $\underline{H}_{-2} = \underline{H}_{-1} = 1$.

Necessary and sufficient condition for $\mathbf{m}_\mathrm{N}$ to be in the interior of the moment space is that $\underline{H}_{2p} > 0$ and $\underline{H}_{2p+1} > 0$. Moreover, when $\mathbf{m}_\mathrm{N}$ is at the boundary of the moment space, some of these Hankel determinants are null.

A discussion on realizable time integration schemes can be found, for example, in [53]. Since the focus of this article is on advection schemes, it suffices to remember that any time-integration scheme convex combination of explicit Euler steps is realizable under some restriction on the integration time step.

As anticipated in the introduction, the objective of this article is to extend the second-order realizable advection scheme of Laurent and Nguyen [1] to unstructured grids. To such a purpose, let us consider a computational grid made of cells of arbitrary shape. The volume of a generic cell c is $\Omega_\mathbf{c}$, and the number of faces of the same cell is $\mathrm{N}_{\mathrm{f},c}$. The subscript *own* indicates the reconstructed values of a variable at the cell face, assuming outgoing flux; the subscript *nei* indicates the reconstructed values of a variable at the cell face if the flux is going into the cell. For consistency with the convention adopted in OpenFOAM, we assume the flux of a transported property to be positive when outgoing with respect to the computational cell owning the face used to define the flux.

Following Vikas et al. [53], and using the nomenclature introduced above, Eq. (2.1) can be rewritten in semi-discrete form to obtain the evolution equation of the moment m_k defined at the center of each computational cells. Such equation reads

$$\frac{\partial m_{k,c}}{\partial t} + \frac{1}{\Omega_c} \sum_{f=0}^{\mathrm{N}_{\mathrm{f},c}-1} \left[m_{k,\mathrm{own},f} \max\left(\mathbf{U}_f \cdot \mathbf{S}_f, 0\right) + m_{k,\mathrm{nei},f} \min\left(\mathbf{U}_f \cdot \mathbf{S}_f, 0\right) \right] = 0, \tag{2.3}$$

where $\mathbf{S}_f$ is the vector normal to the surface of the cell face f, belonging to cell c, with magnitude equal to the surface area of the cell face $|\mathbf{S}_f|$ and pointing outward of the cell, and $\mathbf{U}_f$ is the reconstructed value of $\mathbf{U}$ at the same face.

A procedure needs to be developed to compute the moments $m_{k,\mathrm{own},f}$ and $m_{k,\mathrm{nei},f}$ to ensure the realizability of the advected moment vector $\mathbf{m}_\mathrm{N}$ because, as discussed in the introduction, conventional finite-volume advection schemes, relying on reconstructions of order higher than one, do not guarantee the realizability of the transported moment set [49].

3. The simplified ζ advection scheme

This section summarizes the realizable ζ simplified advection scheme [1], which represents the foundation of the scheme for unstructured grids subject of the present work. This scheme was developed for one-dimensional problems with uniform spatial discretization and extended to multiple dimensional problems with Cartesian grids through dimensional splitting. Only the one-dimensional scheme is presented here.

In the ζ simplified scheme, the moment vector $\mathbf{m}_\mathrm{N}$ is advected by considering the auxiliary vector $\boldsymbol{\zeta}_{\mathrm{N}-1} = (\zeta_0, \dots, \zeta_{N-2})$, with

$$\zeta_p = \frac{\underline{H}_{p+1}\underline{H}_{p-2}}{\underline{H}_p\underline{H}_{p-1}}, p = 0, 1, \dots, \mathrm{N}-2, \tag{3.1}$$

where $\underline{H}_{2p}$ and $\underline{H}_{2p+1}$ are the Hankel determinants in Eq. (1.2).

These quantities are efficiently calculated using the QD algorithm [57], the Chebychev algorithm [1] or the Wheeler algorithm [59]. The adoption of the last two approaches is particularly convenient in the context of quadrature-based moment methods because the coefficients required to calculate the quantities ζ_p correspond to the coefficients of the recurrence relationship of the orthogonal polynomials used to determine the quadrature formula.

Once the vector $\boldsymbol{\zeta}_{\mathrm{N}-1}$ has been defined, the ζ simplified advection scheme consists of the following steps:

1. The zero-order moment m_0 is advected using a conventional MUSCL scheme with the minmod limiter.
2. The values of ζ_p are reconstructed at each side of each cell face using the same MUSCL-type reconstruction as for m_0, which ensures the positivity of the quantities ζ_p is preserved.
3. An additional limitation is conditionally applied to the slopes used to perform the reconstruction at the previous point, if needed to ensure the realizability of the advected

moment set. This additional limitation is defined based on the realizability of the moment vector [1,60]

$$\mathbf{m}^* = 3\mathbf{m} - \mathbf{m}^+ - \mathbf{m}^-, \tag{3.2}$$

where $\mathbf{m}$ is the moment vector at the cell center, $\mathbf{m}^+$ and $\mathbf{m}^-$ are the moment vectors computed from the values ζ_p^+ and ζ_p^- reconstructed at cell faces. If the moment vector defined by Eq. (3.2) is not realizable, an iterative procedure is used to reduce the slopes used to reconstruct ζ_p on cell faces, to ensure the realizability of the moment vector at the cost of the accuracy of the reconstruction. Details of this procedure are reported in Laurent and Nguyen (2017), and its generalization to unstructured grids is discussed in this work.

4. Once the reconstructed face values of ζ_p which ensure the realizability of the entire advected moment set are found, the moment vectors $\mathbf{m}^+$ and $\mathbf{m}^-$ are recomputed from the values ζ_p^+ and ζ_p^- [61].
5. The integration of the advection term proceeds computing the flux of each moment at each cell face based on the direction of the velocity at the centroid of the same face and updating the values of the moments at the cell center, as prescribed by Eq. (2.3).

The scheme summarized above was proven [1] to preserve the realizability of the advected moment vector if the time step is adapted to maintain the value of the CFL number below 1/3, while converging to second-order accuracy with a sufficiently refined spatial discretization.

The extension of the ζ simplified scheme to unstructured grids requires the generalization of the definition of $\mathbf{m}^*$ in Eq. (3.2), and of the CFL condition, which is expected to become a function of the number of faces of the grid cell.

4. Generalization to unstructured grids

The generalization of the ζ simplified scheme to unstructured grids is achieved in this work by considering a co-located grid arrangement, where all the variables are stored at cell centroids, with the exclusion of the values at the boundary of the computational domain, where values are stored at face centroids. This approach is illustrated, for example, in Ferziger and Peric [13], and was adopted in the computational toolkit [56,62,63] used in this work.

In order to evaluate the advection term considering the grid arrangement described above, values of

the velocity and of the advected quantities need to be known at cell faces. The reconstruction of the advected scalars, moments in our case, is illustrated in Sec. 4.1, following the approach implemented into OpenFOAM [56,62], based on the work of several authors [64–70] and summarized by Darwish and Moukalled [71].

4.1. Reconstruction on unstructured grids

Let us consider a generic advected quantity whose value, stored at the centroid C of a generic cell of the computational mesh, is indicated with ψ_{C}. The value of ψ reconstructed on the cell face f is [67,71]

$$\psi_{\mathrm{f}} = \psi_{\mathrm{C}} + \frac{1}{2}\mathcal{L}(r_{\mathrm{f}})(\psi_{\mathrm{D}} - \psi_{\mathrm{C}}), \tag{4.1}$$

where $\mathcal{L}(r_{\mathrm{f}})$ is the limiter function, which depends on the ratio [71]

$$r_{\mathrm{f}} = \frac{2\nabla\psi_{\mathrm{C}} \cdot \mathbf{r}_{\mathrm{CD}}}{\psi_{\mathrm{D}} - \psi_{\mathrm{C}}} - 1, \tag{4.2}$$

with $r_f \geq 0$ for TVD schemes [71].

In these expressions, ψ_{D} represents the downwind value at the center of the neighbor cell sharing face f with the cell whose center is situated in C. The vector $\mathbf{r}_{\mathrm{CD}}$ joins the centers C and D of the same two neighbor cells. The definition of the limiter function for a minmod limiter is [66,71]:

$$\mathcal{L}(r_{\mathrm{f}}) = \max(0, \min(1, r_{\mathrm{f}})). \tag{4.3}$$

The evaluation of the gradient of ψ over the cell C in Eq. (4.2) is detailed in Darwish and Moukalled [71]. A second-order least-squares method is used in this work to ensure accuracy on grids with cells of arbitrary shape and non-uniform spatial distribution.

4.2. Additional limitation for moment sets at the boundary of the moment space

The condition in Eq. (3.2), used to identify cases when an additional limitation is needed for the reconstructed values of the quantities ζ_p, needs to be generalized to be applicable to unstructured grids. To this purpose, let $\mathrm{N_f}$ be the maximum number of faces of a cell of the computational grid. Since arbitrary shaped cells are considered, this value can be different for any cell, without additional complication of the numerical scheme. Let also the flux of the quantities be positive when outgoing with respect to the computational cell under consideration. The number of faces characterized by outgoing flux with respect to the owner cell will be indicated with $\mathrm{N_{f,o,c}}$. The first step required to

extend the ζ simplified scheme [1] to unstructured grids consists in computing the value of the zero-order moment m_0 and of the components ζ_p of the vector $\boldsymbol{\zeta}_{\mathrm{N}-1}$ at both sides of each face internal to the computational domain[3]. The symbols $m_{0,\mathrm{own}}$ and $m_{0,\mathrm{nei}}$ are used to indicate the values of m_0 reconstructed, respectively, at the side of the owner and neighbor cell. Similarly, $\zeta_{p,\mathrm{own}}$ and $\zeta_{p,\mathrm{nei}}$ indicate the values of ζ_p at the side of the owner cell and of the neighbor cell to the face, as discussed for m_0.

Once the reconstruction of m_0 and $\boldsymbol{\zeta}_{\mathrm{N}-1}$ is obtained, the moments at each side of each cell face are recomputed from the vectors $(m_{0,\mathrm{own}}, \boldsymbol{\zeta}_{\mathrm{N}-1,\mathrm{own}})$ and $(m_{0,\mathrm{nei}}, \boldsymbol{\zeta}_{\mathrm{N}-1,\mathrm{nei}})$ using the algorithm proposed by Skibinsky [61].

At this point, it is determined if an additional limitation needs to be applied to the reconstructed values of ζ_p, to guarantee the realizability of the advected moment set. This is achieved by defining, in each computational cell c, the quantity

$$m_{k,c}^{+} = \sum_{f=0}^{\mathrm{N_{f,o,c}}-1} m_{k,c,f}, \tag{4.4}$$

where $m_{k,c,f} = m_{k,\mathrm{own},f}$ is the value of the moment m_k reconstructed on face f of cell c, where the flux is outgoing. The quantities $m_{k,c}^{+}$ are then used to define the components of the $\mathbf{m}^*$ vector as

$$m_{k,c}^{*} = (\mathrm{N_{f,o,c}} + 1) m_{k,c} - m_{k,c}^{+}, \qquad k = 0, \dots, \mathrm{N} - 1. \tag{4.5}$$

If the number of realizable moments in $\mathbf{m}^*$ is smaller than the number of realizable moments in the cell, an additional limitation needs to be applied to the reconstructed values of the ζ_p quantities. The procedure is analogous to the one described in [1], which is summarized here in a slightly modified fashion to clarify the details of its implementation into OpenFOAM.

While not strictly necessary, the reconstructed values $\zeta_{p,\mathrm{f}}$ at cell faces are decomposed, for convenience in the implementation, into the value obtained from a first-order reconstruction $\zeta_{p,\mathrm{f,up}}$, and into a high-order correction $\zeta_{p,\mathrm{f,h.o.}}$ This correction is then multiplied by a coefficient $\lambda_{p,\mathrm{f}} \in [0, 1]$, which is the limiter that needs to be determined to ensure the realizability of the advected

[3] Faces of the computational grid are indicated as *internal* when they do not belong to a boundary of the computational domain. Values of moments at boundary faces are either known, if imposed at the boundary, or obtained from the only neighbor cell which owns the boundary face.

moment set:

$$\zeta_{p,\mathrm{f}} = \zeta_{p,\mathrm{f,up}} + \lambda_{p,\mathrm{f}}\zeta_{p,\mathrm{f,h.o.}}. \tag{4.6}$$

The value of $\lambda_{p,\mathrm{f}}$ is found by considering the values of the limiters determined based on the owner and the neighbor cell of face f, and taking their minimum:

$$\lambda_{p,\mathrm{f}} = \min(\lambda_{p,\mathrm{own}}, \lambda_{p,\mathrm{nei}}). \tag{4.7}$$

In each cell, the value of the limiter $\lambda_{k,c}$ is determined as follows:

1. Set $\lambda_{p,c} = 1$.
2. Compute $\mathbf{m}_c^+$ using $\zeta_{l,\mathrm{f}}$ with $l \leq k$, and with $\zeta_{p,\mathrm{f,up}}$, for $l > p$.
3. Recompute $\mathbf{m}_c^*$ using the values of $\mathbf{m}_c^+$ found at the previous step.
4. Calculate the number of realizable moments in $\mathbf{m}_c^*$. If this number is smaller than the number of realizable moments in $\mathbf{m}_c$, set $\lambda_{p,c} = 1/2$.
5. Recompute $\zeta_{p,c}$ from Eq. (4.6).
6. Repeat the calculation at points 2 – 4 with the new value of $\zeta_{p,c}$. If the number of realizable moments in the updated $\mathbf{m}_c^*$ is smaller than the number of realizable moments in $\mathbf{m}_c$, set $\lambda_{p,c} = 0$.
7. Repeat from 1 for all p to define the limiter for each of the elements of $\boldsymbol{\zeta}_{\mathrm{N-1}}$.

Once the cell values $\lambda_{p,c}$ are defined for each cell, the face limiters are computed according to Eq. (4.7), and the reconstructed values of $\boldsymbol{\zeta}_{\mathrm{N-1,own}}$ and $\boldsymbol{\zeta}_{\mathrm{N-1,nei}}$ are updated applying Eq. (4.6) to each of their components. The updated moment vectors $\mathbf{m}_{\mathrm{N-1,own}}$ and $\mathbf{m}_{\mathrm{N-1,nei}}$ are then found using the Skibinsky [61] algorithm to account for the additional limitation determined according to the procedure described above. These updated moment sets are used to define the advection term in the moment conservation equations (Eq. (2.3)).

4.3. Courant – Friedrichs - Lewy condition for realizability

The proposed scheme ensures the realizability of the advected moment set if the time step Δt used to perform the integration satisfies a CFL condition obtained at the cell faces at which the flux is outgoing. If we consider an arbitrary cell with $\mathrm{N_{f,c}}$ faces, $\mathrm{N_{f,o,c}}$ of which have an outgoing flux, the maximum value of the CFL number is

$$\mathrm{CFL}_{\max} = \frac{1}{2}\max_{\mathrm{c}}\left(\frac{\sum_f \mathbf{U}_f \cdot \mathbf{S}_f}{V_{\mathrm{c}}}\right)\Delta t, \tag{4.8}$$

where V_{c} is the cell volume, $\mathbf{U}_f$ the flow velocity at the cell face f with surface $\mathbf{S}_f$, belonging to cell c, and Δt is the time step. Using the same notation, the maximum value of the time step to ensure moment realizability is defined by the condition

$$\max_c\left[\max_f\left(\frac{\mathbf{U}_f \cdot \mathbf{S}_f}{V_c}\right)\right]\Delta t \leq \min\frac{1}{\mathrm{N_{f,o,c}} + 1}, \tag{4.9}$$

where the minimum function on the right-hand side of Eq. (4.9) is introduced to account for the different number of faces cells may have in unstructured grids.

This is proven by considering the moment vector $\mathbf{m}^n_{\mathrm{N},c}$ at time t_n, and integrate it according to Eq. (2.3) with a backward Euler step to obtain the updated moment vector $\mathbf{m}^{n+1}_{\mathrm{N},c}$ at time t_{n+1}. Assuming $\mathbf{U}_f \cdot \mathbf{S}_f > 0$ for $f = 1, \dots, \mathrm{N_{f,o,c}}$ and $\mathbf{U}_f \cdot \mathbf{S}_f \leq 0$ for $f = \mathrm{N_{f,o,c}}, \dots, \mathrm{N_{f,c}}$, it is possible to write:

$$\mathbf{m}^{n+1}_{\mathrm{N,c}} = \underbrace{\mathbf{m}^n_{\mathrm{N,c}} - \frac{\Delta t}{V_{\mathrm{c}}}\sum_{f=1}^{\mathrm{N_{f,o,c}}}\mathbf{m}^n_{\mathrm{N,own},f}\mathbf{U}_f \cdot \mathbf{S}_f}_{\mathcal{A}} \underbrace{- \frac{\Delta t}{V_{\mathrm{c}}}\sum_{f=\mathrm{N_{f,o,c}}+1}^{\mathrm{N_{f,c}}}\mathbf{m}^n_{\mathrm{N,c}}\mathbf{U}_f \cdot \mathbf{S}_f}_{\mathcal{B}}. \tag{4.10}$$

in which only the term $\mathcal{A}$ is of interest in the proof because $\mathcal{B}$ is necessarily realizable. The moment vector $\mathbf{m}^n_{\mathrm{N,c}}$ can be expressed in terms of $\mathbf{m}^*_{\mathrm{c}}$

$$\mathbf{m}^n_{\mathrm{N,c}} = \frac{1}{\mathrm{N_{f,o,c}} + 1}\left(\mathbf{m}^*_{\mathrm{c}} + \sum_{f=1}^{\mathrm{N_{f,o,c}}}\mathbf{m}^n_{\mathrm{N,own},f}\right), \tag{4.11}$$

consequently:

$$\mathcal{A} = \frac{1}{\mathrm{N_{f,o,c}} + 1}\mathbf{m}^*_{\mathrm{c}} + \sum_{f=1}^{\mathrm{N_{f,o,c}}}\left(\frac{1}{\mathrm{N_{f,o,c}} + 1} - \frac{\Delta t}{V_{\mathrm{c}}}\mathbf{U}_f \cdot \mathbf{S}_f\right)\mathbf{m}^n_{\mathrm{N,own},f}. \tag{4.12}$$

By imposing the term in the summation is positive to ensure the realizability of the moment vector, the condition

$$\frac{1}{\mathrm{N_{f,o,c}} + 1} \geq \frac{\Delta t}{V_{\mathrm{c}}}\mathbf{U}_f \cdot \mathbf{S}_f \tag{4.13}$$

is obtained, which, applied to each value of f and c leads to Eq. (4.9).

It is worth observing that the definition in Eq. (4.9) automatically includes faces internal to the computational domain and faces lying on the boundary. Consequently, no special consideration for boundary conditions where the flux is imposed is needed to correctly define the maximum CFL required to ensure moment realizability.

The condition in Eq. (4.13) reduces, in the case of a uniform one-dimensional discretization with step Δx, to

$$\Delta t \leq \frac{\Delta \mathrm{x}}{2U_f} \tag{4.14}$$

because $\mathrm{N_{f,o,c}} = 1$, assuming there is no mass and momentum source terms localized in the computational cell. The time-step restriction in multi-dimensional cases requires to consider how many cell faces have outgoing fluxes. Assuming a uniform discretization in both directions of a two-dimensional case, the condition in Eq. (4.13) becomes:

$$\Delta t \leq \frac{\Delta \mathrm{x}}{\left(\mathrm{N_{f,o,c}} + 1\right)U_f}. \tag{4.15}$$

Since $\mathrm{N_{f,o,c}} \in \{1, 2, 3\}$, the maximum restriction on the time-step size is possible when $\mathrm{N_{f,o,c}} = 3$, which leads to

$$\Delta t \leq \frac{\Delta \mathrm{x}}{4U_f}. \tag{4.16}$$

Following a similar reasoning, in a triangular grid, assuming triangles are equilateral with side length ℓ, the most restrictive constraint is achieved when two faces have outgoing flux $(\mathrm{N_{f,o,c}} = 2)$, leading to

$$\Delta t \leq \frac{\ell}{3U_f}. \tag{4.17}$$

In comparison, the upwind scheme ensures realizability of the advected moment vector if [54]

$$\Delta t \leq \frac{\Delta x}{U_f}. \tag{4.18}$$

5. Test cases

5.1. One-dimensional advection of moments well inside the moment space

The first test case for the implementation of the numerical scheme consists in a one-dimensional advection problem on $[0,1]$ with periodic boundary conditions and with constant velocity equal to $\mathbf{U} = (1, 0, 0)$. The considered moment set, used as initial condition for the test case, is in the interiod of the moment space and defined through the following auxiliary functions:

$$\alpha(x) = 3.5 + 1.5 \sin 2\pi x, \tag{5.1}$$

$$\beta(x) = 3.5 - 1.5 \cos 2\pi x, \tag{5.2}$$

$$\widetilde{m}_k(x) = \begin{cases} 1 & k = 0 \\ \dfrac{m_{k-1}(x)[\alpha(x) + k - 1]}{\alpha(x) + \beta(x) + k - 1} & k < \mathrm{N'} \end{cases} \tag{5.3}$$

$$\Theta(x) = \begin{cases} \dfrac{1}{2}\left\{1 + \tanh\left[\tan\left(2x - \dfrac{1}{2}\right)\right]\right\} & x \leq \dfrac{1}{2} \\ \dfrac{1}{2}\left\{1 + \tanh\left[\tan\left(\dfrac{1}{2} - 2x\right)\right]\right\} & x > \dfrac{1'}{2} \end{cases} \tag{5.4}$$

which allow the values of the initial moments to be computed as

$$m_k(x) = \widetilde{m}_k(x)\Theta(x). \tag{5.5}$$

The solutions obtained considering eight moments with 50, 100, 500 and 1000 uniformly distributed cells, after two flow-through times, are reported in Fig. 1. It is possible to observe from Fig. 1(a) that the grid resolutions of 50 and 100 cells are not sufficient to accurately preserve the moments because of the numerical diffusion caused by coarse grid resolution. However, the realizability of the moment set is preserved. The accuracy of the results significantly improves in the case of a grid resolution of 500 cells (Fig. 1(c)), which allows the analytical solution to be reproduced nearly exactly. The solution obtained on the grid with 1000 cells (Fig. 1(d)) is visually indistinguishable from the one calculated on 500 cells. Results are in agreement with those reported in the literature [1].

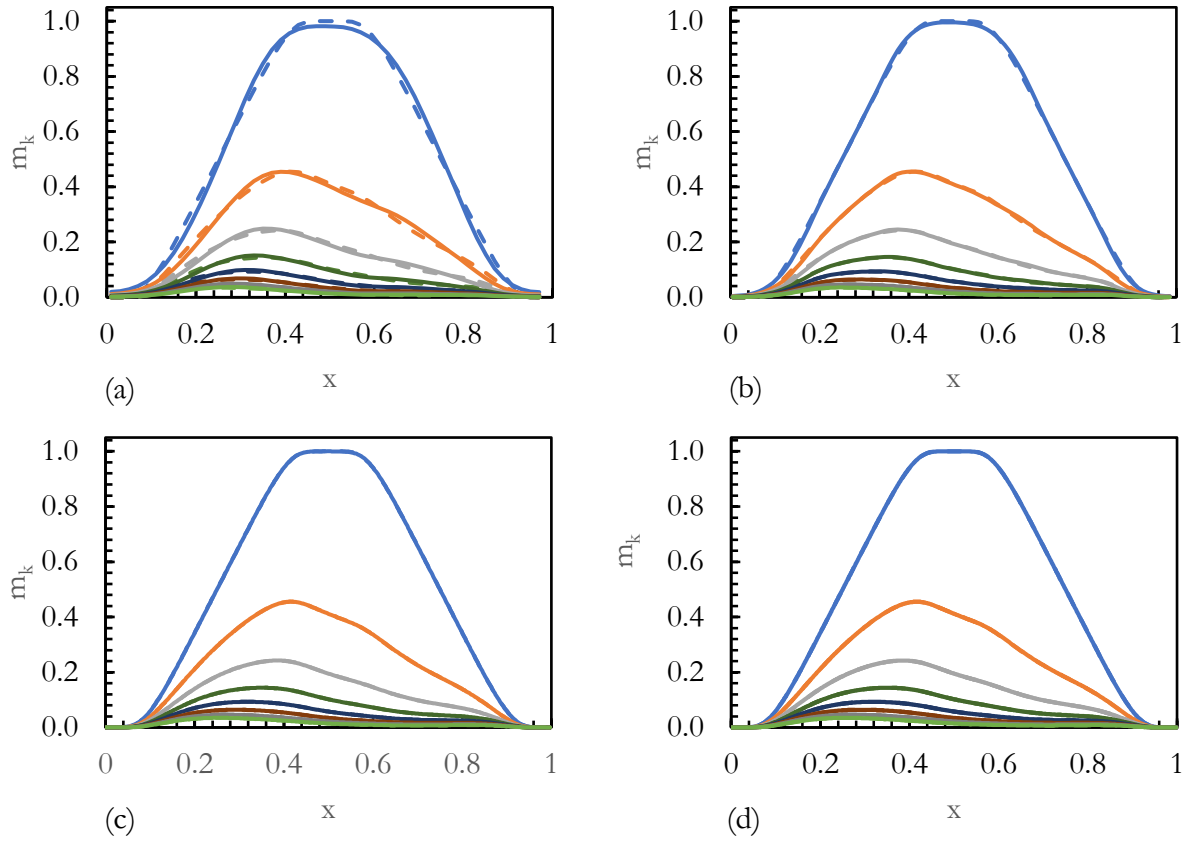


Fig. 1: Moments of the regular NDF. Analytical solution (dashed lines) and numerical prediction (solid lines) with (a) 50, (b) 100, (c) 500 and (d) 1000 cells – Zeta scheme.

In order to quantify the order of accuracy of the numerical scheme, the error of the numerical prediction with respect to the exact solution is calculated in each computational cell as

$$e_k = \frac{|m_{k,exact} - m_{k,num}|}{m_{k,exact}}. \tag{5.6}$$

The logarithm of the error is reported as a function of the logarithm of the grid size $h = \Delta x$ in Fig. 2, which illustrates that the second order of accuracy of the advection scheme is recovered for all the moments of the transported moment vector. Numerical values of the errors reported in Fig. 2 are shown in Table 1.

The test case presented in this section serves as preliminary verification of the implementation of the ζ simplified scheme into OpenFOAM. However, the additional limitation of the ζ_p quantities was not triggered while performing the simulations required to produce the results in Fig. 1. Therefore, a different test case was considered to also examine this aspect of the implementation of the scheme, as

illustrated in Sec. 5.2, where a case involving a multimodal distribution, with moments potentially at the boundary of the moment space is considered.

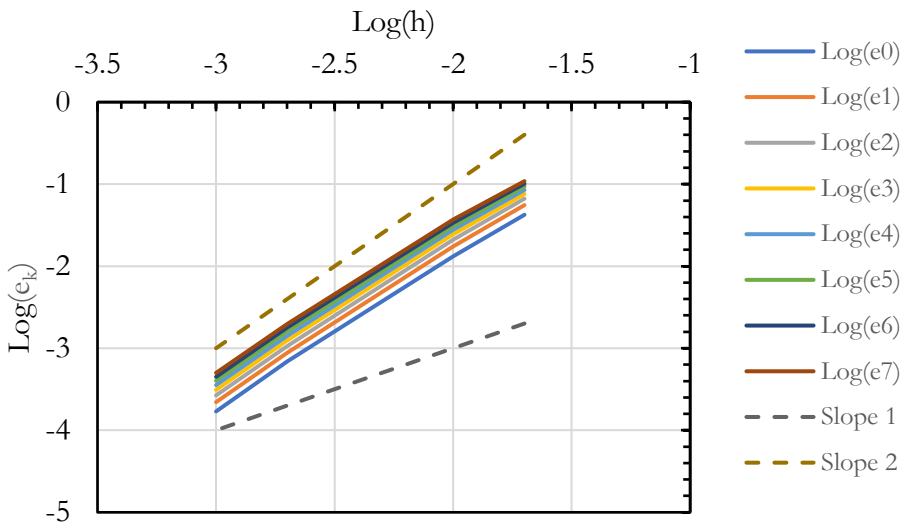


Fig. 2: Logarithm of the error in the numerical prediction of the moments of the regular NDF case as a function of the logarithm of the grid size – Zeta scheme.

Table 1: Numerical values of the errors reported in Fig. 2 and corresponding order of the reconstruction of each moment.

Cells	**h**	**e_0**	**Order m_0**	**e_1**	**Order m_1**	**e_2**	**Order m_2**	**e_3**	**Order m_3**
50	0.02	0.0423		0.0555		0.0665		0.0759	
100	0.01	0.0132	1.6851	0.0173	1.6792	0.0210	1.6625	0.0244	1.6359
500	0.002	0.0007	1.8318	0.0009	1.8512	0.0011	1.8509	0.0012	1.8481
1000	0.001	0.0002	2.0240	0.0002	1.9997	0.0003	2.0047	0.0003	2.0055
Cells	**h**	**e_4**	**Order m_4**	**e_5**	**Order m_5**	**e_6**	**Order m_6**	**e_7**	**Order m_7**
50	0.02	0.0849		0.0934		0.1013		0.1087	
100	0.01	0.0276	1.6187	0.0307	1.6048	0.0337	1.5862	0.0368	1.5614
500	0.002	0.0014	1.8426	0.0016	1.8331	0.0018	1.8235	0.0020	1.8160
1000	0.001	0.0004	2.0023	0.0004	2.0006	0.0004	1.9994	0.0005	1.9851

5.2. One-dimensional advection of moments at the boundary of the moment space

The second test case considers moments of a bimodal NDF, which can be at boundary of the moment space. The initial condition is defined with the help of three weight functions, whose value depends on the position in the computational domain x as indicated in Eqs. (5.7) - (5.9).

$$w_1(x) = \begin{cases} 16x^2(1-x)^2 & 0 \le x \le 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.7}$$

$$w_2(x) = \begin{cases} \frac{256}{81}(4x-1)^2(1-x)^2 & \frac{1}{4} \le x \le 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.8}$$

$$w_3(x) = \begin{cases} 9(3x-1)^2(1-x)^2 & \frac{1}{3} \le x \le 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.9}$$

Two other auxiliary functions are introduced as follows:

$$\lambda_2 = \begin{cases} \lambda_{2,\min} = 2 \cdot 10^{-2} & 0 \le x \le \frac{1}{3} \\ \lambda_{2,\min}(2-3x)^2(6x-1) + \lambda_{\max}(3x-1)^2(5-6x) & \frac{1}{3} < x \le \frac{2}{3}, \\ \lambda_{2,\max} = 0.7 & \frac{2}{3} < x \le 1 \end{cases} \tag{5.10}$$

$$\kappa_2 = \begin{cases} \kappa_{2,\min} = 3 & 0 \le x \le \frac{1}{3} \\ \kappa_{2,\min}(2-3x)^2(6x-1) + \lambda_{\max}(3x-1)^2(5-6x) & \frac{1}{3} < x \le \frac{2}{3}, \\ \kappa_{2,\max} = 10 & \frac{2}{3} < x \le 1 \end{cases} \tag{5.11}$$

With these definitions, the value of the initial moments is found according to the expression:

$$m_k(x) = w_1(x)x_{i,1}^k + w_2(x)\left(2x_{i,1}\right)^k + w_3(x)\lambda^k(x)\Gamma\left(1+\frac{k}{\kappa_2}\right), \tag{5.12}$$

where $x_{i,1} = 0.02$. The same cases considered in Sec. 5.1 were repeated with the initial condition defined by Eq. (5.12).

The comparison of the numerical solution and the exact solution is reported in Fig. 3. Similarly, to what is observed in the case of the moments of a regular NDF, also in the case of the moments of the bimodal distribution, the grid resolutions of 50 and 100 cells do not allow to exactly preserve the moments, which are satisfactorily preserved on the grid with 500 and 1000 cells. However, the ζ scheme ensures the realizability of the transported moment vector.

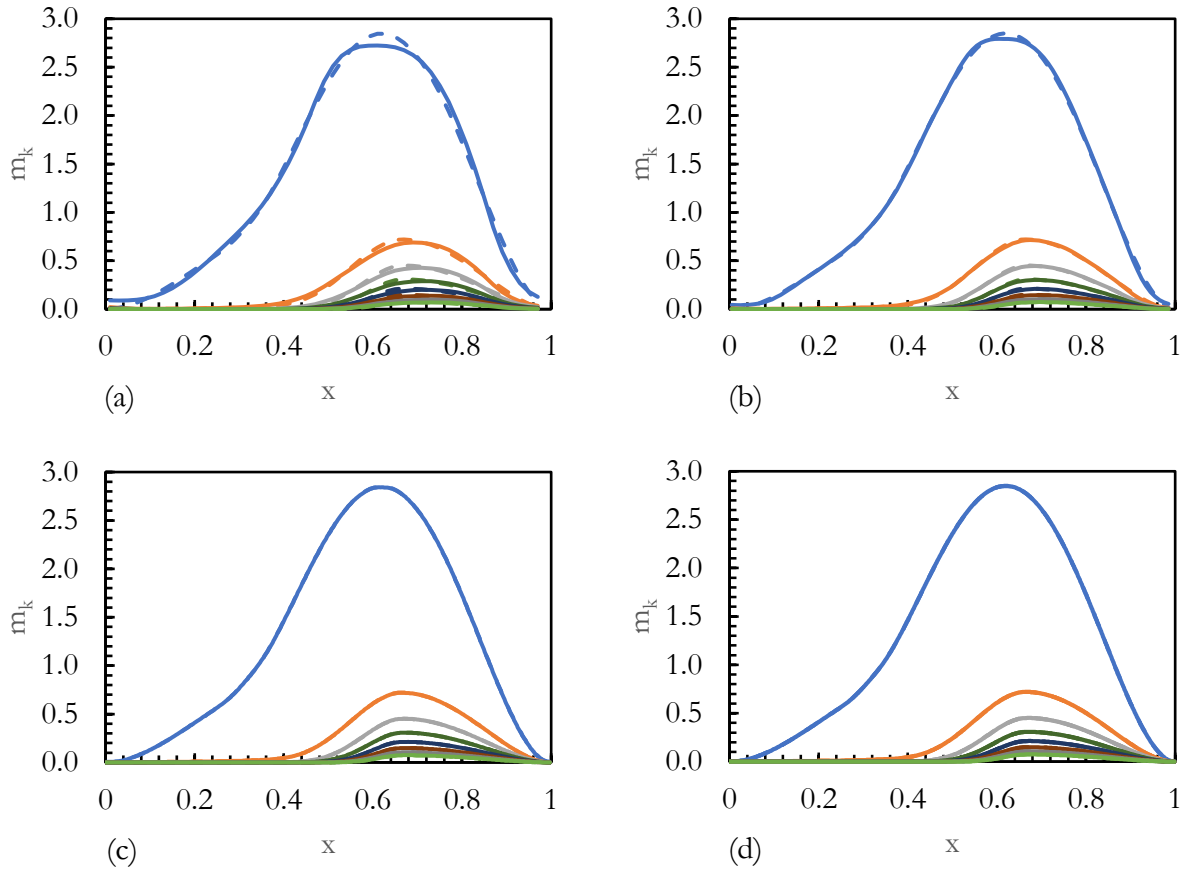


Fig. 3: Moments of the bimodal NDF. Analytical solution (dashed lines) and numerical prediction (solid lines) with (a) 50, (b) 100, (c) 500 and (d) 1000 cells – Zeta scheme.

The behavior of the error, defined by Eq. (5.7), in the case of the bimodal NDF is reported in Fig. 4 and the corresponding numerical values are shown in Table 2.

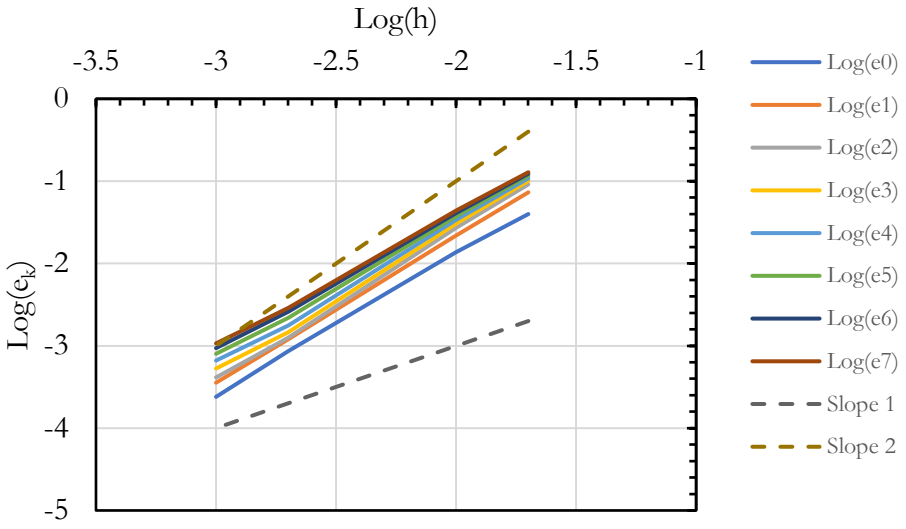


Fig. 4: Logarithm of the error in the numerical prediction of the moments of the bimodal NDF case as a function of the logarithm of the grid size – Zeta scheme.

Table 2: Numerical values of the errors reported in Fig. 4 and corresponding order of the reconstruction of each moment.

Cells	**h**	**e_0**	**Order m_0**	**e_1**	**Order m_1**	**e_2**	**Order m_2**	**e_3**	**Order m_3**
50	0.02	0.0399		0.0731		0.0914		0.1007	
100	0.01	0.0137	1.5393	0.0218	1.7427	0.0271	1.7557	0.0304	1.7284
500	0.002	0.0009	1.7199	0.0012	1.7999	0.0013	1.9073	0.0015	1.8837
1000	0.001	0.0002	1.8448	0.0004	1.7585	0.0004	1.6072	0.0005	1.4687
Cells	**h**	**e_4**	**Order m_4**	**e_5**	**Order m_5**	**e_6**	**Order m_6**	**e_7**	**Order m_7**
50	0.02	0.1083		0.1165		0.1231		0.1284	
100	0.01	0.0339	1.6751	0.0374	1.6374	0.0408	1.5929	0.0439	1.5488
500	0.002	0.0018	1.8360	0.0022	1.7668	0.0026	1.7121	0.0029	1.6885
1000	0.001	0.0007	1.4168	0.0008	1.4466	0.0009	1.4692	0.0011	1.4385

The chart shows that nearly second-order accuracy is achieved for low-order moments. However, the intervention of the additional limiter on the ζ quantities impacts the accuracy of high-order moments. In any case, the order of accuracy of the ζ scheme remains higher than one in all the cases considered in this test, which represents an extreme condition, with moments close to the boundary of the moment space.

5.3. Two-dimensional advection of moments of a regular NDF

The implementation of the second-order moment-preserving ζ scheme was tested in a two-dimensional case considering a computational domain defined as $[0, 0.5] \times [0, 0.5]$, with a steady Taylor-Green velocity field defined as:

$$\mathbf{U} = (\sin 2\pi x \cos 2\pi y\,, -\cos 2\pi x \sin 2\pi y) \tag{5.13}$$

The velocity on cell faces, needed to evaluate the advection flux, was computed by linearly interpolating the velocity field defined in Eq. (5.13) onto the cell faces. The initial value of the moments was defined as follows:

$$m_k(z) = \frac{m_{k-1}(\alpha_{2D}(z) + k - 1)}{\alpha_{2D}(z) + \beta_{2D}(z) + k - 1}\gamma_{2D}(z), \tag{5.14}$$

where

$$z = 8\sqrt{\left(x - \frac{1}{8}\right)^2 + \left(y - \frac{1}{8}\right)^2}, \tag{5.15}$$

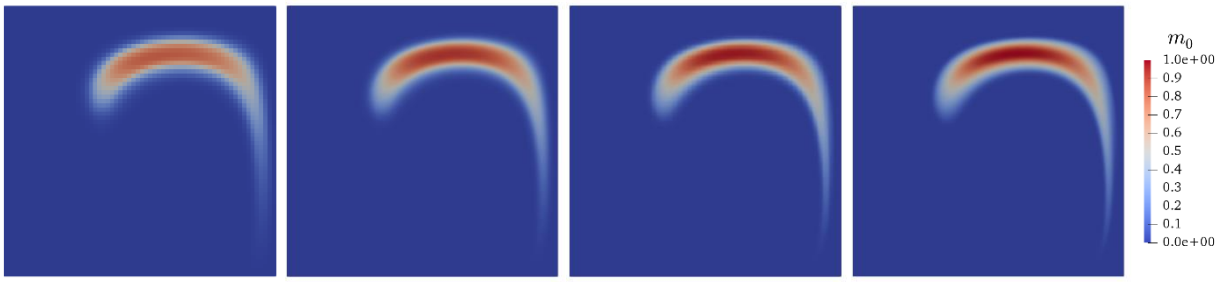
$$\alpha_{2D}(z) = 3.5 + 1.5 \sin 2\pi(1 - z), \tag{5.16}$$

$$\beta_{2D}(z) = 3.5 - 1.5 \cos 2\pi(1 - z), \tag{5.17}$$

$$\gamma_{2D}(z) = \frac{1}{2}\left\{1 + \tanh\left[\tan \pi\left(\frac{1}{2} - z\right)\right]\right\}. \tag{5.18}$$

5.3.1. Calculations on quadrilateral grids

The case illustrated in this section was simulated considering five grid resolutions with increasing level of refinement, respectively with 64, 128, 256, 512 in each spatial direction, with a maximum CFL number of 0.2. Results showing the predicted zero-order moment at $t = 0.8$ are shown in Fig. 5.



(a) (b) (c) (d)

Fig. 5: Zero-order moment of the regular NDF used in the two-dimensional test case on quadrilateral grids at t = 0.8. (a) 64^2 cells, (b) 128^2 cells, (c) 256^2 cells, (d) 512^2 cells. Zeta scheme.

It is apparent from Fig. 5 that the numerical solution obtained on the two finest grid resolutions with

256^2 and 512^2 cells, do not show significant differences. However, to provide a quantitative measure of the accuracy of the zeta scheme, and to validate its implementation into OpenFOAM in comparison

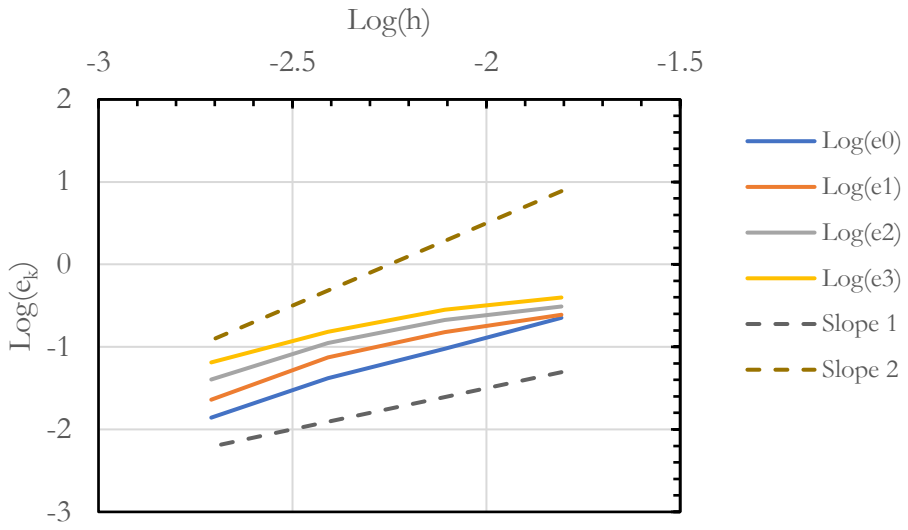


Fig. 6: Logarithm of the error in the two-dimensional numerical tests on hexahedral grid with regular NDF as a function of the logarithm of the grid size.

to [1], the order of accuracy of the ζ scheme is quantified numerically by taking the solution obtained on a uniform quadrilateral grid with 1024^2 cells as reference. The graph of the error is reported in Fig. 6 (numerical values in Table 3), which shows that the slope of the error curves tends to 2 for higher grid resolutions, with a reduction of the formal of order of accuracy for moments of higher order.

Table 3: Numerical values of the errors reported in Fig. 6 and corresponding order of the reconstruction of each moment.

Cells	h	e_0	Order m_0	e_1	Order m_1	e_2	Order m_2	e_3	Order m_3
64	0.0156	0.2253		0.2451		0.3089		0.3970	
128	0.0078	0.0946	1.2527	0.1506	0.7025	0.2115	0.5466	0.2823	0.4921
256	0.0039	0.0419	1.1748	0.0746	1.0130	0.1110	0.9299	0.1531	0.8829
512	0.0020	0.0139	1.5931	0.0229	1.7030	0.0403	1.4637	0.0649	1.2384

The time evolution of the first-order moment of the NDF and of the value of the corresponding limiter $\lambda_{0,c}$ are reported in Fig. 7.

It is observed that $\lambda_{0,c} = 1$ in most of the solution domain. The regions where a complete limitation is applied ($\lambda_{0,c} = 0$), reducing the local accuracy to first order, is often where the solution presents very small positive values of the moment. The additional limitation through $\lambda_{0,c}$ preserves the realizability of the moment vector, while showing a limited impact on the accuracy of the solution, as

previously illustrated by the numerical study of the order of the scheme (Fig. 6).

5.3.2. Calculations on triangular grids

The two-dimensional case presented in this section was also considered to establish the accuracy of the ζ advection scheme on triangular grids. Four grids were considered in these simulations, whose properties are summarized in Table 4. The value of the grid spacing h was calculated according to [72], as

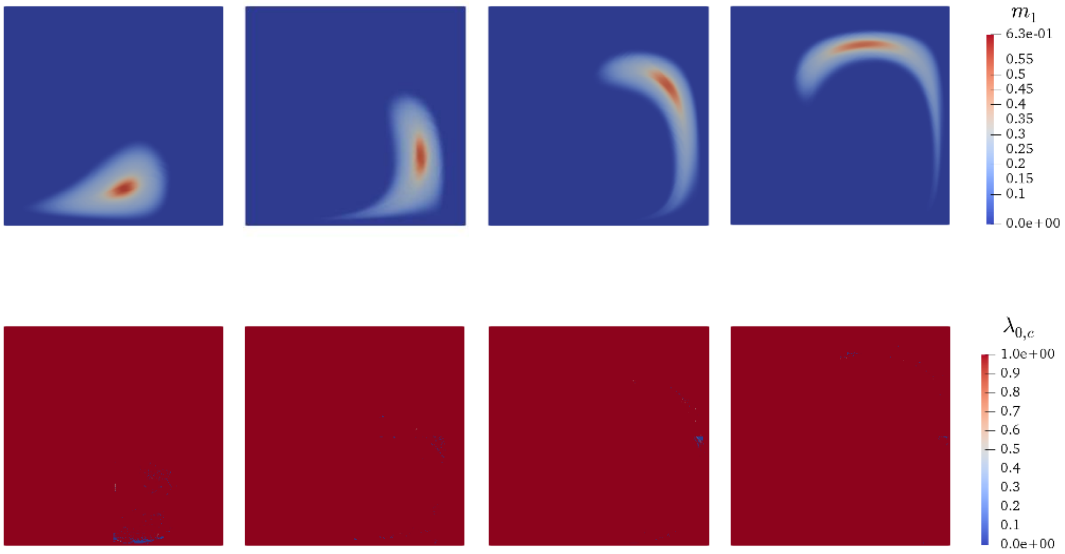


Fig. 7: Time evolution of the first-order moment of the regular NDF considered in the test case on quadrilateral grid (resolution shown: 1024^2): (a) t = 0.2 s; (b) t = 0.4 s; (c) t = 0.6 s; (d) t = 0.8 s.

$$h = \sqrt{\frac{1}{N_{cells}} \sum_i (\Delta A_i)}, \tag{5.19}$$

where ΔA_i is the area of each cell polygon.

Table 4: Properties of the unstructured grids considered in the numerical test cases.

Grid	N. of cells	Domain area	h
A	3938	0.25	$7.97 \cdot 10^{-3}$
B	15894	0.25	$3.97 \cdot 10^{-3}$
C	64688	0.25	$1.97 \cdot 10^{-3}$
D	255596	0.25	$9.89 \cdot 10^{-4}$

The contour plots of the zero-order moment obtained with the ζ scheme on the triangular grids at $t = 0.8$ s, are reported in Fig. 8, which shows the results are visually identical to those obtained on quadrilateral grids and reported in Fig. 5.

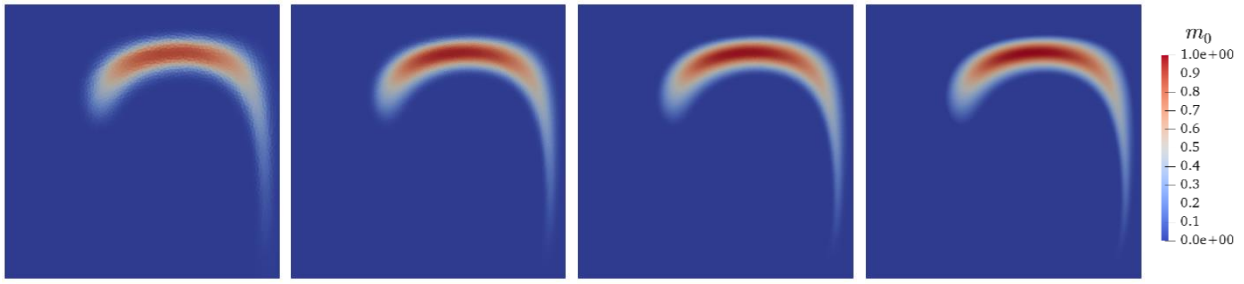


Fig. 8: Zero-order moment of the regular NDF used in the two-dimensional test case on triangular grids at t = 0.8. (a) Grid A, (b) Grid B, (c) Grid C, (d) Grid D. Zeta scheme.

The behavior of the formal order of accuracy of the ζ simplified scheme on triangular grids is shown in Fig. 9 (numerical values in Table 5), which shows comparable trends for all the moments to those observed on quadrilateral grids (Fig. 6), with the second-order accuracy recovered on the finest grids, for the lower-order moments.

5.3.3. Comparison of first-order scheme and ζ simplified scheme

Results obtained with the first-order scheme on a quadrilateral grid are shown in Fig. 10 to highlight

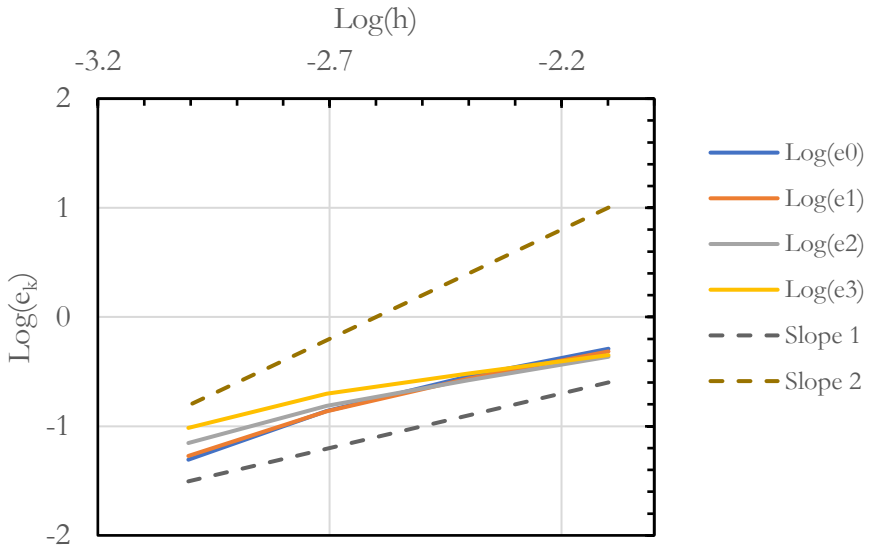
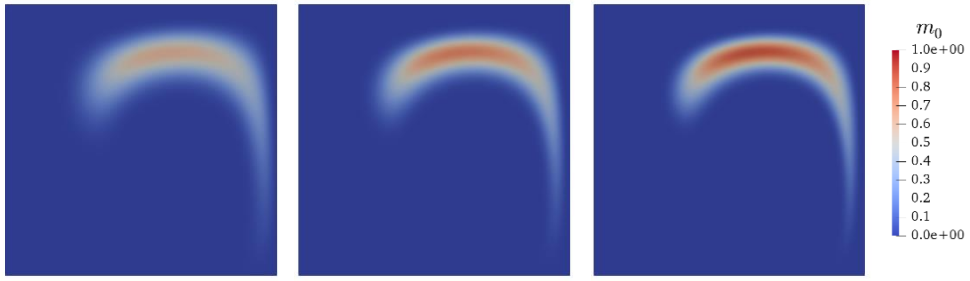


Fig. 9: Logarithm of the error in the two-dimensional numerical tests on triangular grids with regular NDF as a function of the logarithm of the grid size.

the significantly higher dissipation introduced by the first-order scheme, typically used for moment transport to avoid compromising the realizability of the advected moment vector.

Table 5: Numerical values of the errors reported in Fig. 9 and corresponding order of the reconstruction of each moment.

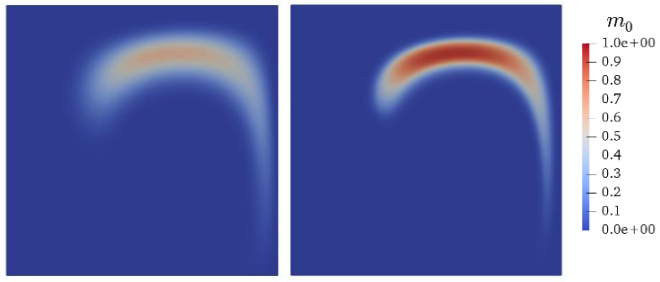
h	e_0	Order m_0	e_1	Order m_1	e_2	Order m_2	e_3	Order m_3
0.0080	0.5123		0.4844		0.4327		0.4483	
0.0040	0.2858	0.8365	0.2720	0.8270	0.2640	0.7084	0.3034	0.5593
0.0020	0.1383	1.0343	0.1367	0.9805	0.1540	0.7681	0.1983	0.6061
0.0010	0.0494	1.4991	0.0534	1.3674	0.0702	1.1437	0.0965	1.0480



(a) (b) (c)

Fig. 10: Zero-order moment field at $t = 0.8$ on a quadrilateral grid with (a) 128^2, (b) 256^2 and (c) 512^2 cells – First-order scheme.

The difference between the first-order scheme and the second-order zeta scheme is further highlighted in Fig. 11, where the zero-order moment obtained at $t = 0.8$ s on a grid with 128^2 cells with the two methods is compared. The results obtained with the ζ simplified scheme clearly shows the reduced dissipation compared to the results obtained with the first-order scheme.



(a) (b)

Fig. 11: Comparison of zero-order moment field obtained on a 128^2 quadrilateral grid at $t = 0.8$, with (a) first-order advection scheme and (b) zeta scheme.

Similarly to what done for the quadrilateral case, the time evolution of the first-order moment of the NDF is shown in Fig. 12, where the fields of m_1 at different times appear to be identical to those shown in Fig. 7 for the quadrilateral case. However, a more careful examination of the behavior of the moment limiter $\lambda_{0,c}$ shows that the region of the computational domain where the limitation is applied is larger for the case with triangular grid compared to the quadrilateral case. Several cells on the front of the region where $m_1 \neq 0$ are affected by the limitation, while, in the quadrilateral case, a smaller number of cells with $\lambda_{0,c} \neq 1$ is observed. The last snapshot (Fig. 12(d)) also shows a small number of cells where $\lambda_{0,c} = 0.5$.

5.4. Two-dimensional advection of moments of a bimodal NDF

The ζ simplified scheme is applied in this section to the case transport of moments of a bi-modal NDF in two dimensions. Only one triangular grid (Grid B in Table 4) is considered.

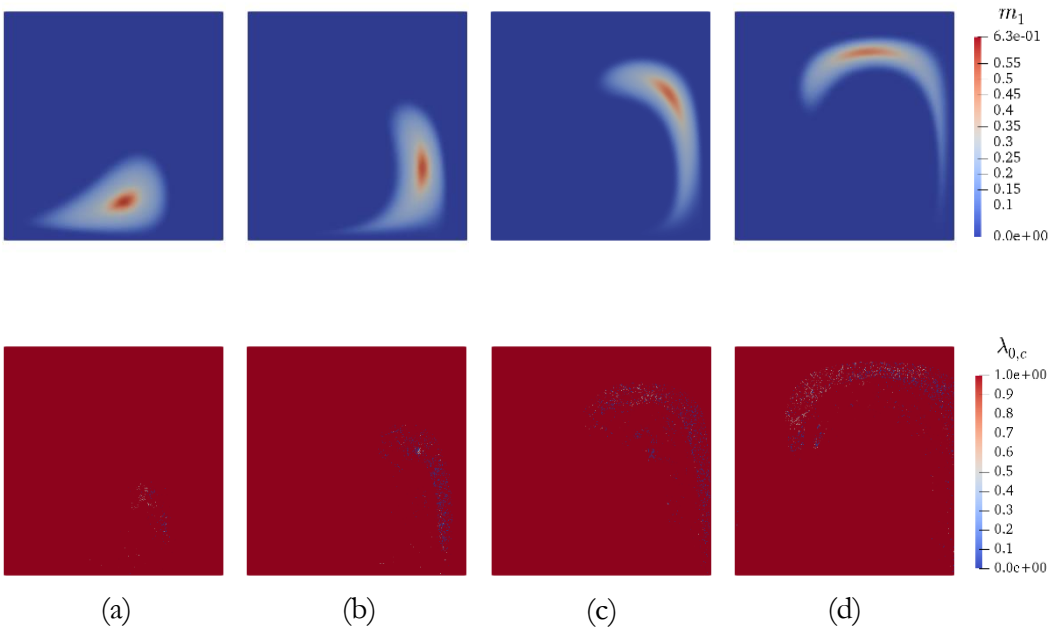


Fig. 12: Time evolution of the first-order moment of the regular NDF in the test case on tetrahedral grid (Grid B) and the corresponding limiter $\lambda_{0,c}$ used in the additional limitation of the ζ simplified scheme. (a) t = 0.2 s; (b) t = 0.4 s; (c) t = 0.6 s; (d) t = 0.8 s.

The initial conditions for the moments are obtained using Eq. (5.12), where the x coordinate is replaced by

$$z = 8\sqrt{\left(x-\frac{1}{8}\right)^2+\left(y-\frac{1}{8}\right)^2}. \tag{5.20}$$

Fig. 13 shows the time evolution of m_0 and m_1 together with the limiter $\lambda_{0,c}$ used in the additional

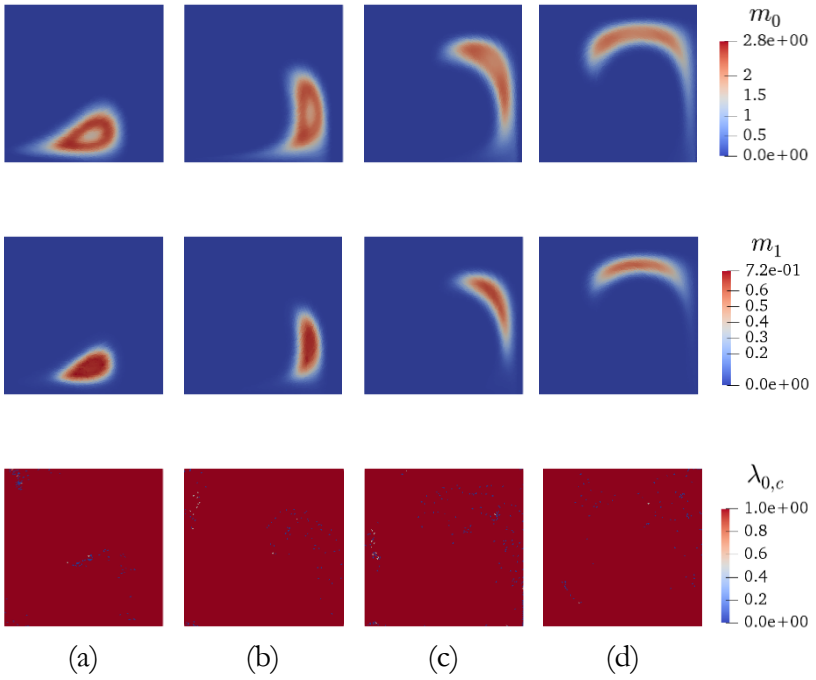


Fig. 13: Time evolution of the first-order moment of the bi-modal NDF in the test case on tetrahedral grid (Grid B) and the corresponding limiter $\lambda_{0,c}$ used in the additional limitation of the ζ simplified scheme. (a) $t = 0.2$ s; (b) $t = 0.4$ s; (c) $t = 0.6$ s; (d) $t = 0.8$ s.

limitation in the ζ simplified scheme for the reconstruction of m_1. The figure shows that to preserve the moment realizability, the additional limitation intervenes in a modest number of cells, as expected because of the moment vectors near the boundary of the moment space.

Finally, it should be noted that the limitation applied to the second-order moment happens in some additional cells, compared to what shown in Fig. 13. This is shown, for example, in Fig. 14 (b), which shows the second-order moment for the case of the bimodal NDF and the corresponding value of the limiter $\lambda_{1,c}$ used in the ζ simplified scheme. The limiter is similarly applied to the third-order moment.

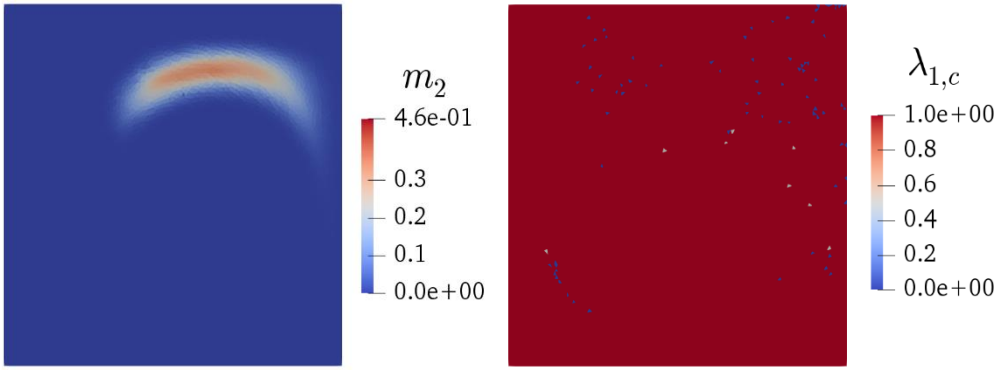


Fig. 14: (a) Secona-order moment of the bi-modal NDF in the test case on tetrahedral grid (Grid B) at $t = 0.8$, and, (b), the corresponding field of the limiter $\lambda_{1,c}$ used in the additional limitation of the ζ simplified scheme.

6. Conclusions

The second-order realizable ζ simplified scheme proposed in Laurent and Nguyen [1] was extended to unstructured grids and implemented into the computational framework OpenQBMM [73], an extension for OpenFOAM [63] that implements quadrature-based moment methods. The implementation was verified by considering first two one-dimensional cases, the first involving moments of a regular NDF and the second where moments of a bimodal NDF are transported.

The proposed implementation of the scheme into OpenQBMM was able to reproduce the results for the same cases reported in Laurent and Nguyen [1], showing convergence to second-order accuracy with grid refinement. The scheme was further verified on a two-dimensional case with uniform hexahedral discretization where the first four moments of a regular NDF are transported. Also in this case, results from Laurent and Nguyen [1] were satisfactorily reproduced. Finally, the same two-dimensional case considered for verification on hexahedral grids was used to establish the accuracy of the scheme on triangular grids. The ζ simplified scheme was able to reproduce the results and the behavior of the error affecting the solution observed on hexahedral grids also on triangular grids.

Future work will consider the development of realizable numerical schemes for multivariate problems, where closure is provided by the conditional quadrature method of moments [74].

7. Source code and test cases

The source code of the OpenQBMM framework [73] used to perform the calculations described in

this article is available in [73,75], where also the test cases presented in the articles are available. The OpenFOAM class implementing the ζ scheme on unstructured grids described in the manuscript is reported in Appedix A.

8. Acknowledgements

The authors would like to gratefully acknowledge the support of the US National Science Foundation under the SI2–SSE award NSF–ACI 1440443 and the support of the French National Research Agency (ANR) under grant ANR-13-TDMO-02 ASMAPE for the ASMAPE project.

A.P. would like to acknowledge the support of the *Jean d'Alembert* fellowship program Idex Paris-Saclay, which supported his stay at Laboratoire EM2C, CNRS, CentraleSupélec, Université Paris-Saclay.

Appedix A

The C++ source code of the main class that implements the ζ scheme in OpenQBMM is reported in this appendix. This code is contained in OpenQBMM 5.0.0 for OpenFOAM 7 [75].

Header file: zetaUnivariateAdvection.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2014-2018 Alberto Passalacqua
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is derivative work of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
```

```
    Foam::zeta

Description
    Second-order univariate moment advection with zeta kinetic scheme.

SourceFiles
    zetaUnivariateAdvection.C

\*---------------------------------------------------------------------------*/

#ifndef zetaUnivariateAdvection_H
#define zetaUnivariateAdvection_H

#include "univariateMomentAdvection.H"


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{
namespace univariateAdvection
{

/*---------------------------------------------------------------------------*\
                           Class zeta Declaration
\*---------------------------------------------------------------------------*/

class zeta
:
    public univariateMomentAdvection
{
    // Private data

        //- Reference to zero-order moment field
        const volScalarField& m0_;

        //- Reconstructed m0 (owner)
        surfaceScalarField m0Own_;

        //- Reconstructed m0 (neighbour)
        surfaceScalarField m0Nei_;

        //- Number of zeta_k values
        label nZetas_;

        //- List of fields of zeta_k (n fields for n + 1 moments)
        PtrList<volScalarField> zetas_;

        //- List of interpolated zeta_k values (neighbour)
        PtrList<surfaceScalarField> zetasNei_;

        //- List of interpolated nodes (owner)
        PtrList<surfaceScalarField> zetasOwn_;

        //- List of interpolated zeta_k values (neighbour)
        PtrList<surfaceScalarField> zetasUpwindNei_;
```

```
        //- List of interpolated nodes (owner)
        PtrList<surfaceScalarField> zetasUpwindOwn_;

        //- List of interpolated zeta_k values (neighbour)
        PtrList<surfaceScalarField> zetasCorrNei_;

        //- List of interpolated nodes (owner)
        PtrList<surfaceScalarField> zetasCorrOwn_;

        //- List of interpolated moments (neighbour)
        PtrList<surfaceScalarField> momentsNei_;

        //- List of interpolated moments (owner)
        PtrList<surfaceScalarField> momentsOwn_;

        //- Field to store the number of faces with outgoing flux per cell
        mutable labelField nFacesOutgoingFlux_;

        //- Field to store the number of realizable moments in each cell
        mutable labelField nRealizableMoments_;

        //- Field to store the number of realizable m* in each cell
        mutable labelField nRealizableMomentsStar_;

        //- List of limiters for zeta_k
        PtrList<surfaceScalarField> limiters_;

        //- List of cell limiters
        PtrList<volScalarField> cellLimiters_;

        //- Face velocity
        const surfaceScalarField& phi_;


    // Private member functions

        //- Compute n values of zeta_k from n + 1 moments
        void computeZetaFields();

        //- Updates reconstructed moments from the values of zeta
        void updateMomentFieldsFromZetas
        (
            const surfaceScalarField& m0f,
            const PtrList<surfaceScalarField>& zetaf,
            PtrList<surfaceScalarField>& mf
        );

        //- Compute n + 1 moments from n values of zeta_k
        void zetaToMoments
        (
            const scalarList& zetaf,
            scalarList& mf,
            scalar m0 = 1.0
        );


protected:
```

```
    // Protected member functions

        //- Calculates the number of cells with outgoing flux
        void countFacesWithOutgoingFlux();

        //- Computes the limiter used for additional limitation
        void limiter();

        //- Reconstructs zeta_k
        void interpolateFields();

        //- Applies additional limitation to zeta_k, if needed
        void limitZetas();


public:

    //- Runtime type information
    TypeName("zeta");


    // Constructors

        //- Construct from univariateMomentSet
        zeta
        (
            const dictionary& dict,
            const scalarQuadratureApproximation& quadrature,
            const surfaceScalarField& phi,
            const word& support
        );


    //- Destructor
    virtual ~zeta();


    // Public member functions

        //- Return the maximum Courant number ensuring moment realizability
        virtual scalar realizableCo() const;

        //- Update moment advection
        virtual void update();
};


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//

} // End namespace univariateAdvection
} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//

#endif

// ************************************************************************
//
```

Class implementation: zetaUnivariateAdvection.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2014-2018 Alberto Passalacqua
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is derivative work of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "zetaUnivariateAdvection.H"
#include "upwind.H"
#include "addToRunTimeSelectionTable.H"

// * * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

namespace Foam
{
namespace univariateAdvection
{
    defineTypeNameAndDebug(zeta, 0);

    addToRunTimeSelectionTable
    (
        univariateMomentAdvection,
        zeta,
        dictionary
    );
```

```
}
}

// * * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

Foam::univariateAdvection::zeta::zeta
(
    const dictionary& dict,
    const scalarQuadratureApproximation& quadrature,
    const surfaceScalarField& phi,
    const word& support
)
:
    univariateMomentAdvection(dict, quadrature, phi, support),
    m0_(moments_(0)),
    m0Own_
    (
        IOobject::groupName("m0OwnZeta", name_),
        fvc::interpolate(m0_, own_, "reconstruct(m0)")
    ),
    m0Nei_
    (
        IOobject::groupName("m0NeiZeta", name_),
        fvc::interpolate(m0_, nei_, "reconstruct(m0)")
    ),
    nZetas_(nMoments_ - 1),
    zetas_(nZetas_),
    zetasNei_(nZetas_),
    zetasOwn_(nZetas_),
    zetasUpwindNei_(nZetas_),
    zetasUpwindOwn_(nZetas_),
    zetasCorrNei_(nZetas_),
    zetasCorrOwn_(nZetas_),
    momentsNei_(nMoments_),
    momentsOwn_(nMoments_),
    nFacesOutgoingFlux_(m0_.size(), 0),
    nRealizableMoments_(m0_.size(), 0),
    nRealizableMomentsStar_(m0_.size(), 0),
    limiters_(nZetas_),
    cellLimiters_(nZetas_),
    phi_(phi)
{
    if (quadrature.momentOrders()[0].size() > 1)
    {
        FatalErrorInFunction
            << "Zeta advection scheme can only be used for" << nl
            << "univariate distributions."
            << abort(FatalError);
    }
    // Populating zeta_k fields and interpolated zeta_k fields
    forAll(zetas_, zetai)
    {
        zetas_.set
        (
            zetai,
            new volScalarField
            (
```

```
            IOobject
            (
                fieldName("zeta", {zetai}),
                phi.mesh().time().timeName(),
                phi.mesh(),
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            phi.mesh(),
            dimensionedScalar("zero", dimless, 0.0)
        )
    );

    zetasNei_.set
    (
        zetai,
        new surfaceScalarField
        (
            IOobject
            (
                fieldName("zetaNei", {zetai}),
                phi.mesh().time().timeName(),
                phi.mesh(),
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            phi.mesh(),
            dimensionedScalar("zero", dimless, 0.0)
        )
    );

    zetasOwn_.set
    (
        zetai,
        new surfaceScalarField
        (
            IOobject
            (
                fieldName("zetaOwn", {zetai}),
                phi.mesh().time().timeName(),
                phi.mesh(),
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            phi.mesh(),
            dimensionedScalar("zero", dimless, 0.0)
        )
    );

    zetasUpwindNei_.set
    (
        zetai,
        new surfaceScalarField
        (
            IOobject
            (
                fieldName("zetaUpwindNei", {zetai}),
```

```
                phi.mesh().time().timeName(),
                phi.mesh(),
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            phi.mesh(),
            dimensionedScalar("zero", dimless, 0.0)
        )
    );

    zetasUpwindOwn_.set
    (
        zetai,
        new surfaceScalarField
        (
            IOobject
            (
                fieldName("zetaUpwindOwn", {zetai}),
                phi.mesh().time().timeName(),
                phi.mesh(),
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            phi.mesh(),
            dimensionedScalar("zero", dimless, 0.0)
        )
    );

    zetasCorrNei_.set
    (
        zetai,
        new surfaceScalarField
        (
            IOobject
            (
                fieldName("zetaCorrNei", {zetai}),
                phi.mesh().time().timeName(),
                phi.mesh(),
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            phi.mesh(),
            dimensionedScalar("zero", dimless, 0.0)
        )
    );

    zetasCorrOwn_.set
    (
        zetai,
        new surfaceScalarField
        (
            IOobject
            (
                fieldName("zetaCorrOwn", {zetai}),
                phi.mesh().time().timeName(),
                phi.mesh(),
                IOobject::NO_READ,
```

```
                IOobject::NO_WRITE
            ),
            phi.mesh(),
            dimensionedScalar("zero", dimless, 0.0)
        )
    );

    limiters_.set
    (
        zetai,
        new surfaceScalarField
        (
            IOobject
            (
                fieldName("zetaLimiter", {zetai}),
                phi.mesh().time().timeName(),
                phi.mesh(),
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            phi.mesh(),
            dimensionedScalar("zero", dimless, 1.0)
        )
    );

    cellLimiters_.set
    (
        zetai,
        new volScalarField
        (
            IOobject
            (
                fieldName("zetaCellLimiter", {zetai}),
                phi.mesh().time().timeName(),
                phi.mesh(),
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            phi.mesh(),
            dimensionedScalar("zero", dimless, 1.0)
        )
    );
}

// Setting face values of moments
forAll(momentsNei_, momenti)
{
    momentsNei_.set
    (
        momenti,
        new surfaceScalarField
        (
            fieldName("momentNeiZeta", {momenti}),
            fvc::interpolate(moments_(momenti))
        )
    );
```

```
        momentsOwn_.set
        (
            momenti,
            new surfaceScalarField
            (
                fieldName("momentOwnZeta", {momenti}),
                fvc::interpolate(moments_(momenti))
            )
        );
    }
}


// * * * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * //

Foam::univariateAdvection::zeta::~zeta()
{}


// * * * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //

void Foam::univariateAdvection::zeta::interpolateFields()
{
    IStringStream m0OwnLimiter("Minmod");
    IStringStream zetaOwnLimiter("Minmod");

    tmp<surfaceInterpolationScheme<scalar>> m0OwnScheme
    (
        fvc::scheme<scalar>(own_, m0OwnLimiter)
    );
    tmp<surfaceInterpolationScheme<scalar>> zetaOwnScheme
    (
        fvc::scheme<scalar>(own_, zetaOwnLimiter)
    );

    IStringStream m0NeiLimiter("Minmod");
    IStringStream zetaNeiLimiter("Minmod");

    tmp<surfaceInterpolationScheme<scalar>> m0NeiScheme
    (
        fvc::scheme<scalar>(nei_, m0NeiLimiter)
    );
    tmp<surfaceInterpolationScheme<scalar>> zetaNeiScheme
    (
        fvc::scheme<scalar>(nei_, zetaNeiLimiter)
    );

    m0Own_ = m0OwnScheme().interpolate(moments_(0));
    m0Nei_ = m0NeiScheme().interpolate(moments_(0));

    forAll(zetas_, zetai)
    {
        zetasNei_[zetai] = zetaNeiScheme().interpolate(zetas_[zetai]);
        zetasOwn_[zetai] = zetaOwnScheme().interpolate(zetas_[zetai]);

        zetasUpwindNei_[zetai] =
            upwind<scalar>(zetas_[zetai].mesh(), nei_).flux(zetas_[zetai]);
```

```
        zetasUpwindOwn_[zetai] =
            upwind<scalar>(zetas_[zetai].mesh(), own_).flux(zetas_[zetai]);

        zetasCorrNei_[zetai] = zetasNei_[zetai] - zetasUpwindNei_[zetai];
        zetasCorrOwn_[zetai] = zetasOwn_[zetai] - zetasUpwindOwn_[zetai];
    }
}

void Foam::univariateAdvection::zeta::zetaToMoments
(
    const scalarList& zetaf,
    scalarList& mf,
    scalar m0
)
{
    scalarSquareMatrix S(nMoments_, 0.0);

    for (label i = 0; i < nZetas_; i++)
    {
        S[0][i] = 1.0;
    }

    for (label i = 1; i < nZetas_; i++)
    {
        for (label j = i; j < nZetas_; j++)
        {
            S[i][j] = S[i][j - 1] + zetaf[j - i]*S[i - 1][j];
        }
    }

    scalarList prod(nMoments_, 1.0);

    prod[1] = zetaf[0];

    for (label i = 2; i < nZetas_; i++)
    {
        prod[i] = prod[i - 1]*zetaf[i - 1];
    }

    // Resetting moments to zero
    mf = 0.0;

    // Computing moments
    mf[0] = 1.0;
    mf[1] = zetaf[0];

    for (label i = 2; i < nMoments_; i++)
    {
        for (label j = 0; j <= i/2; j++)
        {
            mf[i] += prod[i - 2*j]*sqr(S[j][i - j]);
        }
    }

    if (m0 != 1.0)
    {
```

```
        for (label mi = 0; mi < nMoments_; mi++)
        {
            mf[mi] *= m0;
        }
    }
}

void Foam::univariateAdvection::zeta::computeZetaFields()
{
    // Cell-center values
    forAll(m0_, celli)
    {
        if (m0_[celli] >= SMALL)
        {
            univariateMomentSet m(nMoments_, support_);

            for (label mi = 0; mi < nMoments_; mi++)
            {
                m[mi] = moments_(mi)[celli];
            }

            nRealizableMoments_[celli] = m.nRealizableMoments();

            scalarList zetas(m.zetas());

            for (label zetai = 0; zetai < nZetas_; zetai++)
            {
                zetas_[zetai][celli] = zetas[zetai];

                if (zetas_[zetai][celli] > 1.0e-7)
                {
                    zetas_[zetai][celli] = zetas[zetai];
                }
                else
                {
                    zetas_[zetai][celli] = 0.0;
                }
            }
        }
    }

    // Boundary conditions
    const volScalarField::Boundary& bf = zetas_[0].boundaryField();

    forAll(bf, patchi)
    {
        const fvPatchScalarField& m0Patch = bf[patchi];

        forAll(m0Patch, facei)
        {
            if (m0_.boundaryField()[patchi][facei] >= SMALL)
            {
                univariateMomentSet m(nMoments_, support_);

                for (label mi = 0; mi < nMoments_; mi++)
                {
                    m[mi] = moments_(mi).boundaryField()[patchi][facei];
```

```
                }

                scalarList zetas(m.zetas());

                for (label zetai = 0; zetai < nZetas_; zetai++)
                {
                    volScalarField& zi = zetas_[zetai];
                    volScalarField::Boundary& ziBf = zi.boundaryFieldRef();
                    ziBf[patchi][facei] = zetas[zetai];
                }
            }
        }
    }
}

void Foam::univariateAdvection::zeta::countFacesWithOutgoingFlux()
{
    const fvMesh& mesh(phi_.mesh());
    const labelList& own = mesh.faceOwner();
    const labelList& nei = mesh.faceNeighbour();

    nFacesOutgoingFlux_ = 0;

    // Counting internal faces with outgoing flux
    for (label facei = 0; facei < mesh.nInternalFaces(); facei++)
    {
        if (phi_[facei] > 0)
        {
            nFacesOutgoingFlux_[own[facei]] += 1;
        }
        else if (phi_[facei] < 0)
        {
            nFacesOutgoingFlux_[nei[facei]] += 1;
        }
    }

    // Adding boundary faces with outgoing flux
    const surfaceScalarField::Boundary& phiBf = phi_.boundaryField();

    forAll(phiBf, patchi)
    {
        const fvsPatchScalarField& phiPf = phiBf[patchi];
        const labelList& pFaceCells = mesh.boundary()[patchi].faceCells();

        forAll(phiPf, pFacei)
        {
            if (phiPf[pFacei] > 0)
            {
                nFacesOutgoingFlux_[pFaceCells[pFacei]] += 1;
            }
        }
    }
}

void Foam::univariateAdvection::zeta::limitZetas()
{
    const labelUList& owner = phi_.mesh().owner();
```

```
    const labelUList& neighb = phi_.mesh().neighbour();
    const scalarField& phiIf = phi_;
    const surfaceScalarField::Boundary& phiBf = phi_.boundaryField();
    const label nInternalFaces = phi_.mesh().nInternalFaces();

    countFacesWithOutgoingFlux();

    forAll(cellLimiters_, li)
    {
        forAll(cellLimiters_[0], celli)
        {
            cellLimiters_[li][celli] = 1.0;
        }
    }

    // First check on m* to identify cells in need of additional limitation
    scalarRectangularMatrix mPluses(nMoments_, m0_.size(), 0.0);

    // Find m+ (moments reconstructed on cell faces with outgoing flux)
    for (label facei = 0; facei < nInternalFaces; facei++)
    {
        const label own = owner[facei];
        const label nei = neighb[facei];

        if (phi_[facei] > 0.0)
        {
            for (label mi = 0; mi < nMoments_; mi++)
            {
                mPluses[mi][own] += momentsOwn_[mi][facei];
            }
        }
        else
        {
            for (label mi = 0; mi < nMoments_; mi++)
            {
                mPluses[mi][nei] += momentsNei_[mi][facei];
            }
        }
    }

    // Adding boundary faces with outgoing flux
    forAll(phiBf, patchi)
    {
        const fvsPatchScalarField& phiPf = phiBf[patchi];

        const labelList& pFaceCells
            = phi_.mesh().boundary()[patchi].faceCells();

        forAll(phiPf, pFacei)
        {
            if (phiPf[pFacei] > 0)
            {
                for (label mi = 0; mi < nMoments_; mi++)
                {
                    mPluses[mi][pFaceCells[pFacei]] +=
                        momentsOwn_[mi][pFacei];
                }
```

```
            }
        }
    }

    // Compute m* and find how many moments are realizable
    univariateMomentSet mStar(nMoments_, support_);

    forAll(m0_, celli)
    {
        if (m0_[celli] > 0)
        {
            for (label mi = 0; mi < nMoments_; mi++)
            {
                mStar[mi]
                    = scalar(nFacesOutgoingFlux_[celli] + 1)
                        *moments_(mi)[celli] - mPluses[mi][celli];
            }

            nRealizableMomentsStar_[celli] = mStar.nRealizableMoments(false);
        }
        else
        {
            nRealizableMomentsStar_[celli] = nRealizableMoments_[celli];
        }
    }

    // In each cell where the the number of realizable m* is less than the
    // number of realizable m, limitation is attempted
    const cellList& mCells(phi_.mesh().cells());

    forAll(m0_, celli)
    {
        if (nRealizableMomentsStar_[celli] < nRealizableMoments_[celli])
        {
            const cell& mCell(mCells[celli]);

            // Start search for the zetas to limit
            for (label p = 0; p < nRealizableMoments_[celli] - 1; p++)
            {
                scalarList mPlus(nMoments_, 0.0);

                // Check if zeta_p needs limiting by evaluating m* with
                // zeta_k, k > p from constant reconstruction

                // Update mPlus for a face to update m*
                forAll(mCell, fi)
                {
                    const label facei = mCell[fi];

                    if (phi_.mesh().isInternalFace(facei))
                    {
                        if (phi_[facei] > 0)
                        {
                            scalarList zOwn(nZetas_, 0.0);
                            scalarList mOwn(nMoments_, 0.0);

                            for (label zi = 0; zi <= p; zi++)
```

```
                {
                    zOwn[zi] = zetasOwn_[zi][facei];
                }

                for (label zi = p + 1; zi < nZetas_; zi++)
                {
                    zOwn[zi] = zetasUpwindOwn_[zi][facei];
                }

                zetaToMoments(zOwn, mOwn, m0Own_[facei]);

                for (label mi = 0; mi < nMoments_; mi++)
                {
                    mPlus[mi] += mOwn[mi];
                }
            }
        }
    }

    // Compute m*
    for (label mi = 0; mi < nMoments_; mi++)
    {
        mStar[mi]
            = scalar(nFacesOutgoingFlux_[celli] + 1)
              *moments_(mi)[celli] - mPlus[mi];
    }

    nRealizableMomentsStar_[celli]
        = mStar.nRealizableMoments(false);

    // Check if zeta_p needs limitation
    if (nRealizableMomentsStar_[celli] <
        nRealizableMoments_[celli])
    {
        mPlus = 0;

        // Limit zeta_p
        forAll(mCell, fi)
        {
            const label facei = mCell[fi];

            if (phi_.mesh().isInternalFace(facei))
            {
                if (phi_[facei] > 0)
                {
                    zetasOwn_[p][facei]
                        = zetasUpwindOwn_[p][facei]
                        + 0.5*(zetasCorrOwn_[p][facei]);

                    cellLimiters_[p][celli] = 0.5;

                    scalarList zOwn(nZetas_);
                    scalarList mOwn(nMoments_, 0.0);

                    for (label zi = 0; zi < p; zi++)
                    {
                        zOwn[zi] = zetasOwn_[zi][facei];
```

```
                                }

                                zOwn[p] = zetasOwn_[p][facei];

                                for (label zi = p + 1; zi < nZetas_; zi++)
                                {
                                    zOwn[zi] = zetasUpwindOwn_[zi][facei];
                                }

                                zetaToMoments(zOwn, mOwn, m0Own_[facei]);

                                for (label mi = 0; mi < nMoments_; mi++)
                                {
                                    mPlus[mi] += mOwn[mi];
                                }
                            }
                        }
                    }

                    // Compute m*
                    for (label mi = 0; mi < nMoments_; mi++)
                    {
                        mStar[mi]
                            = scalar(nFacesOutgoingFlux_[celli] + 1)
                              *moments_(mi)[celli] - mPlus[mi];
                    }

                    nRealizableMomentsStar_[celli]
                        = mStar.nRealizableMoments(false);

                    if
                    (
                        nRealizableMomentsStar_[celli]
                      < nRealizableMoments_[celli]
                    )
                    {
                        cellLimiters_[p][celli] = 0.0;
                    }
                }
            }
        }
    }

    // Setting limiters on internal faces based on cell limiters
    forAll(phiIf, facei)
    {
        const label own = owner[facei];
        const label nei = neighb[facei];

        if (phi_[facei] > 0)
        {
            for (label zi = 0; zi < nZetas_; zi++)
            {
                limiters_[zi][facei] = cellLimiters_[zi][own];
            }
        }
        else
```

```
        {
            for (label zi = 0; zi < nZetas_; zi++)
            {
                limiters_[zi][facei] = cellLimiters_[zi][nei];
            }
        }
    }

    // Setting limiters on boundary faces
    forAll(phiBf, patchi)
    {
        const fvsPatchScalarField& phiPf = phiBf[patchi];

        const labelList& pFaceCells
            = phi_.mesh().boundary()[patchi].faceCells();

        forAll(phiPf, pFacei)
        {
            if (phiPf[pFacei] > 0)
            {
                for (label zi = 0; zi < nZetas_; zi++)
                {
                    limiters_[zi][pFacei]
                        = cellLimiters_[zi][pFaceCells[pFacei]];
                }
            }
        }
    }

    for (label zi = 1; zi < nZetas_; zi++)
    {
        zetasOwn_[zi] = zetasUpwindOwn_[zi] +
            limiters_[zi]*zetasCorrOwn_[zi];

        zetasNei_[zi] = zetasUpwindNei_[zi] +
            limiters_[zi]*zetasCorrNei_[zi];
    }
}

Foam::scalar Foam::univariateAdvection::zeta::realizableCo() const
{
    const fvMesh& mesh(phi_.mesh());
    const labelList& own = mesh.faceOwner();
    const labelList& nei = mesh.faceNeighbour();

    scalarField internalCo(m0_.size(), 0.0);

    for (label facei = 0; facei < mesh.nInternalFaces(); facei++)
    {
        if (phi_[facei] > 0)
        {
            internalCo[own[facei]] += 1;
        }
        else if (phi_[facei] < 0)
        {
            internalCo[nei[facei]] += 1;
        }
```

```
    }

    internalCo = 1.0/(internalCo + 1.0);

    return gMin(internalCo);
}

void Foam::univariateAdvection::zeta::update()
{
    if (m0_.size() != nFacesOutgoingFlux_.size())
    {
        nFacesOutgoingFlux_.resize(m0_.size());
        nRealizableMoments_.resize(m0_.size());
        nRealizableMomentsStar_.resize(m0_.size());
    }

    // Compute zeta fields
    computeZetaFields();

    // Reconstructing zeta_k on cell faces
    interpolateFields();

    // Recompute moments at sides of cell faces
    updateMomentFieldsFromZetas(m0Nei_, zetasNei_, momentsNei_);
    updateMomentFieldsFromZetas(m0Own_, zetasOwn_, momentsOwn_);

    // Apply additional limitation to zeta_k if needed
    limitZetas();

    // Recompute moments at sides of cell faces
    updateMomentFieldsFromZetas(m0Nei_, zetasNei_, momentsNei_);
    updateMomentFieldsFromZetas(m0Own_, zetasOwn_, momentsOwn_);

    // Calculate moment advection term
    dimensionedScalar zeroPhi("zero", phi_.dimensions(), 0.0);

    forAll(divMoments_, divi)
    {
        divMoments_(divi) =
            fvc::surfaceIntegrate
            (
                momentsNei_[divi]*min(phi_, zeroPhi)
              + momentsOwn_[divi]*max(phi_, zeroPhi)
            );
    }
}

void Foam::univariateAdvection::zeta::updateMomentFieldsFromZetas
(
    const surfaceScalarField& m0f,
    const PtrList<surfaceScalarField>& zetaf,
    PtrList<surfaceScalarField>& mf
)
{
    forAll(zetaf[0], facei)
    {
        scalarList zf(nZetas_);
```

```
        for (label zetai = 0; zetai < nZetas_; zetai++)
        {
            zf[zetai] = zetaf[zetai][facei];
        }

        scalarList mFace(nMoments_, 0.0);
        zetaToMoments(zf, mFace, m0f[facei]);

        for (label mi = 0; mi < nMoments_; mi++)
        {
            mf[mi][facei] = mFace[mi];
        }
    }

    // Boundary conditions
    const surfaceScalarField::Boundary& bf = zetaf[0].boundaryField();

    forAll(bf, patchi)
    {
        const fvsPatchScalarField& m0Patch = bf[patchi];

        forAll(m0Patch, facei)
        {
            scalarList zf(nZetas_);

            for (label zetai = 0; zetai < nZetas_; zetai++)
            {
                zf[zetai] = zetaf[zetai].boundaryField()[patchi][facei];
            }

            scalarList mFace(nMoments_, 0.0);
            zetaToMoments(zf, mFace, m0f.boundaryField()[patchi][facei]);

            for (label mi = 0; mi < nMoments_; mi++)
            {
                mf[mi].boundaryFieldRef()[patchi][facei] = mFace[mi];
            }
        }
    }
}

// ************************************************************************* //
```

References

[1] F. Laurent, T.T. Nguyen, Realizable second-order finite-volume schemes for the advection of moment sets of the particle size distribution, J. Comput. Phys. 337 (2017) 309–338. doi:10.1016/j.jcp.2017.02.046.

[2] D. Ramkrishna, Population balances: theory and applications to particulate systems in engineering, Academic Press, San Diego, CA, 2000.

[3] D.L. Marchisio, R.O. Fox, Computational Models for Polydisperse Particulate and Multiphase Systems, Cambridge University Press, 2013.

[4] C. Cercignani, Rarefied Gas Dynamics: From Basic Concepts to Actual Calculations, 1st ed., Cambridge University Press, 2000.
[5] C. Cercignani, The Boltzmann Equation and Its Applications, Springer, New York, 1988.
[6] A.D. Fokker, Die mittlere Energie rotierender elektrischer Dipole im Strahlungsfeld, Ann. Phys. 348 (1914) 810–820. doi:10.1002/andp.19143480507.
[7] M. Planck, Über einen Satz der statistischen Dynamik und seine Erweiterung in der Quantentheorie, Sitzungsberichte Preuss. Akad. Wiss. Zu Berl. 24 (1917) 324–341.
[8] A. Kolmogoroff, Über die analytischen Methoden in der Wahrscheinlichkeitsrechnung, Math. Ann. 104 (1931) 415–458. doi:10.1007/BF01457949.
[9] S. Chapman, T.G. Cowling, The mathematical theory of non-uniform gases, 2nd ed., Cambridge University Press, Cambridge, U.K., 1961.
[10] H. Struchtrup, The BGK-model with velocity-dependent collision frequency, Contin. Mech. Thermodyn. 9 (1997) 23–31. doi:10.1007/s001610050053.
[11] R. McGraw, Description of aerosol dynamics by the quadrature method of moments, Aerosol Sci. Technol. 27 (1997) 255–265. doi:10.1080/02786829708965471.
[12] R.J. LeVeque, Finite Volume Methods for Hyperbolic Problems, 1st ed., Cambridge University Press, 2002.
[13] J.H. Ferziger, M. Peric, Computational Methods for Fluid Dynamics, 3rd ed., Springer, 2001.
[14] V. Vikas, C.D. Hauck, Z.J. Wang, R.O. Fox, Radiation transport modeling using extended quadrature method of moments, J. Comput. Phys. 246 (2013) 221–241. doi:10.1016/j.jcp.2013.03.028.
[15] V. Vikas, C. Yuan, Z.J. Wang, R.O. Fox, Modeling of bubble-column flows with quadrature-based moment methods, Chem. Eng. Sci. 66 (2011) 3058–3070. doi:10.1016/j.ces.2011.03.009.
[16] A. Wick, T.-T. Nguyen, F. Laurent, R.O. Fox, H. Pitsch, Modeling soot oxidation with the Extended Quadrature Method of Moments, Proc. Combust. Inst. 36 (2017) 789–797. doi:10.1016/j.proci.2016.08.004.
[17] D.L. Wright, R. McGraw, D.E. Rosner, Bivariate Extension of the Quadrature Method of Moments for Modeling Simultaneous Coagulation and Sintering of Particle Populations, J. Colloid Interface Sci. 236 (2001) 242–251. doi:10.1006/jcis.2000.7409.
[18] C. Yoon, R. McGraw, Representation of generally mixed multivariate aerosols by the quadrature method of moments: I. Statistical foundation, J. Aerosol Sci. 35 (2004) 561–576. doi:10.1016/j.jaerosci.2003.11.003.
[19] J. Akroyd, A.J. Smith, L.R. McGlashan, M. Kraft, Numerical investigation of DQMoM-IEM as a turbulent reaction closure, Chem. Eng. Sci. 65 (2010) 1915–1924. doi:10.1016/j.ces.2009.11.010.
[20] V. Alopaeus, M. Laakkonen, J. Aittamaa, Solution of population balances with breakage and agglomeration by high-order moment-conserving method of classes, Chem. Eng. Sci. 61 (2006) 6732–6752. doi:10.1016/j.ces.2006.07.010.
[21] P.J. Attar, P. Vedula, Direct quadrature method of moments solution of the Fokker–Planck equation, J. Sound Vib. 317 (2008) 265–272. doi:10.1016/j.jsv.2008.02.037.
[22] A. Buffo, M. Vanni, D.L. Marchisio, Multidimensional population balance model for the simulation of turbulent gas–liquid systems in stirred tank reactors, Chem. Eng. Sci. 70 (2012) 31–44. doi:10.1016/j.ces.2011.04.042.
[23] T.L. Chan, Y.H. Liu, C.K. Chan, Direct quadrature method of moments for the exhaust particle formation and evolution in the wake of the studied ground vehicle, J. Aerosol Sci. 41 (2010) 553–568. doi:10.1016/j.jaerosci.2010.03.005.
[24] S. de Chaisemartin, L. Fréret, D. Kah, F. Laurent, R.O. Fox, J. Reveillon, M. Massot, Eulerian models for turbulent spray combustion with polydispersity and droplet crossing, Comptes

Rendus Mécanique. 337 (2009) 438–448. doi:10.1016/j.crme.2009.06.016.
[25] O. Desjardins, R.O. Fox, P. Villedieu, A quadrature-based moment method for dilute fluid-particle flows, J. Comput. Phys. 227 (2008) 2514–2539. doi:10.1016/j.jcp.2007.10.026.
[26] R.B. Diemer, J.H. Olson, A moment methodology for coagulation and breakage problems: Part 2—moment models and distribution reconstruction, Chem. Eng. Sci. 57 (2002) 2211–2228. doi:10.1016/S0009-2509(02)00112-4.
[27] P. Donde, H. Koo, V. Raman, A multivariate quadrature based moment method for LES based modeling of supersonic combustion, J. Comput. Phys. 231 (2012) 5805–5821. doi:10.1016/j.jcp.2012.04.031.
[28] R. Fan, D.L. Marchisio, R.O. Fox, Application of the direct quadrature method of moments to polydisperse gas-solid fluidized beds, Powder Technol. 139 (2004) 7–20. doi:10.1016/j.powtec.2003.10.005.
[29] J.L. Favero, P.L.C. Lage, The dual-quadrature method of generalized moments using automatic integration packages, Comput. Chem. Eng. 38 (2012) 1–10. doi:10.1016/j.compchemeng.2011.11.010.
[30] R.O. Fox, A quadrature-based third-order moment method for dilute gas-particle flows, J. Comput. Phys. 227 (2008) 6313–6350. doi:10.1016/j.jcp.2008.03.014.
[31] R.O. Fox, Higher-order quadrature-based moment methods for kinetic equations, J. Comput. Phys. 228 (2009) 7771–7791. doi:10.1016/j.jcp.2009.07.018.
[32] D. Kah, F. Laurent, M. Massot, S. Jay, A high order moment method simulating evaporation and advection of a polydisperse liquid spray, J. Comput. Phys. 231 (2012) 394–422. doi:10.1016/j.jcp.2011.08.032.
[33] D. Kah, F. Laurent, L. Fréret, S. de Chaisemartin, R. Fox, J. Reveillon, M. Massot, Eulerian quadrature-based moment models for dilute polydisperse evaporating sprays, Flow Turbul. Combust. 85 (2010) 649–676. doi:10.1007/s10494-010-9286-z.
[34] D. L. Marchisio, R. Dennis Vigil, R. O. Fox, Implementation of the quadrature method of moments in CFD codes for aggregation–breakage problems, Chem. Eng. Sci. 58 (2003) 3337–3351. doi:10.1016/S0009-2509(03)00211-2.
[35] C. Laurent, G. Lavergne, P. Villedieu, Quadrature method of moments for modeling multi-component spray vaporization, Int. J. Multiph. Flow. 36 (2010) 51–59. doi:10.1016/j.ijmultiphaseflow.2009.08.005.
[36] E. Madadi-Kandjani, A. Passalacqua, An extended quadrature-based moment method with log-normal kernel density functions, Chem. Eng. Sci. 131 (2015) 323–339. doi:10.1016/j.ces.2015.04.005.
[37] A. Passalacqua, R.O. Fox, R. Garg, S. Subramaniam, A fully coupled quadrature-based moment method for dilute to moderately dilute fluid–particle flows, Chem. Eng. Sci. 65 (2010) 2267–2283. doi:10.1016/j.ces.2009.09.002.
[38] M. Pigou, J. Morchain, P. Fede, M.-I. Penet, G. Laronze, New developments of the Extended Quadrature Method of Moments to solve Population Balance Equations, J. Comput. Phys. 365 (2018) 243–268. doi:10.1016/j.jcp.2018.03.027.
[39] M. Petitti, M. Vanni, D.L. Marchisio, A. Buffo, F. Podenzani, Simulation of coalescence, break-up and mass transfer in a gas–liquid stirred tank with CQMOM, Chem. Eng. J. 228 (2013) 1182–1194. doi:10.1016/j.cej.2013.05.047.
[40] L.F.L.R. Silva, R.C. Rodrigues, J.F. Mitre, P.L.C. Lage, Comparison of the accuracy and performance of quadrature-based methods for population balance problems with simultaneous breakage and aggregation, Comput. Chem. Eng. 34 (2010) 286–297. doi:10.1016/j.compchemeng.2009.11.005.
[41] M. Strumendo, H. Arastoopour, Solution of PBE by MOM in finite size domains, Chem. Eng.

Sci. 63 (2008) 2624–2640. doi:10.1016/j.ces.2008.02.010.
[42] C. Yoon, R. McGraw, Representation of generally mixed multivariate aerosols by the quadrature method of moments: II. Aerosol dynamics, J. Aerosol Sci. 35 (2004) 577–598. doi:10.1016/j.jaerosci.2003.11.012.
[43] V. Raman, R.O. Fox, Modeling of Fine-Particle Formation in Turbulent Flames, Annu. Rev. Fluid Mech. 48 (2016) 159–190. doi:10.1146/annurev-fluid-122414-034306.
[44] B. Kong, R.O. Fox, A solution algorithm for fluid–particle flows across all flow regimes, J. Comput. Phys. 344 (2017) 575–594. doi:10.1016/j.jcp.2017.05.013.
[45] B. Kong, R.O. Fox, H. Feng, J. Capecelatro, R. Patel, O. Desjardins, R.O. Fox, Euler–euler anisotropic gaussian mesoscale simulation of homogeneous cluster-induced gas–particle turbulence, AIChE J. 63 (n.d.) 2630–2643. doi:10.1002/aic.15686.
[46] J.C. Heylmun, B. Kong, A. Passalacqua, R.O. Fox, A quadrature-based moment method for polydisperse bubbly flows, Comput. Phys. Commun. (2019). doi:10.1016/j.cpc.2019.06.005.
[47] F. Laurent, V. Santoro, M. Noskov, M.D. Smooke, A. Gomez, M. Massot, Accurate treatment of size distribution effects in polydisperse spray diffusion flames: multi-fluid modelling, computations and experiments, Combust. Theory Model. 8 (2004) 385–412. doi:10.1088/1364-7830/8/2/010.
[48] F. Doisneau, M. Arienti, J. Oefelein, On Multi-Fluid models for spray-resolved LES of reacting jets, Proc. Combust. Inst. 36 (2017) 2441–2450. doi:10.1016/j.proci.2016.07.120.
[49] D.L. Wright, Numerical advection of moments of the particle size distribution in Eulerian models, J. Aerosol Sci. 38 (2007) 352–369. doi:10.1016/j.jaerosci.2006.11.011.
[50] D.L. Marchisio, R.O. Fox, Solution of population balance equations using the direct quadrature method of moments, J. Aerosol Sci. 36 (2005) 43–73. doi:10.1016/j.jaerosci.2004.07.009.
[51] R. McGraw, Correcting transport errors during advection of aerosol and cloud moment sequences in Eulerian models, Brookhaven National Laboratory, 2006. https://www.bnl.gov/envsci/pubs/pdf/2012/BNL-96992-2012-BC.pdf.
[52] M. Essadki, S. de Chaisemartin, M. Massot, F. Laurent, A. Larat, S. Jay, Adaptive Mesh Refinement and High Order Geometrical Moment Method for the Simulation of Polydisperse Evaporating Sprays, Oil Gas Sci. Technol. – Rev. D'IFP Energ. Nouv. 71 (2016) 61. doi:10.2516/ogst/2016012.
[53] V. Vikas, Z.J. Wang, A. Passalacqua, R.O. Fox, Realizable high-order finite-volume schemes for quadrature-based moment methods, J. Comput. Phys. 230 (2011) 5328–5352. doi:10.1016/j.jcp.2011.03.038.
[54] V. Vikas, Z.J. Wang, R.O. Fox, Realizable high-order finite-volume schemes for quadrature-based moment methods applied to diffusion population balance equations, J. Comput. Phys. 249 (2013) 162–179. doi:10.1016/j.jcp.2013.05.002.
[55] G. Alldredge, F. Schneider, A realizability-preserving discontinuous Galerkin scheme for entropy-based moment closures for linear kinetic equations in one space dimension, J. Comput. Phys. 295 (2015) 665–684. doi:10.1016/j.jcp.2015.04.034.
[56] H.G. Weller, G. Tabor, H. Jasak, C. Fureby, A tensorial approach to computational continuum mechanics using object-oriented techniques, Comput. Phys. 12 (1998) 620–631. doi:10.1063/1.168744.
[57] H. Dette, The Theory of Canonical Moments with Applications in Statistics, Probability, and Analysis, John Wiley & Sons, 1997.
[58] H. Dette, W.J. Studden, Matrix measures, moment spaces and Favard's theorem for the interval [0,1] and [0,∞), Linear Algebra Its Appl. 345 (2002) 169–193. doi:10.1016/S0024-3795(01)00493-1.
[59] J.C. Wheeler, Modified moments and Gaussian quadratures, Rocky Mt. J. Math. 4 (1974) 287–

296. doi:10.1216/RMJ-1974-4-2-287.
[60] C. Berthon, Stability of the MUSCL Schemes for the Euler Equations, Commun. Math. Sci. 3 (2005) 133–157.
[61] M. Skibinsky, Extreme nth moments for distributions on [0, 1] and the inverse of a moment space map, J. Appl. Probab. 5 (1968) 693–701. doi:10.2307/3211931.
[62] H. Jasak, Error analysis and estimation for the finite volume method with applications to fluid flows, Imperial College of Science, Technology and Medicine, 1996.
[63] OpenFOAM, The OpenFOAM Foundation, (2018). http://openfoam.org/.
[64] C.W.S. Bruner, Parallelization of the Euler Equations on Unstructured Grids, Virginia Tech, 1996. https://vtechworks.lib.vt.edu/handle/10919/30397 (accessed July 13, 2018).
[65] S. Chakravarthy, S. Osher, High resolution applications of the Osher upwind scheme for the Euler equations, in: 6th Comput. Fluid Dyn. Conf. Danvers, American Institute of Aeronautics and Astronautics, 1983. doi:10.2514/6.1983-1943.
[66] P.L. Roe, Characteristic-based schemes for the Euler equations, Annu. Rev. Fluid Mech. 18 (1986) 337–365. doi:10.1146/annurev.fl.18.010186.002005.
[67] P.L. Roe, Some contributions to the modelling of discontinuous flows, in: 1985: pp. 163–193. http://adsabs.harvard.edu/abs/1985ams..conf..163R (accessed July 13, 2018).
[68] P. Sweby, High resolution schemes using flux limiters for hyperbolic conservation laws, SIAM J. Numer. Anal. 21 (1984) 995–1011. doi:10.1137/0721062.
[69] B. van Leer, Towards the ultimate conservative difference scheme III. Upstream-centered finite-difference schemes for ideal compressible flow, J. Comput. Phys. 23 (1977) 263–275. doi:10.1016/0021-9991(77)90094-8.
[70] B. van Leer, Towards the ultimate conservative difference scheme. II. Monotonicity and conservation combined in a second-order scheme, J. Comput. Phys. 14 (1974) 361–370. doi:10.1016/0021-9991(74)90019-9.
[71] M.S. Darwish, F. Moukalled, TVD schemes for unstructured grids, Int. J. Heat Mass Transf. 46 (2003) 599–611. doi:10.1016/S0017-9310(02)00330-7.
[72] ASME, Procedure for Estimation and Reporting of Uncertainty Due to Discretization in CFD Applications, J. Fluids Eng. 130 (2008) 078001-078001–4. doi:10.1115/1.2960953.
[73] OpenQBMM, An open-source implementation of Quadrature-Based Moment Methods, (2018). doi:10.5281/zenodo.591651.
[74] C. Yuan, R.O. Fox, Conditional quadrature method of moments for kinetic equations, J. Comput. Phys. 230 (2011) 8216–8246. doi:10.1016/j.jcp.2011.07.020.
[75] A. Passalacqua, J. Heylmun, M. Icardi, E. Madadi, P. Bachant, X. Hu, OpenQBMM 5.0.0 for OpenFOAM 7, Zenodo, 2019. doi:10.5281/zenodo.3471804.