PyFR Documentation

Release 2.0.0

Imperial College London

Mar 05, 2024

CONTENTS

PyFR 2.0.0 is an open-source flow solver that uses the high-order flux reconstruction method. For more information on the PyFR project visit our website, or to ask a question visit our forum.

Contents:

CHAPTER

ONE

INSTALLATION

1.1 Quick-start

PyFR 2.0.0 can be installed using pip and virtualenv, as shown in the quick-start guides below.

1.1.1 macOS

It is assumed that the Xcode Command Line Tools and Homebrew are already installed. Follow the steps below to setup the OpenMP backend on macOS:

1. Install MPI:

```
brew install mpi4py
```

2. Download and install libxsmm and set the library path:

```
git clone https://github.com/libxsmm/libxsmm.git
cd libxsmm
make -j4 STATIC=0 BLAS=0
export PYFR_XSMM_LIBRARY_PATH=`pwd`/lib/libxsmm.dylib
```

3. Make a venv and activate it:

```
python3.10 -m venv pyfr-venv
source pyfr-venv/bin/activate
```

4. Install PyFR:

```
pip install pyfr
```

5. Add the following to your *Configuration File (.ini)*:

```
[backend-openmp]
cc = gcc-13
```

Note the version of the compiler which must support the `openmp` flag. This has been tested on macOS 13.6.2 with an Apple M1 Max.

1.1.2 Ubuntu

Follow the steps below to setup the OpenMP backend on Ubuntu:

1. Install Python and MPI:

```
sudo apt install python3 python3-pip libopenmpi-dev openmpi-bin
pip3 install virtualenv
```

2. Download and install libxsmm and set the library path:

```
git clone https://github.com/libxsmm/libxsmm.git
cd libxsmm
make -j4 STATIC=0 BLAS=0
export PYFR_XSMM_LIBRARY_PATH=`pwd`/lib/libxsmm.so
```

3. Make a virtualenv and activate it:

```
python3 -m virtualenv pyfr-venv
source pyfr-venv/bin/activate
```

4. Install PyFR:

```
pip install pyfr
```

This has been tested on Ubuntu 22.04.

1.2 Compiling from source

PyFR can be obtained here. To install the software from source, use the provided `setup.py` installer or add the root PyFR directory to `PYTHONPATH` using:

```
user@computer ~/PyFR$ export PYTHONPATH=.:$PYTHONPATH
```

When installing from source, we strongly recommend using pip and virtualenv to manage the Python dependencies.

1.2.1 Dependencies

PyFR 2.0.0 has a hard dependency on Python 3.10+ and the following Python packages:

1. gimmik >= 3.1.1
2. h5py >= 2.10
3. mako >= 1.0.0
4. mpi4py >= 3.0
5. numpy >= 1.26.4
6. platformdirs >= 2.2.0
7. pytools >= 2016.2.1
8. rtree >= 1.0.1

Note that due to a bug in NumPy, PyFR is not compatible with 32-bit Python distributions.

1.2.1.1 CUDA Backend

The CUDA backend targets NVIDIA GPUs with a compute capability of 3.0 or greater. The backend requires:

1. CUDA >= 11.4

1.2.1.2 HIP Backend

The HIP backend targets AMD GPUs which are supported by the ROCm stack. The backend requires:

1. ROCm >= 6.0.0
2. rocBLAS >= 4.0.0

1.2.1.3 Metal Backend

The Metal backend targets Apple silicon GPUs. The backend requires:

1. pyobjc-framework-Metal >= 9.0

1.2.1.4 OpenCL Backend

The OpenCL backend targets a range of accelerators including GPUs from AMD, Intel, and NVIDIA. The backend requires:

1. OpenCL >= 2.1
2. Optionally CLBlast

Note that when running on NVIDIA GPUs the OpenCL backend may terminate with a segmentation fault after the simulation has finished. This is due to a long-standing bug in how the NVIDIA OpenCL implementation handles sub-buffers. As it occurs during the termination phase—after all data has been written out to disk—the issue does *not* impact the functionality or correctness of PyFR.

1.2.1.5 OpenMP Backend

The OpenMP backend targets multi-core x86-64 and ARM CPUs. The backend requires:

1. GCC >= 12.0 or another C compiler with OpenMP 5.1 support
2. libxsmm >= commit bf5313db8bf2edfc127bb715c36353e610ce7c04 in the `main` branch compiled as a shared library (STATIC=0) with BLAS=0.

In order for PyFR to find libxsmm it must be located in a directory which is on the library search path. Alternatively, the path can be specified explicitly by exporting the environment variable `PYFR_XSMM_LIBRARY_PATH=/path/to/libxsmm.so`.

1.2.1.6 Parallel

To partition meshes for running in parallel it is also necessary to have one of the following partitioners installed:

1. METIS >= 5.2
2. SCOTCH >= 7.0

In order for PyFR to find these libraries they must be located in a directory which is on the library search path. Alternatively, the paths can be specified explicitly by exporting environment variables e.g. `PYFR_METIS_LIBRARY_PATH=/path/to/libmetis.so` and/or `PYFR_SCOTCH_LIBRARY_PATH=/path/to/libscotch.so`.

1.2.1.7 Ascent

To run the *[soln-plugin-ascent]* plugin, MPI, VTK-m, and Conduit are required. VTK-m is a supplimentary VTK library, and Conduit is a library that implements the data classes used in Ascent. Detailed information on compilation and installation of Conduit and Ascent can be found in the respective documentation. Ascent must be version >=0.9.0. When compiling Ascent a renderer must be selected to be compiled, currently PyFR only supports the VTK-h option that comes with Ascent. The paths to the libraries may need to be set as an environment variable. For example, on linux you will need:

```
PYFR_CONDUIT_LIBRARY_PATH=/path/to/libconduit.so
PYFR_ASCENT_MPI_LIBRARY_PATH=/path/to/libascent_mpi.so
```

Currently the plugin requires that Ascent and Conduit are 64-bit, this is default when compiling in most cases.

CHAPTER

TWO

USER GUIDE

For information on how to install PyFR see *Installation*.

2.1 Running PyFR

PyFR 2.0.0 uses three distinct file formats:

1. `.ini` — configuration file
2. `.pyfrm` — mesh file
3. `.pyfrs` — solution file

The following commands are available from the `pyfr` program:

1. `pyfr import` — convert a Gmsh .msh file into a PyFR .pyfrm file.

   Example:

   ```
   pyfr import mesh.msh mesh.pyfrm
   ```

2. `pyfr partition` — partition or repartition an existing mesh and associated solution files.

   Example:

   ```
   pyfr partition 2 mesh.pyfrm solution.pyfrs outdir/
   ```

   Here, the newly partitioned mesh and solution are placed into the directory *outdir*. Multiple solutions can be provided. Time-average files can also be partitioned, too.

   For mixed grids one must include the `-e` flag followed by weights for each element type, or the `balanced` argument. Further details can be found in the *performance guide*.

3. `pyfr run` — start a new PyFR simulation. Example:

   ```
   pyfr run mesh.pyfrm configuration.ini
   ```

4. `pyfr restart` — restart a PyFR simulation from an existing solution file. Example:

   ```
   pyfr restart mesh.pyfrm solution.pyfrs
   ```

   It is also possible to restart with a different configuration file. Example:

   ```
   pyfr restart mesh.pyfrm solution.pyfrs configuration.ini
   ```

5. `pyfr export` — convert a PyFR `.pyfrs` file into an unstructured VTK `.vtu` or `.pvtu` file. If a `-k` flag is provided with an integer argument then `.pyfrs` elements are converted to high-order VTK cells which are exported, where the order of the VTK cells is equal to the value of the integer argument. Example:

```
pyfr export -k 4 mesh.pyfrm solution.pyfrs solution.vtu
```

   If a `-d` flag is provided with an integer argument then `.pyfrs` elements are subdivided into linear VTK cells which are exported, where the number of sub-divisions is equal to the value of the integer argument. Example:

```
pyfr export -d 4 mesh.pyfrm solution.pyfrs solution.vtu
```

   If no flags are provided then `.pyfrs` elements are converted to high-order VTK cells which are exported, where the order of the cells is equal to the order of the solution data in the `.pyfrs` file.

   By default all of the fields in the `.pyfrs` file will be exported. If only a specific field is desired this can be specified with the `-f` flag; for example `-f density -f velocity` will only export the *density* and *velocity* fields.

PyFR can be run in parallel. To do so prefix `pyfr` with `mpiexec -n <cores/devices>`. Note that the mesh must be pre-partitioned, and the number of cores or devices must be equal to the number of partitions.

2.2 Configuration File (.ini)

The .ini configuration file parameterises the simulation. It is written in the INI format. Parameters are grouped into sections. The roles of each section and their associated parameters are described below. Note that both `;` and `#` may be used as comment characters. Additionally, all parameter values support environment variable expansion.

2.2.1 Backends

These sections detail how the solver will be configured for a range of different hardware platforms. If a hardware specific backend section is omitted, then PyFR will fall back to built-in default settings.

2.2.1.1 [backend]

Parameterises the backend with

1. `precision` — number precision, note not all backends support double precision:

   `single` | `double`

2. **`memory-model` — if to enable support for large working sets;**
   should be `normal` unless a memory-model error is encountered:

   `normal` | `large`

3. `rank-allocator` — MPI rank allocator:

   `linear` | `random`

4. `collect-wait-times` — if to track MPI request wait times or not:

   `True` | `False`

5. `collect-wait-times-len` — size of the wait time history buffer:

   int

Example:

```
[backend]
precision = double
rank-allocator = linear
```

2.2.1.2 [backend-cuda]

Parameterises the CUDA backend with

1. `device-id` — method for selecting which device(s) to run on:

   int | `round-robin` | `local-rank` | `uuid`

2. `mpi-type` — type of MPI library that is being used:

   `standard` | `cuda-aware`

3. `cflags` — additional NVIDIA realtime compiler (`nvrtc`) flags:

   string

4. `cublas-nkerns` — number of kernel algorithms to try when benchmarking, defaults to 512:

   int

5. `gimmik-nkerns` — number of kernel algorithms to try when benchmarking, defaults to 8:

   int

6. `gimmik-nbench` — number of benchmarking runs for each kernel, defaults to 5:

   int

Example:

```
[backend-cuda]
device-id = round-robin
mpi-type = standard
```

2.2.1.3 [backend-hip]

Parameterises the HIP backend with

1. `device-id` — method for selecting which device(s) to run on:

   int | `local-rank` | `uuid`

2. `mpi-type` — type of MPI library that is being used:

   `standard` | `hip-aware`

3. `rocblas-nkerns` — number of kernel algorithms to try when benchmarking, defaults to 2048:

   int

4. `gimmik-nkerns` — number of kernel algorithms to try when benchmarking, defaults to 8:

   int

5. `gimmik-nbench` — number of benchmarking runs for each kernel, defaults to 5:

   int

Example:

```
[backend-hip]
device-id = local-rank
mpi-type = standard
```

2.2.1.4 [backend-metal]

Parameterises the Metal backend with

1. `gimmik-max-nnz` — cutoff for GiMMiK in terms of the number of non-zero entires in a constant matrix, defaults to 2048:

   int

2. `gimmik-nkerns` — number of kernel algorithms to try when benchmarking, defaults to 18:

   int

3. `gimmik-nbench` — number of benchmarking runs for each kernel, defaults to 40:

   int

Example:

```
[backend-metal]
gimmik-max-nnz = 512
```

2.2.1.5 [backend-opencl]

Parameterises the OpenCL backend with

1. `platform-id` — for selecting platform id:

   int | *string*

2. `device-type` — for selecting what type of device(s) to run on:

   `all` | `cpu` | `gpu` | `accelerator`

3. `device-id` — for selecting which device(s) to run on:

   int | *string* | `local-rank` | `uuid`

4. `gimmik-max-nnz` — cutoff for GiMMiK in terms of the number of non-zero entires in a constant matrix, defaults to 2048:

   int

5. `gimmik-nkerns` — number of kernel algorithms to try when benchmarking, defaults to 8:

   int

6. `gimmik-nbench` — number of benchmarking runs for each kernel, defaults to 5:

   int

Example:

```
[backend-opencl]
platform-id = 0
device-type = gpu
```

(continues on next page)

(continued from previous page)

```
device-id = local-rank
gimmik-max-nnz = 512
```

2.2.1.6 [backend-openmp]

Parameterises the OpenMP backend with

1. `cc` — C compiler:

   string

2. `cflags` — additional C compiler flags:

   string

3. `alignb` — alignment requirement in bytes; must be a power of two and at least 32:

   int

4. `schedule` — OpenMP loop scheduling scheme:

   `static` | `dynamic` | `dynamic, n` | `guided` | `guided, n`

   where *n* is a positive integer.

Example:

```
[backend-openmp]
cc = gcc
```

2.2.2 Systems

These sections setup and control the physical system being solved, as well as charateristics of the spatial and temporal schemes to be used.

2.2.2.1 [constants]

Sets constants used in the simulation

1. `gamma` — ratio of specific heats for `euler` | `navier-stokes`:

   float

2. `mu` — dynamic viscosity for `navier-stokes`:

   float

3. `nu` — kinematic viscosity for `ac-navier-stokes`:

   float

4. `Pr` — Prandtl number for `navier-stokes`:

   float

5. `cpTref` — product of specific heat at constant pressure and reference temperature for `navier-stokes` with Sutherland's Law:

   float

6. `cpTs` — product of specific heat at constant pressure and Sutherland temperature for `navier-stokes` with Sutherland's Law:

   float

7. `ac-zeta` — artificial compressibility factor for `ac-euler` | `ac-navier-stokes`

   float

Other constant may be set by the user which can then be used throughout the `.ini` file.

Example:

```
[constants]
; PyFR Constants
gamma = 1.4
mu = 0.001
Pr = 0.72

; User Defined Constants
V_in = 1.0
P_out = 20.0
```

2.2.2.2 [solver]

Parameterises the solver with

1. `system` — governing system:

   `euler` | `navier-stokes` | `ac-euler` | `ac-navier-stokes`

   where

   `euler` requires

   - `shock-capturing` — shock capturing scheme:

     `none` | `entropy-filter`

   `navier-stokes` requires

   - `viscosity-correction` — viscosity correction:

     `none` | `sutherland`

   - `shock-capturing` — shock capturing scheme:

     `none` | `artificial-viscosity` | `entropy-filter`

2. `order` — order of polynomial solution basis:

   int

3. `anti-alias` — type of anti-aliasing:

   `flux` | `surf-flux` | `flux, surf-flux`

Example:

```
[solver]
system = navier-stokes
order = 3
anti-alias = flux
```

(continues on next page)

(continued from previous page)

```
viscosity-correction = none
shock-capturing = entropy-filter
```

2.2.2.3 [solver-time-integrator]

Parameterises the time-integration scheme used by the solver with

1. `formulation` — formulation:

   `std` | `dual`

   where

   `std` requires

   - `scheme` — time-integration scheme

     `euler` | `rk34` | `rk4` | `rk45` | `tvd-rk3`

   - `tstart` — initial time

     float

   - `tend` — final time

     float

   - `dt` — time-step

     float

   - `controller` — time-step controller

     `none` | `pi`

     where

     `pi` only works with `rk34` and `rk45` and requires

     - `atol` — absolute error tolerance

       float

     - `rtol` — relative error tolerance

       float

     - `errest-norm` — norm to use for estimating the error

       `uniform` | `l2`

     - `safety-fact` — safety factor for step size adjustment (suitable range 0.80-0.95)

       float

     - `min-fact` — minimum factor by which the time-step can change between iterations (suitable range 0.1-0.5)

       float

     - `max-fact` — maximum factor by which the time-step can change between iterations (suitable range 2.0-6.0)

       float

     - `dt-max` — maximum permissible time-step

float

`dual` requires

- `scheme` — time-integration scheme

  `backward-euler` | `sdirk33` | `sdirk43`

- `pseudo-scheme` — pseudo time-integration scheme

  `euler` | `rk34` | `rk4` | `rk45` | `tvd-rk3` | `vermeire`

- `tstart` — initial time

  float

- `tend` — final time

  float

- `dt` — time-step

  float

- `controller` — time-step controller

  `none`

- `pseudo-dt` — pseudo time-step

  float

- `pseudo-niters-max` — minimum number of iterations

  int

- `pseudo-niters-min` — maximum number of iterations

  int

- `pseudo-resid-tol` — pseudo residual tolerance

  float

- `pseudo-resid-norm` — pseudo residual norm

  `uniform` | `l2`

- `pseudo-controller` — pseudo time-step controller

  `none` | `local-pi`

  where

  `local-pi` only works with `rk34` and `rk45` and requires

  - `atol` — absolute error tolerance

    float

  - `safety-fact` — safety factor for pseudo time-step size adjustment (suitable range 0.80-0.95)

    float

  - `min-fact` — minimum factor by which the local pseudo time-step can change between iterations (suitable range 0.98-0.998)

    float

- `max-fact` — maximum factor by which the local pseudo time-step can change between iterations (suitable range 1.001-1.01)

  float

- `pseudo-dt-max-mult` — maximum permissible local pseudo time-step given as a multiplier of `pseudo-dt` (suitable range 2.0-5.0)

  float

Example:

```
[solver-time-integrator]
formulation = std
scheme = rk45
controller = pi
tstart = 0.0
tend = 10.0
dt = 0.001
atol = 0.00001
rtol = 0.00001
errest-norm = l2
safety-fact = 0.9
min-fact = 0.3
max-fact = 2.5
```

2.2.2.4 [solver-dual-time-integrator-multip]

Parameterises multi-p for dual time-stepping with

1. `pseudo-dt-fact` — factor by which the pseudo time-step size changes between multi-p levels:

   float

2. `cycle` — nature of a single multi-p cycle:

   `[(order, nsteps), (order, nsteps), ... (order, nsteps)]`

   where `order` in the first and last bracketed pair must be the overall polynomial order used for the simulation, `order` can only change by one between subsequent bracketed pairs, and `nsteps` is a non-negative rational number.

Example:

```
[solver-dual-time-integrator-multip]
pseudo-dt-fact = 2.3
cycle = [(3, 0.1), (2, 0.1), (1, 0.2), (0, 1.4), (1, 1.1), (2, 1.1), (3, 4.5)]
```

Used in the following Examples:

1. *2D Incompressible Cylinder Flow*

2.2.2.5 [solver-entropy-filter]

Parameterises entropy filter for shock capturing with

1. `d-min` — minimum allowable density:

   float

2. `p-min` — minimum allowable pressure:

   float

3. `e-tol` — entropy tolerance:

   float

Example:

```
[solver-entropy-filter]
d-min = 1e-6
p-min = 1e-6
e-tol = 1e-6
```

Used in the following Examples:

1. *2D Double Mach Reflection*
2. *2D Viscous Shock Tube*

2.2.2.6 [solver-artificial-viscosity]

Parameterises artificial viscosity for shock capturing with

1. `max-artvisc` — maximum artificial viscosity:

   float

2. `s0` — sensor cut-off:

   float

3. `kappa` — sensor range:

   float

Example:

```
[solver-artificial-viscosity]
max-artvisc = 0.01
s0 = 0.01
kappa = 5.0
```

2.2.2.7 [solver-interfaces]

Parameterises the interfaces with

1. `riemann-solver` — type of Riemann solver:

   `rusanov` | `hll` | `hllc` | `roe` | `roem` | `exact`

   where

   `hll` | `hllc` | `roe` | `roem` | `exact` do not work with `ac-euler` | `ac-navier-stokes`

2. `ldg-beta` — beta parameter used for LDG:

   float

3. `ldg-tau` — tau parameter used for LDG:

   float

Example:

```
[solver-interfaces]
riemann-solver = rusanov
ldg-beta = 0.5
ldg-tau = 0.1
```

2.2.2.8 [soln-filter]

Parameterises an exponential solution filter with

1. `nsteps` — apply filter every `nsteps`:

   int

2. `alpha` — strength of filter:

   float

3. `order` — order of filter:

   int

4. `cutoff` — cutoff frequency below which no filtering is applied:

   int

Example:

```
[soln-filter]
nsteps = 10
alpha = 36.0
order = 16
cutoff = 1
```

2.2.3 Boundary and Initial Conditions

These sections allow users to set the boundary and initial conditions of calculations.

2.2.3.1 [soln-bcs-*name*]

Parameterises constant, or if available space (x, y, [z]) and time (t) dependent, boundary condition labelled *name* in the .pyfrm file with

1. `type` — type of boundary condition:

   `ac-char-riem-inv | ac-in-fv | ac-out-fp | char-riem-inv | no-slp-adia-wall | no-slp-isot-wall | no-slp-wall | slp-adia-wall | slp-wall | sub-in-frv | sub-in-ftpttang | sub-out-fp | sup-in-fa | sup-out-fn`

   where

   `ac-char-riem-inv` only works with `ac-euler | ac-navier-stokes` and requires

   - `ac-zeta` — artificial compressibility factor for boundary (increasing `ac-zeta` makes the boundary less reflective allowing larger deviation from the target state)

     float

   - `niters` — number of Newton iterations

     int

   - `p` — pressure

     float | *string*

   - `u` — x-velocity

     float | *string*

   - `v` — y-velocity

     float | *string*

   - `w` — z-velocity

     float | *string*

   `ac-in-fv` only works with `ac-euler | ac-navier-stokes` and requires

   - `u` — x-velocity

     float | *string*

   - `v` — y-velocity

     float | *string*

   - `w` — z-velocity

     float | *string*

   `ac-out-fp` only works with `ac-euler | ac-navier-stokes` and requires

   - `p` — pressure

     float | *string*

   `char-riem-inv` only works with `euler | navier-stokes` and requires

   - `rho` — density

float | *string*

- `u` — x-velocity

  float | *string*

- `v` — y-velocity

  float | *string*

- `w` — z-velocity

  float | *string*

- `p` — static pressure

  float | *string*

`no-slp-adia-wall` only works with `navier-stokes`

`no-slp-isot-wall` only works with `navier-stokes` and requires

- `u` — x-velocity of wall

  float

- `v` — y-velocity of wall

  float

- `w` — z-velocity of wall

  float

- `cpTw` — product of specific heat capacity at constant pressure and temperature of wall

  float

`no-slp-wall` only works with `ac-navier-stokes` and requires

- `u` — x-velocity of wall

  float

- `v` — y-velocity of wall

  float

- `w` — z-velocity of wall

  float

`slp-adia-wall` only works with `euler` | `navier-stokes`

`slp-wall` only works with `ac-euler` | `ac-navier-stokes`

`sub-in-frv` only works with `navier-stokes` and requires

- `rho` — density

  float | *string*

- `u` — x-velocity

  float | *string*

- `v` — y-velocity

  float | *string*

- `w` — z-velocity

float | *string*

`sub-in-ftpttang` only works with `navier-stokes` and requires

- `pt` — total pressure

  float

- `cpTt` — product of specific heat capacity at constant pressure and total temperature

  float

- `theta` — azimuth angle (in degrees) of inflow measured in the x-y plane relative to the positive x-axis

  float

- `phi` — inclination angle (in degrees) of inflow measured relative to the positive z-axis

  float

`sub-out-fp` only works with `navier-stokes` and requires

- `p` — static pressure

  float | *string*

`sup-in-fa` only works with `euler` | `navier-stokes` and requires

- `rho` — density

  float | *string*

- `u` — x-velocity

  float | *string*

- `v` — y-velocity

  float | *string*

- `w` — z-velocity

  float | *string*

- `p` — static pressure

  float | *string*

`sup-out-fn` only works with `euler` | `navier-stokes`

Example:

```
[soln-bcs-bcwallupper]
type = no-slp-isot-wall
cpTw = 10.0
u = 1.0
```

Simple periodic boundary conditions are supported; however, their behaviour is not controlled through the `.ini` file, instead it is handled at the mesh generation stage. Two faces may be taged with `periodic_x_l` and `periodic_x_r`, where `x` is a unique identifier for the pair of boundaries. Currently, only periodicity in a single cardinal direction is supported, for example, the planes `(x,y,0)`` and `(x,y,10)`.

2.2.3.2 [soln-ics]

Parameterises space (x, y, [z]) dependent initial conditions with

1. `rho` — initial density distribution for `euler` | `navier-stokes`:

   string

2. `u` — initial x-velocity distribution for `euler` | `navier-stokes` | `ac-euler` | `ac-navier-stokes`:

   string

3. `v` — initial y-velocity distribution for `euler` | `navier-stokes` | `ac-euler` | `ac-navier-stokes`:

   string

4. `w` — initial z-velocity distribution for `euler` | `navier-stokes` | `ac-euler` | `ac-navier-stokes`:

   string

5. `p` — initial static pressure distribution for `euler` | `navier-stokes` | `ac-euler` | `ac-navier-stokes`:

   string

Example:

```
[soln-ics]
rho = 1.0
u = x*y*sin(y)
v = z
w = 1.0
p = 1.0/(1.0+x)
```

2.2.4 Nodal Point Sets

Solution point sets must be specified for each element type that is used and flux point sets must be specified for each interface type that is used. If anti-aliasing is enabled then quadrature point sets for each element and interface type that is used must also be specified. For example, a 3D mesh comprised only of prisms requires a solution point set for prism elements and flux point sets for quadrilateral and triangular interfaces.

2.2.4.1 [solver-interfaces-line{-mg-p*order*}]

Parameterises the line interfaces, or if -mg-p*order* is suffixed the line interfaces at multi-p level *order*, with

1. `flux-pts` — location of the flux points on a line interface:

   `gauss-legendre` | `gauss-legendre-lobatto`

2. `quad-deg` — degree of quadrature rule for anti-aliasing on a line interface:

   int

3. `quad-pts` — name of quadrature rule for anti-aliasing on a line interface:

   `gauss-legendre` | `gauss-legendre-lobatto`

Example:

```
[solver-interfaces-line]
flux-pts = gauss-legendre
quad-deg = 10
quad-pts = gauss-legendre
```

2.2.4.2 [solver-interfaces-quad{-mg-p*order*}]

Parameterises the quadrilateral interfaces, or if -mg-p*order* is suffixed the quadrilateral interfaces at multi-p level *order*, with

1. `flux-pts` — location of the flux points on a quadrilateral interface:

   `gauss-legendre` | `gauss-legendre-lobatto`

2. `quad-deg` — degree of quadrature rule for anti-aliasing on a quadrilateral interface:

   int

3. `quad-pts` — name of quadrature rule for anti-aliasing on a quadrilateral interface:

   `gauss-legendre` | `gauss-legendre-lobatto` | `witherden-vincent`

Example:

```
[solver-interfaces-quad]
flux-pts = gauss-legendre
quad-deg = 10
quad-pts = gauss-legendre
```

2.2.4.3 [solver-interfaces-tri{-mg-p*order*}]

Parameterises the triangular interfaces, or if -mg-p*order* is suffixed the triangular interfaces at multi-p level *order*, with

1. `flux-pts` — location of the flux points on a triangular interface:

   `alpha-opt` | `williams-shunn`

2. `quad-deg` — degree of quadrature rule for anti-aliasing on a triangular interface:

   int

3. `quad-pts` — name of quadrature rule for anti-aliasing on a triangular interface:

   `williams-shunn` | `witherden-vincent`

Example:

```
[solver-interfaces-tri]
flux-pts = williams-shunn
quad-deg = 10
quad-pts = williams-shunn
```

2.2.4.4 [solver-elements-quad{-mg-p*order*}]

Parameterises the quadrilateral elements, or if -mg-p*order* is suffixed the quadrilateral elements at multi-p level *order*, with

1. `soln-pts` — location of the solution points in a quadrilateral element:

   `gauss-legendre` | `gauss-legendre-lobatto`

2. `quad-deg` — degree of quadrature rule for anti-aliasing in a quadrilateral element:

   int

3. `quad-pts` — name of quadrature rule for anti-aliasing in a quadrilateral element:

   `gauss-legendre` | `gauss-legendre-lobatto` | `witherden-vincent`

Example:

```
[solver-elements-quad]
soln-pts = gauss-legendre
quad-deg = 10
quad-pts = gauss-legendre
```

2.2.4.5 [solver-elements-tri{-mg-p*order*}]

Parameterises the triangular elements, or if -mg-p*order* is suffixed the triangular elements at multi-p level *order*, with

1. `soln-pts` — location of the solution points in a triangular element:

   `alpha-opt` | `williams-shunn`

2. `quad-deg` — degree of quadrature rule for anti-aliasing in a triangular element:

   int

3. `quad-pts` — name of quadrature rule for anti-aliasing in a triangular element:

   `williams-shunn` | `witherden-vincent`

Example:

```
[solver-elements-tri]
soln-pts = williams-shunn
quad-deg = 10
quad-pts = williams-shunn
```

2.2.4.6 [solver-elements-hex{-mg-p*order*}]

Parameterises the hexahedral elements, or if -mg-p*order* is suffixed the hexahedral elements at multi-p level *order*, with

1. `soln-pts` — location of the solution points in a hexahedral element:

   `gauss-legendre` | `gauss-legendre-lobatto`

2. `quad-deg` — degree of quadrature rule for anti-aliasing in a hexahedral element:

   int

3. `quad-pts` — name of quadrature rule for anti-aliasing in a hexahedral element:

   `gauss-legendre` | `gauss-legendre-lobatto` | `witherden-vincent`

Example:

```
[solver-elements-hex]
soln-pts = gauss-legendre
quad-deg = 10
quad-pts = gauss-legendre
```

2.2.4.7 [solver-elements-tet{-mg-p*order*}]

Parameterises the tetrahedral elements, or if -mg-p*order* is suffixed the tetrahedral elements at multi-p level *order*, with

1. `soln-pts` — location of the solution points in a tetrahedral element:

   `alpha-opt` | `shunn-ham`

2. `quad-deg` — degree of quadrature rule for anti-aliasing in a tetrahedral element:

   int

3. `quad-pts` — name of quadrature rule for anti-aliasing in a tetrahedral element:

   `shunn-ham` | `witherden-vincent`

Example:

```
[solver-elements-tet]
soln-pts = shunn-ham
quad-deg = 10
quad-pts = shunn-ham
```

2.2.4.8 [solver-elements-pri{-mg-p*order*}]

Parameterises the prismatic elements, or if -mg-p*order* is suffixed the prismatic elements at multi-p level *order*, with

1. `soln-pts` — location of the solution points in a prismatic element:

   `alpha-opt~gauss-legendre-lobatto` | `williams-shunn~gauss-legendre` | `williams-shunn~gauss-legendre-lobatto`

2. `quad-deg` — degree of quadrature rule for anti-aliasing in a prismatic element:

   int

3. `quad-pts` — name of quadrature rule for anti-aliasing in a prismatic element:

   `williams-shunn~gauss-legendre` | `williams-shunn~gauss-legendre-lobatto` | `witherden-vincent`

Example:

```
[solver-elements-pri]
soln-pts = williams-shunn~gauss-legendre
quad-deg = 10
quad-pts = williams-shunn~gauss-legendre
```

2.2.4.9 [solver-elements-pyr{-mg-p*order*}]

Parameterises the pyramidal elements, or if -mg-p*order* is suffixed the pyramidal elements at multi-p level *order*, with

1. `soln-pts` — location of the solution points in a pyramidal element:

   `gauss-legendre` | `gauss-legendre-lobatto`

2. `quad-deg` — degree of quadrature rule for anti-aliasing in a pyramidal element:

   int

3. `quad-pts` — name of quadrature rule for anti-aliasing in a pyramidal element:

   `witherden-vincent`

Example:

```
[solver-elements-pyr]
soln-pts = gauss-legendre
quad-deg = 10
quad-pts = witherden-vincent
```

2.2.5 Plugins

Plugins allow for powerful additional functionality to be swapped in and out. There are two classes of plugin available; solution plugins which are prefixed by `soln-` and solver plugins which are prefixed by `solver-`. It is possible to create multiple instances of the same solution plugin by appending a suffix, for example:

```
[soln-plugin-writer]
...

[soln-plugin-writer-2]
...

[soln-plugin-writer-three]
...
```

Certain plugins also expose functionality via a CLI, which can be invoked independently of a PyFR run.

2.2.5.1 Solution Plugins

2.2.5.1.1 [soln-plugin-ascent]

Uses Alpine Ascent to plot on-the-fly . The following parameters can then be set:

1. `nsteps` — produce the plots every `nsteps` time steps:

   int

2. `division` — the level of linear subdivison to use

   int

3. `region` — region to visualise, specified as either the entire domain using `*` or a combination of the geometric shapes specified in *Regions*:

   `*` | `shape(args, ...)`

There are then three components that can be used to build plots. Scenes which define the render, Pipelines that can be used to apply filers and build sequenceces of data manipulations, and Fields which are used to define field expresions.

1. `field-{name}` — this is an extension to the Ascent library where users define expressions for the fields used. This can either be a scalar or a vector, where the latter is defined by a comma separted list of expressions.

   string | *string*, *string* (, *string*)

2. `scene-{name}` — a scene to plot with Ascent options passed in a dictionary. Each scene needs a field, and the expression for that field must have been set either via a field command or a pipeline. Additionally, one or multiple `render-{name}` dictionaries must be defined to configure the rendering of the scene. Multiple render dictionaries give multiple views of the same scene.

   dict

3. `pipeline-{name}` — a pipeline of data manipulations that can be used within a scene. The value is a dictionary containing the valid configuration options. Pipeline objects can be stacked together to form a pipeline of filters by making a list of dictionaries. Finally, the q-criterion and vorticity filters require that a field called velocity is defined.

   dict | [*dict*]

Example:

```
[soln-plugin-ascent]
nsteps = 200
division = 5

field-kenergy = 0.5*rho*(u*u + v*v)
scene-ke = {'render-1': {'image-name': 'ke-{t:.1f}'}, 'field': 'kenergy', 'type':
↪'pseudocolor'}

field-mom = rho*u, rho*v
pipeline-amom = {'type': 'vector_magnitude', 'field': 'mom', 'output-name': 'mag'}
scene-va = {'type': 'pseudocolor', 'pipeline': 'amom', 'field': 'mag', 'render-1': {
↪'image-width': 128, 'image-name': 'm1-{t:4.2f}'}, 'render-2': {'image-width': 256,
↪'image-name': 'm2-{t:4.2f}'}}
```

Note that setting `nsteps` to be too small can have a significant impact on performance as generating each image has overhead and may require some MPI communication to occur.

This plugin also exposes functionality via a CLI. The following functions are available

1. `pyfr ascent render` — render an image from a pre-existing mesh and solution file. It must be run with the same number of ranks as partitions in the mesh. By default it will use settings from the first section of the settings file that it is passed. Alternatively, a specific section name can be provided. In both cases all other sections are ignored.

   Example:

   ```
   pyfr ascent render mesh.pyfrm solution.pyfrs settings.ini
   ```

2.2.5.1.2 [soln-plugin-dtstats]

Write time-step statistics out to a CSV file. Parameterised with

1. `flushsteps` — flush to disk every `flushsteps`:

   int

2. `file` — output file path; should the file already exist it will be appended to:

   string

3. `header` — if to output a header row or not:

   boolean

Example:

```
[soln-plugin-dtstats]
flushsteps = 100
file = dtstats.csv
header = true
```

2.2.5.1.3 [soln-plugin-fluidforce-*name*]

Periodically integrates the pressure and viscous stress on the boundary labelled `name` and writes out the resulting force and moment (if requested) vectors to a CSV file. Parameterised with

1. `nsteps` — integrate every `nsteps`:

   int

2. `file` — output file path; should the file already exist it will be appended to:

   string

3. `header` — if to output a header row or not:

   boolean

4. `morigin` — origin used to compute moments (optional):

   `(x, y, [z])`

5. `quad-deg-{etype}` — degree of quadrature rule for fluid force integration, optionally this can be specified for different element types:

   int

6. `quad-pts-{etype}` — name of quadrature rule (optional):

   string

Example:

```
[soln-plugin-fluidforce-wing]
nsteps = 10
file = wing-forces.csv
header = true
quad-deg = 6
morigin = (0.0, 0.0, 0.5)
```

2.2.5.1.4 [soln-plugin-fwh]

Use Ffowcs Williams–Hawkings equation to approximate far field noise in a uniformly moving medium:

1. `tstart` — time at which to start sampling, default is `0`:

   float

2. `dt` — time step between samples:

   float

3. `file` — output file path; should the file already exist it will be appended to:

   string

4. `header` — if to output a header row or not:

   boolean

5. `surface` — a region the surface of which is saple for the FWH sovler, only use a combination of the geometric shapes specified in *Regions*:

   `shape(args, ...)`

6. `quad-deg` — degree of surface quadrature rule (optional):

   int

7. `quad-pts-{etype}` — name of surface quadrature rule (optional):

   string

8. `observer-pts` — the obversation point in the far field at which noise is approximated:

   `[(x, y), (x, y), ...]|[(x, y, z), (x, y, z), ...]`

9. `rho, u, v, (w), p, (c)` — the constant far field properties of the flow. For incompressible calculations the sound speed `c` and the dnsity `rho` must be given:

   float

Example:

```
[soln-plugin-fwh]
file = fwh.csv
region = box((1, -5), (10, 5))
header = true
tstart = 10
dt = 1e-2
observer-pts = [(1, 10), (1, 30), (1, 100), (1, 300)]

rho = 1
u = 1
v = 0
p = 10
```

2.2.5.1.5 [soln-plugin-integrate]

Integrate quantities over the compuational domain. Parameterised with:

1. `nsteps` — calculate the integral every `nsteps` time steps:

   int

2. `file` — output file path; should the file already exist it will be appended to:

   string

3. `header` — if to output a header row or not:

   boolean

4. `quad-deg` — degree of quadrature rule (optional):

   int

5. `quad-pts-{etype}` — name of quadrature rule (optional):

   string

6. **`norm` — sets the degree and calculates an L_p norm,**
   otherwise standard integration is performed:

   float | `inf` | `none`

7. `region` — region to integrate, specified as either the entire domain using `*` or a combination of the geometric shapes specified in *Regions*:

   `*` | `shape(args, ...)`

8. `int-`*name* — expression to integrate, written as a function of the primitive variables and gradients thereof, the physical coordinates [x, y, [z]] and/or the physical time [t]; multiple expressions, each with their own *name*, may be specified:

   string

Example:

```
[soln-plugin-integrate]
nsteps = 50
file = integral.csv
header = true
quad-deg = 9
vor1 = (grad_w_y - grad_v_z)
vor2 = (grad_u_z - grad_w_x)
vor3 = (grad_v_x - grad_u_y)

int-E = rho*(u*u + v*v + w*w)
int-enst = rho*(%(vor1)s*%(vor1)s + %(vor2)s*%(vor2)s + %(vor3)s*%(vor3)s)
```

2.2.5.1.6 [soln-plugin-nancheck]

Periodically checks the solution for NaN values. Parameterised with

1. `nsteps` — check every `nsteps`:

   int

Example:

```
[soln-plugin-nancheck]
nsteps = 10
```

2.2.5.1.7 [soln-plugin-pseudostats]

Write pseudo-step convergence history out to a CSV file. Parameterised with

1. `flushsteps` — flush to disk every `flushsteps`:

   int

2. `file` — output file path; should the file already exist it will be appended to:

   string

3. `header` — if to output a header row or not:

   boolean

Example:

```
[soln-plugin-pseudostats]
flushsteps = 100
file = pseudostats.csv
header = true
```

2.2.5.1.8 [soln-plugin-residual]

Periodically calculates the residual and writes it out to a CSV file. Parameterised with

1. `nsteps` — calculate every `nsteps`:

   int

2. `file` — output file path; should the file already exist it will be appended to:

   string

3. `header` — if to output a header row or not:

   boolean

4. **`norm` — sets the degree and calculates an L_p norm,**
   default is `2`:

   float | `inf`

Example:

```
[soln-plugin-residual]
nsteps = 10
file = residual.csv
header = true
norm = inf
```

2.2.5.1.9 [soln-plugin-sampler]

Periodically samples specific points in the volume and writes them out to a CSV file. Parameterised with

1. `nsteps` — sample every `nsteps`:

   int

2. `samp-pts` — list of points to sample:

   `[(x, y), (x, y), ...]|[(x, y, z), (x, y, z), ...]`

3. `format` — output variable format:

   `primitive|conservative`

4. `file` — output file path; should the file already exist it will be appended to:

   string

5. `header` — if to output a header row or not:

   boolean

Example:

```
[soln-plugin-sampler]
nsteps = 10
samp-pts = [(1.0, 0.7, 0.0), (1.0, 0.8, 0.0)]
format = primitive
file = point-data.csv
header = true
```

2.2.5.1.10 [soln-plugin-tavg]

Time average quantities. Parameterised with

1. `nsteps` — accumulate the average every `nsteps` time steps:

   int

2. `dt-out` — write to disk every `dt-out` time units:

   float

3. `tstart` — time at which to start accumulating average data:

   float

4. `mode` — output file accumulation mode:

   `continuous|windowed`

   In continuous mode each output file contains average data from `tstart` up until the time at which the file is written. In windowed mode each output file only contains average data for the preceding

`dt-out` time units. The default is `windowed`. Average data files obtained using the windowed mode can be accumulated after-the-fact using the CLI.

5. `std-mode` — standard deviation reporting mode:

   `summary | all`

   If to output full standard deviation fields or just summary statistics. In lieu of a complete field, summary instead reports the maximum and average standard deviation for each field. The default is `summary` with `all` doubling the size of the resulting files.

6. `basedir` — relative path to directory where outputs will be written:

   string

7. `basename` — pattern of output names:

   string

8. `precision` — output file number precision:

   `single | double`

   The default is `single`. Note that this only impacts the output, with statistic accumulation *always* being performed in double precision.

9. `region` — region to be written, specified as either the entire domain using `*`, a combination of the geometric shapes specified in *Regions*, or a sub-region of elements that have faces on a specific domain boundary via the name of the domain boundary:

   `*` | `shape(args, ...)` | *string*

10. `avg-`*name* — expression to time average, written as a function of the primitive variables and gradients thereof; multiple expressions, each with their own *name*, may be specified:

    string

11. `fun-avg-`*name* — expression to compute at file output time, written as a function of any ordinary average terms; multiple expressions, each with their own *name*, may be specified:

    string

Example:

```
[soln-plugin-tavg]
nsteps = 10
dt-out = 2.0
mode = windowed
basedir = .
basename = files-{t:06.2f}

avg-u = u
avg-v = v
avg-uu = u*u
avg-vv = v*v
avg-uv = u*v

fun-avg-upup = uu - u*u
fun-avg-vpvp = vv - v*v
fun-avg-upvp = uv - u*v
```

This plugin also exposes functionality via a CLI. The following functions are available

1. `pyfr tavg merge` — average together multiple time average files into a single time average file. The averaging times are read from the file and do not need to be evenly spaced in time.

   Example:

```
pyfr tavg merge avg-1.00.pyfrs avg-2.00.pyfrs avg-10.00.pyfrs merged_avg.pyfrs
```

2.2.5.1.11 [soln-plugin-writer]

Periodically write the solution to disk in the pyfrs format. Parameterised with

1. `dt-out` — write to disk every `dt-out` time units:

   float

2. `basedir` — relative path to directory where outputs will be written:

   string

3. `basename` — pattern of output names:

   string

4. `post-action` — command to execute after writing the file:

   string

5. `post-action-mode` — how the post-action command should be executed:

   `blocking` | `non-blocking`

4. `region` — region to be written, specified as either the entire domain using `*`, a combination of the geometric shapes specified in *Regions*, or a sub-region of elements that have faces on a specific domain boundary via the name of the domain boundary:

   `*` | `shape(args, ...)` | *string*

Example:

```
[soln-plugin-writer]
dt-out = 0.01
basedir = .
basename = files-{t:.2f}
post-action = echo "Wrote file {soln} at time {t} for mesh {mesh}."
post-action-mode = blocking
region = box((-5, -5, -5), (5, 5, 5))
```

2.2.5.2 Solver Plugins

2.2.5.2.1 [solver-plugin-source]

Injects solution, space (x, y, [z]), and time (t) dependent source terms with

1. `rho` — density source term for `euler` | `navier-stokes`:

   string

2. `rhou` — x-momentum source term for `euler` | `navier-stokes` :

   string

3. `rhov` — y-momentum source term for `euler` | `navier-stokes` :

   string

4. `rhow` — z-momentum source term for `euler` | `navier-stokes` :

   string

5. `E` — energy source term for `euler` | `navier-stokes` :

   string

6. `p` — pressure source term for `ac-euler` | `ac-navier-stokes`:

   string

7. `u` — x-velocity source term for `ac-euler` | `ac-navier-stokes`:

   string

8. `v` — y-velocity source term for `ac-euler` | `ac-navier-stokes`:

   string

9. `w` — w-velocity source term for `ac-euler` | `ac-navier-stokes`:

   string

Example:

```
[solver-plugin-source]
rho = t
rhou = x*y*sin(y)
rhov = z*rho
rhow = 1.0
E = 1.0/(1.0+x)
```

2.2.5.2.2 [solver-plugin-turbulence]

Injects synthetic eddies into a region of the domain. Parameterised with

1. `avg-rho` — average free-stream density:

   float

2. `avg-u` — average free-stream velocity magnitude:

   float

3. `avg-mach` — averge free-stream Mach number:

   float

4. `turbulence-intensity` — percentage turbulence intensity:

   float

5. `turbulence-length-scale` — turbulent length scale:

   float

6. `sigma` — standard deviation of Gaussian sythetic eddy profile:

   float

7. `centre` — centre of plane on which synthetic eddies are injected:

(*float*, *float*, *float*)

8. `y-dim` — y-dimension of plane:

   float

9. `z-dim` — z-dimension of plane:

   float

10. `rot-axis` — axis about which plane is rotated:

    (*float*, *float*, *float*)

11. `rot-angle` — angle in degrees that plane is rotated:

    float

Example:

```
[solver-plugin-turbulence]
avg-rho = 1.0
avg-u = 1.0
avg-mach = 0.2
turbulence-intensity = 1.0
turbulence-length-scale = 0.075
sigma = 0.7
centre = (0.15, 2.0, 2.0)
y-dim = 3.0
z-dim = 3.0
rot-axis = (0, 0, 1)
rot-angle = 0.0
```

2.2.5.3 Regions

Certain plugins are capable of performing operations on a subset of the elements inside the domain. One means of constructing these element subsets is through parameterised regions. Note that an element is considered part of a region if *any* of its nodes are found to be contained within the region. Supported regions:

Rectangular cuboid `box(x0, x1)`
A rectangular cuboid defined by two diametrically opposed vertices. Valid in both 2D and 3D.

Conical frustum `conical_frustum(x0, x1, r0, r1)`
A conical frustum whose end caps are at *x0* and *x1* with radii *r0* and *r1*, respectively. Only valid in 3D.

Cone `cone(x0, x1, r)`
A cone of radius *r* whose centre-line is defined by *x0* and *x1*. Equivalent to `conical_frustum(x0, x1, r, 0)`. Only valid in 3D.

Cylinder `cylinder(x0, x1, r)`
A circular cylinder of radius *r* whose centre-line is defined by *x0* and *x1*. Equivalent to `conical_frustum(x0, x1, r, r)`. Only valid in 3D.

Cartesian ellipsoid `ellipsoid(x0, a, b, c)`
An ellipsoid centred at *x0* with Cartesian coordinate axes whose extents in the *x*, *y*, and *z* directions are given by *a*, *b*, and *c*, respectively. Only valid in 3D.

Sphere `sphere(x0, r)`
A sphere centred at *x0* with a radius of *r*. Equivalent to `ellipsoid(x0, r, r, r)`. Only valid in 3D.

All region shapes also support rotation. In 2D this is accomplished by passing a trailing *rot=angle* argument where *angle* is a rotation angle in degrees; for example `box((-5, 2), (2, 0), rot=30)`. In 3D the syntax is *rot=(phi, theta, psi)* and corresponds to a sequence of Euler angles in the so-called *ZYX convention*. Region expressions can also be added and subtracted together arbitrarily. For example `box((-10, -10, -10), (10, 10, 10)) - sphere((0, 0, 0), 3)` will result in a cube-shaped region with a sphere cut out of the middle.

2.2.6 Additional Information

The *INI* file format is very versatile. A feature that can be useful in defining initial conditions is the substitution feature and this is demonstrated in the *[soln-plugin-integrate]* plugin example.

To prevent situations where you have solutions files for unknown configurations, the contents of the `.ini` file are added as an attribute to `.pyfrs` files. These files use the HDF5 format and can be straightforwardly probed with tools such as h5dump.

In several places within the `.ini` file expressions may be used. As well as the constant `pi`, expressions containing the following functions are supported:

1. `+, -, *, /` — basic arithmetic
2. `sin, cos, tan` — basic trigonometric functions (radians)
3. `asin, acos, atan, atan2` — inverse trigonometric functions
4. `exp, log` — exponential and the natural logarithm
5. `tanh` — hyperbolic tangent
6. `pow` — power, note `**` is not supported
7. `sqrt` — square root
8. `abs` — absolute value
9. `min, max` — two variable minimum and maximum functions, arguments can be arrays

CHAPTER

THREE

DEVELOPER GUIDE

3.1 A Brief Overview of the PyFR Framework

3.1.1 Where to Start

The symbolic link `pyfr.scripts.pyfr` points to the script `pyfr.scripts.main`, which is where it all starts! Specifically, the function `process_run` calls the function `_process_common`, which in turn calls the function `get_solver`, returning an Integrator – a composite of a *Controller* and a *Stepper*. The Integrator has a method named `run`, which is then called to run the simulation.

3.1.2 Controller

A *Controller* acts to advance the simulation in time. Specifically, a *Controller* has a method named `advance_to` which advances a *System* to a specified time. There are three types of physical-time *Controller* available in PyFR 2.0.0:

StdNoneController **Click to show**

StdPIController **Click to show**

DualNoneController **Click to show**

Types of physical-time *Controller* are related via the following inheritance diagram:

There are two types of pseudo-time *Controller* available in PyFR 2.0.0:

DualNonePseudoController **Click to show**

DualPIPseudoController **Click to show**

Types of pseudo-time *Controller* are related via the following inheritance diagram:

3.1.3 Stepper

A *Stepper* acts to advance the simulation by a single time-step. Specifically, a *Stepper* has a method named `step` which advances a *System* by a single time-step. There are eight types of *Stepper* available in PyFR 2.0.0:

StdEulerStepper **Click to show**

StdRK4Stepper **Click to show**

StdRK34Stepper **Click to show**

StdRK45Stepper **Click to show**

StdTVDRK3Stepper **Click to show**

DualBackwardEulerStepper **Click to show**

SDIRK33Stepper **Click to show**

SDIRK43Stepper **Click to show**

Types of *Stepper* are related via the following inheritance diagram:

3.1.4 PseudoStepper

A *PseudoStepper* acts to advance the simulation by a single pseudo-time-step. They are used to converge implicit *Stepper* time-steps via a dual time-stepping formulation. There are five types of *PseudoStepper* available in PyFR 2.0.0:

DualRK4PseudoStepper **Click to show**

DualTVDRK3PseudoStepper **Click to show**

DualEulerPseudoStepper **Click to show**

DualRK34PseudoStepper **Click to show**

DualRK45PseudoStepper **Click to show**

Types of *PseudoStepper* are related via the following inheritance diagram:

3.1.5 System

A *System* holds information/data for the system, including *Elements*, *Interfaces*, and the *Backend* with which the simulation is to run. A *System* has a method named `rhs`, which obtains the divergence of the flux (the 'right-hand-side') at each solution point. The method `rhs` invokes various kernels which have been pre-generated and loaded into queues. A *System* also has a method named `_gen_kernels` which acts to generate all the kernels required by a particular *System*. A kernel is an instance of a 'one-off' class with a method named `run` that implements the required kernel functionality. Individual kernels are produced by a kernel provider. PyFR 2.0.0 has various types of kernel provider. A *Pointwise Kernel Provider* produces point-wise kernels such as Riemann solvers and flux functions etc. These point-wise kernels are specified using an in-built platform-independent templating language derived from Mako, henceforth referred to as *PyFR-Mako*. There are four types of *System* available in PyFR 2.0.0:

ACEulerSystem **Click to show**

ACNavierStokesSystem **Click to show**

EulerSystem **Click to show**

NavierStokesSystem **Click to show**

Types of *System* are related via the following inheritance diagram:

3.1.6 Elements

An *Elements* holds information/data for a group of elements. There are four types of *Elements* available in PyFR 2.0.0:

ACEulerElements **Click to show**

ACNavierStokesElements **Click to show**

EulerElements **Click to show**

NavierStokesElements **Click to show**

Types of *Elements* are related via the following inheritance diagram:

3.1.7 Interfaces

An *Interfaces* holds information/data for a group of interfaces. There are eight types of (non-boundary) *Interfaces* available in PyFR 2.0.0:

ACEulerIntInters **Click to show**

ACEulerMPIInters **Click to show**

ACNavierStokesIntInters **Click to show**

ACNavierStokesMPIInters **Click to show**

EulerIntInters **Click to show**

EulerMPIInters **Click to show**

NavierStokesIntInters **Click to show**

NavierStokesMPIInters **Click to show**

Types of (non-boundary) *Interfaces* are related via the following inheritance diagram:

3.1.8 Backend

A *Backend* holds information/data for a backend. There are five types of *Backend* available in PyFR 2.0.0:

CUDABackend **Click to show**

class pyfr.backends.cuda.base.**CUDABackend**(*cfg*)

_malloc_checked(*nbytes*)

_malloc_impl(*nbytes*)

alias(*obj*, *aobj*)

blocks = False

commit()

const_matrix(*initval*, *dtype=None*, *tags={}*)

graph()

kernel(*name*, **args*, ***kwargs*)

property lookup

malloc(*obj*, *extent*)

matrix(*ioshape*, *initval=None*, *extent=None*, *aliases=None*, *tags={}*, *dtype=None*)

matrix_slice(*mat*, *ra*, *rb*, *ca*, *cb*)

name = 'cuda'

ordered_meta_kernel(*kerns*)

run_graph(*graph*, *wait=False*)

run_kernels(*kernels*, *wait=False*)

unordered_meta_kernel(*kerns*, *splits=None*)

view(*matmap*, *rmap*, *cmap*, *rstridemap=None*, *vshape=()*, *tags={}*)

wait()

xchg_matrix(*ioshape*, *initval=None*, *extent=None*, *aliases=None*, *tags={}*)

xchg_matrix_for_view(*view*, *tags={}*)

xchg_view(*matmap*, *rmap*, *cmap*, *rstridemap=None*, *vshape=()*, *tags={}*)

HIPBackend **Click to show**

class pyfr.backends.hip.base.**HIPBackend**(*cfg*)

_malloc_checked(*nbytes*)

_malloc_impl(*nbytes*)

alias(*obj*, *aobj*)

blocks = False

commit()

const_matrix(*initval*, *dtype=None*, *tags={}*)

graph()

kernel(*name*, **args*, ***kwargs*)

property lookup

malloc(*obj*, *extent*)

matrix(*ioshape*, *initval=None*, *extent=None*, *aliases=None*, *tags={}*, *dtype=None*)

matrix_slice(*mat*, *ra*, *rb*, *ca*, *cb*)

name = 'hip'

ordered_meta_kernel(*kerns*)

run_graph(*graph*, *wait=False*)

run_kernels(*kernels*, *wait=False*)

unordered_meta_kernel(*kerns*, *splits=None*)

view(*matmap*, *rmap*, *cmap*, *rstridemap=None*, *vshape=()*, *tags={}*)

wait()

xchg_matrix(*ioshape*, *initval=None*, *extent=None*, *aliases=None*, *tags={}*)

xchg_matrix_for_view(*view*, *tags={}*)

xchg_view(*matmap*, *rmap*, *cmap*, *rstridemap=None*, *vshape=()*, *tags={}*)

OpenCLBackend **Click to show**

class pyfr.backends.opencl.base.**OpenCLBackend**(*cfg*)

_malloc_checked(*nbytes*)

_malloc_impl(*nbytes*)

alias(*obj*, *aobj*)

blocks = False

commit()

const_matrix(*initval*, *dtype=None*, *tags={}*)

graph()

kernel(*name*, **args*, ***kwargs*)

property lookup

malloc(*obj*, *extent*)

matrix(*ioshape*, *initval=None*, *extent=None*, *aliases=None*, *tags={}*, *dtype=None*)

matrix_slice(*mat*, *ra*, *rb*, *ca*, *cb*)

name = 'opencl'

ordered_meta_kernel(*kerns*)

run_graph(*graph*, *wait=False*)

run_kernels(*kernels*, *wait=False*)

unordered_meta_kernel(*kerns*, *splits=None*)

view(*matmap*, *rmap*, *cmap*, *rstridemap=None*, *vshape=()*, *tags={}*)

wait()

xchg_matrix(*ioshape*, *initval=None*, *extent=None*, *aliases=None*, *tags={}*)

xchg_matrix_for_view(*view*, *tags={}*)

xchg_view(*matmap*, *rmap*, *cmap*, *rstridemap=None*, *vshape=()*, *tags={}*)

OpenMPBackend **Click to show**

class pyfr.backends.openmp.base.**OpenMPBackend**(*cfg*)

_malloc_checked(*nbytes*)

_malloc_impl(*nbytes*)

alias(*obj*, *aobj*)

blocks = True

commit()

const_matrix(*initval*, *dtype=None*, *tags={}*)

graph()

kernel(*name*, **args*, ***kwargs*)

property krunner

property lookup

malloc(*obj*, *extent*)

matrix(*ioshape*, *initval=None*, *extent=None*, *aliases=None*, *tags={}*, *dtype=None*)

matrix_slice(*mat*, *ra*, *rb*, *ca*, *cb*)

name = 'openmp'

ordered_meta_kernel(*kerns*)

run_graph(*graph*, *wait=False*)

run_kernels(*kernels*, *wait=False*)

unordered_meta_kernel(*kerns*, *splits=None*)

view(*matmap*, *rmap*, *cmap*, *rstridemap=None*, *vshape=()*, *tags={}*)

wait()

xchg_matrix(*ioshape*, *initval=None*, *extent=None*, *aliases=None*, *tags={}*)

xchg_matrix_for_view(*view*, *tags={}*)

xchg_view(*matmap*, *rmap*, *cmap*, *rstridemap=None*, *vshape=()*, *tags={}*)

MetalBackend **Click to show**

class pyfr.backends.metal.base.**MetalBackend**(*cfg*)

_malloc_checked(*nbytes*)

_malloc_impl(*nbytes*)

alias(*obj*, *aobj*)

blocks = False

commit()

const_matrix(*initval*, *dtype=None*, *tags={}*)

graph()

kernel(*name*, **args*, ***kwargs*)

property lookup

malloc(*obj*, *extent*)

matrix(*ioshape*, *initval=None*, *extent=None*, *aliases=None*, *tags={}*, *dtype=None*)

matrix_slice(*mat*, *ra*, *rb*, *ca*, *cb*)

name = 'metal'

ordered_meta_kernel(*kerns*)

run_graph(*graph*, *wait=False*)

run_kernels(*kernels*, *wait=False*)

unordered_meta_kernel(*kerns*, *splits=None*)

view(*matmap*, *rmap*, *cmap*, *rstridemap=None*, *vshape=()*, *tags={}*)

wait()

xchg_matrix(*ioshape*, *initval=None*, *extent=None*, *aliases=None*, *tags={}*)

xchg_matrix_for_view(*view*, *tags={}*)

xchg_view(*matmap*, *rmap*, *cmap*, *rstridemap=None*, *vshape=()*, *tags={}*)

Types of *Backend* are related via the following inheritance diagram:

3.1.9 Pointwise Kernel Provider

A *Pointwise Kernel Provider* produces point-wise kernels. Specifically, a *Pointwise Kernel Provider* has a method named `register`, which adds a new method to an instance of a *Pointwise Kernel Provider*. This new method, when called, returns a kernel. A kernel is an instance of a 'one-off' class with a method named `run` that implements the required kernel functionality. The kernel functionality itself is specified using *PyFR-Mako*. Hence, a *Pointwise Kernel Provider* also has a method named `_render_kernel`, which renders *PyFR-Mako* into low-level platform-specific code. The `_render_kernel` method first sets the context for Mako (i.e. details about the *Backend* etc.) and then uses Mako to begin rendering the *PyFR-Mako* specification. When Mako encounters a `pyfr:kernel` an instance of a *Kernel Generator* is created, which is used to render the body of the `pyfr:kernel`. There are four types of *Pointwise Kernel Provider* available in PyFR 2.0.0:

CUDAPointwiseKernelProvider **Click to show**

class pyfr.backends.cuda.provider.**CUDAPointwiseKernelProvider**(**args*, ***kwargs*)

_benchmark(*kfunc*, *nbench=4*, *nwarmup=1*)

_build_arglst(*dims*, *argn*, *argt*, *argdict*)

_build_kernel(*name*, *src*, *argtypes*, *argn=[]*)

_instantiate_kernel(*dims*, *fun*, *arglst*, *argm*, *argv*)

_render_kernel(*name*, *mod*, *extrns*, *tplargs*)

kernel_generator_cls = None

register(*mod*)

HIPPointwiseKernelProvider **Click to show**

class pyfr.backends.hip.provider.**HIPPointwiseKernelProvider**(**args*, ***kwargs*)

_benchmark(*kfunc*, *nbench=4*, *nwarmup=1*)

_build_arglst(*dims*, *argn*, *argt*, *argdict*)

_build_kernel(*name*, *src*, *argtypes*, *argn=[]*)

_instantiate_kernel(*dims*, *fun*, *arglst*, *argm*, *argv*)

_render_kernel(*name*, *mod*, *extrns*, *tplargs*)

kernel_generator_cls = None

register(*mod*)

OpenCLPointwiseKernelProvider **Click to show**

class pyfr.backends.opencl.provider.**OpenCLPointwiseKernelProvider**(**args*, ***kwargs*)

_benchmark(*kfunc*, *nbench=4*, *nwarmup=1*)

_build_arglst(*dims*, *argn*, *argt*, *argdict*)

_build_kernel(*name*, *src*, *argtypes*, *argn=[]*)

_build_program(*src*)

_instantiate_kernel(*dims*, *fun*, *arglst*, *argm*, *argv*)

_render_kernel(*name*, *mod*, *extrns*, *tplargs*)

kernel_generator_cls = None

register(*mod*)

OpenMPPointwiseKernelProvider **Click to show**

class pyfr.backends.openmp.provider.**OpenMPPointwiseKernelProvider**(*backend*)

_build_arglst(*dims*, *argn*, *argt*, *argdict*)

_build_function(*name*, *src*, *argtypes*, *restype=None*)

_build_kernel(*name*, *src*, *argtypes*, *argnames=[]*)

_build_library(*src*)

_get_arg_cls(*argtypes*)

_instantiate_kernel(*dims*, *fun*, *arglst*, *argm*, *argv*)

_render_kernel(*name*, *mod*, *extrns*, *tplargs*)

kernel_generator_cls

alias of *OpenMPKernelGenerator*

register(*mod*)

MetalPointwiseKernelProvider **Click to show**

class pyfr.backends.metal.provider.**MetalPointwiseKernelProvider**(**args*, ***kwargs*)

_benchmark(*kfunc*, *nbench=40*, *nwarmup=25*)

_build_arglst(*dims*, *argn*, *argt*, *argdict*)

_build_kernel(*name*, *src*, *argtypes*, *argn=[]*)

_build_program(*src*)

_instantiate_kernel(*dims*, *fun*, *arglst*, *argm*, *argv*)

_render_kernel(*name*, *mod*, *extrns*, *tplargs*)

kernel_generator_cls = None

register(*mod*)

typemap = {<class 'ctypes.c_float'>: (c_float(0.0), 4), <class 'ctypes.c_int'>: (c_int(0), 4), <class 'ctypes.c_long'>: (c_long(0), 8), <class 'ctypes.c_ulong'>: (c_ulong(0), 8)}

Types of *Pointwise Kernel Provider* are related via the following inheritance diagram:

3.1.10 Kernel Generator

A *Kernel Generator* renders the *PyFR-Mako* in a `pyfr:kernel` into low-level platform-specific code. Specifically, a *Kernel Generator* has a method named `render`, which applies *Backend* specific regex and adds *Backend* specific 'boiler plate' code to produce the low-level platform-specific source – which is compiled, linked, and loaded. There are four types of *Kernel Generator* available in PyFR 2.0.0:

CUDAKernelGenerator **Click to show**

class pyfr.backends.cuda.generator.**CUDAKernelGenerator**(**args*, ***kwargs*)

_deref_arg(*arg*)

_deref_arg_array_1d(*arg*)

_deref_arg_array_2d(*arg*)

_deref_arg_view(*arg*)

_gid = 'ixdtype_t(blockIdx.x)*blockDim.x + threadIdx.x'

_lid = ('threadIdx.x', 'threadIdx.y')

_match_arg(*arg*)

_preload_arg(*arg*)

_render_body(*body*)

_render_body_preamble(*body*)

_render_spec()

_shared_prfx = '__shared__'

_shared_sync = '__syncthreads()'

argspec()

block1d = None

block2d = None

ldim_size(*name*, *factor=1*)

needs_ldim(*arg*)

render()

HIPKernelGenerator **Click to show**

class pyfr.backends.hip.generator.**HIPKernelGenerator**(**args*, ***kwargs*)

_deref_arg(*arg*)

_deref_arg_array_1d(*arg*)

_deref_arg_array_2d(*arg*)

_deref_arg_view(*arg*)

_gid = 'ixdtype_t(blockIdx.x)*blockDim.x + threadIdx.x'

_lid = ('threadIdx.x', 'threadIdx.y')

_match_arg(*arg*)

_preload_arg(*arg*)

_render_body(*body*)

_render_body_preamble(*body*)

_render_spec()

_shared_prfx = '__shared__'

_shared_sync = '__syncthreads()'

argspec()

block1d = None

block2d = None

ldim_size(*name*, *factor=1*)

needs_ldim(*arg*)

render()

OpenCLKernelGenerator **Click to show**

class pyfr.backends.opencl.generator.**OpenCLKernelGenerator**(**args*, ***kwargs*)

_deref_arg(*arg*)

_deref_arg_array_1d(*arg*)

_deref_arg_array_2d(*arg*)

_deref_arg_view(*arg*)

_gid = 'get_global_id(0)'

_lid = ('get_local_id(0)', 'get_local_id(1)')

_match_arg(*arg*)

_preload_arg(*arg*)

_render_body(*body*)

_render_body_preamble(*body*)

_render_spec()

_shared_prfx = '__local'

_shared_sync = 'work_group_barrier(CLK_GLOBAL_MEM_FENCE)'

argspec()

block1d = None

block2d = None

ldim_size(*name*, *factor=1*)

needs_ldim(*arg*)

render()

OpenMPKernelGenerator **Click to show**

class pyfr.backends.openmp.generator.**OpenMPKernelGenerator**(*name*, *ndim*, *args*, *body*, *fpdtype*, *ixdtype*)

_deref_arg(*arg*)

_deref_arg_array_1d(*arg*)

_deref_arg_array_2d(*arg*)

_deref_arg_view(*arg*)

_displace_arg(*arg*)

_match_arg(*arg*)

_render_args(*argn*)

_render_body(*body*)

_render_body_preamble(*body*)

argspec()

ldim_size(*name*, *factor=1*)

needs_ldim(*arg*)

render()

MetalKernelGenerator **Click to show**

class pyfr.backends.metal.generator.**MetalKernelGenerator**(**args*, ***kwargs*)

_deref_arg(*arg*)

_deref_arg_array_1d(*arg*)

_deref_arg_array_2d(*arg*)

_deref_arg_view(*arg*)

_gid = '_tpig.x'

_lid = ('_tpitg.x', '_tpitg.y')

_match_arg(*arg*)

_preload_arg(*arg*)

_render_body(*body*)

_render_body_preamble(*body*)

_render_spec()

_shared_prfx = 'threadgroup'

_shared_sync = 'threadgroup_barrier(mem_flags::mem_threadgroup)'

argspec()

block1d = None

block2d = None

ldim_size(*name*, *factor=1*)

needs_ldim(*arg*)

render()

Types of *Kernel Generator* are related via the following inheritance diagram:

3.2 PyFR-Mako

3.2.1 PyFR-Mako Kernels

PyFR-Mako kernels are specifications of point-wise functionality that can be invoked directly from within PyFR. They are opened with a header of the form:

```
<%pyfr:kernel name='kernel-name' ndim='data-dimensionality' [argument-name='argument-
↪intent argument-attribute argument-data-type' ...]>
```

where

1. `kernel-name` — name of kernel

   string

2. `data-dimensionality` — dimensionality of data

   int

3. `argument-name` — name of argument

   string

4. `argument-intent` — intent of argument

   `in` | `out` | `inout`

5. `argument-attribute` — attribute of argument

   `mpi` | `scalar` | `view`

6. `argument-data-type` — data type of argument

   string

and are closed with a footer of the form:

```
</%pyfr:kernel>
```

3.2.2 PyFR-Mako Macros

PyFR-Mako macros are specifications of point-wise functionality that cannot be invoked directly from within PyFR, but can be embedded into PyFR-Mako kernels. PyFR-Mako macros can be viewed as building blocks for PyFR-mako kernels. They are opened with a header of the form:

```
<%pyfr:macro name='macro-name' params='[parameter-name, ...]'>
```

where

1. `macro-name` — name of macro

   string

2. `parameter-name` — name of parameter

   string

and are closed with a footer of the form:

```
</%pyfr:macro>
```

PyFR-Mako macros are embedded within a kernel using an expression of the following form:

```
${pyfr.expand('macro-name', ['parameter-name', ...])};
```

where

1. `macro-name` — name of the macro

   string

2. `parameter-name` — name of parameter

   string

3.2.3 Syntax

3.2.3.1 Basic Functionality

Basic functionality can be expressed using a restricted subset of the C programming language. Specifically, use of the following is allowed:

1. `+,-,*,/` — basic arithmetic
2. `sin, cos, tan` — basic trigonometric functions
3. `exp` — exponential
4. `pow` — power
5. `fabs` — absolute value
6. `output = ( condition ? satisfied :  unsatisfied )` — ternary if
7. `min` — minimum

8. `max` — maximum

However, conditional if statements, as well as for/while loops, are not allowed.

3.2.3.2 Expression Substitution

Mako expression substitution can be used to facilitate PyFR-Mako kernel specification. A Python expression `expression` prescribed thus `${expression}` is substituted for the result when the PyFR-Mako kernel specification is interpreted at runtime.

Example:

```
E = s[${ndims - 1}]
```

3.2.3.3 Conditionals

Mako conditionals can be used to facilitate PyFR-Mako kernel specification. Conditionals are opened with `% if condition:` and closed with `% endif`. Note that such conditionals are evaluated when the PyFR-Mako kernel specification is interpreted at runtime, they are not embedded into the low-level kernel.

Example:

```
% if ndims == 2:
    fout[0][1] += t_xx;     fout[1][1] += t_xy;
    fout[0][2] += t_xy;     fout[1][2] += t_yy;
    fout[0][3] += u*t_xx + v*t_xy + ${-c['mu']*c['gamma']/c['Pr']}*T_x;
    fout[1][3] += u*t_xy + v*t_yy + ${-c['mu']*c['gamma']/c['Pr']}*T_y;
% endif
```

3.2.3.4 Loops

Mako loops can be used to facilitate PyFR-Mako kernel specification. Loops are opened with `% for condition:` and closed with `% endfor`. Note that such loops are unrolled when the PyFR-Mako kernel specification is interpreted at runtime, they are not embedded into the low-level kernel.

Example:

```
% for i in range(ndims):
    rhov[${i}] = s[${i + 1}];
    v[${i}] = invrho*rhov[${i}];
% endfor
```

CHAPTER

FOUR

PERFORMANCE TUNING

The following sections contain best practices for *tuning* the performance of PyFR. Note, however, that it is typically not worth pursuing the advice in this section until a simulation is working acceptably and generating the desired results.

4.1 OpenMP Backend

4.1.1 AVX-512

When running on an AVX-512 capable CPU Clang and GCC will, by default, only make use of 256-bit vectors. Given that the kernels in PyFR benefit meaningfully from longer vectors it is desirable to override this behaviour. This can be accomplished through the `cflags` key as:

```
[backend-openmp]
cflags = -mprefer-vector-width=512
```

4.1.2 Cores vs. threads

PyFR does not typically derive any benefit from SMT. As such the number of OpenMP threads should be chosen to be equal to the number of physical cores.

4.1.3 Loop Scheduling

By default PyFR employs static scheduling for loops, with work being split evenly across cores. For systems with a single type of core this is usually the right choice. However, on heterogeneous systems it typically results in load imbalance. Here, it can be beneficial to experiment with the *guided* and *dynamic* loop schedules as:

```
[backend-openmp]
schedule = dynamic, 5
```

4.1.4 MPI processes vs. OpenMP threads

When using the OpenMP backend it is recommended to employ *one MPI rank per NUMA zone*. For most systems each socket represents its own NUMA zone. Thus, on a two socket system it is suggested to run PyFR with two MPI ranks, with each process being bound to a single socket. The specifics of how to accomplish this depend on both the job scheduler and MPI distribution.

4.1.5 Asynchronous MPI progression

The parallel scalability of the OpenMP backend depends *heavily* on MPI having support for asynchronous progression; that is to say the ability for non-blocking send and receive requests to complete *without* the need for the host application to make explicit calls into MPI routines. A lack of support for asynchronous progression prevents PyFR from being able to overlap computation with communication.

4.2 CUDA Backend

4.2.1 CUDA-aware MPI

PyFR is capable of taking advantage of CUDA-aware MPI. This enables CUDA device pointers to be directly to passed MPI routines. Under the right circumstances this can result in improved performance for simulations which are near the strong scaling limit. Assuming mpi4py has been built against an MPI distribution which is CUDA-aware this functionality can be enabled through the `mpi-type` key as:

```
[backend-cuda]
mpi-type = cuda-aware
```

4.3 HIP Backend

4.3.1 HIP-aware MPI

PyFR is capable of taking advantage of HIP-aware MPI. This enables HIP device pointers to be directly to passed MPI routines. Under the right circumstances this can result in improved performance for simulations which are near the strong scaling limit. Assuming mpi4py has been built against an MPI distribution which is HIP-aware this functionality can be enabled through the `mpi-type` key as:

```
[backend-hip]
mpi-type = hip-aware
```

4.4 Partitioning

4.4.1 METIS vs SCOTCH

The partitioning module in PyFR includes support for both METIS and SCOTCH. Both usually result in high-quality decompositions. However, for long running simulations on complex geometries it may be worth partitioning a grid with both and observing which decomposition performs best.

4.4.2 Mixed grids

When running PyFR in parallel on mixed element grids it is necessary to take some additional care when partitioning the grid. A good domain decomposition is one where each partition contains the same amount of computational work. For grids with a single element type the amount of computational work is very well approximated by the number of elements assigned to a partition. Thus the goal is simply to ensure that all of the partitions have roughly the same number of elements. However, when considering mixed grids this relationship begins to break down since the computational cost of one element type can be appreciably more than that of another.

There are two main solutions to this problem. The first is to require that each partition contain the same number of elements of each type. For example, if partitioning a mesh with 500 quadrilaterals and 1,500 triangles into two parts, then a sensible goal is to aim for each domain to have 250 quadrilaterals and 750 triangles. Irrespective of what the relative performance differential between the element types is, both partitions will have near identical amounts of work. In PyFR this is known as the *balanced* approach and can be requested via:

```
pyfr partition -e balanced ...
```

This approach typically works well when the number of partitions is small. However, for larger partition counts it can become difficult to achieve such a balance whilst simultaneously minimising communication volume. Thus, in order to obtain a good decomposition a secondary approach is required in which each type of element in the domain is assigned a *weight*. Element types which are more computationally intensive are assigned a larger weight than those that are less intensive. Through this mechanism the total cost of each partition can remain balanced. Unfortunately, the relative cost of different element types depends on a variety of factors, including:

- The polynomial order.
- If anti-aliasing is enabled in the simulation, and if so, to what extent.
- The hardware which the simulation will be run on.

Weights can be specified when partitioning the mesh as `-e shape:weight`. For example, if on a particular system a quadrilateral is found to be 50% more expensive than a triangle this can be specified as:

```
pyfr partition -e quad:3 -e tri:2 ...
```

If precise profiling data is not available regarding the performance of each element type in a given configuration a helpful rule of thumb is to under-weight the dominant element type in the domain. For example, if a domain is 90% tetrahedra and 10% prisms then, absent any additional information about the relative performance of tetrahedra and prisms, a safe choice is to assume the prisms are appreciably *more* expensive than the tetrahedra.

4.4.3 Detecting load imbalances

PyFR includes code for monitoring the amount of time each rank spends waiting for MPI transfers to complete. This can be used, among other things, to detect load imbalances. Such imbalances are typically observed on mixed-element grids with an incorrect weighting factor. Wait time tracking can be enabled as:

```
[backend]
collect-wait-times = true
```

with the resulting statistics being recorded in the `[backend-wait-times]` section of the `/stats` object which is included in all PyFR solution files. This can be extracted as:

```
h5dump -d /stats -b --output=stats.ini soln.pyfrs
```

Note that the number of graphs depends on the system, and not all graphs initiate MPI requests. The average amount of time each rank spends waiting for MPI requests per right hand side evaluation can be obtained by vertically summing all of the `-median` fields together.

There exists an inverse relationship between the amount of computational work a rank has to perform and the amount of time it spends waiting for MPI requests to complete. Hence, ranks which spend comparatively less time waiting than their peers are likely to be overloaded, whereas those which spend comparatively more time waiting are likely to be underloaded. This information can then be used to explicitly re-weight the partitions and/or the per-element weights.

4.5 Scaling

The general recommendation when running PyFR in parallel is to aim for a parallel efficiency of $\epsilon \simeq 0.8$ with the parallel efficiency being defined as:

$$\epsilon = \frac{1}{N}\frac{T_1}{T_N},$$

where N is the number of ranks, T_1 is the simulation time with one rank, and T_N is the simulation time with N ranks. This represents a reasonable trade-off between the overall time-to-solution and efficient resource utilisation.

4.6 Parallel I/O

PyFR incorporates support for parallel file I/O via HDF5 and will use it automatically where available. However, for this work several prerequisites must be satisfied:

- HDF5 must be explicitly compiled with support for parallel I/O.
- The mpi4py Python module *must* be compiled against the same MPI distribution as HDF5. A version mismatch here can result in subtle and difficult to diagnose errors.
- The h5py Python module *must* be built with support for parallel I/O.

After completing this process it is highly recommended to verify everything is working by trying the h5py parallel HDF5 example.

4.7 Plugins

A common source of performance issues is running plugins too frequently. PyFR records the amount of time spent in plugins in the `[solver-time-integrator]` section of the `/stats` object which is included in all PyFR solution files. This can be extracted as:

```
h5dump -d /stats -b --output=stats.ini soln.pyfrs
```

Given the time steps taken by PyFR are typically much smaller than those associated with the underlying physics there is seldom any benefit to running integration and/or time average accumulation plugins more frequently than once every 50 steps. Further, when running with adaptive time stepping there is no need to run the NaN check plugin. For simulations with fixed time steps, it is not recommended to run the NaN check plugin more frequently than once every 10 steps.

4.8 Start-up Time

The start-up time required by PyFR can be reduced by ensuring that Python is compiled from source with profile guided optimisations (PGO) which can be enabled by passing `--enable-optimizations --with-lto` to the `configure` script.

It is also important that NumPy be configured to use an optimised BLAS/LAPACK distribution. Further details can be found in the NumPy building from source guide.

CHAPTER

FIVE

EXAMPLES

Test cases are available in the PyFR-Test-Cases repository. It is important to note, however, that these examples are all relatively small 2D/3D simulations and, as such, are *not* suitable for scalability or performance studies.

5.1 Euler Equations

5.1.1 2D Euler Vortex

Proceed with the following steps to run a parallel 2D Euler vortex simulation on a structured mesh:

1. Navigate to the `PyFR-Test-Cases/2d-euler-vortex` directory:

```
cd PyFR-Test-Cases/2d-euler-vortex
```

2. Run pyfr to convert the Gmsh mesh file into a PyFR mesh file called `euler-vortex.pyfrm`:

```
pyfr import euler-vortex.msh euler-vortex.pyfrm
```

3. Run pyfr to partition the PyFR mesh file into two pieces:

```
pyfr partition 2 euler-vortex.pyfrm .
```

4. Run pyfr to solve the Euler equations on the mesh, generating a series of PyFR solution files called `euler-vortex*.pyfrs`:

```
mpiexec -n 2 pyfr -p run -b cuda euler-vortex.pyfrm euler-vortex.ini
```

5. Run pyfr on the solution file `euler-vortex-100.0.pyfrs` converting it into an unstructured VTK file called `euler-vortex-100.0.vtu`:

```
pyfr export euler-vortex.pyfrm euler-vortex-100.0.pyfrs euler-vortex-100.0.vtu
```

6. Visualise the unstructured VTK file in Paraview

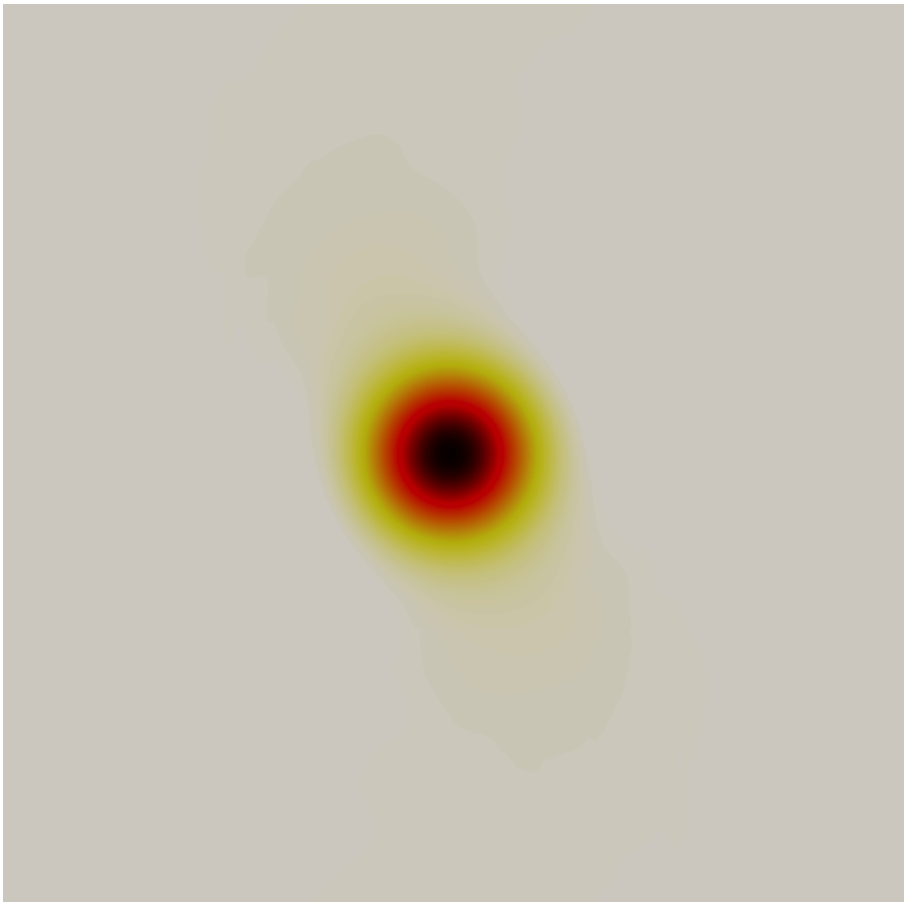

Fig. 1: Colour map of density distribution at 100 time units.

5.1.2 2D Double Mach Reflection

Proceed with the following steps to run a serial 2D double Mach reflection simulation on a structured mesh:

1. Navigate to the `PyFR-Test-Cases/2d-double-mach-reflection` directory:

```
cd PyFR-Test-Cases/2d-double-mach-reflection
```

2. Unzip the Gmsh mesh file file and run pyfr to covert it into a PyFR mesh file called `double-mach-reflection.pyfrm`:

```
unxz double-mach-reflection.msh.xz
pyfr import double-mach-reflection.msh double-mach-reflection.pyfrm
```

3. Run pyfr to solve the compressible Euler equations on the mesh, generating a series of PyFR solution files called `double-mach-reflection-*.pyfrs`:

```
pyfr -p run -b cuda double-mach-reflection.pyfrm double-mach-reflection.ini
```

4. Run pyfr on the solution file `double-mach-reflection-0.20.pyfrs` converting it into an unstructured VTK file called `double-mach-reflection-0.20.vtu`:

```
pyfr export double-mach-reflection.pyfrm double-mach-reflection-0.20.pyfrs double-
↪mach-reflection-0.20.vtu
```

5. Visualise the unstructured VTK file in Paraview

Fig. 2: Colour map of density distribution at 0.2 time units.

5.2 Navier–Stokes Equations

5.2.1 2D Couette Flow

Proceed with the following steps to run a serial 2D Couette flow simulation on a mixed unstructured mesh:

1. Navigate to the `PyFR-Test-Cases/2d-couette-flow` directory:

```
cd PyFR-Test-Cases/2d-couette-flow
```

2. Run pyfr to covert the Gmsh mesh file into a PyFR mesh file called `couette-flow.pyfrm`:

```
pyfr import couette-flow.msh couette-flow.pyfrm
```

3. Run pyfr to solve the Navier-Stokes equations on the mesh, generating a series of PyFR solution files called `couette-flow-*.pyfrs`:

```
pyfr -p run -b cuda couette-flow.pyfrm couette-flow.ini
```

4. Run pyfr on the solution file `couette-flow-040.pyfrs` converting it into an unstructured VTK file called `couette-flow-040.vtu`:

```
pyfr export couette-flow.pyfrm couette-flow-040.pyfrs couette-flow-040.vtu
```

5. Visualise the unstructured VTK file in Paraview

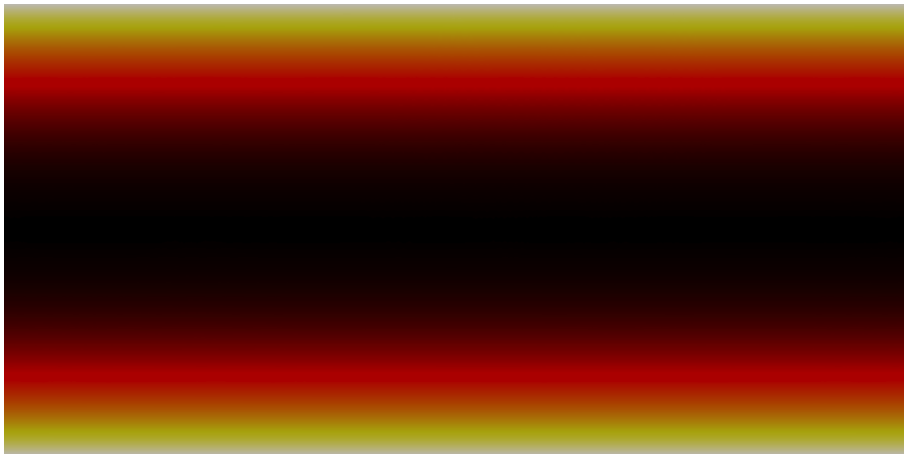

Fig. 3: Colour map of steady-state density distribution.

5.2.2 2D Incompressible Cylinder Flow

Proceed with the following steps to run a serial 2D incompressible cylinder flow simulation on a mixed unstructured mesh:

1. Navigate to the `PyFR-Test-Cases/2d-inc-cylinder` directory:

```
cd PyFR-Test-Cases/2d-inc-cylinder
```

2. Run pyfr to covert the Gmsh mesh file into a PyFR mesh file called `inc-cylinder.pyfrm`:

```
pyfr import inc-cylinder.msh inc-cylinder.pyfrm
```

3. Run pyfr to solve the incompressible Navier-Stokes equations on the mesh, generating a series of PyFR solution files called `inc-cylinder-*.pyfrs`:

```
pyfr -p run -b cuda inc-cylinder.pyfrm inc-cylinder.ini
```

4. Run pyfr on the solution file `inc-cylinder-75.00.pyfrs` converting it into an unstructured VTK file called `inc-cylinder-75.00.vtu`:

```
pyfr export inc-cylinder.pyfrm inc-cylinder-75.00.pyfrs inc-cylinder-75.00.vtu
```

5. Visualise the unstructured VTK file in Paraview

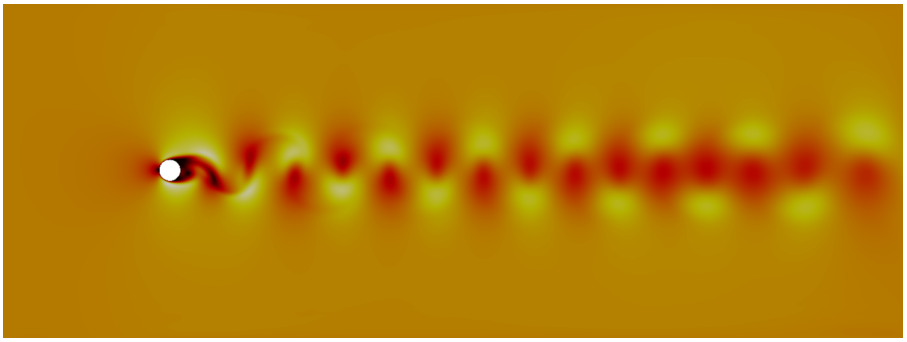

Fig. 4: Colour map of velocity magnitude distribution at 75 time units.

5.2.3 2D Viscous Shock Tube

Proceed with the following steps to run a serial 2D viscous shock tube simulation on a structured mesh:

1. Navigate to the `PyFR-Test-Cases/2d-viscous-shock-tube` directory:

```
cd PyFR-Test-Cases/2d-viscous-shock-tube
```

2. Unzip the the Gmsh mesh file and run pyfr to covert it into a PyFR mesh file called `viscous-shock-tube.pyfrm`:

```
unxz viscous-shock-tube.msh.xz
pyfr import viscous-shock-tube.msh viscous-shock-tube.pyfrm
```

3. Run pyfr to solve the compressible Navier-Stokes equations on the mesh, generating a series of PyFR solution files called `viscous-shock-tube-*.pyfrs`:

```
pyfr -p run -b cuda viscous-shock-tube.pyfrm viscous-shock-tube.ini
```

4. Run pyfr on the solution file `viscous-shock-tube-1.00.pyfrs` converting it into an unstructured VTK file called `viscous-shock-tube-1.00.vtu`:

```
pyfr export viscous-shock-tube.pyfrm viscous-shock-tube-1.00.pyfrs viscous-shock-
↪tube-1.00.vtu
```

5. Visualise the unstructured VTK file in Paraview

5.2.4 3D Triangular Aerofoil

Proceed with the following steps to run a serial 3D triangular aerofoil simulation with inflow turbulence:

1. Navigate to the `PyFR-Test-Cases/3d-triangular-aerofoil` directory:

```
cd PyFR-Test-Cases/3d-triangular-aerofoil
```

2. Unzip the Gmsh mesh file file and run pyfr to covert it into a PyFR mesh file called `triangular-aerofoil.pyfrm`:

```
unxz triangular-aerofoil.msh.xz
pyfr import triangular-aerofoil.msh triangular-aerofoil.pyfrm
```

Fig. 5: Colour map of density distribution at 1 time unit.

3. Run pyfr to solve the Navier-Stokes equations on the mesh, generating a series of PyFR solution files called `triangular-aerofoil-*.pyfrs`:

```
pyfr -p run -b cuda triangular-aerofoil.pyfrm triangular-aerofoil.ini
```

4. Run pyfr on the solution file `triangular-aerofoil-5.00.pyfrs` converting it into an unstructured VTK file called `triangular-aerofoil-5.00.vtu`:

```
pyfr export triangular-aerofoil.pyfrm triangular-aerofoil-5.00.pyfrs triangular-
↪aerofoil-5.00.vtu
```

5. Visualise the unstructured VTK file in Paraview

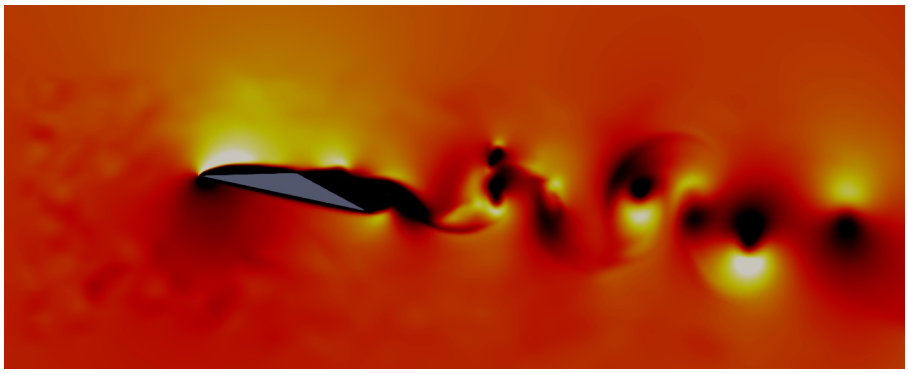

Fig. 6: Colour map of velocity magnitude distribution at 5 time units.

6. If you have installed *Ascent* you can run the same case with the *[soln-plugin-ascent]* plugin activated, which will produce a series of .png images that can then be merged into an animation using a utility such as ffmpeg:

```
pyfr -p run -b cuda triangular-aerofoil.pyfrm triangular-aerofoil-ascent.ini
```

5.2.5 3D Taylor-Green

Proceed with the following steps to run a serial 3D Taylor-Green simulation:

1. Navigate to the `PyFR-Test-Cases/3d-taylor-green` directory:

```
cd PyFR-Test-Cases/3d-taylor-green
```

2. Unzip the Gmsh mesh file file and run pyfr to covert it into a PyFR mesh file called `taylor-green.pyfrm`:

```
unxz taylor-green.msh.xz
pyfr import taylor-green.msh taylor-green.pyfrm
```

3. Run pyfr to solve the Navier-Stokes equations on the mesh, generating a series of PyFR solution files called `taylor-green-*.pyfrs`:

```
pyfr -p run -b cuda taylor-green.pyfrm taylor-green.ini
```

4. Run pyfr on the solution file `taylor-green-5.00.pyfrs` converting it into an unstructured VTK file called `taylor-green-5.00.vtu`:

```
pyfr export taylor-green.pyfrm taylor-green-5.00.pyfrs taylor-green-5.00.vtu
```

5. Visualise the unstructured VTK file in Paraview

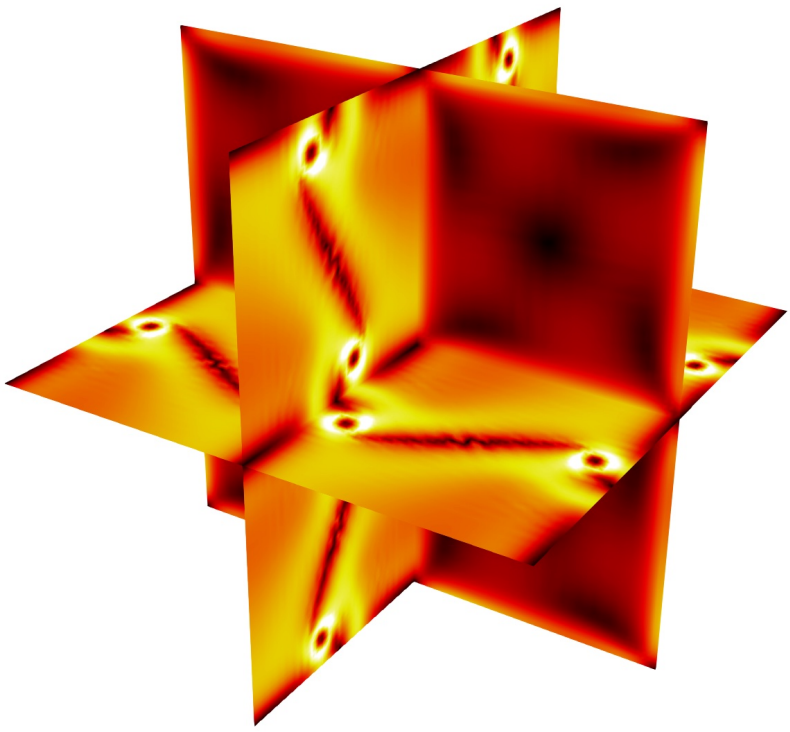

Fig. 7: Colour map of velocity magnitude distribution at 5 time units.

6. If you have installed *Ascent* you can run the same case with the *[soln-plugin-ascent]* plugin activated, which will produce a series of .png images that can then be merged into an animation using a utility such as ffmpeg:

```
pyfr -p run -b cuda taylor-green.pyfrm taylor-green-ascent.ini
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

G

H

K

L

M

N

O

R

T

U

V

W

X