PyFR v2.0.3: Towards Industrial Adoption of Scale-Resolving Simulations

Freddie D. Witherden[a], Peter E. Vincent[b,*], Will Trojak[c], Yoshiaki Abe[d], Amir Akbarzadeh[a], Semih Akkurt[b], Mohammad Alhawwary[a], Lidia Caros[b], Tarik Dzanic[e], Giorgio Giangaspero[f], Arvind S. Iyer[g], Antony Jameson[a], Marius Koch[h], Niki Loppi[i], Sambit Mishra[a], Rishit Modi[a], Gonzalo Sáez-Mischlich[j,k], Jin Seok Park[l], Brian C. Vermeire[m] and Lai Wang[n]

[a]*Department of Ocean Engineering, Texas A&M University, 3145 TAMU, College Station, 77843, TX, USA*
[b]*Department of Aeronautics, Imperial College London, South Kensington, London, SW7 2AZ, UK*
[c]*IBM Research UK, Hartree Centre, Warrington, WA4 4AD, UK*
[d]*Institute of Fluid Science, Tohoku University, Sendai, JP*
[e]*Lawrence Livermore National Laboratory, Livermore, CA, USA*
[f]*Siemens Digital Industries Software, Shepherds Bush Road, London, W6 7NL, UK*
[g]*Astrome Technologies, Bangalore, IN*
[h]*NVIDIA Corporation, Fasanenstr 81, Berlin, 10623, DE*
[i]*NVIDIA Corporation, Porkkalankatu 1, Helsinki, 00180, FI*
[j]*Luminary, 101 S Ellsworth Ave Suite 600, San Mateo, CA, USA*
[k]*ISAE SUPAERO, University of Toulouse, 31400 Toulouse, FR*
[l]*Department of Aerospace Engineering, Inha University, Incheon, KR*
[m]*Department of Mechanical, Industrial, and Aerospace Engineering, Concordia University, Montreal, QC, CA*
[n]*Gaithersburg, MD, USA*

ARTICLE INFO

Keywords:
High-order accuracy
Flux reconstruction
Computational fluid dynamics

ABSTRACT

PyFR is an open-source cross-platform computational fluid dynamics framework based on the high-order Flux Reconstruction approach, specifically designed for undertaking high-accuracy scale-resolving simulations in the vicinity of complex engineering geometries. Since the initial release of PyFR v0.1.0 in 2013, a range of new capabilities have been added to the framework, with a view to enabling industrial adoption of the capability. This paper provides details of those enhancements as released in PyFR v2.0.3, explains efforts to grow an engaged developer and user community, and provides latest performance and scaling results on up to 1024 AMD Instinct MI250X accelerators of Frontier at ORNL (each with two GCDs), and up to 2048 NVIDIA GH200 GPUs on Alps at CSCS.

PROGRAM SUMMARY
Program Title: PyFR
CPC Library link to program files:
Developer's repository link: https://github.com/PyFR/PyFR
Code Ocean capsule:
Licensing provisions: BSD 3-clause
Programming language: Python (generating C/OpenMP, CUDA, OpenCL, HIP, Metal)
Nature of problem: Accurate and efficient scale-resolving simulation of industrial flows.
Solution method: Massively parallel cross-platform implementation of high-order accurate Flux Reconstruction schemes.

1. Introduction

Computational Fluid Dynamics (CFD) is used by high-value industries across the world to reduce costs and improve product performance. The majority of industrial CFD is undertaken using Reynolds-Averaged Navier–Stokes (RANS) simulations, which time-average unsteady phenomena, including turbulence, and replace the 'missing physics' with a model. However, it is well established that RANS approaches have limited applicability when flow is separated

p.vincent@ic.ac.uk (P.E. Vincent)
ORCID(s):

and unsteady. To overcome this limitation, higher-fidelity scale-resolving methods can be used, such as Large Eddy Simulations (LES), Implicit Large Eddy Simulations (ILES) and Direct Numerical Simulations (DNS), with DNS being the most accurate; fully resolving all physics of the governing Navier–Stokes equations. However, the cost of a scale-resolving simulation is typically orders-of-magnitude higher than that of a RANS simulation, and thus the use of scale-resolving methods in industry has until recently been considered intractable.

Our vision with PyFR has been to develop and deliver an open-source Python framework based on the high-order accurate Flux Reconstruction (FR) approach [1] that enables real-world scale-resolving simulations to be undertaken in tractable time, at scale, by both academic and industrial practitioners—helping advance industrial CFD capabilities from their current 'RANS plateau'. It is one of several international efforts towards industrial adoption of scale-resolving simulations, including Nektar++ [2, 3], hpMusic [4], HiPSTAR [5], and charLES (now acquired and distributed for industrial usage by as Fidelity LES Solver by Cadence Inc) [6, 7].

The first version of PyFR—v0.1.0—was released in late 2013 [8]. It supported solving the compressible Euler and Navier–Stokes equations on unstructured grids of hexahedral elements, and was able to target both conventional CPUs and NVIDIA GPUs via a novel domain specific language based on Mako. In the past decade, there have been over 30 subsequent releases of PyFR, adding a wide range of new capabilities with a view to enabling industrial adoption, culminating in the current release of PyFR v2.0.3 which is described in this paper.

2. Flux Reconstruction

The Flux Reconstruction (FR) approach [1] implemented in PyFR is a form of discontinuous spectral element method [9, 10]. As a brief overview, consider the first-order hyperbolic conservation law

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

where α is the field variable index, $u_\alpha = u_\alpha(\mathbf{x}, t)$ are the conservative field variables, and $\mathbf{f}_\alpha = \mathbf{f}_\alpha(u_\alpha)$ are the fluxes of u_α. In order to solve Eq. (1) using FR in a domain $\mathbf{\Omega}$, one must tessellate the domain with N non-overlapping conforming elements $\mathbf{\Omega}_n$ as

$$\mathbf{\Omega} = \bigcup_{n=1}^{N} \mathbf{\Omega}_n, \qquad \bigcap_{n=1}^{N} \mathbf{\Omega}_n = \varnothing, \tag{2}$$

see, for example, Fig. 1. Without loss of generality, we assume in this example all $\mathbf{\Omega}_n$ are of the same type.

Each element $\mathbf{\Omega}_n$ can then be mapped to a reference element $\hat{\mathbf{\Omega}}$ via a mapping function $\mathcal{M}_n$ defined as

$$\mathbf{x} = \mathcal{M}_n(\tilde{\mathbf{x}}), \qquad \tilde{\mathbf{x}} = \mathcal{M}_n^{-1}(\mathbf{x}),$$

and geometric Jacobian matrices can be defined from the mapping functions as

$$\begin{aligned} \mathbf{J}_n = J_{nij} &= \frac{\partial \mathcal{M}_{ni}}{\partial \tilde{x}_j}, \qquad & J_n &= \det \mathbf{J}_n, \\ \mathbf{J}_n^{-1} = J_{nij}^{-1} &= \frac{\partial \mathcal{M}_{ni}^{-1}}{\partial \tilde{x}_j}, \qquad & J_n^{-1} &= \det \mathbf{J}_n^{-1} = \frac{1}{J_n}. \end{aligned}$$

For each $\mathbf{\Omega}_n$, these Jacobian matrices can be used to transform Eq. (1) into reference element space as

$$\frac{\partial u_{n\alpha}}{\partial t} + J_n^{-1} \tilde{\nabla} \cdot \tilde{\mathbf{f}}_{n\alpha} = 0 \quad \text{and} \quad \tilde{\mathbf{f}}_{n\alpha} = \tilde{\mathbf{f}}_{n\alpha}(\tilde{\mathbf{x}}, t) = J_n(\tilde{\mathbf{x}}) \mathbf{J}_n^{-1}(\mathcal{M}_n(\tilde{\mathbf{x}})) \mathbf{f}_{n\alpha}(\mathcal{M}_n(\tilde{\mathbf{x}}), t), \tag{3}$$

where $u_{n\alpha}$ and $\mathbf{f}_{n\alpha}$ are the solution and flux in $\mathbf{\Omega}_n$, respectively, and $\tilde{\nabla} = \partial/\partial \tilde{x}_i$.

We can proceed to define a set of solution points $\tilde{\mathbf{x}}_\zeta^{(u)}$ in the reference element (see Fig. 1), where ζ is the solution point index which satisfies $0 \leqslant \zeta < N^{(u)}$ and $N^{(u)}$ is the number of solution points in the reference element. Now a nodal basis set $\ell_\zeta^{(u)}(\tilde{\mathbf{x}})$ can be defined in the reference element, where the nodal basis polynomials $\ell_\zeta^{(u)}$ satisfy $\ell_\zeta^{(u)}(\tilde{\mathbf{x}}_\sigma^{(u)}) = \delta_{\zeta\sigma}$ where δ_{ij} is the Kronecker delta. We can also define a set of flux points $\tilde{\mathbf{x}}_\zeta^{(f)}$ on the surface of the reference element

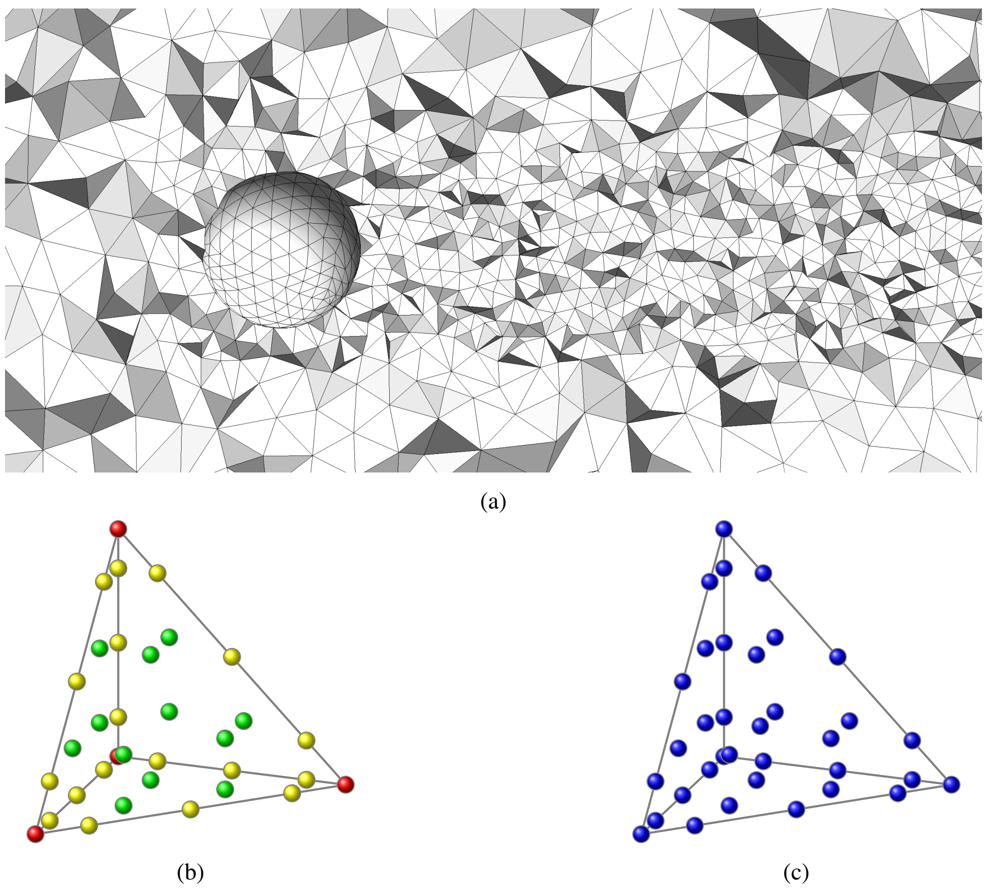


Figure 1: Example of an unstructured curved-element tetrahedral mesh around a sphere (a), fourth-order α-optimised flux points for a tetrahedron (b), where yellow and red indicate doubly and triply collocated points, respectively, and fourth-order α-optimised solution points for a tetrahedron (c).

(see Fig. 1), where ζ is now the flux point index which satisfies $0 \leqslant \zeta < N^{(f)}$ and $N^{(f)}$ is the number of flux points on the surface of the reference element. These flux points are constrained such that flux points from adjoining elements always conform at element interfaces.

The first step in the FR approach is to obtain the discontinuous solution at each flux point $u^{(f)}_{\sigma n\alpha}$ from the solution at the solution points $u^{(u)}_{\zeta n\alpha}$ as

$$u^{(f)}_{\sigma n\alpha} = u^{(u)}_{\zeta n\alpha}\ell^{(u)}_{\zeta}(\tilde{\mathbf{x}}^{(f)}_{\sigma}). \tag{4}$$

The second step is to obtain a transformed common normal interface flux at each flux point $\tilde{f}^{C(f_\perp)}_{\sigma n\alpha}$ from the discontinuous solution at the flux point $u^{(f)}_{\sigma n}$, the discontinuous solution at the conforming flux point in the relevant adjoining element $u'^{(f)}_{\sigma n}$, and the surface normal at the flux point $\mathbf{n}^{(f)}_{\sigma n}$ as

$$\tilde{f}^{C(f_\perp)}_{\sigma n\alpha} = \mathfrak{F}_\alpha(u^{(f)}_{\sigma n}, u'^{(f)}_{\sigma n}, \hat{\mathbf{n}}^{(f)}_{\sigma n}). \tag{5}$$

where $\mathfrak{F}_\alpha$ is *e.g.* an appropriate Riemann solver.

The third step is to calculate the transformed discontinuous flux at the solution points $\tilde{\mathbf{f}}^{(u)}_{\sigma n\alpha}$ from the solution at solution points $\mathbf{u}^{(u)}_{\sigma n\alpha}$ using the system flux function. These values can then be used to calculate the transformed normal discontinuous flux at the flux points $\tilde{f}^{(f_\perp)}_{\sigma n\alpha}$ as

$$\tilde{f}^{(f_\perp)}_{\sigma n\alpha} = \mathbf{n}^{(f)}_{\sigma n} \cdot \tilde{\mathbf{f}}^{(u)}_{\zeta n\alpha}\ell^{(u)}_{\zeta}(\tilde{\mathbf{x}}^{(f)}_{\sigma}). \tag{6}$$

Finally, the transformed divergence of the transformed continuous flux $(\tilde{\nabla} \cdot \tilde{\mathbf{f}})^{(u)}_{\zeta n \alpha}$ can be obtained by propagating the difference between $\tilde{f}^{C(f_\perp)}_{\sigma n \alpha}$ and $\tilde{f}^{(f_\perp)}_{\sigma n \alpha}$ at each flux point into the element using a flux correction function $\mathbf{g}^{(f)}_{\sigma}$, and combining with values of $\tilde{\mathbf{f}}^{(u)}_{\sigma n \alpha}$ as

$$(\tilde{\nabla} \cdot \tilde{\mathbf{f}})^{(u)}_{\zeta n \alpha} = \left[\tilde{\nabla} \cdot \tilde{\mathbf{g}}^{(f)}_{\sigma}(\tilde{\mathbf{x}})\left\{\mathfrak{F}_\alpha f^{(f_\perp)}_{\sigma n \alpha} - f^{(f_\perp)}_{\sigma n \alpha}\right\} + \tilde{\mathbf{f}}^{(u)}_{\nu n \alpha} \cdot \tilde{\nabla} \ell^{(u)}_{\nu}(\tilde{\mathbf{x}})\right]_{\tilde{\mathbf{x}} = \tilde{\mathbf{x}}^{(u)}_{\zeta}}, \tag{7}$$

which can then be used to update the solution at the solution points $u^{(u)}_{\zeta n \alpha}$ via a suitable explicit time integration scheme.

3. New Capabilities

3.1. Cross-Platform Performance

The cross-platform performance of PyFR is enabled through *backends* which can utilise various matrix multiplication kernels and a Mako derived domain specific language (DSL) which achieves complete feature parity across all backends, as per Fig. 2.

Backends. PyFR v0.1.0 had a `C/OpenMP` backend for CPUs and a `CUDA` backend for NVIDIA GPUs. However, since 2013 additional vendors have entered the high-end GPU market including AMD, Intel, and even Apple. As such, PyFR v2.0.3 now contain an additional `HIP` backend for AMD GPUs, an `OpenCL` backend for all GPUs, and a `Metal` backend for Apple GPUs.

DSL. The capabilities and performance of the DSL have been improved in PyFR v2.0.3. In particular, there is language-level support for *reductions*. A kernel argument can now be annotated according to `reduce(op)` where `op` is a reduction operator such as `min`. Whatever value the kernel assigns to this argument will then be automatically and safely reduced with its current value in memory. On the performance side, the DSL is now capable of detecting situations where read-only kernel arguments are likely to be subject to reuse. When running on GPU platforms, the DSL will automatically take care of loading these arguments into shared memory. This helps to reduce pressure on the L1 and L2 caches.

Matrix multiplications. Many operations within an FR time-step can be cast in the form of

$$\mathsf{C} \leftarrow \mathsf{AB} + \beta \mathsf{C},$$

where A is a constant operator matrix, B is an input state matrix, and C is an output state matrix. PyFR v0.1.0 simply offloaded these operations to a platform-specific dense BLAS library such as cuBLAS or OpenBLAS. However, when operating on elements with a tensor-product structure, the operator matrices can exhibit a significant degree of sparsity. This can lead to suboptimal performance in cases where the arithmetic intensity of the operation is beyond that of the underlying hardware.

This issue has been addressed by incorporating additional matrix-multiplication providers into PyFR v2.0.3. When running on with the CUDA and HIP backends, PyFR v2.0.3 will use the GiMMiK [11] library to generate a suite of bespoke fully-unrolled kernels for each A. During the code generation process, GiMMiK automatically elides multiplications through by zero, thus reducing the arithmetic intensity of the operation. The generated kernels are then competitively benchmarked against those provided by the dense BLAS library, with PyFR automatically selecting the fastest kernel for each operation. This auto-tuning is performed autonomously by PyFR at run-time and does not require any direction from the user. When running on CPUs, a similar result is accomplished through the use of libxsmm [12] which includes its own built-in support for automatically choosing between dense and sparse kernels.

Finally, in situations where the flux points are a strict subset of the solution points, additional logic has been incorporated into PyFR to avoid the need for multiplications entirely. This leads to further memory and memory bandwidth savings.

Data layout. The primary data structure in PyFR is an m by n row-major matrix where, up to padding, m is proportional to the number of solution/flux points and n is equal to the product of the number of elements and the number of field variables. It follows that there is a degree of freedom regarding how these field variables are packed along a row. This can be characterised by the stride Δj between two subsequent field variables. The choice of $\Delta j = 1$

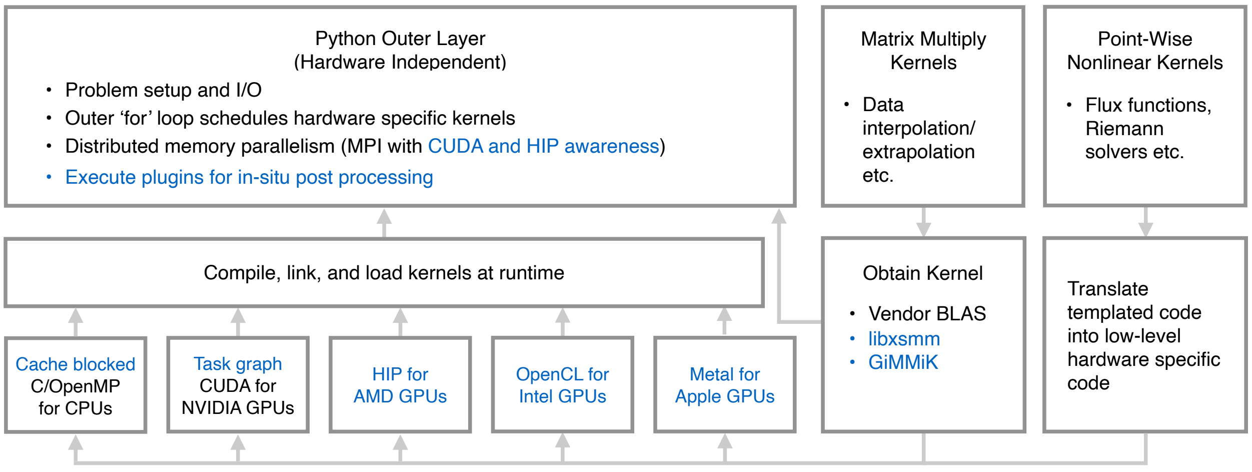

Python Outer Layer
(Hardware Independent)
Problem setup and I/O
Outer 'for' loop schedules hardware specific kernels
Distributed memory parallelism (MPI with CUDA and HIP awareness)
Execute plugins for in-situ post processing
Matrix Multiply Kernels
Data interpolation/ extrapolation etc.
Point-Wise Nonlinear Kernels
Flux functions, Riemann solvers etc.
Compile, link, and load kernels at runtime
Obtain Kernel
Vendor BLAS
libxsmm
GiMMiK
Translate templated code into low-level hardware specific code
Cache blocked C/OpenMP for CPUs
Task graph CUDA for NVIDIA GPUs
HIP for AMD GPUs
OpenCL for Intel GPUs
Metal for Apple GPUs

Figure 2: Overview of how PyFR achieves cross-platform performance. New functionality is marked in blue.

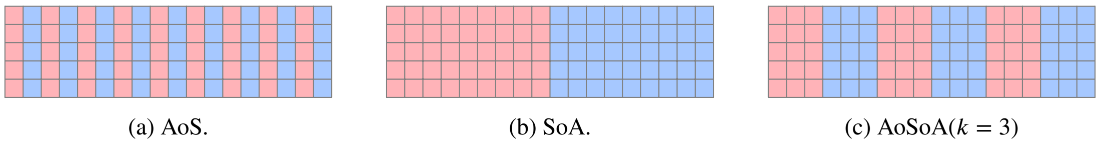


Figure 3: Data layout methodologies for packing multiple field variables into the rows of a matrix.

results in an array of structures (AoS) arrangement, $\Delta j = N_E$ where N_E is the number of elements results in a structure of arrays (SoA) arrangement, and $\Delta j = k$ results in a hybrid array of structure of arrays (AoSoA) approach. An illustration of these arrangements can be seen in Fig. 3.

For simplicity, PyFR v0.1.0 used an SoA approach. However, although this structure is readily amenable to vectorisation, it has some limitations. Firstly, the large stride between field variables decreases the efficiency of caches since adjacent field variables are unlikely to reside in the same cache line. Secondly, it is not friendly to hardware pre-fetchers: an SoA structure with v field variables appears to a CPUs pre-fetcher like v *separate* arrays. Since CPUs are only capable of pre-fetching a finite number of data streams, this can lead to stalls. Finally, given a pointer to one field variable at one point, it is not possible to access the next field variable unless one also knows N_E. To avoid these issues, PyFR v2.0.3 employs the more sophisticated AoSoA packing. The value of k is chosen automatically by the backend based on the vector length of the underlying hardware.

Additionally, when running on CPUs, PyFR v2.0.3 incorporates an additional level of blocking. Rather than allocating a single row-major matrix with n columns, the C/OpenMP backend instead allocates q smaller matrices each with $\sim n/q$ columns. The value of q is chosen to ensure that an entire block can easily remain resident in local caches and serves to further improve data locality.

Task graphs. Strong scaling has also been improved by adding first-class support for *task graphs*. The idea is to exploit the fact that PyFR, as with many scientific codes, repeatedly calls the same sequence of kernels. By treating each kernel as a vertex in a graph and the dependencies between kernels as edges, it is possible—on an *a priori* basis—to form a task graph corresponding to a single right-hand side evaluation. This has two key advantages:

1. It presents the underlying runtime with extra opportunities for extracting parallelism by enabling it to safely identify kernels which can be run in parallel.
2. It enables a substantial reduction in interface overhead since, once constructed, task graphs can be launched with just a single function call as opposed to one function call per kernel.

An example of a task graph for a 2D Navier–Stokes simulation on a mixed-element grid can be seen in Fig. 4. Looking at the graph, we observe that there are four root nodes. As these root nodes are—by definition—independent, it is possible for all four of the kernels to be executed in parallel. Similarly, we observe that the three bcconu boundary

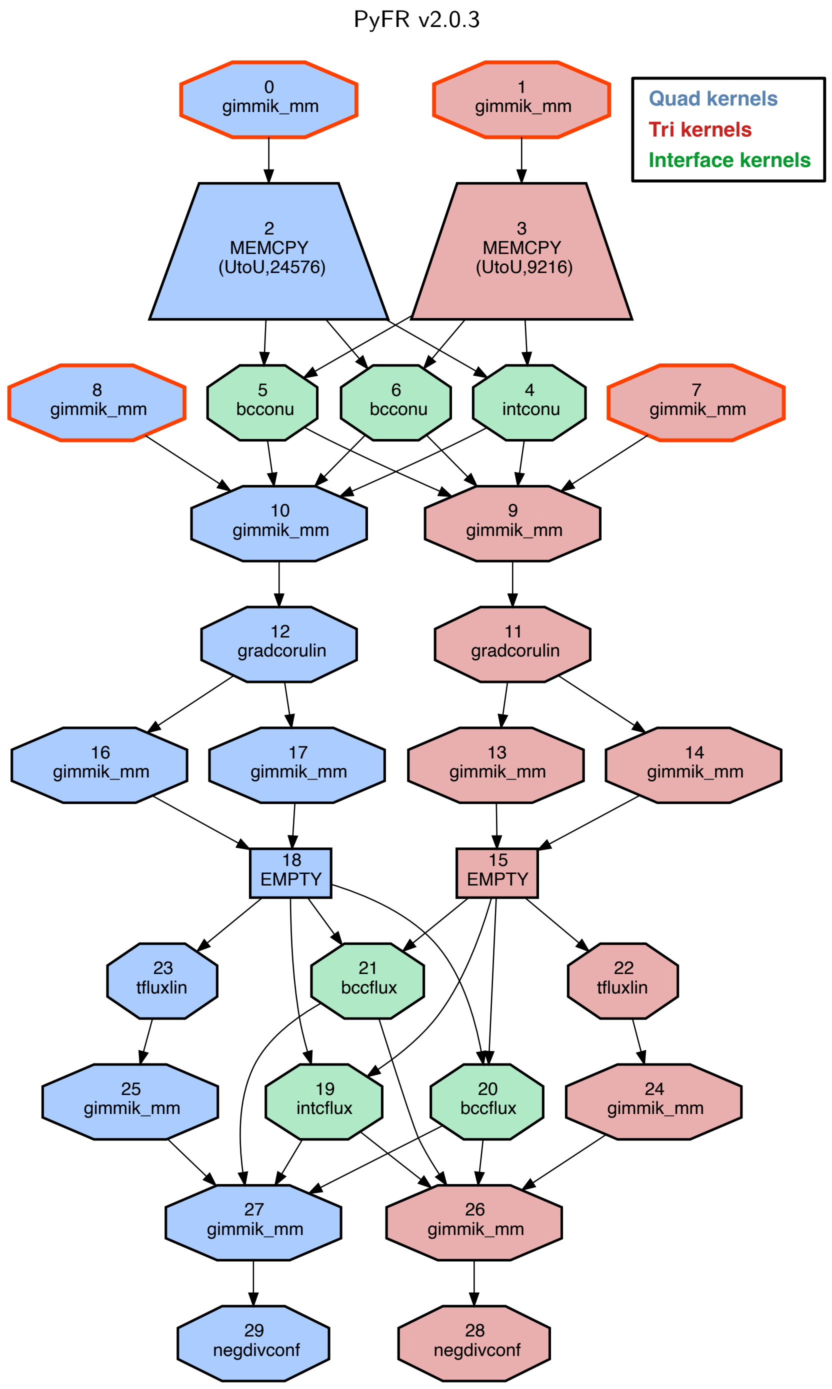


Figure 4: A task graph generated by the NVIDIA `cuGraphDebugDotPrint` API for a 2D mixed-element Navier–Stokes test case. Blue shading indicates kernels for quadrilateral elements, red shading kernels for triangular elements, and green shading for interface kernels. The four root nodes of the graph are marked with red borders.

condition kernels are also independent, such that these can also be executed in parallel. Indeed, careful inspection of the graph shows that there are *always* at least two kernels which may be executed at the same time. When exploited by a backend, this parallelism can improve GPU utilisation which, in turn, leads to improved strong scaling.

On the CUDA backend, PyFR task graphs map directly onto native CUDA graphs. Since the NVIDIA A100 generation, there is hardware support for task graph acceleration, enabling further reductions in overhead. While HIP does support task graphs, preliminary studies show their performance to be inferior to launching kernels directly. As

```
// Kernel 1
for (int i = 0; i < n; i++)
    a[i] += b[i];

// Kernel 2
for (int i = 0; i < n; i++)
    a[i] += c[i];
```

(a) Without blocking; bandwidth $\sim 6n$.

```
// Blocking loop with b = block size
for (int j = 0; j < n; j += b) {
    // Kernel 1
    for (int i = j; i < j + b; i++)
        a[i] += b[i];

    // Kernel 2
    for (int i = j; i < j + b; i++)
        a[i] += c[i];
}
```

(b) With blocking; bandwidth $\sim 4n$.

Figure 5: Example of how cache blocking can be applied to a pair of array addition kernels. In the blocked version when the second kernel updates `a[i]` it will hit in cache, thus saving a write to and read back from main memory.

such, task graphs on the HIP backend are emulated by submitting kernels to a stream in a serial fashion. A similar approach is used on Metal. On OpenCL, task graphs are emulated using out-of-order queues and events. This enables the runtime to identify and exploit inter-kernel parallelism but does not decrease API overhead.

To demonstrate the benefits of native task graphs, we consider the 2D Incompressible Cylinder Flow test case `2d-inc-cylinder` available from the PyFR Test Case repository on GitHub. This is small mixed-element case has a high API overhead, and without task graphs, the run-time on an NVIDIA V100 GPUs is 256 s. However, with task graphs, this reduces to 122 s.

Cache blocking. A powerful means of reducing the memory bandwidth requirements of a code on conventional CPUs is *cache blocking* [13]. The idea is to improve data locality by changing the order in which kernels are called. An example of this can be seen in Fig. 5 which shows how a pair of array addition kernels can be rearranged to reduce bandwidth requirements. A key advantage of cache blocking compared with alternative approaches, such as kernel fusion, is that the kernels themselves do not require modification; all that changes is the arguments to the kernels.

Historically, cache blocking has not been viable for high-order codes due to the size of the intermediate arrays which are generated by kernels. For example, an Intel Ivy Bridge CPU core from 2013 only has 256 KiB of L2 cache which is shared between executable code and data. As a point of reference, for the Euler equations, storing the solution and flux for just eight $\wp = 4$ hexahedra at double precision requires 160 kB. Since 2016, however, there has been a marked increase in the size of private caches, with Intel Golden Cove CPU cores having 2 MiB. The specifics involved in cache blocking FR are detailed in [13, 14] and can improve performance by a factor of two. Within PyFR, cache blocking is accomplished by calling auxiliary methods on task graphs stating which kernels in the graph are suitable for blocking transformations. The interface also contains support for eliminating temporary arrays which can further improve performance.

Multi-node capabilities. Distributed memory parallelism is accomplished via MPI using the mpi4py wrappers [15, 16]. As the message format is standardised across all backends, it is possible for different ranks to employ different backends, thus enabling heterogeneous computing from a homogeneous codebase [17]. In order to improve scalability, the backend interface in PyFR v2.0.3 has been enhanced to allow backends to directly pass GPU device pointers to MPI routines. As such, PyFR v2.0.3 is fully capable of exploiting GPUDirect RDMA on NVIDIA platforms via CUDA Aware MPI, along with its analogue on AMD platforms via HIP Aware MPI. The impact of this technology depends on both the underlying hardware and the degree to which a simulation is strong scaled. In the most extreme cases, twofold performance improvements have been observed when running on clusters of NVIDIA A100 GPUs [18].

3.2. Numerical Stability

PyFR is often used to conduct under-resolved DNS (uDNS), also referred to as ILES of turbulent flow. On account of this under-resolution, the FR scheme is subject to aliasing-driven instabilities, which can cause the simulation to diverge [19]. Additionally, FR schemes exhibit instabilities when solutions contain discontinuities such as shocks.

PyFR v0.1.0 had no specialised capabilities for handling either scenario, beyond simply increasing grid resolution. However, as of PyFR v2.0.3, there are now four separate stabilisation strategies available.

Modal filtering. The simplest stabilisation technique in PyFR v2.0.3 is modal filtering, wherein high-order modes of the solution are periodically filtered as outlined in [10]. This approach is conservative and numerically inexpensive. However, it is an indiscriminate approach - with the filtering applied uniformly across the domain irrespective of whether it is required, and it exposes several free parameters including the filter strength, filter frequency and cut-off modes.

Anti-aliasing. As noted in [19], the origin of aliasing-driven instabilities is the use of a collocation-type projection of the fluxes. PyFR v2.0.3 resolves this issue by using quadrature to perform a least-squares projection of the flux instead. To do this, PyFR employs a series of state-of-the art quadrature rules generated using Polyquad [20], which out-perform those in literature. Although computationally expensive relative to simply performing a collocation projection of the fluxes, studies have shown the results to be markedly superior to those produced by modal filtering [21].

Artificial viscosity. Primarily intended for shock capturing, artificial viscosity is another stabilisation approach provided by PyFR v2.0.3, which dynamically adds extra viscosity into elements whose solutions are exhibiting Gibbs-type phenomena. Based around the widely adopted approach of [22], the method is functional but, as with modal filtering, requires a degree of parameterisation. Additionally, whilst the additional kernels are not particularly expensive—at least within the context of an advection-diffusion type problem such as the Navier–Stokes equations—the process of adding viscosity can have a negative impact on the maximum stable explicit time step. As such, the overall cost of the approach can be high.

Entropy filtering. The final approach provided by PyFR v2.0.3 for stabilisation and shock capturing is entropy filtering [23–26]. This is based around selectively applying a modal filter to elements which violate positivity of density, positivity of pressure or a minimum entropy condition. The filter strength is determined iteratively on a per-element basis, with the goal being to apply as little filtering as possible. As the indicators for instability are physics-based, this method does not typically require any explicit parameterisation. The utility of the approach is demonstrated in the 2D Double Mach Reflection test case `2d-double-mach-reflection` and the 2D Viscous Shock Tube test case `2d-viscous-shock-tube` available from the PyFR Test Case repository on Github.

3.3. Mixed Elements and Domain Decomposition

PyFR v0.1.0 only included support for three element types: quadrilaterals and triangles in two dimensions and hexahedra in three dimensions. Given the difficulties of all-hexahedral meshing around complex geometries, this represented a significant limitation. PyFR v2.0.3 addresses this limitation by adding in complete support for prisms and tetrahedra and partial support for pyramids. Specifically, the pyramid support requires that the quadrilateral base be affine. Given that a major application for pyramids is as a transition layer between a tetrahedral near-field and a hexahedral far-field, this restriction is relatively minor.

One practical complication which arises when running on mixed grids is domain decomposition. The relative performance of different element types is affected by around half a dozen simulation parameters including: the polynomial order, location of solution and flux points and use of anti-aliasing, to name but three. A consequence of this is that when partitioning a grid, it is not possible to employ a single set of element weighting factors. Employing incorrect weighting factors can lead to load imbalances which negatively impact strong scaling. Compared with v0.1.0, PyFR v2.0.3 contains two major improvements in this area.

Firstly, whereas v0.1.0 required grids to be partitioned by the mesh generation software, v2.0.3 includes built-in support for partitioning and re-partitioning both mesh and solution files. This is accomplished by having PyFR call out to the METIS [27] and SCOTCH [28] libraries. Using this functionality, it is relatively simple to experiment with different weightings and change them in concert with the simulation parameters; for example, when restarting a simulation at a higher polynomial order, this functionality can be used to appropriately re-weight the mesh. Moreover, to aid this process, PyFR also includes support for tracking MPI wait times. This information can be used to identify load imbalances between domains, which the user can then employ to derive more appropriate weights.

Secondly, there is also support for *balanced* partitioning wherein PyFR attempts to assign the same number of elements of each type to each domain. This ensures optimal load balancing irrespective of the relative performance

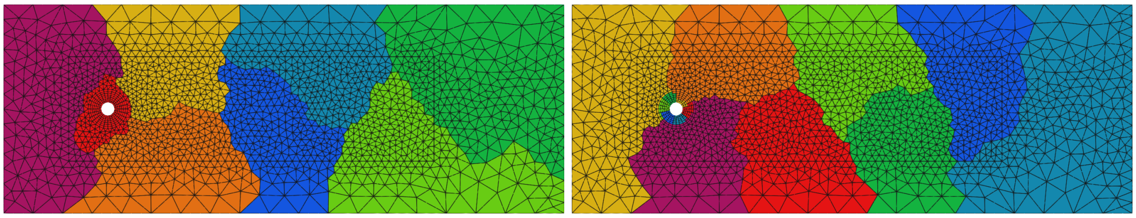

(a) Weighted. (b) Balanced.

Figure 6: Partitionings of a mixed grid with a quadrilateral boundary layer and triangular far-field partitioned into eight parts. For the weighted strategy, each quadrilateral was assigned 3/2 the weight of a triangle.

differential between element types. However, as element types are not uniformly distributed throughout the domain—for example one might have a prismatic boundary layer and a tetrahedral wake—balanced partitioning can lead to partitions becoming non-contiguous when the number of partitions is large. Examples of weighted and balanced partitioning can be seen in Fig. 6 for the 2D Incompressible Cylinder Flow test case `2d-inc-cylinder` available from the PyFR Test Case repository on GitHub.

3.4. Curved Elements

To realise the benefits of high-order schemes, it is necessary to employ grids which, by finite volume standards, are relatively coarse. In order to still accurately represent the underlying geometry, it is therefore important for the elements themselves to be curved. This is accomplished by associating metric terms—which take the form of a 2×2 or 3×3 matrix—within each element.

PyFR v0.1.0 employed the so-called cross-product metric. However, with this approach, the polynomial order of the spatial metric is twice that of the shape function for curved grids—possibly exceeding the order of the solution basis. If this is the case, then the metric terms may not be discretised accurately due to truncation and aliasing errors. In particular, the divergence of the approximated metric terms may become non-zero, which results in a lack of free-stream preservation, *i.e.* the solver cannot maintain a uniform free-stream flow solution.

To overcome this issue, PyFR v2.0.3 instead employs the conservative metric [29, 30], which preserves a uniform free-stream flow even when discretised. Specifically, this approach constructs the metric terms as the curl of the function, thus ensuring they are always divergence-free irrespective of any errors in the function approximation. The approach greatly increases the robustness of PyFR when running on curved grids. An example of its impact can be seen in Fig. 7.

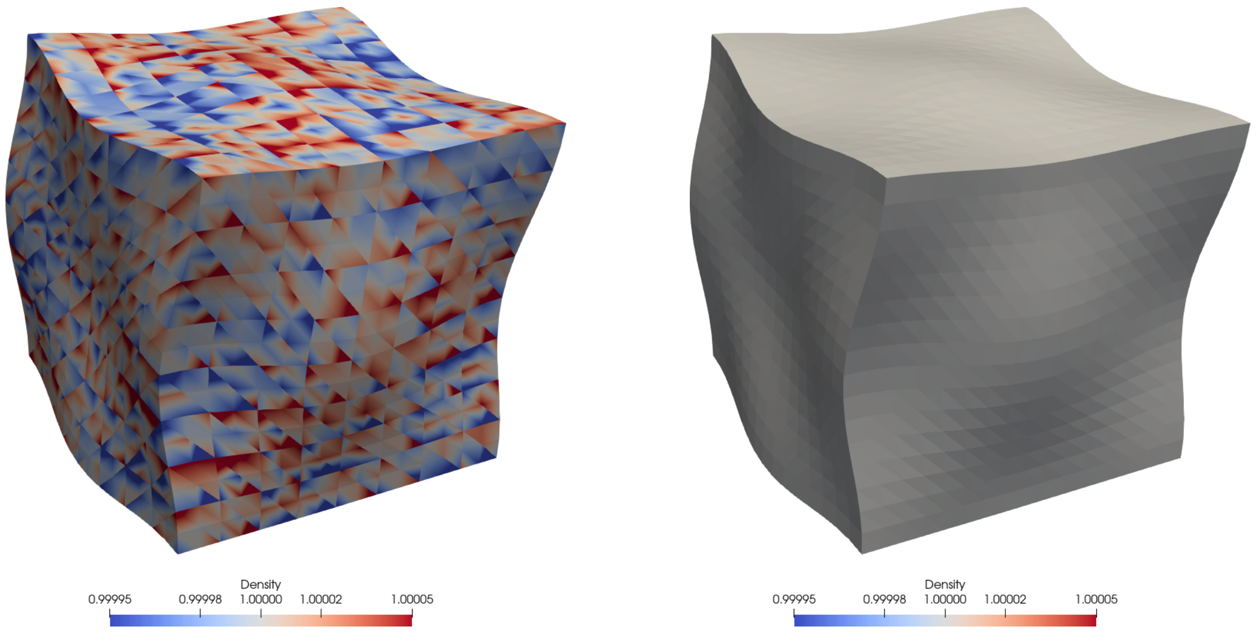
Furthermore, in many real-world grids, only elements in and around the boundary layer are actually curved. PyFR v2.0.3 takes advantage of this fact by identifying linear elements and, in lieu of computing metric terms for each solution point on an *a priori* basis, instead determines them on the fly based off the geometry of the element. This can lead to a substantial saving in memory bandwidth. For example, in a $\wp = 4$ hexahedral element there are $(\wp + 1)^3 = 125$ solution points and hence $3^2 \times 125 = 1,125$ metric terms. However, if the element is linear, then the metric terms are entirely determined by the corner vertices which only involve $3 \times 8 = 24$ terms.

3.5. Adaptive Time Stepping

When using explicit time stepping, the run-time of a simulation is directly proportional to the time step size. However, for non-linear problems, it can be challenging to accurately estimate the maximum stable step size. PyFR v2.0.3 avoids these issues by including support for low storage Runge–Kutta methods with *embedded pairs*. These make it possible to inexpensively obtain an estimate for the numerical error incurred when taking a time step [31]. Once suitably normalised, this error is then used to decide if a time step should be accepted or rejected. Moreover, it is also be used to adapt the step size; increasing for accepted steps and decreasing for rejected steps.

3.6. Incompressible Euler and Navier–Stokes

PyFR v0.1.0 included a compressible Euler and Navier–Stokes solver. In PyFR v2.0.3, support has also been added for the incompressible Navier–Stokes equations. This is accomplished via a combination of the artificial compressibility method of [32] with the dual-time approach of [33]. The result is an iterative scheme which builds extensively upon the



(a) Cross-product metric as used in PyFR v0.1.0.

(b) Conservative metric as used in PyFR v2.0.3.

Figure 7: Simulation of free-stream flow using second-order solution polynomials on a cubically-curved tetrahedral mesh with cross-product metric as used in PyFR v0.1.0 (a), and conservative metric as used in PyFR v2.0.3 (b).

fast residual evaluation capability of PyFR. Convergence is accelerated through a combination of polynomial multigrid [34] and variable local time stepping [35].

3.7. File Formats

The mesh and solution file formats for PyFR v0.1.0 were based around NumPy `.npy` and `.npz` files. Although simple to read and write from Python, they are difficult to access from other environments. Moreover, the formats themselves had no provisions for parallel I/O. For these reasons, PyFR v2.0.3 employs a new set of file formats based around the industry-standard HDF5 [36, 37]. A key advantage of the HDF5 format is its hierarchical nature and the ability to attach arbitrary attributes to most data sets. Moreover, data arrays stored by PyFR use 64-bit rather than 32-bit integers. This enables a partition to have in excess of four billion elements and serves to further future-proof the format for at least the next decade.

To aid in reproducibility, PyFR v2.0.3 solution files embed *all* of the configuration files that have been employed in the simulation up until the current time. This makes it possible to account for the common situation wherein a simulation is started with one configuration file, run for a period of time, and then restarted with a different configuration.

Output format support has also been enhanced since PyFR v0.1.0. Specifically, PyFR v2.0.3 now supports exporting high-order VTU files. This enables the high-order nature of the solution to be preserved throughout more of the post-processing pipeline. Furthermore, there is also support for generating parallel VTU files which can be more efficient in multiprocessing environments.

3.8. Plugin Architecture

The PyFR *plugin* infrastructure provides a lightweight means of adding new capabilities to the code base. Written in pure Python, plugins are capable of adding new command line arguments, periodically post-processing the solution, and adding source-terms to the solver. Examples of a selection of plugins provided with PyFR v2.0.3 are detailed below.

Point sampling. The `soln-plugin-sampler` is capable of periodically sampling a set of points in the domain. At start-up, the plugin automatically determines which element each sample point is inside, and then performs a series of Newton iterations to invert the physical-to-reference space mapping.

Time averaging. The `soln-plugin-tavg` computes the time-average of one or more arbitrary functions. The functions, which are specified in the configuration file, can be parameterised by both the primitive variables and gradients thereof. Beyond computing time-averages, the plugin also computes variances, which can be used for the purposes of uncertainty quantification. Command line support is also included for merging together multiple time-average files. This is particularly useful in environments where there is a limit on job run time.

Force calculation. The `soln-plugin-fluidforce` can be used to compute the net force on boundaries which, in turn, can be used to obtain aerodynamic quantities such as lift and drag. The plugin breaks out separately the pressure and viscous components of the recorded forces.

Turbulence generation. The `solver-plugin-turbulence` implements the synthetic eddy method of [38, 39], which allows turbulence to be injected into any portion of the domain. Specifically, isotropic eddies are injected via a source term formulation, where the turbulence intensity and length scale of the eddies can be specified. The implementation is designed to scale efficiently and minimise memory bandwidth requirements. Specifically, by pre-computing and caching element intersections of every injected eddy *before* the simulation starts, the cost of the implementation is able to scale as the number of eddies intersecting an element at a given time—which for sensible grid resolutions will remain small—as opposed to scaling with the overall number of injected eddies, which can be substantial for large domains and small turbulent length scales. Additionally, the implementation also innovates by passing single unsigned 32-bit integer seeds to define multiple characteristics of a given eddy. These are then unrolled by a device side implementation of a PCG random number generator [40] to produce the actual random characteristics of the eddy. This saves memory bandwidth *cf.* pre-computing random eddy characteristics a priori and passing them as an array of (potentially 64-bit) floats.

In-situ visualisation. The `soln-plugin-ascent` provides in-situ visualisation capabilities and is powered by the lightweight Ascent library [41]. Using the plugin, it is possible to produce complex renderings of the current simulation state without having to write any intermediate files to disk. This enables efficient visualisation of large-scale simulations, for which writing solutions to disk for after-the-fact post-processing is unfeasible.

4. Developer and User Community

Over the past decade, an international community of developers and users has grown around PyFR, drawn from across academia and industry. Key to this growth has been the open-source nature of PyFR, which removes many international and inter-institutional barriers to collaborative code development practices. Code development has also been supported at a technical level by hosting the code base in a Git repository on GitHub, which provides a wide range of tooling and helps define best-practice collaborative processes. Also of importance has been maintaining comprehensive and up-to-date documentation using Sphinx, which is auto-deployed to Read the Docs on each release, as well as providing developer and user support via a forum hosted on Discourse. Finally, in 2020, we launched a virtual PyFR seminar series on Cassyni, which comprises invited talks and discussions on a range of topics related to the theory of high-order FR schemes, their implementation in PyFR and their application to industrially relevant flow problems.

Our user base has successfully applied PyFR to a wide range of fundamental, applied, and industrial flow problems, including studies of flow over turbine cascades [42, 43] with MTU Aero Engines (see Fig. 8), flow over high-rise buildings [44] with Arup (see Fig. 9), flow over Martian rotorcraft aerofoils [45, 46] with NASA (see Fig. 10), flow over supersonic re-entry capsules [47] led by NASA, flow over projectiles [48] led by the Agency for Defense Development in South Korea, flow over wind turbines [49, 50], and flow in thermoacoustic engines [51, 52], as well as studies of airfoil noise reduction [53, 54], flow control [55, 56], wall roughness [57], the Coanda effect [58], surrogate model development [59, 60] and fundamental aspects of channel flow [56, 61]. More recently, PyFR has also been used to enable, for the first time, ILES-based optimisation of turbine cascades [62] and DNS-based optimisation of Martian rotorcraft aerofoils [63].

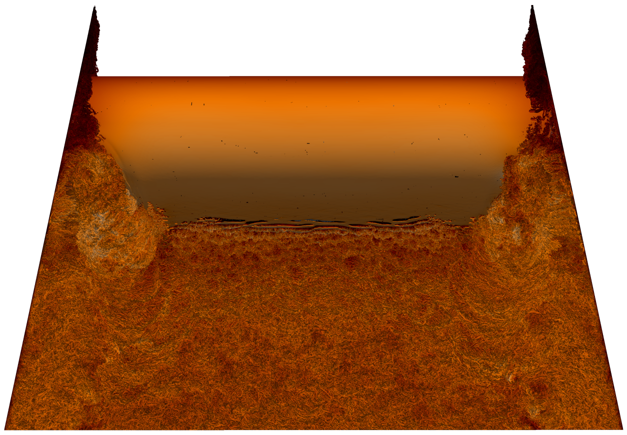


Figure 8: Instantaneous snapshot of a Q-criteria iso-surface coloured by velocity magnitude above the suction side of an MTU-T161 low pressure turbine blade obtained using the compressible Navier–Stokes solver in PyFR. Image is from Fig. 12 of Iyer et. al [43]. Copyright Iyer et al. Reused with permission.

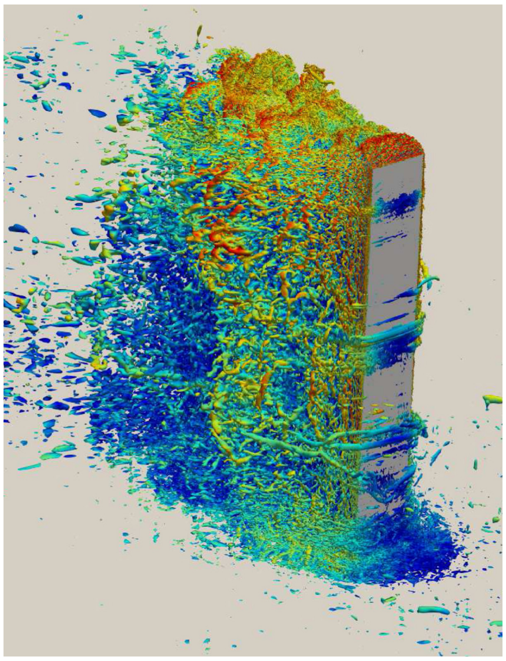


Figure 9: Instantaneous snapshot of a Q-criteria iso-surface coloured by velocity magnitude around a model high-rise building obtained using the incompressible Navier–Stokes solver in PyFR. Image is from Fig. 9 of Giangaspero et. al [44]. Copyright Giangaspero et al. Reused with permission.

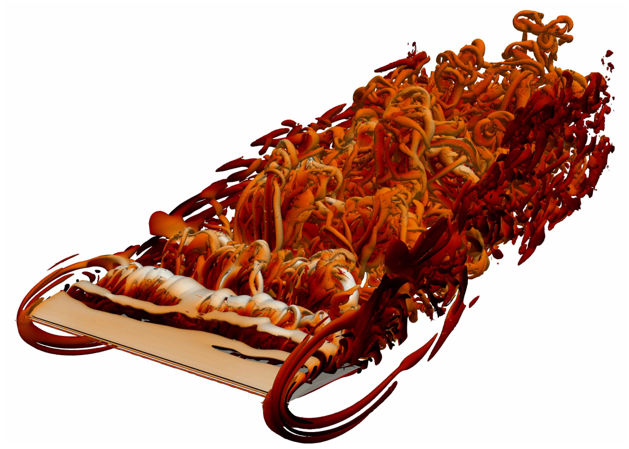


Figure 10: Instantaneous snapshot of a Q-criteria iso-surface coloured by velocity magnitude around a triangular aerofoil for a Martian helicopter obtained using the compressible Navier–Stokes solver in PyFR. Image is from Fig. 10 of Caros et. al [64]. Copyright Caros et al. Reused with permission.

5. Accuracy, Performance, and Scaling

5.1. Accuracy

To demonstrate the accuracy of PyFR v2.0.3 for high-speed flows, we consider a supersonic Taylor–Green vortex test case at a Mach number of 1.25 and a Reynolds number of 1, 600, which was studied in Lusher and Sandham [65] and used to benchmark the accuracy and shock-resolving capabilities of several solvers in Chapelier et al. [66]. Specifically, we solve the compressible Navier-Stokes equations in a domain $-\pi \leq x, y, z \leq \pi$, subject to the following initial conditions,

$$u(t=0,\mathbf{x}) = \sin(x)\cos(y)\cos(z), \tag{8a}$$

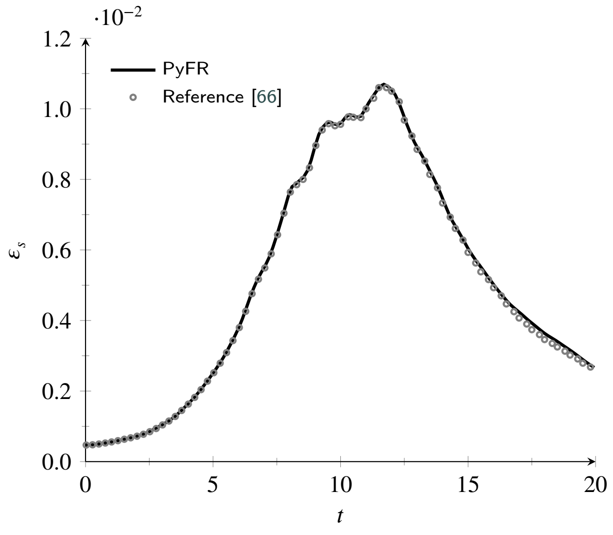
$$v(t=0,\mathbf{x}) = -\cos(x)\sin(y)\cos(z), \tag{8b}$$

$$w(t=0,\mathbf{x}) = 0, \tag{8c}$$

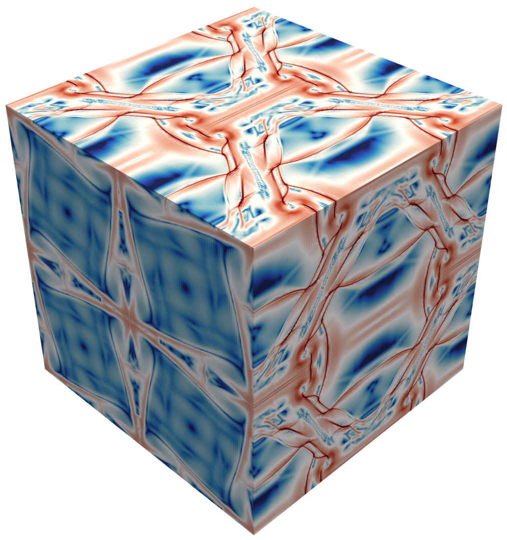
$$p(t=0,\mathbf{x}) = p_0 + \frac{1}{16}\left[\cos(2x)+\cos(2y)\right]\left[2+\cos(2z)\right], \tag{8d}$$

$$\rho(t=0,\mathbf{x}) = p(t=0,\mathbf{x})/p_0, \tag{8e}$$

where p_0 and the reference dynamic viscosity were selected to achieve the desired Mach and Reynolds numbers based on length, velocity and density scales of unity, and a dynamic viscosity computed using Sutherland's law with a reference temperature of 273K [67]. For comparison with the results in Chapelier et al. [66], the simulations were performed using computational meshes consisting of 16^3, 32^3, 64^3 and 128^3 hexahedral elements, with a third-order polynomials approximating the solution within each element (corresponding to 64^3, 128^3, 256^3 and 512^3 DoFs, respectively). Gauss–Legendre–Lobatto flux and solution points were used, and entropy filtering was employed as a shock capturing approach.



(a) Enstrophy.



(b) Schlieren.

Figure 11: Plot of enstrophy as a function of time (left) and a Schlieren-type representation of the density gradient norm at $t = 6$ (right) for the supersonic Taylor–Green vortex case computed with $N = 128^3$ hexahedral elements (512^3 DoFs), along with the reference data of Chapelier et al. [66].

Fig. 11 shows solenoidal dissipation (enstrophy), defined as

$$\varepsilon_s = \frac{1}{(2\pi)^3}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} \mu\,\boldsymbol{\omega}\cdot\boldsymbol{\omega}\;\mathrm{d}x\mathrm{d}y\mathrm{d}z, \tag{9}$$

as a function of time, as well as a Schlieren-type representation of the density gradient norm at $t = 6$ for a case computed with $N = 128^3$ hexahedral elements (512^3 DoFs). The predicted enstrophy profiles are found to be in

Table 1
Summary of the solvers and numerical methods used for comparison.

Solver	Numerical method	Order of accuracy	Shock capturing method
PyFR	Flux Reconstruction	4	Entropy filter
CODA [68]	Modal Discontinuous Galerkin	4	Artificial viscosity
FLEXI [69]	Nodal Discontinuous Galerkin	4	Subcell finite volume
SD3D [70]	Spectral difference	4	Artificial viscosity
OpenSBLI [71]	Finite difference	6	Targeted ENO (TENO)

excellent agreement with the reference data of Chapelier et al. [66], computed using a highly resolved (2048^3 DoFs) high-order finite difference targeted ENO (TENO) scheme, indicating that PyFR can accurately resolve small-scale turbulent flow structures in supersonic flows. This test case allows for further comparison against the results of various solvers presented in Chapelier et al. [66]. In particular, we compare to similar discontinuous finite element-type schemes (e.g., Discontinuous Galerkin, spectral difference, etc.) with identical mesh resolution and approximation order, the details of which are summarized in Table 1, as well as the high-order finite difference TENO scheme which was used to compute the reference results.

Fig. 12 shows dilatational dissipation, defined as

$$\varepsilon_d = \frac{4}{3(2\pi)^3} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} \mu \left(\boldsymbol{\nabla} \cdot \mathbf{u}\right)^2 \, \mathrm{d}x\mathrm{d}y\mathrm{d}z, \tag{10}$$

as a function of time for cases computed using 64^3, 128^3, 256^3 and 512^3 DoFs in comparison to the results of the solvers in Chapelier et al. [66]. The simulations from PyFR generally show less shock dissipation, resulting in dilatational dissipation profiles which are closer to the reference data for a given resolution, indicating that the entropy filtering shock capturing approach does not introduce excessive numerical dissipation and can sharply resolve shock profiles.

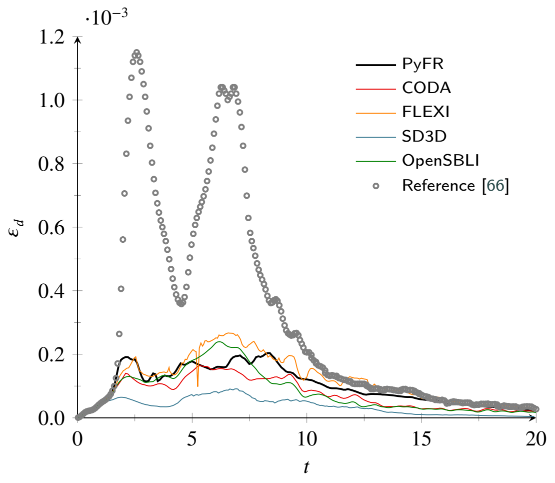
5.2. Performance and Scaling

PyFR has previously been used to undertake petascale simulations on a range of the world's largest GPU supercomputers, including Piz Daint at CSCS and Titan at ORNL. Overall performance and strong and weak scaling has been demonstrated previously in this context, and indeed simulations undertaken with PyFR were shortlisted for the Gordon Bell Prize in 2016 [72], achieving 13.7 DP-PFLOP/s (58 % of theoretical peak) using 18,000 NVIDIA Tesla K20X GPUs on Titan.

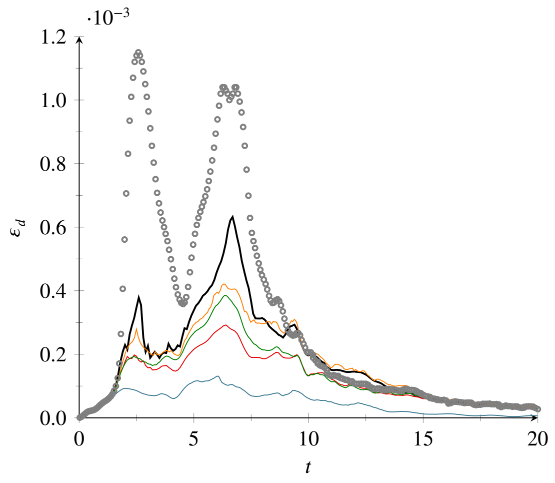
To demonstrate the performance and scaling characteristics of PyFR v2.0.3, we consider a subsonic version of the Taylor–Green vortex test case described above, run with double precision arithmetic on Frontier at ORNL using AMD Instinct MI250X accelerators, and on Alps at CSCS using NVIDIA GH200 GPUs. For this case, the reference pressure was modified to achieve a Mach number of 0.08, and the dynamic viscosity was set to be constant. The computational mesh consisted of $13,891,500$ tetrahedral elements, with seventh-order solution polynomials used to represent the solution within each element, and alpha-optimised flux and solution points were employed. Fig. 13 plots enstrophy as a function of time for a case run on 512 AMD Instinct MI250X accelerators of Frontier (each with two GCDs). Results are found to be in excellent agreement with the reference data of van Rees et al. [73].

Table 2 and Table 3 present strong scaling of the test case on Frontier and Alps, respectively, where we note that on Frontier PyFR was run without HIP-Aware MPI, whereas on Alps, PyFR was run with CUDA-Aware MPI. In terms of scaling, we observe that at 2048 ranks, both platforms deliver similar scalability numbers. However, on account of the superior baseline performance, the NVIDIA system is transferring $\sim$3.7 times more data over the interconnect. Given both systems make use of the Cray Slingshot interconnect, this suggests that the network is *not* the limiting factor on Frontier. Rather, it is more likely related to our inability to run with HIP-aware MPI on Frontier, and the inability of PyFR to employ native HIP graphs due to unresolved issues in the HIP runtime.

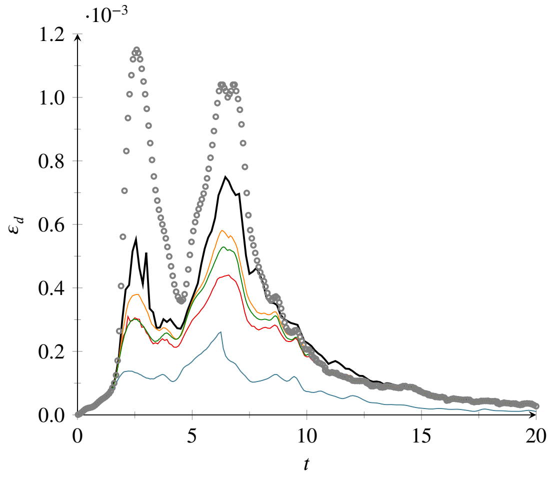
In terms of absolute performance, we note that a single NVIDIA GH200 is $\sim$3.7 times faster than one GCD of an AMD MI250X. A substantial portion of this can be explained by the differences in peak memory bandwidth. A GH200 uses HBM3 memory with a peak bandwidth of 4 TiB / s, whereas the MI250X uses HBM2e memory with a peak bandwidth per-GCD of 1.6 TiB / s. This gives a ratio of 2.5. Moreover, micro-benchmarks on the MI250X indicate that peak bandwidth is only reliably achieved for kernels with a 1:1 read-to-write ratio. Outside of this regime,



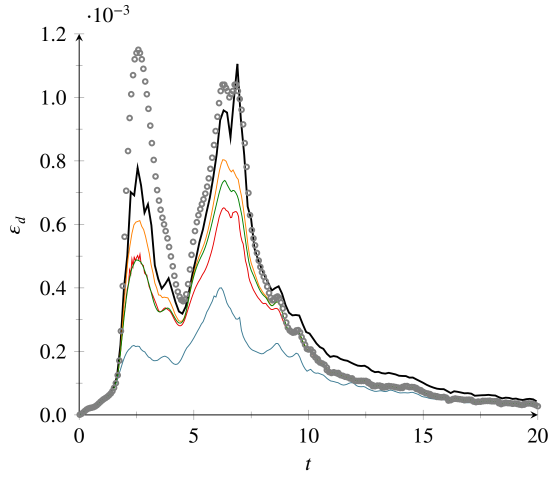
(a) $N_{\mathrm{DoFs}} = 64^3$.



(b) $N_{\mathrm{DoFs}} = 128^3$.



(c) $N_{\mathrm{DoFs}} = 256^3$.



(d) $N_{\mathrm{DoFs}} = 512^3$.

Figure 12: Dilatational dissipation as a function of time for the supersonic Taylor–Green vortex case computed with varying mesh resolution in comparison to the results of the solvers in Chapelier et al. [66].

bandwidths closer to ~1.2 TiB / s are more commonly observed. Such a discrepancy is not observed on NVIDIA hardware, however. Accounting for this gives us a revised performance ratio of ~3.3 which is similar to what is actually observed. The remaining performance differences are likely due to the superior caching setup of the NVIDIA GPU which has—in the absence of shared memory allocations—some 208 KiB available per SM, whereas AMD only provides 16 KiB per CU. Similarly, whereas NVIDIA provide 50 MiB of shared L2 cache, AMD only provide 8 MiB. These caches are important for the interface kernels which have an irregular memory access pattern.

Finally, we can make a comparison between absolute performance almost a decade ago using PyFR v0.2.2 on an NVIDIA K40c GPU [17] with current absolute performance. Specifically, data from Table 6 of [17] for a tetrahedrally-dominated mesh with fourth-order solution polynomials in each element gives an absolute performance of 0.122 GDoF/s per K40c GPU, whereas the 16 rank case from Table 3 here gives an absolute performance of 6.004 GDoF/s per GH200 GPU. This leads to an absolute performance improvement ratio of 49.2, accounting for the totality of both hardware and software improvements over the period, and where we note the ratio is conservative since use of seventh vs. fourth order solution polynomials necessitates substantially more FLOPs/DoF. This conservative estimate of an almost 50× performance increase over the last decade constitutes a substantial step towards the industrial adoption of scale-resolving simulations.

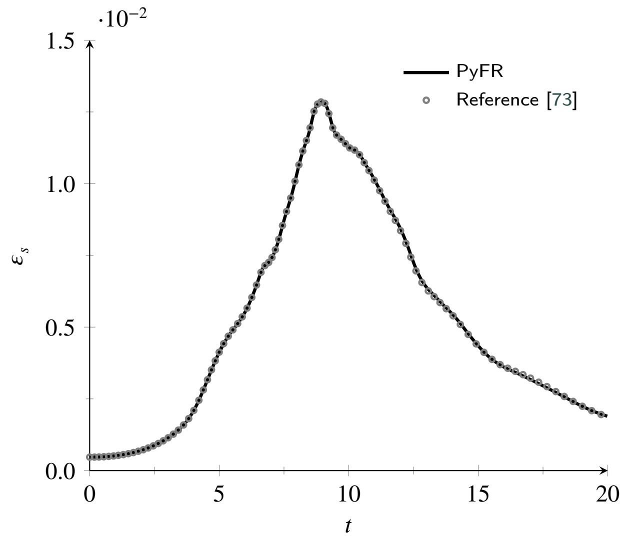


Figure 13: Plot of enstrophy as a function of time for the subsonic Taylor–Green vortex case run with PyFR on 512 AMD Instinct MI250X accelerators of Frontier (each with two GCDs), along with the reference data of van Rees et al. [73].

Table 2
Strong scalability of PyFR on Frontier for the Taylor–Green vortex test case using compressible Navier–Stokes solver on a mesh with $12 \times 105^3 = 13,891,500$ tetrahedral elements and seventh-order solution polynomials used to represent the solution within each element. The speedup is relative to 8 AMD Mi250X accelerators each with two GCDs. HIP-Aware MPI was not employed.

# Ranks	16	32	64	128	256	512	1024	2048
GDoF/s	26.21	50.87	98.70	193.37	369.10	669.80	1133.48	1537.68
Speedup	1.0	1.94	3.77	7.38	14.08	25.56	43.25	58.68
Efficiency	1.00	0.97	0.94	0.92	0.88	0.80	0.68	0.46

Table 3
Strong scalability of PyFR on Alps for the Taylor–Green vortex test case using compressible Navier-Stokes solver on a mesh with $12 \times 105^3 = 13,891,500$ tetrahedral elements and seventh-order solution polynomials used to represent the solution within each element. The speedup is relative to 16 GH200 GPUs. CUDA-Aware MPI was employed.

# Ranks	16	32	64	128	256	512	1024	2048
GDoF/s	96.07	199.60	404.73	806.95	1599.12	2861.56	4632.03	5753.96
Speedup	1.0	2.08	4.21	8.40	16.64	29.79	48.21	59.89
Efficiency	1.00	1.04	1.05	1.05	1.04	0.93	0.75	0.47

6. Conclusions

Since the initial release of PyFR v0.1.0 in 2013 [8], a range of new capabilities have been added to the framework, with a view to enabling industrial adoption. In this work, we have provided details of these enhancements as released in PyFR v2.0.3, including improvements to cross-platform performance (new backends, extensions of the DSL, new matrix multiplication providers, improvements to the data layout, use of task graphs) and improvements to numerical stability (modal filtering, anti-aliasing, artificial viscosity, entropy filtering), as well as the addition of prismatic, tetrahedral and pyramid shaped elements, improved domain decomposition support for mixed element grids, improved handling of curved element meshes, the addition of an adaptive time-stepping capability, the addition of incompressible Euler and Navier-Stokes solvers, improvements to file formats and the development of a plugin architecture. We have also explained efforts to grow an engaged developer and user community and provided a range of examples that demonstrate how our user base is applying PyFR to solve a wide range of fundamental, applied and industrial flow problems. Finally, we have demonstrated the accuracy of PyFR v2.0.3 for a supersonic Taylor-Green vortex case, with shocks and turbulence, and provided latest performance and scaling results on up to 1024 AMD Instinct MI250X accelerators of Frontier at ORNL (each with two GCDs) and up to 2048 Nvidia GH200 GPUs of Alps at CSCS. We note that absolute performance of PyFR accounting for the totality of both hardware and software improvements has, conservatively, increased by almost 50× over the last decade.

Acknowledgements

FDW would like to acknowledge support from the Air Force Office of Scientific Research via grants FA9550-21-1-0190 ("Enabling next-generation heterogeneous computing for massively parallel high-order compressible CFD") and FA9550-23-1-0232 ("Next Generation High-Order Methods for Multi-Physics Multi-Scale Problems") under the direction of Dr. Fariba Fahroo. PEV would like to acknowledge support from the Engineering and Physical Sciences Research Council via awards EP/K027379/1, EP/R030340/1, EP/R029423/1, EP/L000407/1, the European Commission via awards 635962 and 814837, the Leverhulme Trust via a Philip Leverhulme Prize, and Innovate UK via award EP/M50676X/1, and BAE Systems Submarines. BCV acknowledges support from the Natural Sciences and Engineering Research Council of Canada (NSERC) under grants RGPAS-2017-507988 and RGPIN-2017-06773. TD would like to thank Jean-Baptiste Chapelier and collaborators for providing comparison data for simulations in this work. An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract DE-AC05-00OR22725. This work also used computational resources of TSUBAME through the HPCI System Research Project (Project ID: hp190061), was supported by JST FOREST Program (Grant Number JPMJFR2124) and JSPS KAKENHI (Grant Number 24K01074), and was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE–AC52–07NA27344.

References

[1] H. T. Huynh. A flux reconstruction approach to high-order schemes including discontinuous galerkin methods. In *18th AIAA Computational Fluid Dynamics Conference*. American Institute of Aeronautics and Astronautics, June 2007. doi: 10.2514/6.2007-4079.

[2] C. D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, et al. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219, 2015.

[3] D. Moxey, C. D. Cantwell, Y. Bao, A. Cassinelli, G. Castiglioni, S. Chun, E. Juda, E. Kazemi, K. Lackhove, J. Marcon, et al. Nektar++: Enhancing the capability and application of high-fidelity spectral/hp element methods. *Computer Physics Communications*, 249:107110, 2020.

[4] Z.J. Wang, Y. Li, F. Jia, G.M. Laskowski, J. Kopriva, U. Paliath, and R. Bhaskaran. Towards industrial large eddy simulation using the FR/CPR method. *Computers & Fluids*, 156:579–589, 2017.

[5] R. D. Sandberg. Development of a new compressible navier-stokes solver for numerical simulations of flows in turbomachinery. *Progress report for HPC Europa++ Transnational Access Project*, 1264, 2008.

[6] G. A. Bres, S. T. Bose, M. Emory, F. E. Ham, O. T. Schmidt, G. Rigas, and T. Colonius. Large-eddy simulations of co-annular turbulent jet using a voronoi-based mesh generation framework. In *2018 AIAA/CEAS Aeroacoustics Conference*, page 3302, 2018.

[7] K. A. Goc, O. Lehmkuhl, G. I. Park, S. T. Bose, and P. Moin. Large eddy simulation of aircraft at affordable cost: a milestone in computational fluid dynamics. *Flow*, 1:E14, 2021.

[8] F. D. Witherden, A. M. Farrington, and P. E. Vincent. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185(11):3028–3040, 2014.

[9] G. Karniadakis and S. J. Sherwin. *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press, USA, 2005.

[10] J. S. Hesthaven and T. Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. Springer Science & Business Media, 2007.

[11] B. D. Wozniak, F. D. Witherden, F. P. Russell, P. E. Vincent, and P. H. J. Kelly. GiMMiK—generating bespoke matrix multiplication kernels for accelerators: Application to high-order computational fluid dynamics. *Computer Physics Communications*, 202:12–22, 2016.

[12] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 981–991. IEEE, 2016.

[13] S. Akkurt, F. D. Witherden, and P. E. Vincent. Cache blocking strategies applied to flux reconstruction. *Computer Physics Communications*, 271:108193, 2022.

[14] S. Akkurt. Cache Blocking Strategies Applied to Flux Reconstruction. *PyFR Seminar Series*, 2021. doi: 10.52843/gpnpwx.

[15] L. Dalcín, R. Paz, and M. Storti. MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005.

[16] L. Dalcín, R. Paz, M. Storti, and J. D'Elía. MPI for Python: Performance improvements and MPI-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655–662, 2008.

[17] F. D. Witherden, B. C. Vermeire, and P. E. Vincent. Heterogeneous computing on mixed unstructured grids with PyFR. *Computers & Fluids*, 120:173–186, 2015.

[18] S. Mishra, F. D. Witherden, D. Chakravorty, L. Perez, and F. Dang. Scaling study of flow simulations on composable cyberinfrastructure. In *Practice and Experience in Advanced Research Computing*, pages 221–225. 2023.

[19] A. Jameson, P. E. Vincent, and P. Castonguay. On the non-linear stability of flux reconstruction schemes. *Journal of Scientific Computing*, 50:434–445, 2012.

[20] F. D. Witherden and P. E. Vincent. On the identification of symmetric quadrature rules for finite element methods. *Computers & Mathematics with Applications*, 69(10):1232–1241, 2015.

[21] J. S Park, F. D Witherden, and P. E Vincent. High-order implicit large-eddy simulations of flow over a NACA0021 aerofoil. *AIAA journal*, 55 (7):2186–2197, 2017.

[22] P.-O. Persson and J. Peraire. Sub-cell shock capturing for discontinuous Galerkin methods. In *44th AIAA aerospace sciences meeting and exhibit*, page 112, 2006.

[23] T. Dzanic and F. D. Witherden. Positivity-preserving entropy-based adaptive filtering for discontinuous spectral element methods. *Journal of Computational Physics*, 468:111501, 2022.

[24] T. Dzanic and F. D. Witherden. Positivity-preserving entropy filtering for the ideal magnetohydrodynamics equations. *Computers & Fluids*, 266:106056, 2023.

[25] T. Dzanic. Positivity-preserving entropy-based adaptive filtering for shock capturing. *PyFR Seminar Series*, 2022. doi: 10.52843/cassyni.pvy6c0.

[26] Will Trojak and Tarik Dzanic. Positivity-preserving discontinuous spectral element methods for compressible multi-species flows. *Computers & Fluids*, 280:106343, August 2024. doi: 10.1016/j.compfluid.2024.106343.

[27] G. Karypis and V. Kumar. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

[28] Cédric Chevalier and François Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel computing*, 34(6-8):318–331, 2008.

[29] D. A. Kopriva. Metric identities and the discontinuous spectral element method on curvilinear meshes. *Journal of Scientific Computing*, 26: 301–327, 2006.

[30] Y. Abe, T. Haga, T. Nonomura, and K. Fujii. On the freestream preservation of high-order conservative flux-reconstruction schemes. *Journal of Computational Physics*, 281:28–54, 2015.

[31] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer, Berlin, 2 edition, 1993.

[32] A. J. Chorin. A numerical method for solving incompressible viscous flow problems. *Journal of computational physics*, 135(2):118–125, 1997.

[33] A. Jameson. Time dependent calculations using multigrid, with applications to unsteady flows past airfoils and wings. In *10th Computational fluid dynamics conference*, page 1596, 1991.

[34] N. A. Loppi, F. D. Witherden, A. Jameson, and P. E. Vincent. A high-order cross-platform incompressible Navier–Stokes solver via artificial compressibility with application to a turbulent jet. *Computer Physics Communications*, 233:193–205, 2018.

[35] N. A. Loppi, F. D. Witherden, A. Jameson, and P. E. Vincent. Locally adaptive pseudo-time stepping for high-order flux reconstruction. *Journal of Computational Physics*, 399:108913, 2019.

[36] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 workshop on array databases*, pages 36–47, 2011.

[37] A. Collette. *Python and HDF5: unlocking scientific data*. O'Reilly Media, Inc., 2013.

[38] G. Giangaspero, F. D. Witherden, and P. E. Vincent. Synthetic turbulence generation for high-order scale-resolving simulations on unstructured grids. *AIAA Journal*, 60(2):1032–1051, 2022.

[39] G. Giangaspero. Synthetic Turbulence Generation in PyFR. *PyFR Seminar Series*, 2021. doi: 10.52843/cassyni.249z01.

[40] Melissa E. O'Neill. Pcg : A family of simple fast space-efficient statistically good algorithms for random number generation. 2014. URL https://api.semanticscholar.org/CorpusID:3489282.

[41] M. Larsen, E. Brugger, H. Childs, and C. Harrison. Ascent: A flyweight in situ library for exascale simulations. In *In Situ Visualization for Computational Science*, pages 255–279. Springer, 2022.

[42] N. F. Afshar. High-Order Implicit Large Eddy Simulation of Flow over a Low-Reynolds Turbine Cascade. *PyFR Seminar Series*, 2021. doi: 10.52843/cassyni.pw9418.

[43] A.S. Iyer, Y. Abe, B.C. Vermeire, P. Bechlars, R.D. Baier, A. Jameson, F.D. Witherden, and P.E. Vincent. High-order accurate direct numerical simulation of flow over a mtu-t161 low pressure turbine blade. *Computers & Fluids*, 226:104989, 2021.

[44] Giorgio Giangaspero, Luca Amerio, Steven Downie, Alberto Zasso, and Peter Vincent. High-order scale-resolving simulations of extreme wind loads on a model high-rise building. *Journal of Wind Engineering and Industrial Aerodynamics*, 230:105169, 2022.

[45] L. C. Roca. DNS-based optimisation of airfoils for Martian helicopters using PyFR. *PyFR Seminar Series*, 2023. doi: 10.52843/cassyni.6r5ry1.

[46] L. C. Roca. Martian Aerodynamics with PyFR. *PyFR Seminar Series*, 2021. doi: 10.52843/47ly7q.

[47] Rathakrishnan Bhaskaran and Eric C. Stern. *Scale Resolving Simulations of Viking '75 Reentry Capsule Wake Flow*. AIAA, 2024. doi: 10.2514/6.2024-4127. URL https://arc.aiaa.org/doi/abs/10.2514/6.2024-4127.

[48] J. S. Park. High-Order Implicit Large-eddy Simulations of Flow around a Projectile using PyFR. *PyFR Seminar Series*, 2021. doi: 10.52843/cassyni.536kkl.

[49] T. Liang. Actuator Line Model for Wind Turbine Wake Prediction with PyFR. *PyFR Seminar Series*, 2024. doi: 10.52843/cassyni.p3fyks.

[50] Tianyang Liang and Changhong Hu. Numerical simulation of wind turbine wake characteristics by flux reconstruction method. *Renewable Energy*, page 121092, 2024. ISSN 0960-1481. doi: https://doi.org/10.1016/j.renene.2024.121092.

[51] N. Blanc. Simulating a Thermoacoustic Engine With PyFR. *PyFR Seminar Series*, 2023. doi: 10.52843/cassyni.dlsjz8.

[52] Nathan Blanc, Michael Laufer, Steven Frankel, and Guy Z. Ramon. High-fidelity numerical simulations of a standing-wave thermoacoustic engine. *Applied Energy*, 360:122817, 2024.

[53] Z. Wan. Noise reduction of serrated trailing edges with implicit large eddy simulation using PyFR. *PyFR Seminar Series*, 2022. doi: 10.52843/cassyni.br5jn1.

[54] Z. Yuan. Numerical simulations of aerofoil tonal noise reduction by roughness elements. *PyFR Seminar Series*, 2023. doi: 10.52843/cassyni.j59tff.

[55] M. Laufer. Implicit LES of NACA 0018 Airfoil with Active Flow Control. *PyFR Seminar Series*, 2021. doi: 10.52843/cassyni.nd09lk.

[56] H. Foysi. Active control of compressible supersonic wall-bounded flow using direct numerical simulations with spanwise velocity modulation at the walls using PyFR. *PyFR Seminar Series*, 2023. doi: 10.52843/cassyni.2g7fx6.

[57] K. Cengiz. Use of High-Order Curved Elements for Direct and Large Eddy Simulation of Flow over Rough Surfaces. *PyFR Seminar Series*, 2023. doi: 10.52843/cassyni.hkqms2.

[58] T. Regev. GPU-Accelerated High-Fidelity Implicit LES of Coanda Cylinder Flow Instabilities. *PyFR Seminar Series*, 2023. doi: 10.52843/cassyni.5yklnq.

[59] Ali Girayhan Özbay. Unsteady 2D Flow Reconstruction around Arbitrary Shapes via Conformal Mapping aided Deep Neural Networks . *PyFR Seminar Series*, 2022. doi: 10.52843/cassyni.s1q5yf.

[60] Ali Girayhan Özbay and Sylvain Laizet. Deep learning fluid flow reconstruction around arbitrary two-dimensional objects from sparse sensors using conformal mappings. *AIP Advances*, 12(4):045126, 04 2022. ISSN 2158-3226. doi: 10.1063/5.0087488. URL https://doi.org/10.1063/5.0087488.

[61] A. S. Iyer, F. D. Witherden, S. I. Chernyshenko, and P. E. Vincent. Identifying eigenmodes of averaged small-amplitude perturbations to turbulent channel flow. *Journal of Fluid Mechanics*, 875:758–780, 2019. doi: 10.1017/jfm.2019.520.

[62] A. Aubry. Gradient-Free Aerodynamic Shape Optimization using PyFR and MADS. *PyFR Seminar Series*, 2021. doi: 10.52843/cassyni.nqp2sp.

[63] Lidia Caros, Oliver Buxton, and Peter Vincent. Optimization of triangular airfoils for martian helicopters using direct numerical simulations. *AIAA Journal*, 61(11):4935–4945, 2023.

[64] Lidia Caros, Oliver Buxton, Tsuyoshi Shigeta, Takayuki Nagata, Taku Nonomura, Keisuke Asai, and Peter Vincent. Direct numerical simulation of flow over a triangular airfoil under martian conditions. *AIAA Journal*, 60(7):3961–3972, 2022.

[65] David J. Lusher and Neil D. Sandham. Assessment of low-dissipative shock-capturing schemes for the compressible Taylor–Green vortex. *AIAA Journal*, 59(2):533–545, February 2021. doi: 10.2514/1.j059672.

[66] Jean-Baptiste Chapelier, David J. Lusher, William Van Noordt, Christoph Wenzel, Tobias Gibis, Pascal Mossier, Andrea Beck, Guido Lodato, Christoph Brehm, Matteo Ruggeri, Carlo Scalo, and Neil Sandham. Comparison of high-order numerical methodologies for the simulation of the supersonic Taylor–Green vortex flow. *Physics of Fluids*, 36(5), May 2024. doi: 10.1063/5.0206359.

[67] William Sutherland. LII. The viscosity of gases and molecular force. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 36(223):507–531, December 1893. doi: 10.1080/14786449308620508.

[68] Pedro Stefanin Volpiani, Jean-Baptiste Chapelier, Axel Schwöppe, Jens Jägersküpper, and Steeve Champagneux. Aircraft simulations using the new CFD software from ONERA, DLR, and Airbus. *Journal of Aircraft*, 61(3):857–869, May 2024. doi: 10.2514/1.c037506.

[69] Nico Krais, Andrea Beck, Thomas Bolemann, Hannes Frank, David Flad, Gregor Gassner, Florian Hindenlang, Malte Hoffmann, Thomas Kuhn, Matthias Sonntag, and Claus-Dieter Munz. FLEXI: A high order discontinuous Galerkin framework for hyperbolic–parabolic conservation laws. *Computers & Mathematics with Applications*, 81:186–219, January 2021. doi: 10.1016/j.camwa.2020.05.004.

[70] J.-B. Chapelier, G. Lodato, and A. Jameson. A study on the numerical dissipation of the spectral difference method for freely decaying and wall-bounded turbulence. *Computers & Fluids*, 139:261–280, November 2016. doi: 10.1016/j.compfluid.2016.03.006.

[71] David J. Lusher, Satya P. Jammy, and Neil D. Sandham. OpenSBLI: Automated code-generation for heterogeneous computing architectures applied to compressible fluid dynamics on structured grids. *Computer Physics Communications*, 267:108063, October 2021. doi: 10.1016/j.cpc.2021.108063.

[72] P. E. Vincent, F. D. Witherden, B. Vermeire, J. S. Park, and A. Iyer. Towards green aviation with python at petascale. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2016.

[73] Wim M. van Rees, Anthony Leonard, D.I. Pullin, and Petros Koumoutsakos. A comparison of vortex and pseudo-spectral methods for the simulation of periodic vortical flows at high reynolds numbers. *Journal of Computational Physics*, 230(8):2794–2805, 2011.