Computer Physics Communications 233 (2018) 193–205

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

# A high-order cross-platform incompressible Navier–Stokes solver via artificial compressibility with application to a turbulent jet[☆]

N.A. Loppi [a,*], F.D. Witherden [b], A. Jameson [b], P.E. Vincent [a]

[a] Department of Aeronautics, Imperial College London, SW7 2AZ, United Kingdom
[b] Department of Aeronautics and Astronautics, Stanford University, CA 94305, USA

ARTICLE INFO

*Article history:*
Received 22 December 2017
Received in revised form 12 June 2018
Accepted 15 June 2018
Available online 25 June 2018

*Keywords:*
Incompressible flows
Artificial compressibility
Flux reconstruction
Modern hardware
Parallel algorithms
Turbulence

ABSTRACT

Modern hardware architectures such as GPUs and manycore processors are characterised by an abundance of compute capability relative to memory bandwidth. This makes them well-suited to solving temporally explicit and spatially compact discretisations of hyperbolic conservation laws. However, classical pressure-projection-based incompressible Navier–Stokes formulations do not fall into this category. One attractive formulation for solving incompressible problems on modern hardware is the method of artificial compressibility. When combined with explicit dual time stepping and a high-order Flux Reconstruction discretisation, the majority of operations can be cast as compute bound matrix–matrix multiplications that are well-suited for GPU acceleration and manycore processing. In this work, we develop a high-order cross-platform incompressible Navier–Stokes solver, via artificial compressibility and dual time stepping, in the PyFR framework. The solver runs on a range of computer architectures, from laptops to the largest supercomputers, via a platform-unified templating approach that can generate/compile CUDA, OpenCL and C/OpenMP code at runtime. The extensibility of the cross-platform templating framework defined within PyFR is clearly demonstrated, as is the utility of $P$-multigrid for convergence acceleration. The platform independence of the solver is verified on Nvidia Tesla P100 GPUs and Intel Xeon Phi 7210 KNL manycore processors with a 3D Taylor–Green vortex test case. Additionally, the solver is applied to a 3D turbulent jet test case at $Re = 10{,}000$, and strong scaling is reported up to 144 GPUs. The new software constitutes the first high-order accurate cross-platform implementation of an incompressible Navier–Stokes solver via artificial compressibility and $P$-multigrid accelerated dual time stepping to be published in the literature. The technology has applications in a range of sectors, including the maritime and automotive industries. Moreover, due to its cross-platform nature, the technology is well placed to remain relevant in an era of rapidly evolving hardware architectures.

**Program summary**

*Program Title:* PyFR v1.7.5
*Program Files doi:* http://dx.doi.org/10.17632/65m665nt9c.1
*Licensing provisions:* BSD 3-clause
*Programming language:* Python, CUDA, OpenCL and C
*Supplementary material:* Configuration and mesh files for the Taylor–Green Vortex and Turbulent jet test cases
*Journal reference of previous version:* Comput. Phys. Commun. **185** (2014) 3028–3040
*Does the new version supersede the previous version?:* Yes
*Reasons for the new version:* Adding support for incompressible flows
*Summary of revisions:* Introducing a new high-order cross-platform incompressible flow solver via artificial compressibility and $P$-multigrid accelerated dual time stepping.
*Nature of problem:* Incompressible Euler and Navier–Stokes equations for solving unsteady turbulent flows.
*Solution method:* Artificial compressibility formulation discretised with a high-order Flux Reconstruction approach in space and $P$-multigrid accelerated dual time stepping in time.

[☆] This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
[*] Corresponding author.
*E-mail address:* n.loppi15@imperial.ac.uk (N.A. Loppi).

https://doi.org/10.1016/j.cpc.2018.06.016

*Additional comments including restrictions and unusual features:* The algorithm targets modern massively parallel hardware platforms. Cross-platform capability is achieved via runtime code generation.

## 1. Introduction

The Computational Fluid Dynamics (CFD) community benefits from a wide range of methods for solving flow problems. Currently, nominally second-order accurate Finite Volume (FV) methods underpin the majority of industrial CFD, due to their robustness, and ability to work on unstructured meshes of complex geometries. However, over the past three decades, significant research has been undertaken into developing methods with increased spatial orders-of-accuracy. These high-order methods have been shown to offer superior accuracy to low-order methods, with the same or lower computational cost [1,2], and have demonstrated potential for performing scale-resolving turbulent simulations more efficiently than their low order counterparts. However, issues remain, especially in terms of industrial adoption. These include a high memory footprint for implicit time stepping, and the challenge of generating high-order curved element meshes [2].

Early attempts at developing high-order accurate spatial discretisations were based on finite difference and pure spectral approaches [3,4], and were hence limited to simple geometries. The first spatially compact high order approach suitable for unstructured grids was the Discontinuous Galerkin (DG) formulation by Reed and Hill [5] for simulating neutron transport. Later, it was popularised and applied to CFD by Cockburn et al. [6,7]. DG is formulated using the integral representation of the governing system by combining a Finite Element type polynomial approximation inside the element, and a FV type interface flux. Other popular high-order schemes with a resemblance to nodal DG are the Spectral Difference (SD) methods by Liu et al. [8,9], which are a generalised version of the staggered grid Chebyshev multidomain method by Kopriva and Kolias [10]. A more recent approach is the Flux Reconstruction (FR) method proposed by Huynh [11] which unifies nodal DG and SD schemes within a single framework. When combined with explicit time stepping, all schemes in the FR framework retain a compact stencil. This spatial locality results in minimal communication between elements, which is of particular importance for modern hardware platforms that are characterised by an excess of compute capability relative to memory bandwidth.

Leveraging the full potential of modern hardware platforms with a pressure based algorithm for solving the incompressible Navier–Stokes equations is difficult since they exhibit significant indirect communication during the elliptic projection stage. The elliptic projection can be bypassed by adopting the Artificial Compressibility Method (ACM) of Chorin [12], which was developed in the late 1960s as an alternative way of solving steady incompressible fluid flow problems. Rather than projecting the pressure with a Poisson equation, a coupling pressure term is added to the continuity equation that drives the system towards a solenoidal velocity field in the limit of steady state. The original formulation preserves the hyperbolic nature of the system, but destroys time accuracy. A common procedure for recovering time accuracy is dual time stepping [13]. In this approach, real time is discretised with a backward difference scheme, whose solution is found by marching the governing equation in pseudo time. In the context of the ACM, the divergence free velocity constraint is satisfied at each physical time step. Adopting explicit dual time stepping as a temporal discretisation and using high-order FR in space, the majority of operations can be cast as compute bound matrix–matrix multiplications that are well-suited for GPUs and manycore processors.

There have been various attempts to apply the ACM in a high-order context. Bassi [14] first succeeded in applying the approach with DG to solve steady incompressible flow problems. Later, Liang et al. [15] extended the method to SD schemes in 2D, providing support for unsteady flows via dual time stepping. Recently, Cox et al. [16] successfully applied the method with FR in 2D, and introduced fully implicit inner iterations. In the current study, we extend the cross-platform high-order CFD framework PyFR (www.pyfr.org) [17,18] to include an incompressible Navier–Stokes solver based on FR, the ACM, dual time stepping, and $P$-multigrid.

The paper is structured as follows. Section 2 gives a brief overview of the high-order FR approach for mixed element unstructured grids. Sections 3 and 4 detail the ACM and how it can be advanced in time using $P$-multigrid accelerated dual time stepping. Section 5 describes our cross-platform implementation in the PyFR framework, that can run on a wide range of hardware architectures. Section 6 verifies the platform independence on Nvidia Tesla P100 GPUs and Intel Xeon Phi 7210 KNL manycore processors with a 3D Taylor–Green vortex test case. Additionally, the utility of $P$-multigrid for convergence acceleration is demonstrated. In Section 7, we apply the solver to a 3D turbulent jet problem at $Re = 10{,}000$, which has relevance to many industrial application areas and natural flow phenomena, such as hydrojet propulsion, cooling systems, and seafloor plumes. Excellent agreement with available experimental data is obtained. Finally, conclusions are drawn in Section 8.

## 2. Flux reconstruction

A summary of the FR method for mixed unstructured grids is given below. For further details see [19–21]. To begin, consider a finite solution domain in Euclidean space $\mathbf{\Omega} \in \mathbb{R}^{N_D}$ with a coordinate system $\mathbf{x} = x_i \in \mathbb{R}^{N_D}$ in $N_D$ dimensions. In this domain, we wish to solve an advection–diffusion problem written in conservative form as

$$\frac{\partial u_\alpha}{\partial t} + \nabla \cdot \mathbf{f}_\alpha = 0, \tag{1}$$

where the conservative field variables are defined as $u_\alpha = u_\alpha(\mathbf{x}, t)$ and the corresponding flux is $\mathbf{f}_\alpha = \mathbf{f}_\alpha(u_\alpha, \nabla u_\alpha)$. The subscript $\alpha$ refers to a single field variable or its flux, $0 < \alpha < N_v$, where $N_v$ is the total number of field variables. A conforming mesh comprising of a set of suitable element types $\varepsilon$ defines the discrete representation of the solution domain. Such elements are for example line elements in $N_D = 1$, quadrilaterals and triangles in $N_D = 2$, and hexahedra in $N_D = 3$. The elements in the discrete domain must satisfy

$$\mathbf{\Omega} = \bigcup_{e \in \varepsilon} \mathbf{\Omega}_e\,, \qquad \mathbf{\Omega}_e = \bigcup_{n=1}^{|\mathbf{\Omega}_e|} \mathbf{\Omega}_n\,, \qquad \mathbf{\Omega} = \bigcap_{e \in \varepsilon} \bigcap_{n=1}^{|\mathbf{\Omega}_e|} \mathbf{\Omega}_n = \emptyset\,, \tag{2}$$

where the subscript $e$ refers to all elements of a certain type and $|\mathbf{\Omega}_e|$ denotes their total count such that $0 \leq n \leq |\mathbf{\Omega}_e|$.

All calculations are performed in a transformed space by mapping each element $\mathbf{\Omega}_e$ into its respective standard element $\tilde{\mathbf{\Omega}}_e \in \mathbb{R}^{N_D}$, represented in a coordinate system $\tilde{\mathbf{x}} = \tilde{x}_i \in \mathbb{R}^{N_D}$. The mapping functions relating the two element spaces are defined as

$$\tilde{\mathbf{x}} = \mathcal{M}_{en}^{-1}(\mathbf{x}), \tag{3}$$

$$\mathbf{x} = \mathcal{M}_{en}(\tilde{\mathbf{x}}). \tag{4}$$

The Jacobian matrices and determinants related to the mapping are

$$\mathbf{J}_{en}^{-1} = J_{enij}^{-1} = \frac{\partial \mathcal{M}_{eni}^{-1}}{\partial x_j}, \qquad \mathbf{J}_{en} = J_{enij} = \frac{\partial \mathcal{M}_{eni}}{\partial x_j}, \tag{5}$$

$$\mathcal{J}_{en}^{-1} = \det \mathbf{J}_{en}^{-1} = \frac{1}{\mathcal{J}_{en}}, \qquad \mathcal{J}_{en} = \det \mathbf{J}_{en}, \tag{6}$$

where $\widetilde{\nabla} = \partial/\partial\tilde{x}_i$. Using the expressions above, the transformed flux and transformed gradient of the solution can be expressed as

$$\tilde{\mathbf{f}}_{en\alpha}(\tilde{\mathbf{x}}, t) = \mathcal{J}_{en}(\tilde{\mathbf{x}})\mathbf{J}_{en}^{-1}(\mathcal{M}_{en}(\tilde{\mathbf{x}}))\mathbf{f}_{en\alpha}(\mathcal{M}_{en}(\tilde{\mathbf{x}}), t), \tag{7}$$

$$\left(\widetilde{\nabla}u\right)_{en\alpha}(\tilde{\mathbf{x}}, t) = \mathbf{J}_{en}^{T}(\tilde{\mathbf{x}})\nabla u_{en\alpha}(\mathcal{M}_{en}(\tilde{\mathbf{x}}), t). \tag{8}$$

Furthermore, Eq. (1) can be written in terms of the transformed divergence of the transformed flux as

$$\frac{\partial u_{en\alpha}}{\partial t} + \mathcal{J}_{en}^{-1}\widetilde{\nabla} \cdot \tilde{\mathbf{f}}_{en\alpha} = 0. \tag{9}$$

To proceed, consider defining a nodal solution basis within a standard element. Take $\tilde{\mathbf{x}}_{ei}^{u}$ to be a set of standard element solution points associated with a given element type, where $0 \le i < N_e$. Examples of such point distributions are Gauss–Legendre or Gauss–Lobatto points, when $N_D = 1$, Witherden-Vincent points [22] for triangles, when $N_D = 2$, and Shunn–Ham points [23] for tetrahedra, when $N_D = 3$. The set of solution points $\tilde{\mathbf{x}}_{ei}^{u}$ within the standard element $\mathbf{\Omega_e}$ can be used to define a nodal basis set $l_{ei}(\tilde{\mathbf{x}})$ of order $P(N_e)$ that spans a polynomial space $\mathcal{P}$. The nodal basis must satisfy the property $l_{ei}(\tilde{\mathbf{x}}_{ej}^{u}) = \delta_{ij}$. Following the conventions in [24], we first form any modal basis $L_{ei}$, typically using Jacobi polynomials that span $\mathcal{P}$, and calculate the entries in the generalised Vandermonde matrix $\mathcal{V}_{eij}$. Second, we construct the nodal basis as $l_{ei}(\tilde{\mathbf{x}}_{ej}^{u}) = \mathcal{V}_{eij}^{-1}L_{ej}$. In addition to the solution points, a set of element-type-specific flux points $\tilde{\mathbf{x}}_{i}^{f}$, where $0 \le i < N_e^{f}$, must be defined on the element boundaries $\partial\mathbf{\Omega}_e$. Two elements sharing a boundary, and hence a set of flux points along the interface, must have a conforming mapping which recovers the same global location for the paired flux points. Thus, all paired flux points, $\mathbf{x}_{eni}^{f} = \mathbf{x}_{e'n'i'}^{f}$ must satisfy

$$\mathcal{M}_{en}(\tilde{\mathbf{x}}_{ei}^{f}) = \mathcal{M}_{e'n'}(\tilde{\mathbf{x}}_{e'i'}^{f}). \tag{10}$$

The first step of the FR approach involves computing the discontinuous solution at the flux points from the interpolating solution polynomial

$$u_{en\alpha}^{f} = u_{ein\alpha}^{u}l_{ei}(\tilde{\mathbf{x}}_{ej}^{f}). \tag{11}$$

These values can be used to find a common interface solution at the flux points via

$$\mathcal{C}u_{ein\alpha}^{f} = \mathcal{C}u_{e'i'n'\alpha}^{f} = \mathcal{C}(u_{ein\alpha}^{f}, u_{e'i'n'\alpha}^{f}), \tag{12}$$

where $\mathcal{C}(u_R, u_L)$ is a scalar function, analogous to a Riemann solver that returns the common solution, where the subscripts R and L denote the right and left states respectively. The next step is to compute the gradient of the solution at the solution points. For this purpose, we form a vector correction function $\mathbf{g}_{ei}^{f}(\tilde{\mathbf{x}})$ which satisfies

$$\hat{\tilde{\mathbf{n}}}_{ej} \cdot \mathbf{g}_{ei}^{f}(\tilde{\mathbf{x}}_{ej}^{f}) = \delta_{ij}. \tag{13}$$

and subsequently define the transformed gradient of the solution at the solution points as

$$\left(\widetilde{\nabla}u\right)_{ein\alpha}^{u} = \left(\hat{\tilde{\mathbf{n}}}\widetilde{\nabla}\right)_{ej} \cdot \mathbf{g}_{ej}^{f}(\tilde{\mathbf{x}}_{ei}^{u})\{\mathcal{C}_\alpha u_{ejn\alpha}^{f} - u_{ejn\alpha}^{f}\} + u_{ekn\alpha}^{u}\widetilde{\nabla}l_{ek}(\tilde{\mathbf{x}}_{ei}^{u}), \tag{14}$$

which transforms into physical space as

$$(\nabla u)_{ein\alpha}^{u} = \mathbf{J}_{ein}^{-T}\left(\widetilde{\nabla}u\right)_{ein\alpha}^{u}, \tag{15}$$

where $\mathbf{J}_{ein}^{-T} = \mathbf{J}_{en}^{-T}(\tilde{\mathbf{x}}_{ei})$. Further, the transformed flux at the solution points can be evaluated as

$$\tilde{\mathbf{f}}_{ein\alpha}^{u} = \mathcal{J}_{ein}\mathbf{J}_{ein}^{-1}\mathbf{f}_\alpha(u_{ein\alpha}^{u}, (\nabla u)_{ein\alpha}^{u}), \tag{16}$$

and the normal component of the transformed flux at the flux points as

$$\tilde{f}_{ein\alpha}^{f\perp} = l_{ej}(\tilde{\mathbf{x}}_{ei}^{f})\hat{\tilde{n}}_{ei} \cdot \tilde{\mathbf{f}}_{ejn\alpha}^{u}. \tag{17}$$

Common normal fluxes at the flux point pairs are found with a Riemann solver $\mathcal{F}(u_R, \nabla u_R, u_L, \nabla u_L, \mathbf{n})$. The common values are assigned as

$$\mathcal{F}_\alpha f_{ein\alpha}^{f\perp} = -\mathcal{F}_\alpha f_{e'i'n'\alpha}^{f\perp} = \mathcal{F}(u_{ein}^{f}, (\nabla u)_{ein}^{f}, u_{e'i'n'}^{f}, (\nabla u)_{e'i'n'}^{f}, \tilde{\mathbf{n}}_{ein}^{f}), \tag{18}$$

where

$$(\nabla u)_{ein\alpha}^{f} = l_{ejn}(\tilde{\mathbf{x}}_{ei}^{f})\left(\widetilde{\nabla}u\right)_{ejn\alpha}^{u}. \tag{19}$$

The normal common interface fluxes transform into the standard element space via

$$\mathcal{F}_\alpha\tilde{f}_{ein\alpha}^{f\perp} = \mathcal{J}_{ein}^{f}\tilde{\mathbf{n}}_{ein}^{f}\mathcal{F}_\alpha f_{ein\alpha}^{f\perp}. \tag{20}$$

The divergence of the continuous flux can be computed with a procedure analogous to Eq. (14), as

$$\left(\widetilde{\nabla} \cdot \tilde{\mathbf{f}}\right)_{ein\alpha}^{u} = \left[\widetilde{\nabla} \cdot \mathbf{g}_{ej}^{f}(\tilde{\mathbf{x}}_{ei}^{u})\{\mathcal{F}_\alpha f_{ejn\alpha}^{f\perp} - f_{ejn\alpha}^{f\perp}\} + \tilde{\mathbf{f}}_{ekn\alpha}^{u} \cdot \widetilde{\nabla}l_{ek}(\tilde{\mathbf{x}}_{ei}^{u})\right]. \tag{21}$$

This serves as the right-hand side in Eq. (9) and the solution can be advanced in time using a suitable time-integration method.

## 3. Artificial compressibility

### 3.1. Artificial compressibility for the Euler equations

In the ACM formulation of the Euler equations, the conservative variables in three dimensions $\mathbf{x} = \{x \ y \ z\}^T$ are

$$u = \begin{Bmatrix} p \\ v_x \\ v_y \\ v_z \end{Bmatrix}, \tag{22}$$

where $p$ is the pressure and $\mathbf{v} = \{v_x \ v_y \ v_z\}^T$ are the velocity components. The total flux is defined as $\mathbf{f} = \mathbf{f}_e$, where $\mathbf{f}_e = f_{ex}\hat{\mathbf{i}} + f_{ey}\hat{\mathbf{j}} + f_{ez}\hat{\mathbf{k}}$,

$$f_{ex} = \begin{Bmatrix} \zeta v_x \\ v_x^2 + p \\ v_x v_y \\ v_x v_z \end{Bmatrix}, f_{ey} = \begin{Bmatrix} \zeta v_y \\ v_y v_x \\ v_y^2 + p \\ v_y v_z \end{Bmatrix}, f_{ez} = \begin{Bmatrix} \zeta v_z \\ v_z v_x \\ v_z v_y \\ v_z^2 + p \end{Bmatrix}, \tag{23}$$

$\zeta$ is the artificial compressibility relaxation factor, and $\hat{\mathbf{i}}, \hat{\mathbf{j}}$ and $\hat{\mathbf{k}}$ are unit vectors in the $x$, $y$ and $z$ directions, respectively.

This formulation of the governing equations has a hyperbolic nature; artificial pressure waves traveling at a finite speed are introduced. These waves distribute the pressure and disappear in the limit of pseudo steady state. Eigenvalues of the inviscid flux Jacobian matrices

$$\mathbf{J}_i = \frac{\partial f_{ei}}{\partial u} \ \forall i \in \mathbf{x} \tag{24}$$

are

$$\lambda_i = \{v_i - c_i \ v_i \ v_i \ v_i + c_i\}^T \ \forall i \in \mathbf{x}, \tag{25}$$

where $c_i = \sqrt{v_i^2 + \zeta}$ is the pseudo speed of sound. The pseudo speed of sound monotonically decreases with decreasing artificial compressibility relaxation factor $\zeta$ which can be adjusted to reduce the system stiffness.

The common interface flux $\mathcal{F}$ only contains the inviscid term such that $\mathcal{F} = \mathcal{F}_e$, and the inviscid interface flux is obtained using a Rusanov Riemann solver [25] as

$$\mathcal{F}_e(u_L, u_R, \hat{\mathbf{n}}) = \hat{\mathbf{n}} \cdot \{\frac{1}{2}(\mathbf{f}_{eL} + \mathbf{f}_{eR})\} + \frac{1}{2}\max(\lambda_n)(u_L - u_R), \tag{26}$$

where $\max(\lambda_n) = |\hat{\mathbf{n}} \cdot \mathbf{v}| + c_n$, with $c_n = \sqrt{(\hat{\mathbf{n}} \cdot \mathbf{v})^2 + \zeta}$ the maximum wave speed in the interface normal direction.

### 3.2. Artificial compressibility for the Navier–Stokes equations

In the ACM formulation of the Navier–Stokes equations, the total flux is defined as $\mathbf{f} = \mathbf{f}_e - \mathbf{f}_v$, where $\mathbf{f}_v = f_{vx}\hat{\mathbf{i}} + f_{vy}\hat{\mathbf{j}} + f_{vz}\hat{\mathbf{k}}$,

$$f_{vx} = \nu \begin{Bmatrix} 0 \\ \dfrac{\partial v_x}{\partial x} \\ \dfrac{\partial v_y}{\partial x} \\ \dfrac{\partial v_z}{\partial x} \end{Bmatrix}, f_{vy} = \nu \begin{Bmatrix} 0 \\ \dfrac{\partial v_x}{\partial y} \\ \dfrac{\partial v_y}{\partial y} \\ \dfrac{\partial v_z}{\partial y} \end{Bmatrix}, f_{vz} = \nu \begin{Bmatrix} 0 \\ \dfrac{\partial v_x}{\partial z} \\ \dfrac{\partial v_y}{\partial z} \\ \dfrac{\partial v_z}{\partial z} \end{Bmatrix}, \tag{27}$$

and $\nu$ is the kinematic viscosity. The total normal interface flux is now defined as $\mathcal{F} = \mathcal{F}_e - \mathcal{F}_v$, where the viscous part $\mathcal{F}_v$ is computed using the local discontinuous Galerkin (LDG) approach as presented in [24],

$$\begin{aligned}\mathcal{F}_v(u_L, \nabla u_L, u_R, \nabla u_R, \hat{\mathbf{n}}) = \hat{\mathbf{n}} \cdot \{(\frac{1}{2} + \beta)\mathbf{f}_{vL} \\ + (\frac{1}{2} - \beta)\mathbf{f}_{vR}\} + \tau(u_L - u_R).\end{aligned} \tag{28}$$

The common interface solution needed for the gradient is computed with

$$\mathcal{C}u^f(u_L, u_R) = \left(\frac{1}{2} - \beta\right) u_L + \left(\frac{1}{2} + \beta\right) u_R. \tag{29}$$

The two free parameters in the expressions above, $\beta$ and $\tau$, control the degree of upwinding/downwinding and the solution jump penalisation at the interface. Common practice is to select $\beta = \pm\frac{1}{2}$ and $0 \leq \tau \leq 1$.

## 4. Dual time stepping

### 4.1. Overview

A dual time stepping formulation for the artificial compressibility Euler/Navier–Stokes equations can be written as

$$\frac{\partial u}{\partial \tau} + \mathbf{I}_c \frac{\partial u}{\partial t} + \nabla \cdot \mathbf{f} = 0, \tag{30}$$

where the first term is a pseudo time derivative which is marched towards zero, resulting in a pseudo steady state. A cancellation matrix $\mathbf{I}_c = \text{diag}\{0\ 1\ 1\ 1\}$ is employed as a coefficient for the physical time derivative to eliminate it from the continuity equation. This ensures that the solution is driven towards a solenoidal velocity field, since at the pseudo steady state the continuity equation with a fixed $\zeta$ yields

$$\lim_{\tau\to\infty} \left[\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}\right] = 0. \tag{31}$$

When the solution is integrated with respect to pseudo time, the real time derivative can be considered as a source term for the right-hand side that is updated at every physical time step.

**Table 1**
Coefficients for the BDFs.

| | $B_0$ | $B_1$ | $B_2$ | $B_3$ |
|---|---|---|---|---|
| Backward-Euler | 1 | −1 | – | – |
| BDF2 | $\frac{3}{2}$ | −2 | $\frac{1}{2}$ | – |
| BDF3 | $\frac{11}{6}$ | −3 | $\frac{1}{2}$ | $-\frac{1}{3}$ |

Defining $\mathcal{R} = R - \mathbf{I}_c S_t$, where $R = -\nabla \cdot \mathbf{f}$ and $S_t = \frac{\partial u}{\partial t}$, Eq. (30) can be expressed as

$$\frac{\partial u}{\partial \tau} = \mathcal{R} = R - \mathbf{I}_c S_t. \tag{32}$$

Discretising the real time derivative with an $M$-stage BDF, and the pseudo time derivative with a $k_{\max}$-stage explicit multistage scheme, we can write a single pseudo iteration as

$$\begin{aligned} u^{n+1,m+1,0} &= u^{n+1,m}\,, \\ u^{n+1,m+1,k} &= u^{n+1,m+1,0} + \alpha_k \Delta\tau \mathcal{R}^{n+1,m+1,k-1} \\ &\quad \text{for } k = 1 \ldots k_{\max}\,, \\ u^{n+1,m+1} &= u^{n+1,m+1,k_{\max}}, \end{aligned} \tag{33}$$

where the superscripts $n$ and $m$ denote real and pseudo time levels, $k$ is the stage index and $\alpha$ is the stage coefficient. Different real time levels are needed for the BDF expansion in the right hand side computation according to

$$\begin{aligned} \mathcal{R}^{n+1,m+1,k-1} &= \frac{1}{\alpha_{PI}}\left[R^{n+1,m+1,k-1} - \mathbf{I}_c S_t^{n+1,m+1,0}\right] \\ &= \frac{1}{\alpha_{PI}}\Bigg[R^{n+1,m+1,k-1} \\ &\quad - \frac{\mathbf{I}_c}{\Delta t}\left(B_0 u^{n+1,m+1,0} + \sum_{\sigma=0}^{M} B_{\sigma+1} u^{n-\sigma}\right)\Bigg], \end{aligned} \tag{34}$$

where $B_\sigma$ are the coefficients of the BDFs as listed in Table 1. In this study, the leading term in the BDF is expressed at the first RK stage, which is a common practice in so-called point implicit source term treatment. The treatment results in a scaling coefficient $\alpha_{PI} = 1 + \alpha_k \Delta\tau B_0/\Delta t$ which limits $\Delta\tau$ if $\Delta\tau \gg \Delta t$. Being explicit in the pseudo time, the coefficient was found to have no advantage, and in this study we default to $\alpha_{PI} = 1$.

The subiteration procedure defined by Eq. (32) is repeated until the solution is deemed to have converged. Specific stopping criteria include the following: residuals being driven to a prescribed tolerance, or a fixed number of pseudo iterations being reached. Subsequently, the pseudo steady-state solution is stored as the physical solution and assigned as the initial value for the next time level $u^{n+2,0} = u^{n+1}$. Furthermore, the remaining terms in $S_t$ are updated accordingly, $u^{n-(\sigma+1)} = u^{n-\sigma}$ for $\sigma = 1 \ldots M$. Algorithm 1 illustrates the dual time stepping approach with a BDF2 expansion.

```
while $t < t_{max}$ do
    while unconverged do
        Solve Equation 32
    end
    Return $u^{n+1}$
    $u^{n-1} = u^n$
    $u^n = u^{n+1}$
    $t = t + \Delta t$
end
```

**Algorithm 1:** Dual Time Stepping with BDF2.

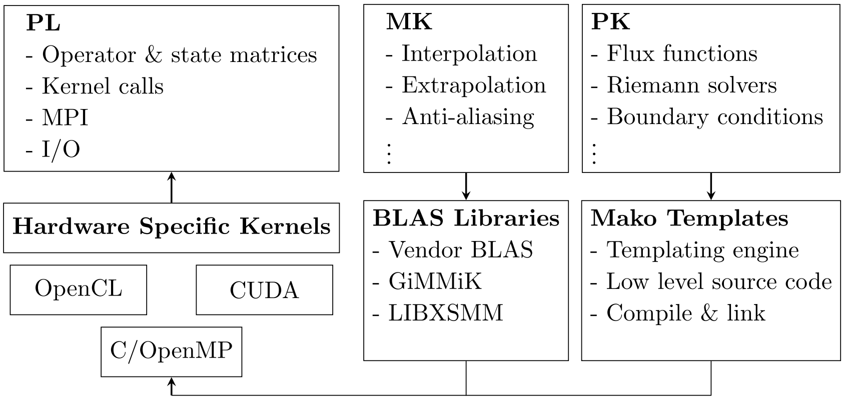


**Fig. 1.** The structure of the PyFR framework.

### 4.2. *P*-Multigrid

The efficiency of the dual time stepping algorithm depends on fast convergence of the pseudo steady-state process. The disadvantage of using explicit schemes in pseudo time is that their stability region is limited by the CFL number making them inefficient at eliminating low frequency error modes on fine grids [26]. The issue can be addressed by adopting implicit schemes to allow much larger pseudo time steps [16]. However, the trade-off with implicit schemes is that their storage requirements are significantly higher due to the size of the flux Jacobian matrices which so far appears to have prohibited their use in large scale 3D applications at high orders. Moreover, many methods for solving the resulting linear system, such as LU-SGS or ILU-GMRES, are not scale invariant and increasing the amount of parallelism reduces the numerical effectiveness of the preconditioner.

The *P*-multigrid technique (perhaps better referred to as multi-*P*) attempts to circumvent the CFL condition by correcting the solution at different polynomial levels. Without altering the computational grid, low order representations of the solution, which exhibit a less restrictive CFL limit, can be used to propagate information with a larger $\Delta\tau$. Another quality of *P*-multigrid is that when the solution is projected to a lower order space, the low frequency error modes appear as higher frequencies respective to the resolution, which allows explicit steppers to eliminate them more effectively.

In the *P*-multigrid pseudo time formulation, the following equation is solved

$$\frac{\partial u_l}{\partial \tau} = \mathcal{R}_l - r_l, \tag{35}$$

where $r_l$ is a multigrid source term arising from lower order representations of the solution and the subscript denotes a *P*-multigrid level $l$ corresponding to any polynomial order between 0 and $P$. At the highest *P*-multigrid level $l = P$, the source term $r_l = 0$. Following the methodologies in [15] and [26], a single *P*-multigrid V-cycle can be formulated according to Algorithm 2. In contrast to [15], our formulation performs the residual defect restriction and *P*-multigrid source evaluation only on the divergence term $R$ and the real-time derivative term is only accounted for during the smoothing iterations. This was found to modestly accelerate convergence with respect to wall-time, since for each restriction operation it reduces the number of pointwise kernel calls by two.

```
for $l \in \{P, \ldots, l_{min} + 1\}$ do
    for $i \in \{0, \ldots, Niters_l\}$ do
        Smooth according to Equation 35
    end
    Calculate residual defect $d_l = r_l - R(u_l)$
    Restrict solution $u^0_{l-1} = \mathrm{I_r}(u_l)$ and store it
    Restrict defect $d_{l-1} = \mathrm{I_r}(d_l)$
    Evaluate source $r_{l-1} = R(u^0_{l-1}) + d_{l-1}$.
end
for $l \in \{l_{min}, \ldots, P\}$ do
    for $i \in \{0, \ldots, Niters_l\}$ do
        Smooth according to Equation 40
    end
    Calculate correction $\Delta_l = u^0_l - u_l$
    Prolongate correction $\Delta_{l+1} = \mathrm{I_p}(\Delta_l)$
    Add correction $u_{l+1} = u_{l+1} + \Delta_{l+1}$
end
for $i \in \{0, \ldots, Niters_{l_{max}}\}$ do
    Post-smoothing at the highest level according to Equation 35
end
```

**Algorithm 2:** A single *P*-multigrid V-cycle between polynomial levels $P$ and $l_{\mathrm{min}}$, where $Niters_l$ denotes the number of smoothing iterations at level $l$.

Restriction and prolongation operations can be performed in several ways. In our work, the restriction operator is constructed using the generalised Vandermonde matrix as

$$\mathrm{I_r} = \mathcal{V}^T_{l-1}\tilde{\mathbf{I}}\mathcal{V}^{-T}_l, \tag{36}$$

where $\tilde{\mathbf{I}}$ is an non-square matrix of zeros, except on its main diagonal, where it has entries of one. The approach corresponds to transforming the nodal solution into a modal representation, removing the highest order mode and transforming back to a nodal basis. Prolongation is simply performed as Lagrangian interpolation according to

$$\mathrm{I_p} = l_{l,ei}(\tilde{\mathbf{x}}_{l+1,ej}). \tag{37}$$

## 5. Implementation

### 5.1. PyFR overview

PyFR is a cross-platform framework for solving advection–diffusion problems using the FR approach. One of the main advantages of PyFR is that it produces platform portable code with a single implementation using Python and Mako. Fig. 1 provides an overview of the framework. PL is the hardware independent Python layer which constitutes the main body of the framework. PL initialises the data layout, precomputes all necessary matrices, and defines the chain of kernel calls that drive the simulation. Distributed memory parallelism and input/output are also handled by PL. Mesh partitioning, when required, is outsourced to METIS [27].

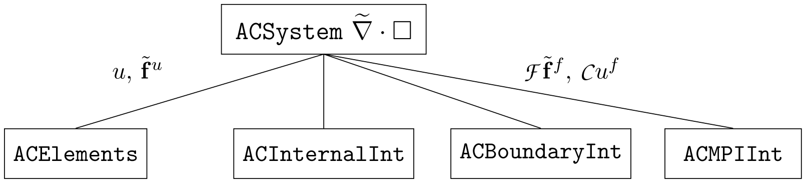

ACSystem $\widetilde{\nabla}\cdot\square$
$u$, $\tilde{\mathbf{f}}^u$
$\mathcal{F}\tilde{\mathbf{f}}^f$, $\mathcal{C}u^f$
ACElements
ACInternalInt
ACBoundaryInt
ACMPIInt

**Fig. 2.** The class hierarchy of the artificial compressibility systems.

```
 1  class ACNavierStokesElements(BaseACFluidElements,
 2                               BaseAdvectionDiffusionElements):
 3      def set_backend(self, backend, nscalupts, nonce):
 4          super().set_backend(backend, nscalupts, nonce)
 5
 6          backend.pointwise.register('pyfr.solvers.acnavstokes.kernels.tflux')
 7
 8          tplargs = dict(ndims=self.ndims, nvars=self.nvars,
 9                         c=self.cfg.items_as('constants', float))
10
11          self.kernels['tdisf'] = lambda: backend.kernel(
12              'tflux', tplargs=tplargs, dims=[self.nupts, self.neles],
13              u=self.scal_upts_inb, smats=self.smat_at('upts'),
14              f=self._vect_upts
15          )
```

**Fig. 3.** Registering the `tflux` PK, which computes the discontinuous flux $\tilde{\mathbf{f}}^u$, to the `ACNavierStokesElements` PL class.

The kernels called by PL can be divided into matrix multiplication kernels MK and pointwise kernels PK. The MK are used for all operations where a time-wise constant operator matrix multiplies a large state matrix, such as interpolation, extrapolation, and anti-aliasing. An example is interpolation of field variables from the solution points onto the flux points, which is performed for a given element type via

$$\mathbf{U}_e^f = \mathbf{M}_e^0 \mathbf{U}_e \,, \tag{38}$$

where

$$(\mathbf{U}_e^u)_{j(n\alpha)} = u_{ejn\alpha}^u \,, \qquad \mathbb{M}(\mathbf{U}_e^u) = N_e \times N_v|\boldsymbol{\Omega}_e| \tag{39}$$

$$(\mathbf{U}_e^f)_{i(n\alpha)} = u_{ein\alpha}^f \,, \qquad \mathbb{M}(\mathbf{U}_e^f) = N_e^f \times N_v|\boldsymbol{\Omega}_e| \tag{40}$$

$$(\mathbf{M}_e^0)_{i(n\alpha)} = l_{ej}(\tilde{\mathbf{x}}_{ei}^f) \,, \qquad \mathbb{M}(\mathbf{M}_e^0) = N_e^f \times N_e. \tag{41}$$

All MK are off-loaded to GEMM (GEneral Matrix Multiply) subroutines from vendor supplied BLAS (Basic Linear Algebra Subprograms) libraries or bespoke GiMMiK [28] and LIBXSMM [29] kernels. The latter two kernel libraries leverage the *a priori* known structure/sparsity of the operator matrices and can considerably reduce wall-clock time in certain circumstances [28].

PK are used for operations that require pointwise calculations, such as computing the non-linear fluxes, Riemann solves, and boundary conditions. They are built at runtime by passing platform-unified Mako kernel templates to a Mako derived templating engine that produces low level platform specific source code. The low level code is then compiled as shared libraries and linked to the PL at runtime. The templating engine automatically generates appropriate indexing for vectorisation and efficient memory accesses which vary across platforms.

### *5.2. Artificial compressibility solvers*

Fig. 2 illustrates the structure of the ACM solvers at the PL, which were implemented as separate modules in the *pyfr/solvers/* directory. In Fig. 2 and discussion below, the class names are unified, representing both ACEuler and ACNavierStokes solvers. At the top of the ACM solver class hierarchy is the `ACSystem` class which inherits a pipeline of kernel calls to perform FR discretisation of an arbitrary advection–diffusion equation as described in Section 2. To complete the discretisation for the ACM equations, `ACSystem` also binds four subclasses: `ACElements`, `ACInternalInt`, `ACMPIInt` and `ACBoundaryInt` that set the state vector $u$ and register all PK for computing the ACM fluxes.

The `ACElements` class registers the PKs for computing fluxes $\tilde{\mathbf{f}}^u$ at internal solution points. The common interface flux treatment $\mathcal{F}\tilde{\mathbf{f}}^f$ is divided into MPI-rank-local PKs and message passing PKs, which are registered in `Internal` and `MPIInt`, respectively. This approach allows the data exchange and rank-local computation to happen simultaneously. The third interface class `BoundaryInt` registers all boundary condition PKs. Additionally, kernels for computing the common interface solution $\mathcal{C}u^f$ are needed for all interface classes that require the viscous flux calculation.

Fig. 3 demonstrates a two step procedure for registering a PK to the PL. As an example, we only consider a PK for computing the discontinuous flux $\tilde{\mathbf{f}}^u$ that is registered to the `ACNavierStokesElements` PL class. All other classes in Fig. 2 are built using an analogous approach. First, the kernel must be registered to the backend by providing the location of the Mako template, which is done on line 6. Second, the kernel must be added to the pointwise kernels dictionary as a lambda function, which is done on line 11. The lambda function must specify keyword arguments for all variables that are passed from PL to PK, which include pointers to the input and output matrices, `u` (state), `smats` (mapping), `f`(flux), and auxiliary constant variables to facilitate templating `tplargs`, `dims`.

Fig. 4 shows the templated implementation of the discontinuous flux kernel `tflux` that we register to PL in Fig. 3. Input and output arguments on lines 5 to 6 correspond to the keyword arguments in the lambda function. The template consists of `inviscid_flux` and `viscous_flux_add` macro functions on the lines 21 and 36, which work as individual building blocks for the template. These macro functions, which form the expression for the inviscid and viscous fluxes, are expanded on lines 10 and 11. Within the macros, the flux expression is unrolled using a Python-like loop syntax which has direct access to the templating variables passed from Python, such as `tplargs` and `dims`. The ${} tags denote a placeholder for variable substitution.

```
<%inherit file='base'/>
<%namespace module='pyfr.backends.base.makoutil' name='pyfr'/>

<%pyfr:kernel name='tflux' ndim='2'
              u='in fpdtype_t[${str(nvars)}]'
              smats='in fpdtype_t[${str(ndims)}][${str(ndims)}]'
              f='inout fpdtype_t[${str(ndims)}][${str(nvars)}]'>
    // Compute the flux (F = Fi + Fv)
    fpdtype_t ftemp[${ndims}][${nvars}];
    ${pyfr.expand('inviscid_flux', 'u', 'ftemp')};
    ${pyfr.expand('viscous_flux_add', 'u', 'f', 'ftemp')};

    // Transform the fluxes
% for i, j in pyfr.ndrange(ndims, nvars):
    f[${i}][${j}] = ${' + '.join('smats[{0}][{1}]*ftemp[{1}][{2}]'
                                 .format(i, k, j)
                                 for k in range(ndims))};
% endfor
</%pyfr:kernel>

<%pyfr:macro name='inviscid_flux' params='s, f'>
    fpdtype_t v[] = ${pyfr.array('s[{i}]', i=(1, ndims + 1))};
    fpdtype_t p = s[0];

    // Mass flux
% for i in range(ndims):
    f[${i}][0] = ${c['ac-zeta']}*v[${i}];
% endfor

    // Momentum fluxes
% for i, j in pyfr.ndrange(ndims, ndims):
    f[${i}][${j + 1}] = v[${i}]*v[${j}]${' + p' if i == j else ''};
% endfor
</%pyfr:macro>

<%pyfr:macro name='viscous_flux_add' params='uin, grad_uin, fout'>
% for i, j in pyfr.ndrange(ndims, ndims):
    fout[${i}][${j+1}] += -${c['nu']}*grad_uin[${i}][${j+1}];
% endfor
</%pyfr:macro>
```

**Fig. 4.** The Mako template to generate the `tflux` PK.

### 5.3. *Boundary conditions*

All boundary conditions were implemented as separate boundary PKs. These PKs impose the boundary conditions via ghost states for the underlying pseudo time system, which converge to physical boundary conditions in the limit of pseudo steady-state. Specifically, three ghost states per boundary condition are required. In the Riemann solver, an inviscid ghost state $\mathscr{B}^{\mathrm{Rie}}u$ replaces the right-hand-side solution state $u_{\mathrm{R}}$. In the LDG approach, a viscous ghost state $\mathscr{B}^{\mathrm{LDG}}u$ replaces right-hand-side solution state $u_{\mathrm{R}}$, and a solution gradient ghost state $\mathscr{B}^{\mathrm{LDG}}\nabla u$ replaces the right-hand-side gradient $\nabla u_{\mathrm{R}}$.

The following boundary conditions are available for the ACM solvers. A velocity inlet condition is imposed via ghost states defined as

$$\mathscr{B}^{\mathrm{Rie}}u = \{p_{\mathrm{L}} \;\; v_x^b \;\; v_y^b \;\; v_z^b\}^T, \tag{42}$$

$$\mathscr{B}^{\mathrm{LDG}}u = \mathscr{B}^{\mathrm{Rie}}u, \tag{43}$$

$$\mathscr{B}^{\mathrm{LDG}}\nabla u = 0, \tag{44}$$

where the superscript $b$ denotes a user-specified free-stream value and the subscript $L$ denotes the domain-side state. A pressure outlet condition is imposed via ghost states defined as

$$\mathscr{B}^{\mathrm{Rie}}u = \{p^b \;\; v_{x\mathrm{L}} \;\; v_{y\mathrm{L}} \;\; v_{z\mathrm{L}}\}^T, \tag{45}$$

$$\mathscr{B}^{\mathrm{LDG}}u = \mathscr{B}^{\mathrm{Rie}}u, \tag{46}$$

$$\mathscr{B}^{\mathrm{LDG}}\nabla u = 0. \tag{47}$$

A no-slip wall is imposed via ghost states defined as

$$\mathscr{B}^{\mathrm{Rie}}u = \{p_{\mathrm{L}} \;\; 2v_x^w - v_{x\mathrm{L}} \;\; 2v_y^w - v_{x\mathrm{L}} \;\; 2v_z^w - v_{x\mathrm{L}}\}^T, \tag{48}$$

$$\mathscr{B}^{\mathrm{LDG}}u = \{p_{\mathrm{L}} \;\; v_x^w \;\; v_y^w \;\; v_z^w\}^T, \tag{49}$$

$$\mathscr{B}^{\mathrm{LDG}}\nabla u = \nabla u_{\mathrm{L}}, \tag{50}$$

where the superscript $w$ denotes the wall-velocities which are zero for stationary wall. A slip-wall is imposed via ghost states defined as

$$\mathscr{B}^{\mathrm{Rie}}u = \{p_{\mathrm{L}} \;\; v_{x\mathrm{L}} - 2\hat{n}_x V_{n\mathrm{L}} \;\; v_{y\mathrm{L}} - 2\hat{n}_y V_{n\mathrm{L}} \;\; v_{z\mathrm{L}} - 2\hat{n}_z V_{n\mathrm{L}}\}^T, \tag{51}$$

$$\mathscr{B}^{\mathrm{LDG}}u = \mathscr{B}^{\mathrm{Rie}}u, \tag{52}$$

$$\mathscr{B}^{\mathrm{LDG}}\nabla u = 0. \tag{53}$$

### 5.4. *P-multigrid accelerated dual time stepping*

Dual time stepping was implemented as a new `DualIntegrator` composite class that comprises of `DualController` (DC), `DualPseudoStepper` (DPS), `DualStepper` (DS) and `MultiP` (MP) subclasses as per Fig. 5. To begin, the `DualIntegrator`/`MultiP` class launches an instance of a `systemcls` for each polynomial order in the *P*-Multigrid cycle, which have the ability to compute the divergence of the flux *R*. Without *P*-multigrid acceleration, a single instance at the solution order *P* is created. In case of multiple systems, only one instance can be active at a time. The active system is a `MultiP` class property which returns a system corresponding to a class variable `self.level`. For example, the system `ACSystem`$_1$ is activated by changing `self.level = 1`.

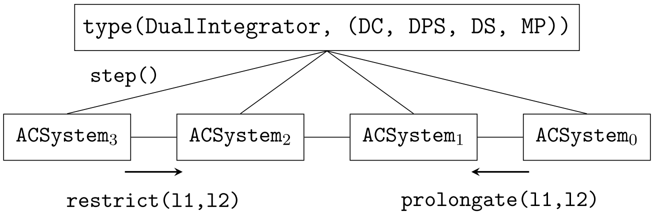


**Fig. 5.** The structure of the $P$-Multigrid implementation consisting of an `DualIntegrator` composite class that launches several instances (levels) of an `ACSystem` class with different polynomial orders.

Additionally, `MultiP` pre-computes and stores the prolongation and restriction operators. The `MultiP` implementation also supports having different GEMM providers (vendor BLAS, GiMMiK, LIBXSMM) and anti-aliasing options across $P$-multigrid levels.

System activation and time-integration are managed by the `DualController` subclass. Specifically, it advances the solution to an arbitrary physical time by calling a sequence of `step()`, `restrict(l1, l2)` and `prolongate(l1, l2)` methods, which form the multigrid cycle. The arguments `l1/l2` are the multigrid levels before/after application of the operator. The cycle can be arbitrary between the levels $P$ and 0, and it is repeated until the solution has converged. The convergence is monitored using platform specific reduction kernels. These kernels were implemented independently for each backend, allowing efficient per-field-variable reduction in the context of an AoSoA (Array of Structures of Arrays) data layout.

The `step()` method, which sets the explicit smoother scheme, is defined in the `DualPseudoStepper` subclass. Current options are the forward-Euler, TVD-RK3 and RK4 schemes. The physical time discretisation method is defined in the `DualStepper` class. The BDF source term is automatically added to appropriate entries of the right-hand-side state matrix during the `step()` call. Options for the physical time discretisations are backward-Euler, BDF2 and BDF3. Both stepper subclasses were structured to facilitate the addition of further schemes using simple Python syntax.

## 6. 3D Taylor–Green vortex at $Re = 1600$

### 6.1. Overview

A 3D Taylor–Green vortex test case at a Reynolds number $Re = 1600$ (based on the diameter and peak velocity of the initial vortices) was studied to verify platform independence and investigate the effect of $P$-multigrid acceleration on performance. In the test case, a set of large vortices interact with each other, transition to turbulence, and finally decay via viscosity. Initial conditions defining vortices with a diameter of 1.0 and a peak velocity of 1.0 were prescribed as

$$v_x = \sin(x)\cos(y)\cos(z)\,, \tag{54}$$

$$v_y = -\sin(x)\cos(y)\cos(z), \tag{55}$$

$$v_z = 0, \tag{56}$$

$$p = 1 + \frac{1}{16}\left[\cos(2x) + \cos(2y)\right]\left[\cos(2z) + 2\right], \tag{57}$$

in a periodic domain $-\pi \le x, y, z \le \pi$. Fig. 6 shows volume renderings of the vorticity magnitude for the initial condition and during the enstrophy peak at $t = 8$.

A total of three double precision simulations were performed on two state-of-the-art platforms. First, the case was run with the CUDA backend on two Nvidia Tesla P100 GPUs with and without $P$-multigrid. Additionally, the multigrid accelerated simulation was repeated with the OpenMP backend on two Intel Xeon Phi 7210 KNL manycore processors. All simulations were launched with an optimal kernel configuration which was found a priori by systematically studying the effect of GiMMiK and LIBXSMM cut-off parameters on the performance. On P100 GPUs, the optimal performance was achieved by offloading all matrix multiplications with number-of-non-zero elements less than 1800 to GiMMiK, which means that only the restriction and prolongation operator between $P = 4$ and $P = 3$ were computed by cuBLAS. On Intel Xeon Phi 7210 KNLs, LIBXSMM outperformed any combination of MKL and GiMMiK and was thereby globally enforced by setting its cut-off based on matrix size to 100,000. Moreover, to achieve optimal performance, the MCDRAM of the Intel Xeon Phi 7210 KNL processor was configured to work in flat mode, and an

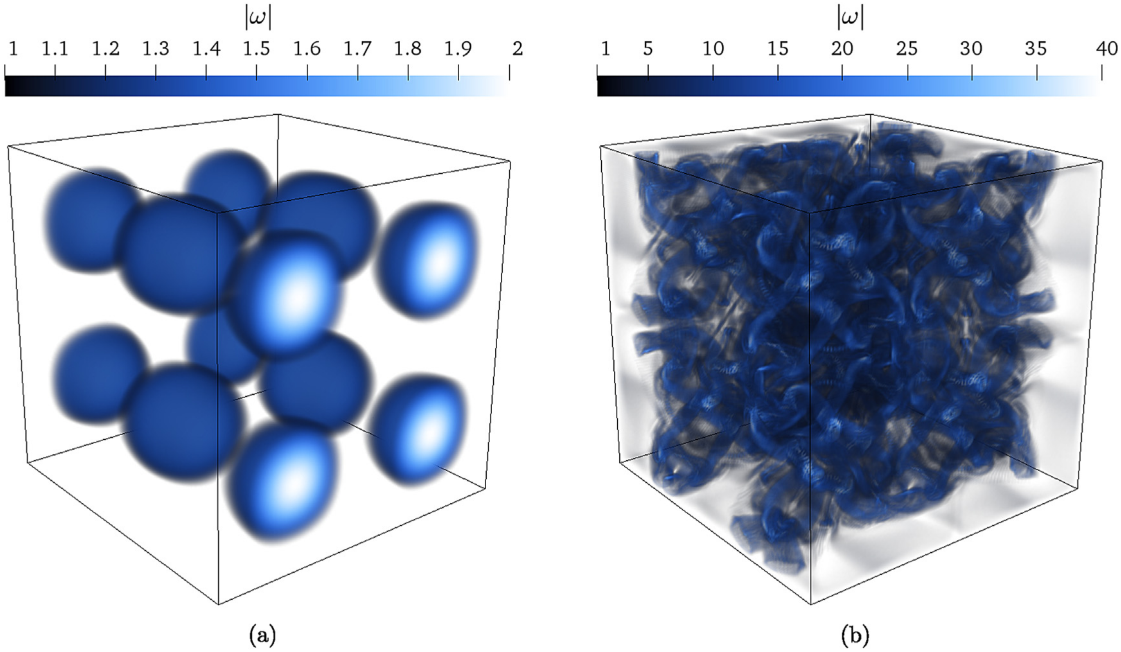


**Fig. 6.** Vorticity magnitude for the Taylor–Green vortex simulation at (a) $t = 0$ and (b) $t = 8$.

**Table 2**
Summary of Taylor–Green vortex simulations.

| | *P*-Multigrid | Iterations | Backend | Platform |
|---|---|---|---|---|
| Case 1 | Off | 75 RK4 | CUDA | P100 |
| Case 2 | On | 3 cycles | CUDA | P100 |
| Case 3 | On | 3 cycles | OpenMP | KNL |

OpenMP/MPI approach with two ranks per node to was used to saturate the interconnects.

All simulations were performed on a hexahedral mesh of 52 × 52 × 52 equisized elements using PyFR v1.7.5. The solution polynomial order used in this study was $P = 4$ with a Gauss–Legendre solution point distribution. No subgrid-scale turbulence model or spatial filtering was applied, and the simulation can be considered as implicit LES or under-resolved DNS. The physical time was integrated using a BDF2 scheme with a time step size $\Delta t = 0.006$. The classical RK4 scheme was used as the pseudo time scheme/smoother in all simulations, and the artificial compressibility factor was fixed as $\beta = 3$. To ensure impartial comparison of the *P*-multigrid and the single-level simulation, all simulations where performed with a pseudo-time step size $\Delta\tau = 0.0014$ that was optimised for the single level case. The optimal $\Delta\tau$ was found by studying the rate of convergence with a constant number of RK4 iterations at the enstrophy peak $t = 8$ via a bisection approach. A 5-level cycle 1-1-1-1-2-1-1-1-3 with the RK4 smoother corresponding to polynomial orders 4-3-2-1-0-1-2-3-4 was used in the *P*-multigrid accelerated simulations, and the pseudo time step sizes at lower levels were increased according to $\Delta\tau_l = 1.85^{4-l}\Delta\tau$. To exclude the effects of convergence monitoring on the measured wall-times, all simulations were performed with a fixed number of pseudo-iterations per physical time step, specifically three cycles for the *P*-multigrid accelerated cases, and 75 RK4 iterations for the single-level case. These values were selected heuristically to ensure both the *P*-multigrid and single-level cases achieved similar levels of convergence in velocity field divergence, which is directly proportional to the pressure residual. Furthermore, it was verified *a posteriori* that the velocity field divergence was on average 1.25 times lower for the *P*-multigrid accelerated cases compared with the single level case, leading to conservative estimates for any *P*-multigrid speed-up.

Table 2 shows a summary of the simulations. The mesh and input files are provided as Electronic Supplementary Material to this paper.

### 6.2. Results

Fig. 7 plots the temporal evolution of $\mathcal{D}$, the solution-point-wise $L2$ norm of the velocity field divergence evaluated at the end of each physical time step, for Cases 1 and 2. It was found that $\mathcal{D}$ was on average 1.25 times lower for the *P*-multigrid accelerated cases compared with the single level case. It can also be seen that the effectiveness of *P*-multigrid is most apparent at the beginning of the simulation, when lengths scales are larger, due to more effective low wave number smoothing. When the large vortices break into smaller structures near the enstrophy peak at $t = 8$, the low wave number smoothing associated with *P*-multigrid becomes less important, and both Case 1 and Case 2 converge to the same level.

Fig. 8 shows the dissipation of the total kinetic energy

$$-\frac{\mathrm{d}E_k}{\mathrm{d}t} = -\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{1}{|\Omega|}\int_{\Omega}\frac{\mathbf{v}\cdot\mathbf{v}}{2}\mathrm{d}\mathbf{x}\right), \tag{58}$$

and the temporal evolution of enstrophy

$$\varepsilon = \frac{1}{|\Omega|}\int_{\Omega}\frac{\boldsymbol{\omega}\cdot\boldsymbol{\omega}}{2}\mathrm{d}\mathbf{x}, \tag{59}$$

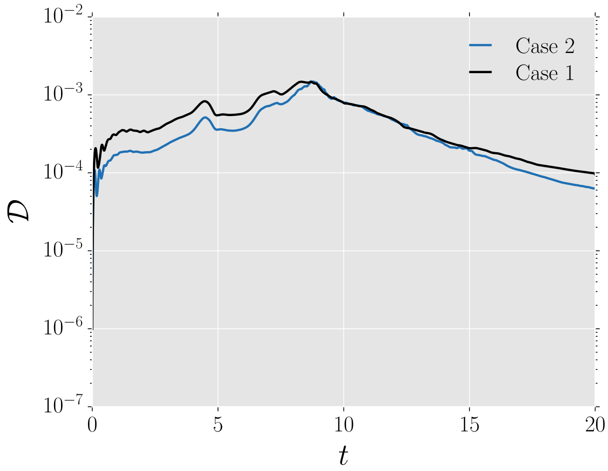


**Fig. 7.** The temporal evolution of the solution-point-wise $L2$ norm of the velocity field divergence $\mathcal{D}$ evaluated at the end of each physical time step.

**Table 3**
Wall-times of Taylor–Green vortex simulations.

| | Case 1 | Case 2 | Case 3 |
|---|---|---|---|
| Wall-time (hh:mm:ss) | 12:45:44 | 03:38:12 | 09:07:14 |

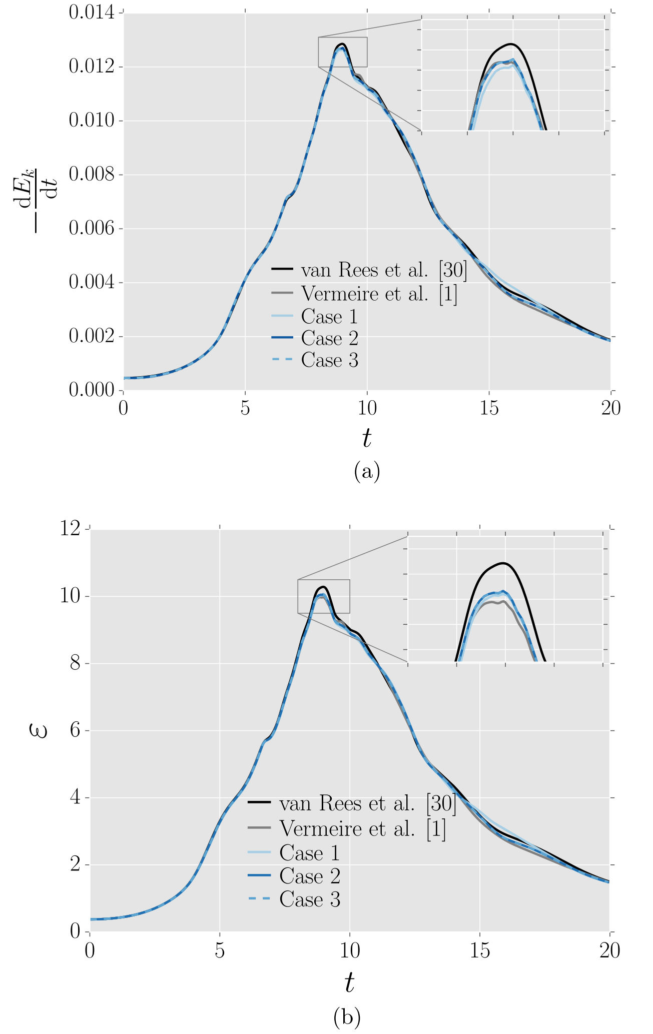
where $\boldsymbol{\omega}$ is the vorticity vector. The volume integrals were computed using quadrature rules, the quadrature nodes being the 4th order Gauss–Legendre solution points. Fig. 8 also includes reference results from van Rees et al. [30] who used an incompressible pseudo-spectral code with $512^3$ degrees of freedom, and compressible PyFR results at $Ma = 0.1$ from Vermeire et al. [1] with identical resolution to the current setup. Three observations can be made. First, results of Case 2 and Case 3 are identical which verifies the platform and backend independence. Second, the *P*-multigrid accelerated Case 2 produces slightly more accurate results than Case 1, which is in line with the better convergence observed in Fig. 7. Third, the *P*-multigrid accelerated Case 2 results are more accurate than the low-Mach compressible solution from Vermeire et al. [1] at the enstrophy peak at $t = 8$, and homogeneous turbulence decay phase $t > 15$, which suggest that the ACM formulation captures the incompressible physics better than a low-Mach approach.

Table 3 shows the wall-time for each case. The results confirm that *P*-multigrid yields an over 3.5 times speed-up compared to single level pseudo time stepping, while maintaining slightly better solution accuracy and convergence. The results also show that the OpenMP backend on Intel Xeon Phi 7210 KNLs is approximately 2.5 times slower than the CUDA backend on Nvidia Tesla P100s. To put the wall-times in context, the compressible $Ma = 0.1$ simulation of Vermeire et al. was repeated on the same Nvidia Tesla P100 hardware as Cases 1 and 2, using the configuration file provided in the Electronic Supplementary Material of [1]. The wall-time of the compressible $Ma = 0.1$ simulation with $P = 4$ and adaptive explicit RK45 time stepping was 05:44:02 on two Nvidia Tesla P100 GPUs. This is 1.6 times longer than the time required for Case 2 to complete, further demonstrating the utility of an ACM approach with *P*-multigrid convergence acceleration.

## 7. Incompressible jet at *Re* = 10,000

### 7.1. Overview

An incompressible 3D turbulent jet test case at a Reynolds number $Re = 10{,}000$ (based on the diameter and peak velocity



(a)

(b)

**Fig. 8.** The temporal evolution of (a) kinetic energy dissipation and (b) enstrophy.

of the jet) was studied to validate the solver, and to assess strong scalability on massively parallel systems. The test case was chosen since experimental data are available for comparison [31], and it has relevance to many industrial application areas and natural flow phenomena, such as hydrojet propulsion, cooling systems, and seafloor plumes. Influential experiments and a general theory of incompressible turbulent round jets are discussed in the review of Lipari and Standsby [32]. Round jets at various Reynolds numbers have also been studied numerically: closest to our setup being DNS by Boersma [33] at $Re = 5000$ and explicitly filtered LES by Bogey and Bailly [34] at $Re = 11{,}000$. However, both of these studies have been performed with compressible codes at higher Mach numbers, 0.6 and 0.9, respectively.

Fig. 9 shows the computational grid in the $y - z$ plane at $x = 0$ together with a schematic of the simulation setup in the $y - x$ plane at $z = 0$. The diameter of the jet was 0.5. The origin was located at the center of the jet as it enters the domain. A two dimensional unstructured circular grid of diameter 24 was extruded in $x$ for a distance of 50 in 250 equally sized steps. The resulting 3D mesh contained 247,250 hexahedral elements in the center of the domain $0 < r < 2.5$, where $r = \sqrt{y^2 + z^2}$, and 596,500 prismatic elements elsewhere. The virtual origin is located at ($x_0$,0,0), which is the starting point of the self-similar region associated with a linear velocity decay and spreading rate [32].

The jet inflow profile with a peak velocity of 1.0 was imposed as

$$V_{\text{jet}}(r) = 0.5 - 0.5\tanh\left[20\left(r - 0.25\right)\right]. \tag{60}$$

A no-slip condition was applied at the boundary surrounding the jet inflow zone, a slip-wall condition was applied at the cylindrical far-field boundary, and a pressure of 10 was imposed at the outlet. A sponge layer was found to be necessary to dissipate the jet before it impinged on the outlet. Specifically, it was imposed via a spatially dependent source term $S$ defined as

$$S = \left(u - u^{\text{out}}\right)\left[0.5 + 0.5\tanh\left(0.5\left(x - 45\right)\right)\right], \tag{61}$$

where $u^{\text{out}} = \{10 \;\; 0 \;\; 0 \;\; 0\}^T$. The mesh and input files are provided as Electronic Supplementary Material to this paper.

### 7.2. Strong scaling

A strong scaling study for the jet test case was undertaken to demonstrate the scalability of the solver and to find optimal runtime parameters for the full physics run. The solution polynomial order was selected as $P = 4$, and the physical time was advanced using BDF2 with $\Delta t = 0.005$. A 5-level $P$-multigrid with an RK4 smoother cycle 1-1-1-1-2-1-1-1-3 was identified to yield good performance in pseudo time. Additionally, due to small and highly curved elements near the center of the domain, flux divergence anti-aliasing of orders between 7 and 4 (7-7-6-5-4-5-6-7-7) was added to increase the stability of the smoothing iterations. The time step sizes between the $P$-multigrid levels were varied with $\Delta\tau_l = 1.7^{4-l}\Delta\tau$, where $\Delta\tau = 0.0007$, and the artificial compressibility parameter was kept constant at $\beta = 2.5$. The simulation was performed with three $P$-multigrid cycles per physical time step, leading to $\mathcal{D} \approx 8 \times 10^{-4}$ as the simulation progressed. The GiMMiK number-of-non-zero cut-off parameter was specified as 3560, which outsourced the restriction/prolongation between levels $P = 3$ and $P = 4$ to cuBLAS, while the rest of the matrix multiplications were performed by GiMMiK. In addition, strong scaling was studied without $P$-multigrid acceleration using single level $P = 4$ pseudo time stepping using 75 iterations, but an otherwise identical setup. Double precision was used throughout.

Fig. 10 shows strong scaling from 9 through 144 Nvidia P100 GPUs with and without $P$-multigrid acceleration. Strong scaling in both cases is almost linear up to 36 Nvidia P100 GPUs after which both cases start to tail off. The $P$-multigrid accelerated case experiences quicker decline due to the presence of low-order iterations with fewer degrees of freedom. For example, on 144 Nvidia P100 GPUs, there are only 1.64 $P = 0$ solution points per CUDA core, whereas the corresponding number at $P = 4$ is over 146. Nevertheless, $P$-multigrid still improves the time to solution on 144 Nvidia P100 GPUs by over a factor of 2.5 compared to single level pseudo time stepping.

### 7.3. Results

The full turbulent jet case was simulated using PyFR v.1.7.5 with $P$-multigrid acceleration and the runtime parameters described in Section 7.2 on 96 Nvidia Tesla P100 GPUs with double precision. The simulation was performed as implicit LES so that no subgrid-scale model or spatial filtering was applied. The simulation was first run to $t = 750$, at which time spurious transients were observed to have been dissipated by the sponge region. Subsequently, the simulation was restarted and turbulent statistics were gathered until $t = 1800$, when the time-averaged quantities were found to be visually converged.

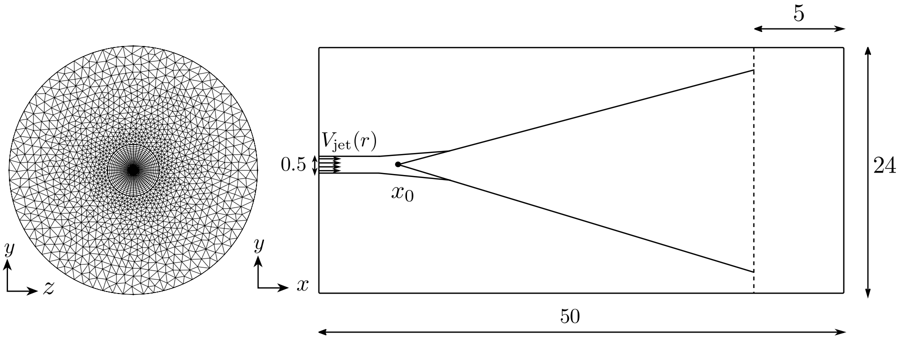


**Fig. 9.** Computational grid in the $y - z$ plane at $x = 0$ together with a schematic of the simulation setup in the $y - x$ plane at $z = 0$. The virtual origin is located at ($x_0$,0,0).

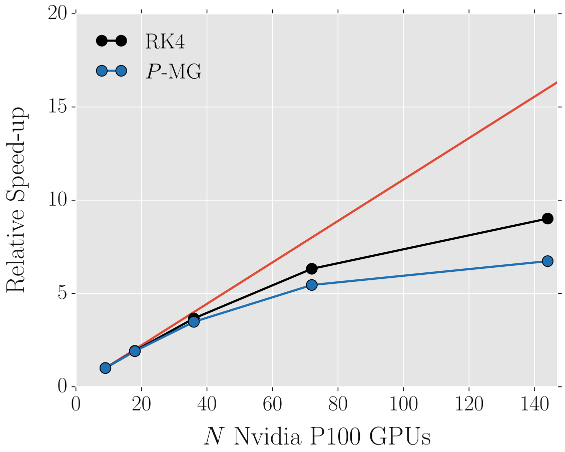


**Fig. 10.** Strong scaling of the incompressible jet on Nvidia Tesla P100 GPUs with P-multigrid (P-MG) and without P-multigrid (RK4). The red line indicates ideal strong scaling.

Fig. 11 shows the volumetrically rendered instantaneous velocity magnitude field at $t = 750$, illustrating the shape of the jet and the resolution of the turbulent scales. In addition, we can see that the outlet sponge region does not have noticeable adverse effects on the jet development.

Figs. 12 and 13 provide a comparison between the computational results and experimental data at $Re = 11{,}000$ from Panchapakesan and Lumley [31]. Fig. 12a shows $1/v_c$ as a function of a shifted coordinate $x - x_0$, where $v_c$ is the time-averaged stream-wise midline velocity and $x_0$ is set as 2.5 to allow comparison with the experimental data. Fig. 12b shows $\overline{v_x}/v_c$ as a function of the self-similarity coordinate $\eta$ defined as

$$\eta = \frac{r}{x - x_0}, \tag{62}$$

where $\overline{v_x}$ is the stream-wise velocity averaged both in time, and over conical planes of constant $\eta$ in the region $10 \le x \le 30$. The profiles are in very good agreement with the experimental data and demonstrate that the correct average decay rate is achieved not only in the midline but also elsewhere in the jet. Fig. 13 shows $\overline{v_x'}/v_c$ and $\overline{v_r'}/v_c$ as a function of $\eta$, where $\overline{v_x'}$ and $\overline{v_r'}$ are the stream-wise and radial root-mean-square velocity fluctuations, respectively, averaged over conical planes of constant $\eta$ in the region $10 \le x \le 30$. Again, the simulation results show very good agreement with experimental data. These results demonstrate that the ACM solver is able to simulate fully turbulent incompressible flows at scale on massively parallel systems.

## 8. Conclusions

A high-order cross-platform incompressible Navier–Stokes solver has been implemented in the PyFR framework using the ACM and *P*-multigrid accelerated dual time stepping. The extensibility of the cross-platform templating framework defined within PyFR has been clearly demonstrated. Platform independence was verified on Nvidia Tesla P100 GPUs and Intel Xeon Phi 7210 KNL manycore processors with a 3D Taylor–Green vortex test case. Additionally, the utility of *P*-multigrid for convergence acceleration was demonstrated; reducing time-to-solution by a factor of 3.5 compared to single level pseudo time stepping. Finally, the solver was applied to a 3D round jet test case at $Re = 10{,}000$, and excellent agreement with experimental data was obtained.

The new software constitutes the first high-order accurate cross-platform implementation of an incompressible Navier–Stokes solver via the ACM and *P*-multigrid accelerated dual time stepping to be published in the literature. The technology has

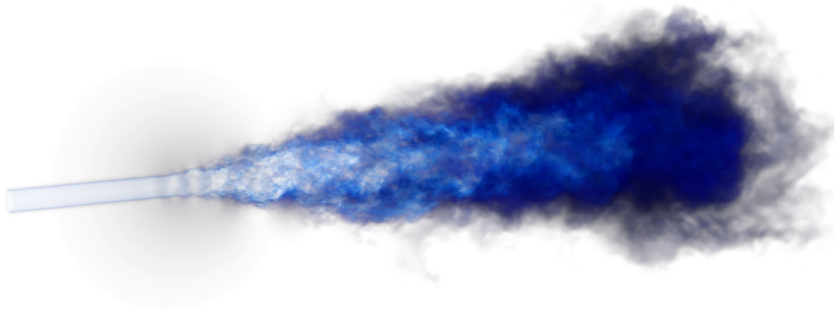


**Fig. 11.** Instantaneous velocity magnitude field of the incompressible jet at $t = 1500$.

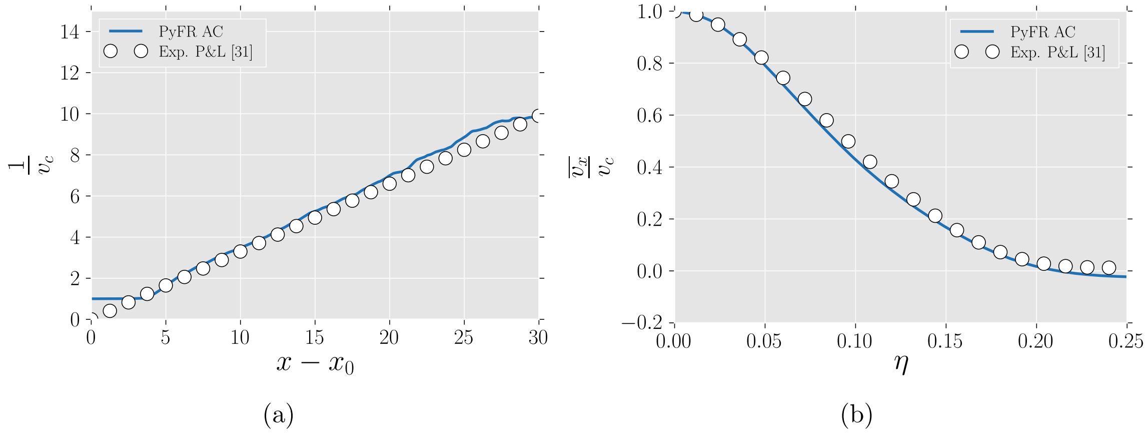


**Fig. 12.** Plot of (a) $1/v_c$ as a function of a shifted coordinate $x - x_0$, where $v_c$ is the time-averaged stream-wise midline velocity and $x_0$ is set as 2.5. Plot of (b) $\overline{v_x}/v_c$ as a function of the self-similarity coordinate $\eta$, where $\overline{v_x}$ is the stream-wise velocity averaged both in time, and over conical planes of constant $\eta$ in the region $10 \leq x \leq 30$.

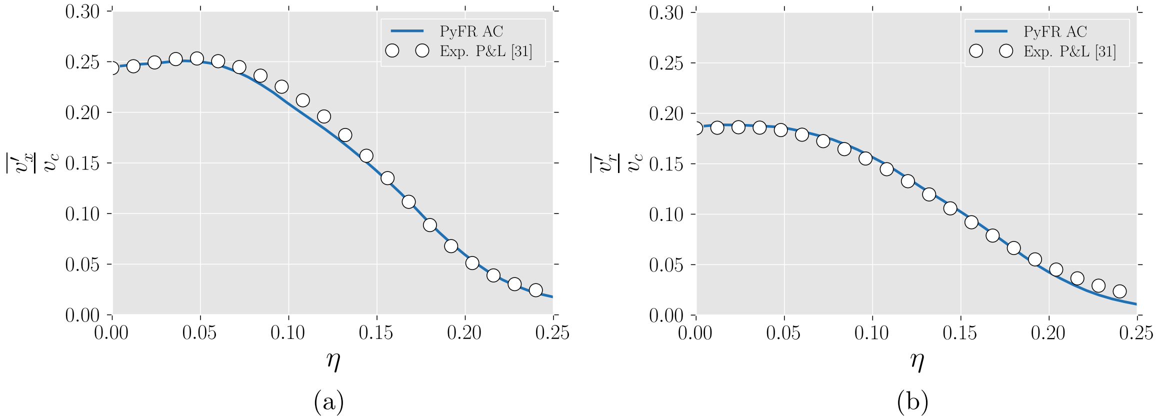


**Fig. 13.** Plot of (a) $\overline{v_x'}/v_c$ as a function of $\eta$, where $\overline{v_x'}$ is the root-mean-square stream-wise velocity fluctuation averaged over conical planes of constant $\eta$ in the region $10 \leq x \leq 30$. Plot of (b) $\overline{v_r'}/v_c$ as a function of $\eta$, where $\overline{v_r'}$ is the root-mean-square radial velocity fluctuation averaged over conical planes of constant $\eta$ in the region $10 \leq x \leq 30$.

applications in a range of sectors, including the maritime and automotive industries. Moreover, due to its cross-platform nature, the technology is well placed to remain relevant in an era of rapidly evolving hardware architectures.

## Acknowledgments

N.A. Loppi and P.E. Vincent would like to thank the Engineering and Physical Sciences Research Council and BAE Systems for their support via an Industrial CASE doctoral training grant and an Early Career Fellowship (EP/K027379/1). F.D. Witherden and A. Jameson would like to thank the Air Force Office of Scientific Research for their support via grant FA9550-14-1-0186. This work was also supported by compute allocations on Piz Daint at the Swiss National Supercomputing Centre under project ID S762 and Peta4-KNL at the Cambridge Service for Data Driven Discovery under project ID CS005.

## Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cpc.2018.06.016.

## References

[1] B.C. Vermeire, F.D. Witherden, P.E. Vincent, J. Comput. Phys. 334 (2017) 497–521.
[2] Z.J. Wang, K. Fidkowski, R. Abgrall, F. Bassi, D. Caraeni, A. Cary, H. Deconinck, R. Hartmann, K. Hillewaert, H.T. Huynh, N. Kroll, G. May, P.O. Persson, B. van Leer, M. Visbal, Internat. J. Numer. Methods Fluids 72 (8) (2013) 811–845.
[3] D. Gottlieb, S.A. Orszag, Numerical Analysis of Spectral Methods: Theory and Applications, Vol. 26, Siam, 1977.
[4] S.K. Lele, J. Comput. Phys. 103 (1) (1992) 16–42.
[5] W.H. Reed, T.R. Hill, Los Alamos Rep. LA-UR-73-479, 1973, p. 10.
[6] B. Cockburn, S. Hou, C.-W. Shu, Math. Comp. 52 (186) (1989) 411–435.
[7] B. Cockburn, S. Hou, C.-W. Shu, Math. Comp. 54 (190) (1990) 545–581.
[8] Y. Liu, M. Vinokur, Z.J. Wang, J. Comput. Phys. 216 (2) (2006) 780–801.
[9] Z.J. Wang, Y. Liu, G. May, A. Jameson, J. Sci. Comput. 32 (1) (2007) 45–71.
[10] D.A. Kopriva, J.H. Kolias, J. Comput. Phys. 125 (1) (1996) 244–261.
[11] H.T. Huynh, 18th AIAA Comp. Fluid Dyn. Conf., 25 - 28 June, Miami, FL, AIAA Pap. 2007-4079.
[12] A.J. Chorin, J. Comput. Phys. 135 (1967) 118–125.
[13] A. Jameson, 10th Comp. Fluid Dyn. Conf., 24 - 26 June, Honolulu, HI, AIAA Pap. 91-1596.
[14] F. Bassi, A. Crivellini, D.A. Di Pietro, S. Rebay, J. Comput. Phys. 218 (2) (2006) 794–815.
[15] C. Liang, A. Chan, X. Liu, A. Jameson, 49th AIAA Aerosp. Sci. Meet. Incl. New Horizons Forum Aerosp. Expo. 4 - 7 January, Orlando, FL, AIAA Pap. 2011-48, 2011.
[16] C. Cox, C. Liang, M.W. Plesniak, J. Comput. Phys. 314 (2016) 414–435.
[17] F.D. Witherden, A.M. Farrington, P.E. Vincent, Comput. Phys. Comm. 185 (11) (2014) 3028–3040.

[18] F.D. Witherden, B.C. Vermeire, P.E. Vincent, Comput. & Fluids 120 (2015) 173–186.
[19] P.E. Vincent, P. Castonguay, A. Jameson, J. Sci. Comput. 47 (1) (2011) 50–72.
[20] P.E. Vincent, P. Castonguay, A. Jameson, J. Comput. Phys. 230 (22) (2011) 8134–8154.
[21] A. Jameson, P.E. Vincent, P. Castonguay, J. Sci. Comput. 50 (2) (2012) 434–445.
[22] F. Witherden, P. Vincent, J. Sci. Comput. 61 (2) (2014) 398–423.
[23] L. Shunn, F. Ham, J. Comput. Appl. Math. 236 (17) (2012) 4348–4364.
[24] J.S. Hesthaven, T. Warburton, Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications, Springer Science & Business Media, 2007.
[25] D. Elsworth, E. Toro, NASA STI/Recon Tech. Rep. N, Vol. 94, 1992.
[26] K.J. Fidkowski, T.A. Oliver, J. Lu, D.L. Darmofal, J. Comput. Phys. 207 (1) (2005) 92–113.
[27] G. Karypis, V. Kumar, SIAM J. Sci. Comput. 20 (1) (1998) 359–392.
[28] B.D. Wozniak, F.D. Witherden, F.P. Russell, P.E. Vincent, P.H.J. Kelly, Comput. Phys. Comm. 202 (2016) 12–22.
[29] A. Heinecke, G. Henry, M. Hutchinson, H. Pabst, Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, 2016, p. 84.
[30] W.M. van Rees, A. Leonard, D.I. Pullin, P. Koumoutsakos, J. Comput. Phys. 230 (8) (2011) 2794–2805.
[31] N.R. Panchapakesan, J.L. Lumley, J. Fluid Mech. 246 (1993) 197–223.
[32] G. Lipari, P.K. Stansby, Flow Turbul. Combust. 87 (1) (2011) 79–114.
[33] B.J. Boersma, Fluid Dynam. Res. 35 (6) (2004) 425–447.
[34] C. Bogey, C. Bailly, J. Fluid Mech. 627 (2009) 129–160.