



# PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach<sup>☆</sup>



F.D. Witherden<sup>\*</sup>, A.M. Farrington, P.E. Vincent

Department of Aeronautics, Imperial College London, SW7 2AZ, United Kingdom

## ARTICLE INFO

### Article history:

Received 6 December 2013

Received in revised form

9 July 2014

Accepted 14 July 2014

Available online 23 July 2014

### Keywords:

High-order

Flux reconstruction

Parallel algorithms

Heterogeneous computing

## ABSTRACT

High-order numerical methods for unstructured grids combine the superior accuracy of high-order spectral or finite difference methods with the geometric flexibility of low-order finite volume or finite element schemes. The Flux Reconstruction (FR) approach unifies various high-order schemes for unstructured grids within a single framework. Additionally, the FR approach exhibits a significant degree of element locality, and is thus able to run efficiently on modern streaming architectures, such as Graphical Processing Units (GPUs). The aforementioned properties of FR mean it offers a promising route to performing affordable, and hence industrially relevant, scale-resolving simulations of hitherto intractable unsteady flows within the vicinity of real-world engineering geometries. In this paper we present PyFR, an open-source Python based framework for solving advection–diffusion type problems on streaming architectures using the FR approach. The framework is designed to solve a range of governing systems on mixed unstructured grids containing various element types. It is also designed to target a range of hardware platforms via use of an in-built domain specific language based on the Mako templating engine. The current release of PyFR is able to solve the compressible Euler and Navier–Stokes equations on grids of quadrilateral and triangular elements in two dimensions, and hexahedral elements in three dimensions, targeting clusters of CPUs, and NVIDIA GPUs. Results are presented for various benchmark flow problems, single-node performance is discussed, and scalability of the code is demonstrated on up to 104 NVIDIA M2090 GPUs. The software is freely available under a 3-Clause New Style BSD license (see [www.pyfr.org](http://www.pyfr.org)).

### Program summary

*Program title:* PyFR v0.1.0

*Catalogue identifier:* AETY\_v1\_0

*Program summary URL:* [http://cpc.cs.qub.ac.uk/summaries/AETY\\_v1\\_0.html](http://cpc.cs.qub.ac.uk/summaries/AETY_v1_0.html)

*Program obtainable from:* CPC Program Library, Queen's University, Belfast, N. Ireland

*Licensing provisions:* New style BSD license

*No. of lines in distributed program, including test data, etc.:* 12 733

*No. of bytes in distributed program, including test data, etc.:* 214 183

*Distribution format:* tar.gz

*Programming language:* Python, CUDA and C.

*Computer:* Variable, up to and including GPU clusters.

*Operating system:* Recent version of Linux/UNIX.

*RAM:* Variable, from hundreds of megabytes to gigabytes.

<sup>☆</sup> This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (<http://www.sciencedirect.com/science/journal/00104655>).

<sup>\*</sup> Corresponding author. Tel.: +44 02075945108.

E-mail addresses: [freddie.witherden08@imperial.ac.uk](mailto:freddie.witherden08@imperial.ac.uk), [freddie@witherden.org](mailto:freddie@witherden.org) (F.D. Witherden).

*Classification:* 6.5, 12.

*External routines:* Python 2.7, numpy, PyCUDA, mpi4py, SymPy, Mako

*Nature of problem:*

Compressible Euler and Navier–Stokes equations of fluid dynamics; potential for any advection–diffusion type problem.

*Solution method:*

High-order flux reconstruction approach suitable for curved, mixed, unstructured grids.

*Unusual features:*

Code makes extensive use of symbolic manipulation and runtime code generation through a domain specific language.

*Running time:*

Many small problems can be solved on a recent workstation in minutes to hours.

© 2014 The Authors. Published by Elsevier B.V.

This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/3.0/>).

## Nomenclature

Throughout we adopt a convention in which dummy indices on the right hand side of an expression are summed. For example  $C_{ijk} = A_{ijl}B_{ilk} \equiv \sum_l A_{ijl}B_{ilk}$  where the limits are implied from the surrounding context. All indices are assumed to be zero-based.

### Functions

$\delta_{ij}$	Kronecker delta
$\det \mathbf{A}$	Matrix determinant
$\dim \mathbf{A}$	Matrix dimensions

### Indices

$e$	Element type
$n$	Element number
$\alpha$	Field variable number
$i, j, k$	Summation indices
$\rho, \sigma, \nu$	Summation indices

### Domains

$\Omega$	Solution domain
$\Omega_e$	All elements in $\Omega$ of type $e$
$\hat{\Omega}_e$	A standard element of type $e$
$\partial \Omega_e$	Boundary of $\Omega_e$
$\Omega_{en}$	Element $n$ of type $e$ in $\Omega$
$ \Omega_e $	Number of elements of type $e$

### Expansions

$\wp$	Polynomial order
$N_D$	Number of spatial dimensions
$N_V$	Number of field variables
$\ell_{e\rho}$	Nodal basis polynomial $\rho$ for element type $e$
$x, y, z$	Physical coordinates
$\tilde{x}, \tilde{y}, \tilde{z}$	Transformed coordinates
$\mathcal{M}_{en}$	Transformed to physical mapping

### Adornments and suffixes

$\tilde{\square}$	A quantity in transformed space
$\hat{\square}$	A vector quantity of unit magnitude
$\square^T$	Transpose
$\square^{(u)}$	A quantity at a solution point
$\square^{(f)}$	A quantity at a flux point
$\square^{(f_\perp)}$	A normal quantity at a flux point

### Operators

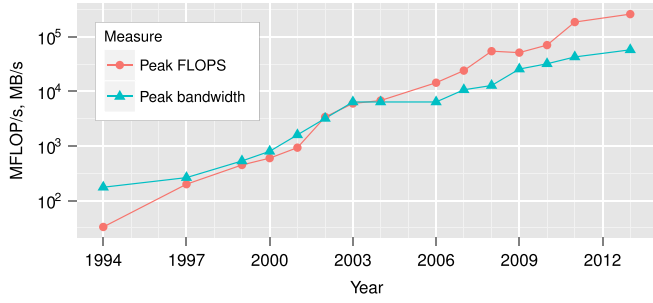
$\mathfrak{C}_\alpha$	Common solution at an interface
$\mathfrak{F}_\alpha$	Common normal flux at an interface

## 1. Introduction

There is an increasing desire amongst industrial practitioners of computational fluid dynamics (CFD) to undertake high-fidelity scale-resolving simulations of transient compressible flows within the vicinity of complex geometries. For example, to improve the design of next generation unmanned aerial vehicles (UAVs), there exists a need to perform simulations – at Reynolds numbers  $10^4$ – $10^7$  and Mach numbers  $M \sim 0.1$ – $1.0$  – of highly separated flow over deployed spoilers/air-brakes; separated flow within serpentine intake ducts; acoustic loading in weapons bays; and flow over entire UAV configurations at off-design conditions. Unfortunately, current generation industry-standard CFD software based on first- or second-order accurate Reynolds Averaged Navier–Stokes (RANS) approaches is not well suited to performing such simulations. Henceforth, there has been significant interest in the potential of high-order accurate methods for unstructured mixed grids, and whether they can offer an efficient route to performing scale-resolving simulations within the vicinity of complex geometries. Popular examples of high-order schemes for unstructured mixed grids include the discontinuous Galerkin (DG) method, first introduced by Reed and Hill [1], and the spectral difference (SD) methods originally proposed under the moniker ‘staggered-grid Chebyshev multidomain methods’ by Kopriva and Kolas in 1996 [2] and later popularised by Sun et al. [3]. In 2007 Huynh proposed the flux reconstruction (FR) approach [4]; a unifying framework for high-order schemes for unstructured grids that incorporates both the nodal DG schemes of [5] and, at least for a linear flux function, any SD scheme. In addition to offering high-order accuracy on unstructured mixed grids, FR schemes are also compact in space, and thus when combined with explicit time marching offer a significant degree of element locality. As such, explicit high-order FR schemes are characterised by a large degree of structured computation.

Over the past two decades improvements in the arithmetic capabilities of processors have significantly outpaced advances in random access memory. Algorithms which have traditionally been compute bound – such as dense matrix–vector products – are now limited instead by the bandwidth to/from memory. This is epitomised in Fig. 1. Whereas the processors of two decades ago had FLOPS-per-byte of  $\sim 0.2$  more recent chips have ratios upwards of  $\sim 4$ . This disparity is not limited to just conventional CPUs. Massively parallel accelerators and co-processors such as the NVIDIA K20X and Intel Xeon Phi 5110P have ratios of 5.24 and 3.16, respectively.

A concomitant of this disparity is that modern hardware architectures are highly dependent on a combination of high speed



**Fig. 1.** Trends in the peak floating point performance (double precision) and memory bandwidth of server-class Intel processors from 1994 to 2013. The quotient of these two measures yields the FLOPS-per-byte of a processor. Source: Data courtesy of Jan Treibig.

caches and/or shared memory to maintain throughput. However, for an algorithm to utilise these efficiently its memory access pattern must exhibit a degree of either spatial or temporal locality. To a first-order approximation the spatial locality of a method is inversely proportional to the amount of memory indirection. On an unstructured grid indirection arises whenever there is coupling between elements. This is potentially a problem for discretisations whose stencil is not compact. Coupling also arises in the context of implicit time stepping schemes. Implementations are therefore very often bound by memory bandwidth. As a secondary trend we note that the manner in which FLOPS are realised has also changed. In the early 1990s commodity CPUs were predominantly scalar with a single core of execution. However in 2013 processors with eight or more cores are not uncommon. Moreover, the cores on modern processors almost always contain vector processing units. Vector lengths up to 256 bits, which permit up to four double precision values to be operated on at once, are not uncommon. It is therefore imperative that compute-bound algorithms are amenable to both multithreading and vectorisation. A versatile means of accomplishing this is by breaking the computation down into multiple, necessarily independent, streams. By virtue of their independence these streams can be readily divided up between cores and vector lanes. This leads directly to the concept of *stream processing*. We will refer to architectures amenable to this form of parallelisation as streaming architectures.

A corollary of the above discussion is that compute intensive discretisations which can be formulated within the stream processing paradigm are well suited to acceleration on current – and likely future – hardware platforms. The FR approach combined with explicit time stepping is an archetypical of this.

Our objective in this paper is to present PyFR, an open-source Python based framework for solving advection–diffusion type problems on streaming architectures using the FR approach. The framework is designed to solve a range of governing systems on mixed unstructured grids containing various element types. It is also designed to target a range of hardware platforms via use of an in-built domain specific language derived from the Mako templating engine. The current release of PyFR is able to solve the compressible Euler and Navier–Stokes equations on unstructured grids of quadrilateral and triangular elements in two-dimensions, and unstructured grids of hexahedral elements in three-dimensions, targeting clusters of CPUs, and NVIDIA GPUs. The paper is structured as follows. In Section 2 we provide an overview of the FR approach for advection–diffusion type problems on mixed unstructured grids. In Section 3 we proceed to describe our implementation strategy, and in Section 4 we present the Euler and Navier–Stokes equations, which are solved by the current release of PyFR. The framework is then validated in Section 5, single-node performance is discussed in Section 6, and scalability of the code

is demonstrated on up to 104 NVIDIA M2090 GPUs in Section 7. Finally, conclusions are drawn in Section 8.

## 2. Flux reconstruction

A brief overview of the FR approach for solving advection–diffusion type problems is given below. Extended presentations can be found elsewhere [4,6–14].

Consider the following advection–diffusion problem inside an arbitrary domain  $\Omega$  in  $N_D$  dimensions

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

where  $0 \leq \alpha < N_V$  is the *field variable index*,  $u_\alpha = u_\alpha(\mathbf{x}, t)$  is a conserved quantity,  $\mathbf{f}_\alpha = \mathbf{f}_\alpha(u, \nabla u)$  is the flux of this conserved quantity and  $\mathbf{x} = x_i \in \mathbb{R}^{N_D}$ . In defining the flux we have taken  $u$  in its unscripted form to refer to all of the  $N_V$  field variables and  $\nabla u$  to be an object of length  $N_D \times N_V$  consisting of the gradient of each field variable. We start by rewriting Eq. (1) as a first order system

$$\frac{\partial u_\alpha}{\partial t} + \nabla \cdot \mathbf{f}_\alpha(u, \mathbf{q}) = 0, \quad (2a)$$

$$\mathbf{q}_\alpha - \nabla u_\alpha = 0, \quad (2b)$$

where  $\mathbf{q}$  is an auxiliary variable. Here, as with  $\nabla u$ , we have taken  $\mathbf{q}$  in its unscripted form to refer to the gradients of all of the field variables.

Take  $\mathcal{E}$  to be the set of available element types in  $\mathbb{R}^{N_D}$ . Examples include quadrilaterals and triangles in two dimensions and hexahedra, prisms, pyramids and tetrahedra in three dimensions. Consider using these various elements types to construct a conformal mesh of the domain such that

$$\Omega = \bigcup_{e \in \mathcal{E}} \Omega_e \quad \text{and} \quad \Omega_e = \bigcup_{n=0}^{|\Omega_e|-1} \Omega_{en} \quad \text{and} \quad \bigcap_{e \in \mathcal{E}} \bigcap_{n=0}^{|\Omega_e|-1} \Omega_{en} = \emptyset,$$

where  $\Omega_e$  refers to all of the elements of type  $e$  inside of the domain,  $|\Omega_e|$  is the number of elements of this type in the decomposition, and  $n$  is an index running over these elements with  $0 \leq n < |\Omega_e|$ . Inside each element  $\Omega_{en}$  we require that

$$\frac{\partial u_{en\alpha}}{\partial t} + \nabla \cdot \mathbf{f}_{en\alpha} = 0, \quad (3a)$$

$$\mathbf{q}_{en\alpha} - \nabla u_{en\alpha} = 0. \quad (3b)$$

It is convenient, for reasons of both mathematical simplicity and computational efficiency, to work in a transformed space. We accomplish this by introducing, for each element type, a standard element  $\hat{\Omega}_e$  which exists in a transformed space,  $\tilde{\mathbf{x}} = \tilde{x}_i$ . Next, assume the existence of a mapping function for each element such that

$$x_i = \mathcal{M}_{eni}(\tilde{\mathbf{x}}), \quad \mathbf{x} = \mathcal{M}_{en}(\tilde{\mathbf{x}}),$$

$$\tilde{x}_i = \mathcal{M}_{eni}^{-1}(\mathbf{x}), \quad \tilde{\mathbf{x}} = \mathcal{M}_{en}^{-1}(\mathbf{x}),$$

along with the relevant Jacobian matrices

$$\mathbf{J}_{en} = J_{enij} = \frac{\partial \mathcal{M}_{eni}}{\partial \tilde{x}_j}, \quad J_{en} = \det \mathbf{J}_{en},$$

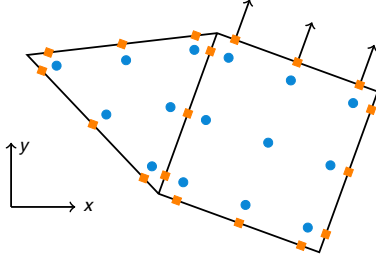
$$\mathbf{J}_{en}^{-1} = J_{enij}^{-1} = \frac{\partial \mathcal{M}_{eni}^{-1}}{\partial x_j}, \quad J_{en}^{-1} = \det \mathbf{J}_{en}^{-1} = \frac{1}{J_{en}}.$$

These definitions provide us with a means of transforming quantities to and from standard element space. Taking the transformed solution, flux, and gradients inside each element to be

$$\tilde{u}_{en\alpha} = \tilde{u}_{en\alpha}(\tilde{\mathbf{x}}, t) = J_{en}(\tilde{\mathbf{x}}) u_{en\alpha}(\mathcal{M}_{en}(\tilde{\mathbf{x}}), t), \quad (4a)$$

$$\tilde{\mathbf{f}}_{en\alpha} = \tilde{\mathbf{f}}_{en\alpha}(\tilde{\mathbf{x}}, t) = J_{en}(\tilde{\mathbf{x}}) \mathbf{J}_{en}^{-1}(\mathcal{M}_{en}(\tilde{\mathbf{x}})) \mathbf{f}_{en\alpha}(\mathcal{M}_{en}(\tilde{\mathbf{x}}), t), \quad (4b)$$

$$\tilde{\mathbf{q}}_{en\alpha} = \tilde{\mathbf{q}}_{en\alpha}(\tilde{\mathbf{x}}, t) = \mathbf{J}_{en}^T(\tilde{\mathbf{x}}) \mathbf{q}_{en\alpha}(\mathcal{M}_{en}(\tilde{\mathbf{x}}), t), \quad (4c)$$



**Fig. 2.** Solution points (blue circles) and flux points (orange squares) for a triangle and quadrangle in physical space. For the top edge of the quadrangle the normal vectors have been plotted. Observe how the flux points at the interface between the two elements are co-located.

and letting  $\tilde{\nabla} = \partial/\partial\tilde{x}_i$ , it can be readily verified that

$$\frac{\partial u_{en\alpha}}{\partial t} + J_{en}^{-1} \tilde{\nabla} \cdot \tilde{\mathbf{f}}_{en\alpha} = 0, \quad (5a)$$

$$\tilde{\mathbf{q}}_{en\alpha} - \tilde{\nabla} u_{en\alpha} = 0, \quad (5b)$$

as required. We note here the decision to multiply the first equation through by a factor of  $J_{en}^{-1}$ . Doing so has the effect of taking  $\tilde{u}_{en} \mapsto u_{en}$  which allows us to work in terms of the physical solution. This is more convenient from a computational standpoint.

We next proceed to associate a set of solution points with each standard element. For each type  $e \in \mathcal{E}$  take  $\{\tilde{\mathbf{x}}_{e\rho}^{(u)}\}$  to be the chosen set of points where  $0 \leq \rho < N_e^{(u)}(\wp)$ . These points can then be used to construct a nodal basis set  $\{\tilde{\ell}_{e\rho}^{(u)}(\tilde{\mathbf{x}})\}$  with the property that  $\tilde{\ell}_{e\rho}^{(u)}(\tilde{\mathbf{x}}_{e\sigma}^{(u)}) = \delta_{\rho\sigma}$ . To obtain such a set we first take  $\{\psi_{e\sigma}(\tilde{\mathbf{x}})\}$  to be any basis which spans a selected order  $\wp$  polynomial space defined inside  $\hat{\Omega}_e$ . Next we compute the elements of the generalised Vandermonde matrix  $\mathcal{V}_{e\rho\sigma} = \psi_{e\sigma}(\tilde{\mathbf{x}}_{e\rho}^{(u)})$ . With these a nodal basis set can be constructed as  $\tilde{\ell}_{e\rho}^{(u)}(\tilde{\mathbf{x}}) = \mathcal{V}_{e\rho\sigma}^{-1} \psi_{e\sigma}(\tilde{\mathbf{x}})$ . Along with the solution points inside of each element we also define a set of flux points on  $\partial\hat{\Omega}_e$ . We denote the flux points for a particular element type as  $\{\tilde{\mathbf{x}}_{e\rho}^{(f)}\}$  where  $0 \leq \rho < N_e^{(f)}(\wp)$ . Let the set of corresponding normalised outward-pointing normal vectors be given by  $\{\hat{\mathbf{n}}_{e\rho}^{(f)}\}$ . It is critical that each flux point pair along an interface share the same coordinates in physical space. For a pair of flux points  $e\rho n$  and  $e'\rho'n'$  at a non-periodic interface this can be formalised as  $\mathcal{M}_{en}(\tilde{\mathbf{x}}_{e\rho}^{(f)}) = \mathcal{M}_{e'n'}(\tilde{\mathbf{x}}_{e'\rho'}^{(f)})$ . A pictorial illustration of this can be seen in Fig. 2.

The first step in the FR approach is to go from the discontinuous solution at the solution points to the discontinuous solution at the flux points

$$u_{e\sigma n\alpha}^{(f)} = u_{e\rho n\alpha}^{(u)} \tilde{\ell}_{e\rho}^{(f)}(\tilde{\mathbf{x}}_{e\sigma}^{(f)}), \quad (6)$$

where  $u_{e\rho n\alpha}^{(u)}$  is an approximate solution of field variable  $\alpha$  inside of the  $n$ th element of type  $e$  at solution point  $\tilde{\mathbf{x}}_{e\rho}^{(u)}$ . This can then be used to compute a common solution

$$\mathcal{C}_\alpha u_{e\rho n\alpha}^{(f)} = \mathcal{C}_\alpha u_{e\rho n\alpha}^{(f)} = \mathcal{C}_\alpha(u_{e\rho n\alpha}^{(f)}, u_{e\rho n\alpha}^{(f)}), \quad (7)$$

where  $\mathcal{C}_\alpha(u_L, u_R)$  is a scalar function that given two values at a point returns a common value. Here we have taken  $\tilde{e\rho n}$  to be the element type, flux point number and element number of the adjoining point at the interface. Since grids in FR are permitted to be unstructured the relationship between  $e\rho n$  and  $\tilde{e\rho n}$  is indirect. This necessitates the use of a lookup table. As the common solution function is permitted to perform upwinding or downwinding of the solution it is in general the case that  $\mathcal{C}_\alpha(u_{e\rho n\alpha}^{(f)}, u_{e\rho n\alpha}^{(f)}) \neq \mathcal{C}_\alpha(u_{e\rho n\alpha}^{(f)}, u_{e\rho n\alpha}^{(f)})$ .

$u_{e\rho n\alpha}^{(f)}$ . Hence, it is important that each flux point pair only be visited once with the same common solution value assigned to both  $\mathcal{C}_\alpha u_{e\rho n\alpha}^{(f)}$  and  $\mathcal{C}_\alpha u_{e\rho n\alpha}^{(f)}$ .

Further, associated with each flux point is a vector correction function  $\mathbf{g}_{e\rho}^{(f)}(\tilde{\mathbf{x}})$  constrained such that

$$\hat{\mathbf{n}}_{e\sigma}^{(f)} \cdot \mathbf{g}_{e\rho}^{(f)}(\tilde{\mathbf{x}}_{e\sigma}^{(f)}) = \delta_{\rho\sigma}, \quad (8)$$

with a divergence that sits in the same polynomial space as the solution. Using these fields we can express the solution to Eq. (5b) as

$$\tilde{\mathbf{q}}_{e\sigma n\alpha}^{(u)} = \left[ \hat{\mathbf{n}}_{e\rho}^{(f)} \cdot \tilde{\nabla} \cdot \mathbf{g}_{e\rho}^{(f)}(\tilde{\mathbf{x}}) \{ \mathcal{C}_\alpha u_{e\rho n\alpha}^{(f)} - u_{e\rho n\alpha}^{(f)} \} + u_{e\rho n\alpha}^{(u)} \tilde{\nabla} \ell_{e\rho}^{(u)}(\tilde{\mathbf{x}}) \right]_{\tilde{\mathbf{x}}=\tilde{\mathbf{x}}_{e\sigma}^{(u)}}, \quad (9)$$

where the term inside the curly brackets is the ‘jump’ at the interface and the final term is an order  $\wp - 1$  approximation of the gradient obtained by differentiating the discontinuous solution polynomial. Following the approaches of Kopriva [15] and Sun et al. [3] we can now compute physical gradients as

$$\mathbf{q}_{e\sigma n\alpha}^{(u)} = \mathbf{J}_{e\sigma n}^{-T(u)} \tilde{\mathbf{q}}_{e\sigma n\alpha}^{(u)}, \quad (10)$$

$$\mathbf{q}_{e\sigma n\alpha}^{(f)} = \ell_{e\rho}^{(u)}(\tilde{\mathbf{x}}_{e\sigma}^{(f)}) \mathbf{q}_{e\rho n\alpha}^{(u)}, \quad (11)$$

where  $\mathbf{J}_{e\sigma n}^{-T(u)} = \mathbf{J}_{e\sigma n}^{-T}(\tilde{\mathbf{x}}_{e\sigma}^{(u)})$ . Having solved the auxiliary equation we are now able to evaluate the transformed flux

$$\tilde{\mathbf{f}}_{e\rho n\alpha}^{(u)} = J_{e\rho n}^{(u)} \mathbf{J}_{e\rho n}^{-1(u)} \tilde{\mathbf{f}}_{e\rho n\alpha}^{(u)}, \quad (12)$$

where  $J_{e\rho n}^{(u)} = \det \mathbf{J}_{en}(\tilde{\mathbf{x}}_{e\rho}^{(u)})$ . This can be seen to be a collocation projection of the flux. With this it is possible to compute the normal transformed flux at each of the flux points

$$\tilde{f}_{e\sigma n\alpha}^{(f)} = \ell_{e\rho}^{(u)}(\tilde{\mathbf{x}}_{e\sigma}^{(f)}) \hat{\mathbf{n}}_{e\sigma}^{(f)} \cdot \tilde{\mathbf{f}}_{e\rho n\alpha}^{(u)}. \quad (13)$$

Considering the physical normals at the flux points we see that

$$\mathbf{n}_{e\sigma n}^{(f)} = n_{e\sigma n}^{(f)} \hat{\mathbf{n}}_{e\sigma n}^{(f)} = \mathbf{J}_{e\sigma n}^{-T(f)} \hat{\mathbf{n}}_{e\sigma}^{(f)}, \quad (14)$$

which is the outward facing normal vector in physical space where  $n_{e\sigma n}^{(f)} > 0$  is defined as the magnitude. As the interfaces between two elements conform we must have  $\hat{\mathbf{n}}_{e\sigma n}^{(f)} = -\hat{\mathbf{n}}_{e'\sigma n'}^{(f)}$ . With these definitions we are now in a position to specify an expression for the common normal flux at a flux point pair as

$$\tilde{\mathcal{F}}_{e\sigma n\alpha}^{(f)} = -\tilde{\mathcal{F}}_{e'\sigma n'\alpha}^{(f)} = \tilde{\mathcal{F}}_\alpha(u_{e\sigma n\alpha}^{(f)}, u_{e'\sigma n'\alpha}^{(f)}, \mathbf{q}_{e\sigma n\alpha}^{(f)}, \mathbf{q}_{e'\sigma n'\alpha}^{(f)}, \hat{\mathbf{n}}_{e\sigma n}^{(f)}). \quad (15)$$

The relationship  $\tilde{\mathcal{F}}_{e\sigma n\alpha}^{(f)} = -\tilde{\mathcal{F}}_{e'\sigma n'\alpha}^{(f)}$  arises from the desire for the resulting numerical scheme to be conservative; a net outward flux from one element must be balanced by a corresponding inward flux on the adjoining element. It follows that  $\tilde{\mathcal{F}}_\alpha(u_L, u_R, \mathbf{q}_L, \mathbf{q}_R, \hat{\mathbf{n}}_L) = -\tilde{\mathcal{F}}_\alpha(u_R, u_L, \mathbf{q}_R, \mathbf{q}_L, -\hat{\mathbf{n}}_L)$ . The common normal fluxes in Eq. (15) can now be taken into transformed space via

$$\tilde{\mathcal{F}}_{e\sigma n\alpha}^{(f)} = J_{e\sigma n}^{(f)} n_{e\sigma n}^{(f)} \tilde{\mathcal{F}}_{e\sigma n\alpha}^{(f)}, \quad (16)$$

$$\tilde{\mathcal{F}}_{e\sigma n\alpha}^{(f)} = J_{e\sigma n}^{(f)} n_{e\sigma n}^{(f)} \tilde{\mathcal{F}}_{e\sigma n\alpha}^{(f)}, \quad (17)$$

where  $J_{e\sigma n}^{(f)} = \det \mathbf{J}_{en}(\tilde{\mathbf{x}}_{e\sigma}^{(f)})$ .

It is now possible to compute an approximation for the divergence of the continuous flux. The procedure is directly analogous to the one used to calculate the transformed gradient in Eq. (9)

$$(\tilde{\nabla} \cdot \tilde{\mathbf{f}})_{e\rho n\alpha}^{(u)} = \left[ \tilde{\nabla} \cdot \mathbf{g}_{e\rho}^{(f)}(\tilde{\mathbf{x}}) \{ \tilde{\mathcal{F}}_{e\sigma n\alpha}^{(f)} - \tilde{f}_{e\sigma n\alpha}^{(f)} \} + \tilde{\mathbf{f}}_{e\rho n\alpha}^{(u)} \cdot \tilde{\nabla} \ell_{e\rho}^{(u)}(\tilde{\mathbf{x}}) \right]_{\tilde{\mathbf{x}}=\tilde{\mathbf{x}}_{e\rho}^{(u)}}, \quad (18)$$

which can then be used to obtain a semi-discretised form of the governing system

$$\frac{\partial u_{e\rho n\alpha}^{(u)}}{\partial t} = -J_{e\rho n}^{-1(u)} (\tilde{\nabla} \cdot \tilde{\mathbf{f}})_{e\rho n\alpha}^{(u)}, \quad (19)$$

where  $J_{e\rho n}^{-1(u)} = \det \mathbf{J}_{en}^{-1}(\tilde{\mathbf{x}}_{e\rho}^{(u)}) = 1/J_{e\rho n}^{(u)}$ .



This semi-discretised form is simply a system of ordinary differential equations in  $t$  and can be solved using one of a number of schemes, e.g. a classical fourth order Runge–Kutta (RK4) scheme.

### 3. Implementation

#### 3.1. Overview

PyFR is a Python based implementation of the FR approach described in Section 2. It is designed to be compact, efficient, and platform portable. Key functionality is summarised in Table 1.

The majority of operations within an FR step can be cast in terms of matrix–matrix multiplications, as detailed in Appendix A. All remaining operations (e.g. flux evaluations) are point-wise, concerning themselves with either a single solution point inside of an element or two collocating flux points at an interface. Hence, in broad terms, there are five salient aspects of an FR implementation, specifically (i) definition of the constant operator matrices detailed in Appendix A, (ii) specification of the state matrices detailed in Appendix A, (iii) implementation of matrix multiply kernels, (iv) implementation of point-wise kernels, and finally (v) handling of distributed memory parallelism and scheduling of kernel invocations. Details regarding how each of the above were achieved in PyFR are presented below.

#### 3.2. Definition of constant operator matrices

Setup of the seven constant operator matrices detailed in Appendix A requires evaluation of various polynomial expressions, and their derivatives, at solution/flux points within each type of standard element. Although conceptually simple, such operations can be cumbersome to code. To keep the codebase compact PyFR makes extensive use of symbolic manipulation via SymPy [16], which brings computer algebra facilities similar to those found in Maple and Mathematica to Python. SymPy has built-in support for most common polynomials and can readily evaluate such expressions to arbitrary precision. Efficiency of the setup phase is not critical, since the operations are only performed once at start-up. Since efficiency is not critical, platform portability is effectively achieved by running such operations on the host CPU in all cases.

#### 3.3. Specification of state matrices

In specifying the state matrices detailed in Appendix A there is a degree of freedom regarding how the field variables of each element are packed along a row. The packing of field variables can be characterised by considering the distance,  $\Delta j$  (in columns) between two subsequent field variables for a given element. The case of  $\Delta j = 1$  corresponds to the array of structures (AoS) packing whereas the choice of  $\Delta j = |\Omega_e|$  leads to the structure of arrays (SoA) packing. A hybrid approach wherein  $\Delta j = k$  with  $k$  being constant results in the AoSoA( $k$ ) approach. An implementation is free to choose between any of these counting patterns so long as it is consistent. For simplicity PyFR uses the SoA packing order across all platforms.

#### 3.4. Matrix multiplication kernels

PyFR defers matrix multiplication to the GEMM family of sub-routines provided a suitable Basic Linear Algebra Subprograms (BLAS) library. BLAS is available for virtually all platforms and optimised versions are often maintained by the hardware vendors themselves (e.g. cuBLAS for NVIDIA GPUs). This approach greatly facilitates development of efficient and platform portable code. We note, however, that the matrix sizes encountered in PyFR are

**Table 1**  
Key functionality of PyFR.

Dimensions	2D, 3D
Elements	Triangles, Quadrilaterals, Hexahedra
Spatial orders	Arbitrary
Time steppers	Euler, RK4, DOPRI5
Precisions	Single, Double
Platforms	CPUs via C/OpenMP, NVIDIA GPUs via CUDA
Communication	MPI
Governing systems	Euler, Compressible Navier–Stokes

```

1 <%pyfr:kernel name='negdivconf' ndim='2'
2           tdivtconf='inout fptype_t[${str(nvars)}]
3           rcpdjac='in fptype_t'>
4 % for i in range(nvars):
5     tdivtconf[${i}] *= -rcpdjac;
6 % endfor
7 </%pyfr:kernel>

```

**Fig. 3.** An example of an extrinsic kernel in PyFR. The template variable  $nvars$  is taken to be the number of field variables,  $N_v$ . The kernel arguments  $tdivtconf$  and  $rcpdjac$  correspond to  $\tilde{\nabla} \cdot \mathbf{f}$  and  $J^{-1}$  respectively with the operation being performed in-place.

not necessarily optimal from a GEMM perspective. Specifically, GEMM is optimised for the multiplication of large square matrices, whereas the constant operator matrices in PyFR are ‘small and square’ with 10–100 rows/columns, and the state matrices are ‘short and fat’ with 10–100 rows and 10 000–100 000 columns. Moreover, we note that the constant operator matrices are known *a priori*, and do not change in time. This *a priori* knowledge could, in theory, be leveraged to design bespoke matrix multiply kernels that are more efficient than GEMM. Development of such bespoke kernels will be a topic of future research.

#### 3.5. Point-wise kernels

Point-wise kernels are specified using a domain specific language implemented in PyFR atop of the Mako templating engine [17]. The templated kernels are then interpreted at runtime, converted to low-level code, compiled, linked and loaded. Currently the templating engine can generate C/OpenMP to target CPUs, and CUDA (via the PyCUDA wrapper [18]) to target NVIDIA GPUs. The use of a domain specific language avoids implementation of each point-wise kernel for each target platform; keeping the codebase compact and platform portable. Runtime code generation also means it is possible to instruct the compiler to emit binaries which are optimised for the current hardware architecture. Such optimisations can result in anything up to a fourfold improvement in performance when compared with architectural defaults.

As an example of a point-wise kernel we consider the evaluation of the right hand side of Eq. (19), which reads  $-J_{\rho n}^{-1(u)}(\tilde{\nabla} \cdot \mathbf{f})_{\rho n \alpha}^{(u)}$ . The operation consists of a point-wise multiplication between the negative reciprocal of the Jacobian and the transformed divergence of the flux at each solution point. Fig. 3 shows how such a kernel can be expressed in the domain specific language of PyFR. There are several points of note. Firstly, the kernel is purely scalar in nature. This is by design; in PyFR point-wise kernels need only prescribe the point-wise operation to be applied. Important choices such as how to vectorise a given operation or how to gather data from memory are all delegated to templating engine. Secondly, we note it is possible to utilise Python when generating the main body of kernels. This capability is showcased on lines four, five and six where it is used to unroll a for loop over each of the field variables. Finally, we also highlight the use of an abstract data type  $fptype\_t$  for floating point variables which permits a single set of kernels to be used for both single and double precision operation. Generated CUDA source for this kernel can be seen in Fig. 4, and the equivalent C kernel can be found in Fig. 5.

### 3.6. Distributed memory parallelism and scheduling

PyFR is capable of operating on high performance computing clusters utilising distributed memory parallelism. This is accomplished through the Message Passing Interface (MPI). All MPI functionality is implemented at the Python level through the mpi4py [19] wrapper. To enhance the scalability of the code care has been taken to ensure that all requests are persistent, point-to-point and non-blocking. Further, the format of data that is shared between ranks has been made backend independent. It is therefore possible to deploy PyFR on heterogeneous clusters consisting of both conventional CPUs and accelerators.

The arrangement of kernel calls required to solve an advection–diffusion problem can be seen in Fig. 6. Our primary objective when scheduling kernels was to maximise the potential for overlapping communication with computation. In order to help achieve this the common interface solution,  $\mathbf{c}_\alpha$ , and common interface flux,  $\mathbf{f}_\alpha$ , kernels have been broken apart into two separate kernels; suffixed in the figure by INT and MPI. PyFR is therefore able to perform a significant degree of rank-local computation while the relevant ghost states are being exchanged.

Our secondary objective when scheduling kernels was to minimise the amount of temporary storage required during the evaluation of  $-\nabla \cdot \mathbf{f}$ . Such optimisations are critical within the context of accelerators which often have an order of magnitude less memory than a contemporary platform. In order to help achieve this  $\mathbf{U}^{(u)}$ ,  $\mathbf{R}^{(u)}$ , and  $\mathbf{R}^{(u)}$  are allowed to alias. By permitting the same storage location to be used for both the inputted solution and the outputted flux divergence it is possible to reduce the storage requirements of the RK schemes. Another opportunity for memory reuse is in the transformed flux function where the incoming gradients,  $\mathbf{Q}^{(u)}$ , can be overwritten with the transformed flux,  $\mathbf{F}^{(u)}$ . A similar approach can be used in the common interface flux function whereby  $\mathbf{U}^{(f)}$  can be updated in-place with the entries of  $\mathbf{D}^{(f)}$  which holds the transformed common normal flux. Moreover,  $\mathbf{C}^{(f)}$  is also able to utilise the same storage as the somewhat larger  $\mathbf{Q}^{(f)}$  array. These optimisations allow PyFR to process over 100 000 curved, unstructured, hexahedral elements at  $\varphi = 3$  inside of a 5 GiB memory footprint.

## 4. Governing systems

### 4.1. Overview

PyFR is a framework for solving various advection–diffusion type problems. In the current release of PyFR two specific governing systems can be solved, specifically the Euler equations for inviscid compressible flow, and the compressible Navier–Stokes equations for viscous compressible flow. Details regarding both are given below.

### 4.2. Euler equations

Using the framework introduced in Section 2 the three dimensional Euler equations can be expressed in conservative form as

$$\mathbf{u} = \begin{Bmatrix} \rho \\ \rho v_x \\ \rho v_y \\ \rho v_z \\ E \end{Bmatrix}, \quad (20)$$

$$\mathbf{f} = \mathbf{f}^{(\text{inv})} = \begin{Bmatrix} \rho v_x & \rho v_y & \rho v_z \\ \rho v_x^2 + p & \rho v_x v_y & \rho v_x v_z \\ \rho v_x v_y & \rho v_y^2 + p & \rho v_y v_z \\ \rho v_x v_z & \rho v_y v_z & \rho v_z^2 + p \\ v_x(E + p) & v_y(E + p) & v_z(E + p) \end{Bmatrix},$$

```

1  global void
2  negdivconf(int _ny, int _nx,
3             const fpdtype_t* __restrict__ rcpdjac_v,
4             int ldrcpdjac,
5             fpdtype_t* __restrict__ tdivtconf_v,
6             int ldtdivtconf)
7  {
8      int _x = blockIdx.x*blockDim.x + threadIdx.x;
9
10     for (int _y = 0; _y < _ny && _x < _nx; ++_y)
11     {
12         fpdtype_t rcpdjac, tdivtconf[4];
13
14         // Load rcpdjac
15         rcpdjac = rcpdjac_v[ldrcpdjac*_y + _x];
16
17         // Load tdivtconf
18         tdivtconf[0] = tdivtconf_v[ldtdivtconf*_y + _x + _nx*0];
19         tdivtconf[1] = tdivtconf_v[ldtdivtconf*_y + _x + _nx*1];
20         tdivtconf[2] = tdivtconf_v[ldtdivtconf*_y + _x + _nx*2];
21         tdivtconf[3] = tdivtconf_v[ldtdivtconf*_y + _x + _nx*3];
22
23         tdivtconf[0] *= -rcpdjac;
24         tdivtconf[1] *= -rcpdjac;
25         tdivtconf[2] *= -rcpdjac;
26         tdivtconf[3] *= -rcpdjac;
27
28         // Store tdivtconf
29         tdivtconf_v[ldtdivtconf*_y + _x + _nx*0] = tdivtconf[0];
30         tdivtconf_v[ldtdivtconf*_y + _x + _nx*1] = tdivtconf[1];
31         tdivtconf_v[ldtdivtconf*_y + _x + _nx*2] = tdivtconf[2];
32         tdivtconf_v[ldtdivtconf*_y + _x + _nx*3] = tdivtconf[3];
33     }
34 }

```

Fig. 4. Generated CUDA source for the template in Fig. 3 for when  $N_V = 4$ .

with  $\mathbf{u}$  and  $\mathbf{f}$  together satisfying Eq. (1). In the above  $\rho$  is the mass density of the fluid,  $\mathbf{v} = (v_x, v_y, v_z)^T$  is the fluid velocity vector,  $E$  is the total energy per unit volume and  $p$  is the pressure. For a perfect gas the pressure and total energy can be related by the ideal gas law

$$E = \frac{p}{\gamma - 1} + \frac{1}{2} \rho \|\mathbf{v}\|^2, \quad (21)$$

where  $\gamma = c_p/c_v$ .

With the fluxes specified all that remains is to prescribe a method for computing the common normal flux,  $\mathbf{f}_\alpha$ , at interfaces as defined in Eq. (15). This can be accomplished using an approximate Riemann solver for the Euler equations. There exist a variety of such solvers as detailed in [20]. A description of those implemented in PyFR can be found in Appendix B.

### 4.3. Compressible Navier–Stokes equations

The compressible Navier–Stokes equations can be viewed as an extension of the Euler equations via the inclusion of viscous terms. Within the framework outlined above the flux now takes the form of  $\mathbf{f} = \mathbf{f}^{(\text{inv})} - \mathbf{f}^{(\text{vis})}$  where

$$\mathbf{f}^{(\text{vis})} = \begin{Bmatrix} 0 & 0 & 0 \\ \mathcal{T}_{xx} & \mathcal{T}_{yx} & \mathcal{T}_{zx} \\ \mathcal{T}_{xy} & \mathcal{T}_{yy} & \mathcal{T}_{zy} \\ \mathcal{T}_{xz} & \mathcal{T}_{yz} & \mathcal{T}_{zz} \\ v_i \mathcal{T}_{ix} + \Delta \partial_x T & v_i \mathcal{T}_{iy} + \Delta \partial_y T & v_i \mathcal{T}_{iz} + \Delta \partial_z T \end{Bmatrix}. \quad (22)$$

In the above we have defined  $\Delta = \mu c_p / P_r$  where  $\mu$  is the dynamic viscosity and  $P_r$  is the Prandtl number. The components of the stress–energy tensor are given by

$$\mathcal{T}_{ij} = \mu (\partial_i v_j + \partial_j v_i) - \frac{2}{3} \mu \delta_{ij} \nabla \cdot \mathbf{v}. \quad (23)$$

Using the ideal gas law the temperature can be expressed as

$$T = \frac{1}{c_v} \frac{1}{\gamma - 1} \frac{p}{\rho}, \quad (24)$$

with partial derivatives thereof being given according to the quotient rule.

```

1 static PYFR_NOINLINE void
2 negdivconf_inner(int _nx,
3                 const fptype_t * restrict rcpdjac_v,
4                 fptype_t * restrict tdivtconf_v0,
5                 fptype_t * restrict tdivtconf_v1,
6                 fptype_t * restrict tdivtconf_v2,
7                 fptype_t * restrict tdivtconf_v3)
8 {
9     PYFR_ALIGNED(rcpdjac_v);
10    PYFR_ALIGNED(tdivtconf_v0);
11    PYFR_ALIGNED(tdivtconf_v1);
12    PYFR_ALIGNED(tdivtconf_v2);
13    PYFR_ALIGNED(tdivtconf_v3);
14
15    for (int _x = 0; _x < _nx; _x++)
16    {
17        fptype_t rcpdjac, tdivtconf[4];
18
19        // Load rcpdjac
20        rcpdjac = rcpdjac_v[_x];
21
22        // Load tdivtconf
23        tdivtconf[0] = tdivtconf_v0[_x];
24        tdivtconf[1] = tdivtconf_v1[_x];
25        tdivtconf[2] = tdivtconf_v2[_x];
26        tdivtconf[3] = tdivtconf_v3[_x];
27
28        tdivtconf[0] *= -rcpdjac;
29        tdivtconf[1] *= -rcpdjac;
30        tdivtconf[2] *= -rcpdjac;
31        tdivtconf[3] *= -rcpdjac;
32
33        // Store tdivtconf
34        tdivtconf_v0[_x] = tdivtconf[0];
35        tdivtconf_v1[_x] = tdivtconf[1];
36        tdivtconf_v2[_x] = tdivtconf[2];
37        tdivtconf_v3[_x] = tdivtconf[3];
38    }
39 }
40
41 void
42 negdivconf(int _ny, int _nx,
43           const fptype_t * restrict rcpdjac_v,
44           int lsdrpdjac,
45           fptype_t * restrict tdivtconf_v,
46           int lsdtdivtconf)
47 {
48     #pragma omp parallel for
49     for (int _y = 0; _y < _ny; _y++)
50         negdivconf_inner(_nx, rcpdjac_v + _y*lsdrpdjac,
51                         tdivtconf_v + (_y*4 + 0)*lsdtdivtconf,
52                         tdivtconf_v + (_y*4 + 1)*lsdtdivtconf,
53                         tdivtconf_v + (_y*4 + 2)*lsdtdivtconf,
54                         tdivtconf_v + (_y*4 + 3)*lsdtdivtconf);
55 }

```

**Fig. 5.** Generated OpenMP annotated C source code for the template in Fig. 3 for when  $N_V = 4$ . The somewhat unconventional structure is necessary to ensure that the kernel is properly vectorised across a range of compilers.

Since the Navier–Stokes equations are an advection–diffusion type system it is necessary to both compute a common solution ( $\mathcal{C}_\alpha$  of Eq. (7)) at element boundaries and augment the inviscid Riemann solver to handle the viscous part of the flux. A popular approach is the LDG method as presented in [5,13]. In this approach the common solution is given  $\forall \alpha$  according to

$$\mathcal{C}(u_L, u_R) = (\frac{1}{2} - \beta)u_L + (\frac{1}{2} + \beta)u_R, \quad (25)$$

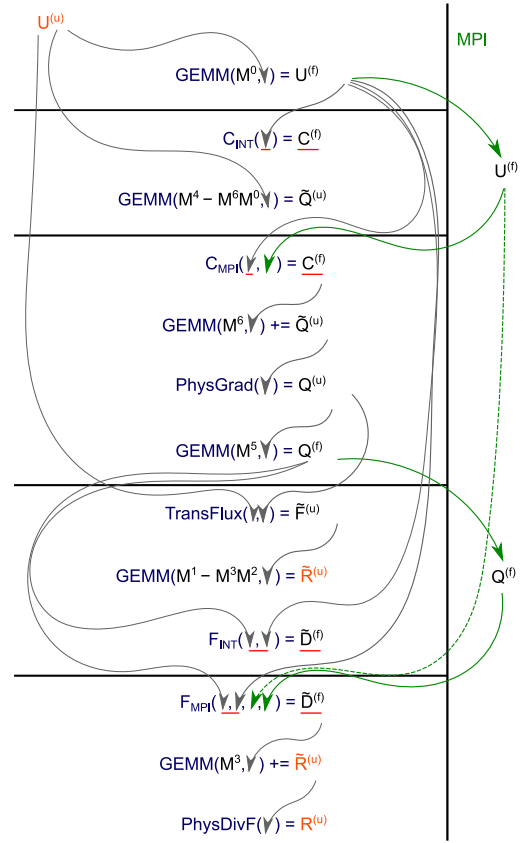
where  $\beta$  controls the degree of upwinding/downwinding. The common normal interface flux is then prescribed, once again  $\forall \alpha$ , according to

$$\mathcal{F}(u_L, u_R, \mathbf{q}_L, \mathbf{q}_R, \hat{\mathbf{n}}_L) = \mathcal{F}^{(\text{inv})} - \mathcal{F}^{(\text{vis})}, \quad (26)$$

where  $\mathcal{F}^{(\text{inv})}$  is a suitable inviscid Riemann solver (see Appendix B) and

$$\mathcal{F}^{(\text{vis})} = \hat{\mathbf{n}}_L \cdot \left\{ (\frac{1}{2} + \beta) \mathbf{f}_L^{(\text{vis})} + (\frac{1}{2} - \beta) \mathbf{f}_R^{(\text{vis})} \right\} + \tau(u_L - u_R), \quad (27)$$

with  $\tau$  being a penalty parameter,  $\mathbf{f}_L^{(\text{vis})} = \mathbf{f}^{(\text{vis})}(u_L, \mathbf{q}_L)$ , and  $\mathbf{f}_R^{(\text{vis})} = \mathbf{f}^{(\text{vis})}(u_R, \mathbf{q}_R)$ . We observe here that if the common solution is upwinded then the common normal flux will be downwinded. Generally,  $\beta = \pm 1/2$  as this results in the numerical scheme having a compact stencil and  $0 \leq \tau \leq 1$ .



**Fig. 6.** Flow diagram showing the stages required to compute  $-\nabla \cdot \mathbf{f}$ . Symbols correspond to those of Appendix A. For simplicity arguments referencing constant data have been omitted. Memory indirection is indicated by red underlines. Synchronisation points are signified by black horizontal lines. Dotted lines correspond to data reuse.

#### 4.3.1. Presentation in two dimensions

The above prescriptions of the Euler and Navier–Stokes equations are valid for the case of  $N_D = 3$ . The two dimensional formulation can be recovered by deleting the fourth rows in the definitions of  $u$ ,  $\mathbf{f}^{(\text{inv})}$  and  $\mathbf{f}^{(\text{vis})}$  along with the third columns of  $\mathbf{f}^{(\text{inv})}$  and  $\mathbf{f}^{(\text{vis})}$ . Vectors are now two dimensional with the velocity being given by  $\mathbf{v} = (v_x, v_y)^T$ .

## 5. Validation

### 5.1. Euler equations: Euler vortex super accuracy

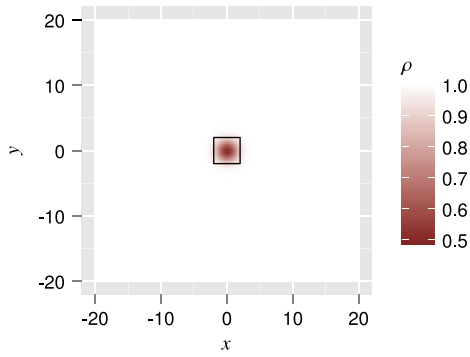
Various authors [4,10] have shown FR schemes exhibit so-called ‘super accuracy’ (an order of accuracy greater than the expected  $\varphi + 1$ ). To confirm PyFR can achieve super accuracy for the Euler equations a square domain  $\Omega = [-20, 20]^2$  was decomposed into four structured quad meshes with spacings of  $h = 1/3$ ,  $h = 2/7$ ,  $h = 1/4$ , and  $h = 2/9$ . Initial conditions were taken to be those of an isentropic Euler vortex in a free-stream

$$\rho(\mathbf{x}, t = 0) = \left\{ 1 - \frac{S^2 M^2 (\gamma - 1) \exp 2f}{8\pi^2} \right\}^{\frac{1}{\gamma-1}}, \quad (28)$$

$$\mathbf{v}(\mathbf{x}, t = 0) = \frac{S y \exp f}{2\pi R} \hat{\mathbf{x}} + \left\{ 1 - \frac{S x \exp f}{2\pi R} \right\} \hat{\mathbf{y}}, \quad (29)$$

$$p(\mathbf{x}, t = 0) = \frac{\rho^\gamma}{\gamma M^2}, \quad (30)$$

where  $f = (1 - x^2 - y^2)/2R^2$ ,  $S = 13.5$  is the strength of the vortex,  $M = 0.4$  is the free-stream Mach number, and  $R = 1.5$



**Fig. 7.** Initial density profile for the vortex in  $\Omega$ . The black box shows the region where the error is calculated.

is the radius. All meshes were configured with periodic boundary conditions along boundaries of constant  $x$ . Along boundaries of constant  $y$  the dynamical variables were fixed according to

$$\begin{aligned} \rho(\mathbf{x} = x\hat{\mathbf{x}} \pm 20\hat{\mathbf{y}}, t) &= 1, \\ \mathbf{v}(\mathbf{x} = x\hat{\mathbf{x}} \pm 20\hat{\mathbf{y}}, t) &= \hat{\mathbf{y}}, \\ p(\mathbf{x} = x\hat{\mathbf{x}} \pm 20\hat{\mathbf{y}}, t) &= \frac{1}{\gamma M^2}, \end{aligned}$$

which are simply the limiting values of the initial conditions. Strictly speaking these conditions, on account of the periodicity, result in the modelling of an infinite array of coupled vortices. The impact of this is mitigated by the observation that the exponentially decaying vortex has a characteristic radius which is far smaller than the extent of the domain. Neglecting these effects the analytic solution of the system in a time  $t$  is simply a translation of the initial conditions.

Using the analytical solution we can define an  $L^2$  error as

$$\sigma(t)^2 = \int_{-2}^2 \int_{-2}^2 [\rho^\delta(\mathbf{x} + \Delta_y(t)\hat{\mathbf{y}}, t) - \rho(\mathbf{x}, t=0)]^2 d^2\mathbf{x}, \quad (31)$$

where  $\rho^\delta(\mathbf{x}, t)$  is the numerical mass density,  $\rho(\mathbf{x}, t=0)$  the analytic mass density, and  $\Delta_y(t)$  is the ordinate corresponding to the centre of the vortex at a time  $t$  and accounts for the fact that the vortex is translating in a free stream velocity of unity in the  $y$  direction. Restricting the region of consideration to a small box centred around the origin serves to further mitigate against the effects of vortices coupling together. The initial mass density along with the  $[-2, -2] \times [2, 2]$  region used to evaluate the error can be seen in Fig. 7. At times,  $t_c$ , when the vortex is centred on the box the error can be readily computed by integrating over each element inside the box and summing the residuals

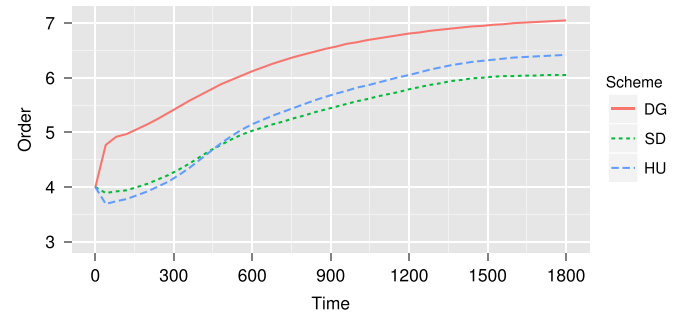
$$\sigma(t_c)^2 = \iint_{\hat{\Omega}_e} [\rho_i^\delta(\tilde{\mathbf{x}}, t_c) - \rho(\mathcal{M}_i(\tilde{\mathbf{x}}), 0)]^2 J_i(\tilde{\mathbf{x}}) d^2\tilde{\mathbf{x}}, \quad (32)$$

where,  $\rho_i^\delta(\tilde{\mathbf{x}}, t_c)$  is the approximate mass density inside of the  $i$ th element, and  $J_i(\tilde{\mathbf{x}})$  the associated Jacobian. These integrals can be approximated by applying Gaussian quadrature

$$\begin{aligned} \sigma(t_c)^2 &\approx J_i(\tilde{\mathbf{x}}_j) [\rho_i^\delta(\tilde{\mathbf{x}}_j, t_c) - \rho(\mathcal{M}_i(\tilde{\mathbf{x}}_j), 0)]^2 \omega_j \\ &= \frac{h^2}{4} [\rho_i^\delta(\tilde{\mathbf{x}}_j, t_c) - \rho(\mathcal{M}_i(\tilde{\mathbf{x}}_j), 0)]^2 \omega_j, \end{aligned} \quad (33)$$

where  $\{\tilde{\mathbf{x}}_j\}$  are abscissa and  $\{\omega_j\}$  the weights of a rule determined for integration inside of a standard quadrilateral. So long as the rule used is of a suitable strength then this will be a very good approximation of the true  $L^2$  error.

Following [10] the initial conditions were laid onto the mesh using a collocation projection with  $\wp = 3$ . The simulation was



**Fig. 8.** Spatial super accuracy observed for a  $\wp = 3$  simulation using DG, SD and HU as defined in [10].

then run with three different flux reconstruction schemes: DG, SD, and HU as defined in [10]. Solution points were placed at a tensor product construction of Gauss–Legendre quadrature points. Common interface fluxes were computed using a Rusanov Riemann solver. To advance the solutions in time a classical fourth order Runge–Kutta method (RK4) was used. The time step was taken to be  $\Delta t = 0.00125$  with  $t = 0..1800$  with solutions written out to disk every 32 000 steps. The order of accuracy of the scheme at a particular time can be determined by plotting  $\log \sigma$  against  $\log h$  and performing a least-squares fit through the four data points. The order is given by the gradient of the fit. A plot of order of accuracy against time for the three schemes can be seen in Fig. 8. We note that the order of accuracy changes as a function of time. This is due to the fact that the error is actually of the form  $\sigma(t) = \sigma_p + \sigma_{so}(t)$  where  $\sigma_p$  is a constant projection error and  $\sigma_{so}$  is a time-dependent spatial operator error. The projection error arises as a consequence of the fourth order collocation projection of the initial conditions onto the mesh. Over time the spatial operator error grows in magnitude and eventually dominates. Only when  $\sigma_{so}(t) \gg \sigma_p$  can the true order of the method be observed. The results here can be seen to be in excellent agreement with those of [10].

## 5.2. Compressible Navier–Stokes equations: Couette flow

Consider the case in which two parallel plates of infinite extent are separated by a distance  $H$  in the  $y$  direction. We treat both plates as isothermal walls at a temperature  $T_w$  and permit the top plate to move at a velocity  $v_w$  in the  $x$  direction with respect to the bottom plate. For simplicity we shall take the ordinate of the bottom plate as zero. In the case of a constant viscosity  $\mu$  the Navier–Stokes equations admit an analytical solution in which

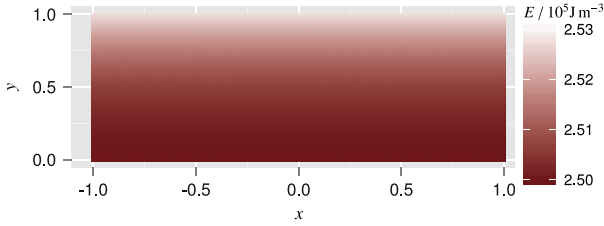
$$\rho(\phi) = \frac{\gamma}{\gamma - 1} \frac{2p}{2c_p T_w + P_r v_w^2 \phi(1 - \phi)}, \quad (34)$$

$$\mathbf{v}(\phi) = v_w \phi \hat{\mathbf{x}}, \quad (35)$$

$$p = p_c, \quad (36)$$

where  $\phi = y/H$  and  $p_c$  is a constant pressure. The total energy is given by the ideal gas law of Eq. (21). On a finite domain the Couette flow problem can be modelled through the imposition of periodic boundary conditions. For a two dimensional mesh periodicity is enforced in  $x$  whereas for three dimensional meshes it is enforced in both  $x$  and  $z$ . To validate the Navier–Stokes solver in PyFR we take  $\gamma = 1.4$ ,  $P_r = 0.72$ ,  $\mu = 0.417$ ,  $c_p = 1005 \text{ J K}^{-1}$ ,  $H = 1 \text{ m}$ ,  $T_w = 300 \text{ K}$ ,  $p_c = 1 \times 10^5 \text{ Pa}$ , and  $v_w = 69.445 \text{ m s}^{-1}$ . These values correspond to a Mach number of 0.2 and a Reynolds number of 200. The plates were modelled as no-slip isothermal walls as detailed in Appendix C.4 of Appendix C. A plot of the resulting energy profile can be seen in Fig. 9. Constant initial conditions are taken as  $\rho = \langle \rho(\phi) \rangle$ ,  $\mathbf{v} = v_w \hat{\mathbf{x}}$ , and  $p = p_c$ . Using the analytical solution





**Fig. 9.** Converged steady state energy profile for the two dimensional Couette flow problem.

we again define an  $L^2$  error as

$$\sigma(t)^2 = \int_{\Omega} [E^{\delta}(\mathbf{x}, t) - E(\mathbf{x})]^2 d^N \mathbf{x} \quad (37)$$

$$= \int_{\Omega_{ei}} [E_{ei}^{\delta}(\tilde{\mathbf{x}}, t) - E(\mathcal{M}_{ei}(\tilde{\mathbf{x}}))]^2 J_{ei}(\tilde{\mathbf{x}}) d^N \tilde{\mathbf{x}} \quad (38)$$

$$\approx [E_{ei}^{\delta}(\tilde{\mathbf{x}}_{ej}, t) - E(\mathcal{M}_{ei}(\tilde{\mathbf{x}}_{ej}))]^2 J_{ei}(\tilde{\mathbf{x}}_{ej}) \omega_{ej}, \quad (39)$$

where  $\Omega$  is the computational domain,  $E^{\delta}(\mathbf{x}, t)$  is the numerical total energy, and  $E(\mathbf{x})$  the analytic total energy. In the third step we have approximated each integral by a quadrature rule with abscissa  $\{\tilde{\mathbf{x}}_{ej}\}$  and weights  $\{\omega_{ej}\}$  inside of an element type  $e$ . Couette flow is a steady state problem and so in the limit of  $t \rightarrow \infty$  the numerical total energy should converge to a solution. Starting from a constant initial condition the  $L^2$  error was computed every 0.1 time units. The simulation was said to have converged when  $\sigma(t)/\sigma(t + 0.1) \leq 1.01$  where  $\sigma$  is the  $L^2$  error. We will denote the time at which this occurs by  $t_{\infty}$ .

Once the system has converged for a range of meshes it is possible to compute the order of accuracy of the scheme. For a given  $\wp$  this is the slope (plus or minus a standard error) of a linear least squares fit of  $\log h \sim \log \sigma(t_{\infty})$  where  $h$  is an approximation of the characteristic grid spacing. The expected order of accuracy is  $\wp + 1$ . In all simulations inviscid fluxes were computed using the Rusanov approach and the LDG parameters were taken to be  $\beta = 1/2$  and  $\tau = 0.1$ . All simulations were performed with DG correction functions and at double precision. Inside tensor product elements Gauss–Legendre solution and flux points were employed. Triangular elements utilised Williams–Shunn solution points and Gauss–Legendre flux points.

**Two dimensional unstructured mixed mesh.** For the two dimensional test cases the computational domain was taken to be  $[-1, 1] \times [0, 1]$ . This domain was then meshed with both triangles and quadrilaterals at four different refinement levels as shown in Fig. 10. The Couette flow problem described above was then solved on each of these meshes. Experimental  $L^2$  errors and orders of accuracy can be seen in Table 2. We note that in all cases the expected order of accuracy was obtained.

**Three dimensional extruded hexahedral mesh.** For this three dimensional case the computational domain was taken to be  $[-1, 1] \times [0, 1] \times [0, 1]$ . Meshes were constructed through first generating a series of unstructured quadrilateral meshes in the  $x$ - $y$  plane. A three layer extrusion was then performed on this meshes to yield a series of hexahedral meshes. Experimental  $L^2$  errors and orders of accuracy for these meshes can be seen in Table 3.

**Three dimensional unstructured hexahedral mesh.** As a further test a domain of dimension  $[0, 1]^3$  was considered. This domain was meshed using completely unstructured hexahedra. Three levels of refinement were used resulting in meshes with 96, 536 and 1004 elements. A cutaway of the most refined mesh can be seen in Fig. 11. Experimental  $L^2$  errors and the resulting orders of accuracy are presented in Table 4. Despite the fully unstructured nature of

**Table 2**

$L^2$  energy error and orders of accuracy for the Couette flow problem on four mixed meshes. The mesh spacing was approximated as  $h \sim N_E^{-1/2}$  where  $N_E$  is the total number of elements in the mesh.

Tris	Quads	$\sigma(t_{\infty}) / \text{J m}^{-3}$			
		$\wp = 1$	$\wp = 2$	$\wp = 3$	$\wp = 4$
2	8	$1.26 \times 10^2$	$5.77 \times 10^{-1}$	$5.54 \times 10^{-3}$	$6.62 \times 10^{-5}$
6	22	$3.56 \times 10^1$	$1.40 \times 10^{-1}$	$6.72 \times 10^{-4}$	$3.91 \times 10^{-6}$
10	37	$2.08 \times 10^1$	$4.35 \times 10^{-2}$	$2.54 \times 10^{-4}$	$8.16 \times 10^{-7}$
16	56	$1.46 \times 10^1$	$3.52 \times 10^{-2}$	$1.09 \times 10^{-4}$	$4.62 \times 10^{-7}$
Order		$2.21 \pm 0.12$	$2.99 \pm 0.32$	$3.97 \pm 0.05$	$5.20 \pm 0.38$

**Table 3**

$L^2$  energy errors and orders of accuracy for the Couette flow problem on three extruded hexahedral meshes. On account of the extrusion  $h \sim N_E^{-1/2}$  where  $N_E$  is the total number of elements in the mesh.

Hexes	$\sigma(t_{\infty}) / \text{J m}^{-3}$		
	$\wp = 1$	$\wp = 2$	$\wp = 3$
78	$3.35 \times 10^1$	$5.91 \times 10^{-2}$	$7.28 \times 10^{-4}$
195	$1.23 \times 10^1$	$1.87 \times 10^{-2}$	$1.15 \times 10^{-4}$
405	$6.15 \times 10^0$	$5.49 \times 10^{-3}$	$2.72 \times 10^{-5}$
Order		$2.06 \pm 0.08$	$3.99 \pm 0.03$

**Table 4**

$L^2$  energy errors and orders of accuracy for the Couette flow problem on three unstructured hexahedral meshes. Mesh spacing was taken as  $h \sim N_E^{-1/3}$  where  $N_E$  is the total number of elements in the mesh.

Hexes	$\sigma(t_{\infty}) / \text{J m}^{-3}$		
	$\wp = 1$	$\wp = 2$	$\wp = 3$
96	$1.91 \times 10^1$	$4.32 \times 10^{-2}$	$5.83 \times 10^{-4}$
536	$8.20 \times 10^0$	$9.11 \times 10^{-3}$	$6.89 \times 10^{-5}$
1004	$3.82 \times 10^0$	$3.22 \times 10^{-3}$	$2.04 \times 10^{-5}$
Order		$1.93 \pm 0.46$	$4.16 \pm 0.44$

the mesh the expected order of accuracy was again obtained in all cases. We do, however, note the higher standard errors associated with these results.

### 5.3. Compressible Navier–Stokes equations: flow over a cylinder

In order to demonstrate the ability of PyFR to solve the unsteady Navier–Stokes equations flow over a cylinder at Reynolds number 3900 and Mach number  $M = 0.2$  was simulated. A cylinder of radius  $1/2$  was placed at  $(0, 0)$  inside of a domain of dimension  $[-18, 30] \times [-10, 10] \times [0, 3.2]$ . This domain was then meshed in the  $x$ - $y$  plane with 4661 quadratically curved quadrilateral elements. Next, this grid was extruded along the  $z$ -axis to yield a total of 46 610 hexahedra. The grid, which can be seen in Fig. 12, was partitioned into four pieces. Along surfaces of  $y = \pm 10$  and  $x = -18$  the inflow boundary condition of Appendix C.2 in Appendix C was imposed. Along the surface of  $x = 30$  the outflow condition of Appendix C.3 in Appendix C was used. Periodic conditions were imposed in the  $z$  direction. On the surface of the cylinder the no-slip isothermal wall condition of Appendix C.4 in Appendix C was imposed. The free-stream conditions were taken to be  $\rho = 1$ ,  $\mathbf{v} = \hat{\mathbf{x}}$ , and  $p = 1/\gamma M^2$ . These were also used as the initial conditions for the simulation. DG correction functions were used with the LDG parameters being  $\beta = 1/2$  and  $\tau = 0.1$ . The ratio of specific heats was taken as  $\gamma = 1.4$  and the Prandtl number as  $Pr = 0.72$ .

The simulation was run with  $\wp = 4$  with four NVIDIA K20c GPUs. It contained some  $29 \times 10^6$  degrees of freedom. Isosurfaces of density captured after the turbulent wake had fully developed can be seen in Fig. 13.

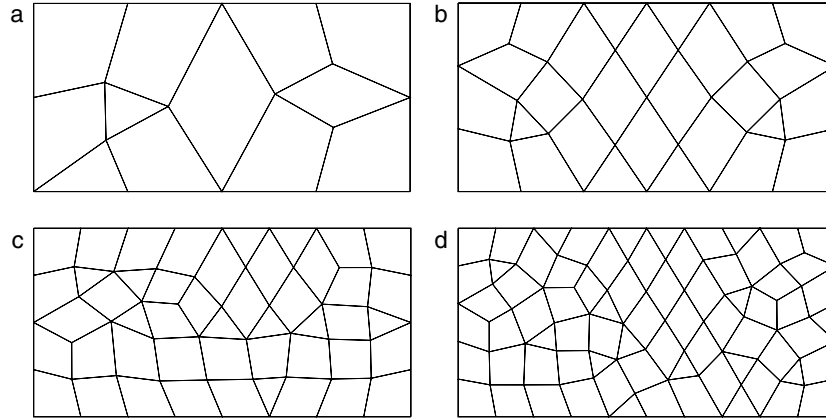


Fig. 10. Unstructured mixed element meshes used for the two dimensional Couette flow problem.

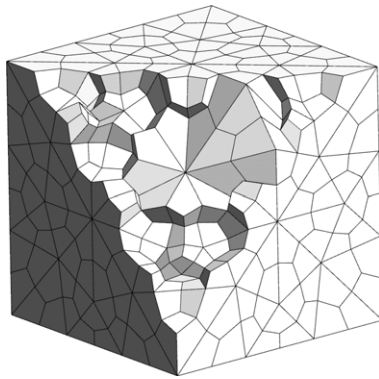


Fig. 11. Cutaway of the unstructured hexahedral mesh with 1004 elements.

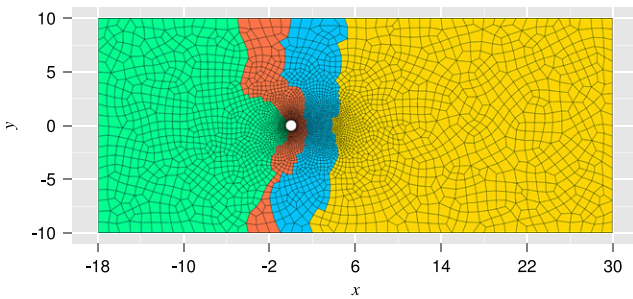


Fig. 12. Cross section in the  $x$ - $y$  plane of the cylinder mesh. Colours indicate the partition to which the elements belong.

## 6. Single node performance

The single node performance of PyFR has been evaluated on an NVIDIA M2090 GPU. This accelerator has a theoretical peak double precision floating point performance of 665 GFLOP/s, and when ECC is disabled the theoretical peak memory bandwidth is 177 GB/s. As points of reference we observe that cuBLAS (CUDA 5.5) is able to obtain 407 GFLOP/s when multiplying a pair of  $4096 \times 4096$  matrices on this hardware, and the maximum device bandwidth obtainable by the bandwidth test application included with the CUDA SDK is 138.9 GiB/s when ECC is disabled. We shall refer to these values as *reference peaks*.

To conduct the evaluation a fully periodic cuboidal domain was meshed with 50 176 hexahedral elements. The double precision Navier–Stokes solver of PyFR was then run on this mesh at orders  $\wp = 2, 3, 4$  with  $\beta = 1/2$ . In conducting the analysis kernels were grouped into one of three categories: matrix multiplications (DGEMM), point-wise kernels with direct memory access patterns

Table 5

Single GPU performance of PyFR for the Navier–Stokes equations when run on an NVIDIA M2090 with ECC disabled. As the memory bandwidth requirements of DGEMM are dependent on the accumulation strategy adopted by the implementation these values have been omitted.

		Order		
		$\wp = 2$	$\wp = 3$	$\wp = 4$
Wall time/%	DGEMM	55.7	66.2	81.4
	PD	24.9	21.5	12.8
	PI	19.4	12.3	5.8
Bandwidth/GiB/s	PD	125.5	125.0	124.8
	PI	124.8	124.3	124.2
Arithmetic/GFLOP/s	DGEMM	205.3	368.1	305.4
	PD	0.7	0.7	0.7
	PI	0.9	0.8	0.9

(PD) and point-wise kernels with some level of indirect memory access (PI). Indirection arises in the computation of  $\mathcal{C}_\alpha$  in Eq. (7) and  $\mathcal{F}_\alpha$  in Eq. (15) and occurs as a consequence of the unstructured nature of PyFR. The resulting breakdowns of wall-clock time, memory bandwidth and floating point operations can be seen in Table 5. It is clear that the majority of floating point operations are concentrated inside the calls to DGEMM with the point-wise operations being heavily memory bandwidth bound. Of this bandwidth some  $\sim 15\%$  was ascribed to register spillage above and beyond that which can be absorbed by the L1 cache.

The high fraction of peak bandwidth obtained by the indirect kernels can be attributed to three factors. Firstly, the constant data required for calculations at interfaces, such as  $\hat{\mathbf{n}}_{\text{esn}}^{(f)}$  and  $J_{\text{esn}}^{(f)} n_{\text{esn}}^{(f)}$ , is ordered to ensure direct (coalesced) access. Secondly, at start-up PyFR attempts to determine an iteration ordering over the various flux-point pairs that will minimise the number of cache misses.

Many of the memory accesses are therefore near-coalesced. Thirdly and finally we highlight the impressive latency-hiding capabilities of the CUDA programming model.

In line with expectations the proportion of time spent performing matrix–matrix multiplications increases as a function of order. When going from  $\wp = 2$  to  $\wp = 3$  a significant portion of the additional compute is offset by the improved performance of cuBLAS. However, when  $\wp = 4$  the performance of these kernels in absolute terms can be seen to regress slightly. This contributes to the greatly increased fraction of wall-clock time spent inside of these kernels. Nevertheless, the achieved rate of 305.4 GFLOP/s is still over 75% of the reference peak. Also in line with expectations is the invariance of the arithmetic performance of the point-wise kernels with respect to order. As the order is varied all that changes is the

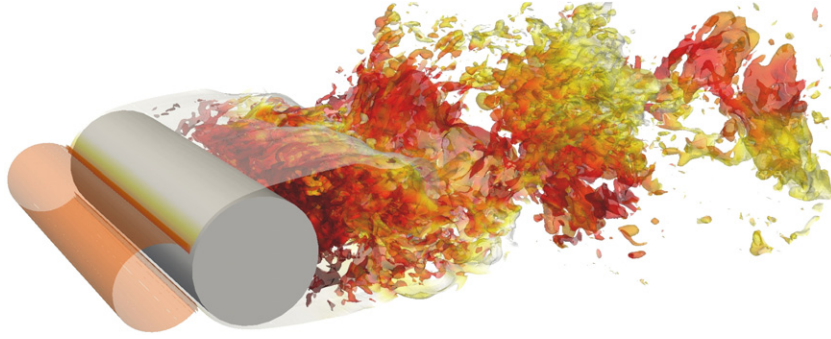


Fig. 13. Isosurfaces of density around the cylinder.

Table 6

Weak scalability of PyFR for the Navier–Stokes equations with  $\phi = 3$ . Runtime is normalised with respect to a single NVIDIA M2090 GPU.

# M2090s	1	2	4	8	16	32	64	104
Runtime	1.00	1.00	1.01	1.01	1.01	1.01	1.01	1.01

Table 7

Strong scalability of PyFR for the Navier–Stokes equations with  $\phi = 3$ . The speedup is relative to a single NVIDIA M2090 GPU.

# M2090s	1	2	4	8	16	32
Speedup	1.00	2.03	3.96	7.48	14.07	26.18

number of points to be processed with the operation itself remaining identical.

## 7. Scalability

The scalability of PyFR has been evaluated on the Emerald GPU cluster. It is housed at the STFC Rutherford Appleton Laboratory and based around 60 HP SL390 nodes with three NVIDIA M2090 GPUs and 24 HP SL390 nodes with eight NVIDIA M2090 GPUs. Nodes are connected by QDR InfiniBand.

For simplicity all runs herein were performed on the eight GPU nodes. As a starting point a domain of dimension  $[-16, 16] \times [-16, 16] \times [0, 1.75]$  was meshed isotropically with  $N_E = 114,688$  structured hexahedral elements. The mesh was configured with completely periodic boundary conditions. When run with the Navier–Stokes solver in PyFR with  $\phi = 3$  the mesh gives a working set of  $\sim 4720$  MiB. This is sufficient to 90% load an M2090 which when ECC is enabled has  $\sim 5250$  MiB memory available to the user. When examining the scalability of a code there are two commonly used metrics. The first of these is weak scalability in which the size of the target problem is increased in proportion to the number of ranks  $N$  with  $N_E \propto N$ . For a code with perfect weak scalability the runtime should remain unchanged as more ranks are added. The second metric is strong scalability wherein the problem size is fixed and the speedup compared to a single rank is assessed. Perfect strong scalability implies that the runtime scales as  $1/N$ .

For the domain outlined above weak scalability was evaluated by increasing the dimensions of the domain according to  $[-16, 16] \times N[-16, 16] \times [0, 1.75]$ . This extension permitted the domain to be trivially decomposed along the  $y$ -axis. The resulting runtimes for  $1 \leq N \leq 104$  can be seen in Table 6. We note that in the  $N = 104$  case that the simulation consisted of some  $3.8 \times 10^9$  degrees of freedom with a working set of  $\sim 485$  GiB.

To study the strong scalability the initial domain was partitioned along the  $x$ - and  $y$ -axes. Each partition contained exactly  $N_E/N_s$ . The resulting speedups for  $1 \leq N \leq 32$  can be seen in Table 7. Up to eight GPUs scalability can be seen to be near perfect. Beyond this the relationship begins to break down. When  $N = 32$  an improvement of 26 can be observed. However, in this case each GPU is loaded to less than 3% and so the result is to be expected.

## 8. Conclusions

In this paper we have described PyFR, an open source Python based framework for solving advection–diffusion type problems on

streaming architectures. The structure and ethos of PyFR has been explained including our methodology for targeting multiple hardware platforms. We have shown that PyFR exhibits spatial super accuracy when solving the 2D Euler equations and the expected order of accuracy when solving Couette flow problem on a range of grids in 2D and 3D. Qualitative results for unsteady 3D viscous flow problems on curved grids have also been presented. Performance of PyFR has been validated on an NVIDIA M2090 GPU in three dimensions. It has been shown that the compute bound kernels are able to obtain between 50% and 90% of reference peak FLOP/s whereas the bandwidth bound point-wise kernels are, across the board, able to obtain in excess of 89% of reference peak bandwidth. The scalability of PyFR has been demonstrated in the strong sense up to 32 NVIDIA M2090s and in the weak sense up to 104 NVIDIA M2090s when solving the 3D Navier–Stokes equations.

## Acknowledgements

The authors would like to thank the Engineering and Physical Sciences Research Council for their support via two Doctoral Training Grants and an Early Career Fellowship (EP/K027379/1). The authors would also like to thank the e-Infrastructure South Centre for Innovation for granting access to the Emerald supercomputer, and NVIDIA for donation of three K20c GPUs.

## Appendix A. Matrix representation

It is possible to cast the majority of operations in an FR step as matrix–matrix multiplications of the form

$$\mathbf{C} \leftarrow c_1 \mathbf{A} \mathbf{B} + c_2 \mathbf{C}, \quad (40)$$

where  $c_{1,2} \in \mathbb{R}$  are constants,  $\mathbf{A}$  is a constant operator matrix, and  $\mathbf{B}$  and  $\mathbf{C}$  are state matrices. To accomplish this we start by introducing the following constant operator matrix

$$(\mathbf{M}_e^0)_{\sigma\rho} = \ell_{e\rho}^{(u)}(\tilde{\mathbf{x}}_{e\sigma}^{(f)}), \quad \dim \mathbf{M}_e^0 = N_e^{(f)} \times N_e^{(u)},$$

and the following state matrices

$$\begin{aligned} (\mathbf{U}_e^{(u)})_{\rho(n\alpha)} &= u_{e\rho n\alpha}^{(u)}, & \dim \mathbf{U}_e^{(u)} &= N_e^{(u)} \times N_V |\Omega_e|, \\ (\mathbf{U}_e^{(f)})_{\sigma(n\alpha)} &= u_{e\sigma n\alpha}^{(f)}, & \dim \mathbf{U}_e^{(f)} &= N_e^{(f)} \times N_V |\Omega_e|. \end{aligned}$$

In specifying the state matrices there is a degree of freedom associated with how the  $N_V$  field variables for each element are packed along a row of the matrix, with the possible packing choices being discussed in Section 3.3. Using these matrices we are able to

reformulate Eq. (6) as

$$\mathbf{U}_e^{(f)} = \mathbf{M}_e^0 \mathbf{U}_e^{(u)}. \quad (41)$$

In order to apply a similar procedure to Eq. (9) we let

$$\begin{aligned} (\mathbf{M}_e^4)_{\rho\sigma} &= [\tilde{\mathbf{V}} \ell_{e\rho}^{(u)}(\tilde{\mathbf{x}})]_{\tilde{\mathbf{x}}=\tilde{\mathbf{x}}_{e\sigma}^{(u)}}, & \dim \mathbf{M}_e^4 &= N_D N_e^{(u)} \times N_e^{(u)}, \\ (\mathbf{M}_e^6)_{\rho\sigma} &= [\hat{\mathbf{n}}_{e\rho}^{(f)} \cdot \tilde{\mathbf{V}} \cdot \mathbf{g}_{e\sigma}^{(f)}(\tilde{\mathbf{x}})]_{\tilde{\mathbf{x}}=\tilde{\mathbf{x}}_{e\sigma}^{(f)}}, & \dim \mathbf{M}_e^6 &= N_D N_e^{(u)} \times N_e^f, \\ (\mathbf{C}_e^{(f)})_{\rho(n\alpha)} &= \mathbf{c}_\alpha u_{e\rho n\alpha}^{(f)}, & \dim \mathbf{C}_e^{(f)} &= N_e^{(f)} \times N_V |\Omega_e|, \\ (\tilde{\mathbf{Q}}_e^{(u)})_{\sigma(n\alpha)} &= \tilde{\mathbf{q}}_{e\sigma n\alpha}^{(u)}, & \dim \tilde{\mathbf{Q}}_e^{(u)} &= N_D N_e^{(u)} \times N_V |\Omega_e|. \end{aligned}$$

Here it is important to qualify assignments of the form  $\mathbf{A}_{ij} = \mathbf{x}$  where  $\mathbf{x}$  is a  $N_D$  component vector. As above there is a degree of freedom associated with the packing. With the benefit of foresight we take the stride between subsequent elements of  $\mathbf{x}$  in a matrix column to be either  $\Delta i = N_e^{(u)}$  or  $\Delta i = N_e^{(f)}$  depending on the context. With these matrices Eq. (9) reduces to

$$\begin{aligned} \tilde{\mathbf{Q}}_e^{(u)} &= \mathbf{M}_e^6 \{\mathbf{C}_e^{(f)} - \mathbf{U}_e^{(f)}\} + \mathbf{M}_e^4 \mathbf{U}_e^{(u)} \\ &= \mathbf{M}_e^6 \{\mathbf{C}_e^{(f)} - \mathbf{M}_e^0 \mathbf{U}_e^{(u)}\} + \mathbf{M}_e^4 \mathbf{U}_e^{(u)} \\ &= \mathbf{M}_e^6 \mathbf{C}_e^{(f)} + \{\mathbf{M}_e^4 - \mathbf{M}_e^6 \mathbf{M}_e^0\} \mathbf{U}_e^{(u)}. \end{aligned} \quad (42)$$

Applying the procedure to Eq. (11) we take

$$\begin{aligned} \mathbf{M}_e^5 &= \text{diag}(\mathbf{M}_e^0, \dots, \mathbf{M}_e^0) & \dim \mathbf{M}_e^5 &= N_D N_e^{(f)} \times N_D N_e^{(u)}, \\ (\mathbf{Q}_e^{(u)})_{\sigma(n\alpha)} &= \mathbf{q}_{e\sigma n\alpha}^{(u)}, & \dim \mathbf{Q}_e^{(u)} &= N_D N_e^{(u)} \times N_V |\Omega_e|, \\ (\mathbf{Q}_e^{(f)})_{\sigma(n\alpha)} &= \mathbf{q}_{e\sigma n\alpha}^{(f)}, & \dim \mathbf{Q}_e^{(f)} &= N_D N_e^{(f)} \times N_V |\Omega_e|, \end{aligned}$$

hence

$$\mathbf{Q}_e^{(f)} = \mathbf{M}_e^5 \mathbf{Q}_e^{(u)}, \quad (43)$$

where we note the block diagonal structure of  $\mathbf{M}_e^5$ . This is a direct consequence of the above choices for  $\Delta i$ . Finally, to rewrite Eq. (18) we write

$$\begin{aligned} (\mathbf{M}_e^1)_{\rho\sigma} &= [\tilde{\mathbf{V}} \ell_{e\rho}^{(u)}(\tilde{\mathbf{x}})]_{\tilde{\mathbf{x}}=\tilde{\mathbf{x}}_{e\sigma}^{(u)}}, & \dim \mathbf{M}_e^1 &= N_e^{(u)} \times N_D N_e^{(u)}, \\ (\mathbf{M}_e^2)_{\rho\sigma} &= [\ell_{e\rho}^{(u)}(\tilde{\mathbf{x}}_{e\sigma}^{(f)}) \hat{\mathbf{n}}_{e\sigma}^{(f)}]^T, & \dim \mathbf{M}_e^2 &= N_e^{(f)} \times N_D N_e^{(u)}, \\ (\mathbf{M}_e^3)_{\rho\sigma} &= [\tilde{\mathbf{V}} \cdot \mathbf{g}_{e\sigma}^{(f)}(\tilde{\mathbf{x}})]_{\tilde{\mathbf{x}}=\tilde{\mathbf{x}}_{e\sigma}^{(u)}}, & \dim \mathbf{M}_e^3 &= N_e^{(u)} \times N_e^{(f)}, \\ (\tilde{\mathbf{D}}_e^{(f)})_{\sigma(n\alpha)} &= \tilde{\mathbf{d}}_{e\sigma n\alpha}^{(f)}, & \dim \tilde{\mathbf{D}}_e^{(f)} &= N_e^{(f)} \times N_V |\Omega_e|, \\ (\tilde{\mathbf{F}}_e^{(u)})_{\rho(n\alpha)} &= \tilde{\mathbf{f}}_{e\rho n\alpha}^{(u)}, & \dim \tilde{\mathbf{F}}_e^{(u)} &= N_D N_e^{(u)} \times N_V |\Omega_e|, \\ (\tilde{\mathbf{R}}_e^{(u)})_{\rho(n\alpha)} &= (\tilde{\mathbf{V}} \cdot \tilde{\mathbf{f}})_{e\rho n\alpha}^{(u)}, & \dim \tilde{\mathbf{R}}_e^{(u)} &= N_e^{(u)} \times N_V |\Omega_e|, \end{aligned}$$

and after substitution of Eq. (13) for  $\tilde{\mathbf{f}}_{e\sigma n\alpha}^{(f)}$  obtain

$$\begin{aligned} \tilde{\mathbf{R}}_e^{(u)} &= \mathbf{M}_e^3 \{\tilde{\mathbf{D}}_e^{(f)} - \mathbf{M}_e^2 \tilde{\mathbf{F}}_e^{(u)}\} + \mathbf{M}_e^1 \tilde{\mathbf{F}}_e^{(u)} \\ &= \mathbf{M}_e^3 \tilde{\mathbf{D}}_e^{(f)} + \{\mathbf{M}_e^1 - \mathbf{M}_e^3 \mathbf{M}_e^2\} \tilde{\mathbf{F}}_e^{(u)}. \end{aligned} \quad (44)$$

## Appendix B. Approximate Riemann solvers

### B.1. Overview

In the following section we take  $u_L$  and  $u_R$  to be the two discontinuous solution states at an interface and  $\hat{\mathbf{n}}_L$  to be the normal vector associated with the first state. For convenience we take  $\mathbf{f}_L^{(\text{inv})} = \mathbf{f}^{(\text{inv})}(u_L)$ , and  $\mathbf{f}_R^{(\text{inv})} = \mathbf{f}^{(\text{inv})}(u_R)$  with inviscid fluxes being prescribed by Eq. (20).

### B.2. Rusanov

Also known as the local Lax–Friedrichs method a Rusanov type Riemann solver imposes inviscid numerical interface fluxes

according to

$$\mathfrak{F}^{(\text{inv})} = \frac{\hat{\mathbf{n}}_L}{2} \cdot \{\mathbf{f}_L^{(\text{inv})} + \mathbf{f}_R^{(\text{inv})}\} + \frac{s}{2}(u_L - u_R), \quad (45)$$

where  $s$  is an estimate of the maximum wave speed

$$s = \sqrt{\frac{\gamma(p_L + p_R)}{\rho_L + \rho_R}} + \frac{1}{2} |\hat{\mathbf{n}}_L \cdot (\mathbf{v}_L + \mathbf{v}_R)|. \quad (46)$$

## Appendix C. Boundary conditions

### C.1. Overview

To incorporate boundary conditions into the FR approach we introduce a set of boundary interface types  $b \in \mathcal{B}$ . At a boundary interface there is only a single flux point: that which belongs to the element whose edge/face is on the boundary. Associated with each boundary type are a pair of functions  $\mathfrak{C}_\alpha^{(b)}(u_L)$  and  $\mathfrak{F}_\alpha^{(b)}(u_L, \mathbf{q}_L, \hat{\mathbf{n}}_L)$  where  $u_L$ ,  $\mathbf{q}_L$ , and  $\hat{\mathbf{n}}_L$  are the solution, solution gradient and unit normals at the relevant flux point. These functions prescribe the common solutions and normal fluxes, respectively.

Instead of directly imposing solutions and normal fluxes it is oftentimes more convenient for a boundary to instead provide ghost states. In its simplest formulation  $\mathfrak{C}_\alpha^{(b)} = \mathfrak{C}_\alpha(u_L, \mathfrak{B}^{(b)} u_L)$  and  $\mathfrak{F}_\alpha^{(b)} = \mathfrak{F}_\alpha(u_L, \mathfrak{B}^{(b)} u_L, \mathbf{q}_L, \mathfrak{B}^{(b)} \mathbf{q}_L, \hat{\mathbf{n}}_L)$  where  $\mathfrak{B}^{(b)} u_L$  is the ghost solution state and  $\mathfrak{B}^{(b)} \mathbf{q}_L$  is the ghost solution gradient. It is straightforward to extend this prescription to allow for the provisioning of different ghost solution states for  $\mathfrak{C}_\alpha$  and  $\mathfrak{F}_\alpha$  and to permit  $\mathfrak{B}^{(b)} \mathbf{q}_L$  to be a function of  $u_L$  in addition to  $\mathbf{q}_L$ .

### C.2. Supersonic inflow

The supersonic inflow condition is parameterised by a free-stream density  $\rho_f$ , velocity  $\mathbf{v}_f$ , and pressure  $p_f$ .

$$\mathcal{B}^{(\text{inv})} u_L = \mathcal{B}^{(\text{ldg})} u_L = \begin{cases} \rho_f \\ \rho_f \mathbf{v}_f \\ p_f / (\gamma - 1) + \frac{\rho_f}{2} \|\mathbf{v}_f\|^2 \end{cases}, \quad (47)$$

$$\mathcal{B}^{(\text{ldg})} \mathbf{q}_L = 0. \quad (48)$$

### C.3. Subsonic outflow

Subsonic outflow boundaries are parameterised by a free-stream pressure  $p_f$ .

$$\mathcal{B}^{(\text{inv})} u_L = \mathcal{B}^{(\text{ldg})} u_L = \begin{cases} \rho_L \\ \rho_L \mathbf{v}_L \\ p_f / (\gamma - 1) + \frac{\rho_L}{2} \|\mathbf{v}_L\|^2 \end{cases}, \quad (49)$$

$$\mathcal{B}^{(\text{ldg})} \mathbf{q}_L = 0. \quad (50)$$

### C.4. No-slip isothermal wall

The no-slip isothermal wall condition depends on the wall temperature  $c_p T_w$  and the wall velocity  $\mathbf{v}_w$ . Usually  $\mathbf{v}_w = 0$ .

$$\mathcal{B}^{(\text{inv})} u_L = \rho_L \begin{cases} 1 \\ 2\mathbf{v}_w - \mathbf{v}_L \\ c_p T_w / \gamma + \frac{1}{2} \|2\mathbf{v}_w - \mathbf{v}_L\|^2 \end{cases}, \quad (51)$$

$$\mathcal{B}^{(\text{ldg})} u_L = \rho_L \begin{cases} 1 \\ \mathbf{v}_w \\ c_p T_w / \gamma + \frac{1}{2} \|\mathbf{v}_w\|^2 \end{cases}, \quad (52)$$

$$\mathcal{B}^{(\text{ldg})} \mathbf{q}_L = \mathbf{q}_L. \quad (53)$$



## References

- [1] W.H. Reed, T.R. Hill, Triangular mesh methods for the neutron transport equation. Technical Report LA-UR-73-479, Los Alamos Scientific Laboratory, 1973.
- [2] David A. Kopriva, John H. Kolas, A conservative staggered-grid Chebyshev multidomain method for compressible flows, *J. Comput. Phys.* 125 (1) (1996) 244–261.
- [3] Yuzhi Sun, Zhi Jian Wang, Yen Liu, High-order multidomain spectral difference method for the Navier–Stokes equations on unstructured hexahedral grids, *Commun. Comput. Phys.* 2 (2) (2007) 310–333.
- [4] H.T. Huynh, A flux reconstruction approach to high-order schemes including discontinuous Galerkin methods, *AIAA paper 4079* (2007) 2007.
- [5] Jan S. Hesthaven, Tim Warburton, *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*, vol. 54, Springer Verlag, New York, 2008.
- [6] P.E. Vincent, P. Castonguay, A. Jameson, A new class of high-order energy stable flux reconstruction schemes, *J. Sci. Comput.* 47 (1) (2011) 50–72.
- [7] P. Castonguay, P.E. Vincent, A. Jameson, A new class of high-order energy stable flux reconstruction schemes for triangular elements, *J. Sci. Comput.* (2011).
- [8] Patrice Castonguay, P.E. Vincent, Antony Jameson, Application of high-order energy stable flux reconstruction schemes to the Euler equations, in: 49th AIAA Aerospace Sciences Meeting, volume 686, 2011.
- [9] A. Jameson, P.E. Vincent, P. Castonguay, On the non-linear stability of flux reconstruction schemes, *J. Sci. Comput.* 50 (2) (2011) 434–445.
- [10] P.E. Vincent, P. Castonguay, A. Jameson, Insights from von Neumann analysis of high-order flux reconstruction schemes, *J. Comput. Phys.* 230 (22) (2011) 8134–8154.
- [11] Patrice Castonguay, Peter E. Vincent, Antony Jameson, A new class of high-order energy stable flux reconstruction schemes for triangular elements, *J. Sci. Comput.* 51 (1) (2012) 224–256.
- [12] D.M. Williams, P. Castonguay, P.E. Vincent, A. Jameson, Energy stable flux reconstruction schemes for advection–diffusion problems on triangles, *J. Comput. Phys.* (2013).
- [13] P. Castonguay, D.M. Williams, P.E. Vincent, A. Jameson, Energy stable flux reconstruction schemes for advection–diffusion problems, *Comput. Methods Appl. Mech. Engrg.* (2013).
- [14] D.M. Williams, A. Jameson, Energy stable flux reconstruction schemes for advection–diffusion problems on tetrahedra, *J. Sci. Comput.* (2013) 1–39.
- [15] David A. Kopriva, A staggered-grid multidomain spectral method for the compressible Navier–Stokes equations, *J. Comput. Phys.* 143 (1) (1998) 125–158.
- [16] SymPy Development Team. SymPy: Python library for symbolic mathematics, 2013.
- [17] Michael Bayer, Mako: Templates for python, 2013.
- [18] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fahih, Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation, *Parallel Comput.* 38 (3) (2012) 157–174.
- [19] Lisandro Dalcin, mpi4py: Mpi for python, 2013.
- [20] Eleuterio F. Toro, *Riemann Solvers and Numerical Methods for Fluid Dynamics: a Practical Introduction*, Springer, 2009.