



zCFDguide Documentation

Release v2023.11.6307-36-ge2de7

Zenotech Ltd

Nov 23, 2023

CONTENTS

| | |
|---|----------|
| 1 Version: v2023.11.6307-36-ge2de7 | 1 |
| 1.1 Overview | 1 |
| 1.2 Installation | 2 |
| 1.3 Running zCFD | 4 |
| 1.4 Getting Started | 11 |
| 1.5 Reference Guide | 64 |
| 1.6 zCFD Functions | 127 |
| 1.7 Mesh Conversion | 133 |
| 1.8 FWH solver | 135 |
| 1.9 zM3 | 141 |
| 1.10 Visualisation | 144 |
| 1.11 Utilities | 145 |

VERSION: V2023.11.6307-36-GE2DE7

1.1 Overview

zCFD is a GPU-accelerated high-performance computational fluid dynamics (CFD) solver.

zCFD is designed to make highly effective use of modern computer architectures. It is structured as a high performance core, executed via a flexible Python based wrapper. zCFD will run on unstructured meshes from a variety of sources, and will export data in many common formats. zCFD offers an easier route to coupling multi-disciplinary capabilities and integrating third-party data sources.

Features include both finite-volume and finite-element, (high order Discontinuous Galerkin) formulations, overset capability and fluid-structure interaction (FSI) running on both CPUs and GPUs. zCFD is capable of performing both steady-state and time accurate simulations, with a range of time marching schemes. The numerical schemes available have been selected for accuracy and robustness. The code provides automatic scaling of turbulent wall functions; automatic determination of inflow / outflow conditions and pre-conditioning to allow for low and high Mach number flow.

zCFD makes use of the MPI protocol to manage execution over a set of compute nodes. The execution process is completely parallel from the start to finish including all the input/output functionality. To support this scalable implementation the mesh input is provided in hdf5 format which is portable and self describing. Filters are provided to convert the most frequently used formats to this internal format.

zCFD uses a single control dictionary to specify all the parameters. This is a Python file containing a python dictionary with certain mandatory keys. The use of Python for the control dictionary enables the user to add custom functions that populate the dictionary at runtime.

The zCFD distribution includes a GPU-accelerated high performance Computational Aero-Acoustics solver (CAA), primarily aimed at broadband, aerodynamically generated noise. This is based on the high order Discontinuous Galerkin (DG) method which solves the Acoustic Perturbation Equations (APE) to predict noise based on aerodynamic sources. zCFD is a time domain solver therefore noise signals can be post-processed to provide narrowband and broadband spectra.

In order to generate aeroacoustic noise predictions, acoustic sources can be specified using a stochastic noise generation technique based on the Fast Random Particle Mesh (FRPM) method, alternatively, acoustic sources from high fidelity CFD simulations can be used.

To improve ease of use, the included zMesh mesh generator will generate appropriate acoustics meshes, permitting optimum time steps, using the CFD surface mesh, thus eliminating further CAD clean up and meshing effort by the user.

1.2 Installation

1.2.1 System Requirements

zCFD is supported on the following Linux distributions:

- RHEL 7+
- CentOS 7+
- Ubuntu 20.04+
- Windows 10/11 via [WSL2](#)

By default zCFD is compiled for x86_64 but is available for ARM and IBM Power on request.

GPU support

zCFD supports running on NVidia GPUs with a compute capability of 6.1 and above (See [Compute Capability](#)).

An AMD GPU (HIP) based version is available on request.

1.2.2 Linux / Unix

If you are running Linux/Unix zCFD is provided as a stand-alone installer containing all the packages required to get started. The installer is available for download from the [zCFD website](#). Once the zCFD installer has been downloaded run the install script and follow the prompts.

```
$ bash zCFD-icx-sse-impi-<INSTALL_VERSION>-Linux-64bit.sh
```

Then copy the license file provided over email into the lic folder in the install directory.

```
$ cp zcfd.lic /<path to zCFD install>/lic/
```

Once zCFD has been installed the zCFD environment can be activated from a bash shell:

```
$ source /<path to zCFD install>/bin/activate  
(zCFD)$
```

1.2.3 Windows

zCFD can be run locally on windows machines if Windows Subsystem for Linux 2 (WSL2) is installed. To install WSL2 simply follow the Microsoft instructions which can be found [here](#).

The chosen Linux distribution shouldn't have an effect on how the code runs, but the Linux distributions listed above are supported.

In order to access your Linux filesystem on Windows it can be useful to add a quick access link in file explorer. To do this, in the address in file explorer enter

```
\wsl$
```

This will show you the root folders for your installed Linux distributions.

The installer is available for download from the [zCFD website](#). For WSL, download the standard Linux distribution. Once the zCFD installer has been downloaded run the install script from within the WSL2 environment and follow the prompts.

```
$ bash zCFD-icc-sse-impi-<INSTALL_VERSION>-Linux-64bit.sh
```

Then copy the license file provided over email into the lic folder in the install directory.

```
$ cp zcfd.lic /<path to zCFD install>/lic/
```

Once zCFD has been installed the zCFD environment can be activated:

```
$ source /<path to zCFD install>/bin/activate
(zCFD)$
```

1.2.4 Licence

zCFD requires a license. If you do not already have a license, please contact zcfd@zenotech.com to obtain one. When zCFD starts it will look in the following locations for a licence file:

1. 'zcfd.lic' in the directory alongside the input files
2. 'zcfd.lic' in the lic directory of the zCFD installation. i.e. /<path to zCFD install>/lic/
3. RLM_LICENSE environment variable

The RLM_LICENSE environment variable can be defined as below.

```
export RLM_LICENSE=/path/to/license/file
```

or, where port is the port number in the license file, and host is the hostname in the license file

```
export RLM_LICENSE=port@server
```

In the case that the system running the simulation cannot access the internet a local license server will be required. See <http://www.reprisesoftware.com/admin/software-licensing.php> for details.

Licence Error codes

Below is a list of some of the error codes that you may encounter due to licensing errors. If the error code you receive is not listed, please contact us.

| Error Code | Description | Possible Remedy |
|------------|---|--|
| -3 | License has expired | Contact us to renew your license |
| -6 | Requested version is not supported | Your license is for an older product version; contact us to upgrade |
| -8 | checkout request for too many licenses | Reduce the number of ranks that you are running the solver on or stop any existing running processes. |
| -21 | No heartbeat | The license heartbeat was not received; check that the license server is still running and that have connectivity to it |
| -22 | All licenses in use | All licenses are currently in use; stop any running processes to free up a license. |
| -37 | License start date has not been reached | Your license has not yet become valid. Contact us if you believe this to be in error |
| -102 | Can't read license data | Check the permissions on the license file on the server |
| -103 | Network error | Check that you have connectivity to the license server; this may involve opening a port on your firewall or connecting to the internet |

continues on next page

Table 1 – continued from previous page

| Error Code | Description | Possible Remedy |
|------------|---|--|
| -104 | Error writing to network | Check your connectivity to the license server; check firewall settings and any endpoint security software you may be running |
| -105 | Error reading from network | Check your connectivity to the license server; check firewall settings and any endpoint security software you may be running |
| -106 | Unexpected response from license server | Check that your license configuration is pointing to the correct server |
| -107 | HELLO message for wrong server | Check that your license configuration is pointing to the correct server |
| -108 | Error in private key | There is an issue with your server side license file - contact us to resolve this issue |
| -109 | Error signing authorization | There is an issue with your server side license file - contact us to resolve this issue |
| -110 | Internal error | An error occurred in the license server - contact us if this error repeats itself |
| -111 | Connection refused at server | Check your firewall settings and that you are pointing at the correct license server |
| -112 | No server to connect to | Check that the server is running and that you are pointing at the correct license server |
| -113 | Bad Communication handshake | Check that the server is running and that you are pointing at the correct license server |
| -114 | Can't get ethernet address | Check that the user that you are running as has correct permissions to query the network card |

1.3 Running zCFD

zCFD is designed to exploit parallelism in the target computing hardware, making use of multiple processes (MPI), multiple threads per CPU process (OpenMP) and / or many threads per GPU. zCFD will handle most parallel functions automatically, so the user just needs to specify how much, and what type, of resource to use for a given run.

Table of Contents

- [Command Line Execution](#)
- [Input Validation](#)
- [Overriding Mesh and Case Names](#)
- [Batch Queue Submission](#)
- [EPIC](#)
- [EPIC Desktops](#)
- [Using Jupyter](#)
- [zCFD Environment Variables](#)

1.3.1 Command Line Execution

zCFD includes an in-built command-line environment, which should be activated either when run directly or via a cluster management and job scheduling system, such as Slurm.

```
>source /INSTALL_LOCATION/zCFD-version/bin/activate
```

This sets up the environment to enable execution of the specific version. When within the command-line environment, the command prompt will show a zCFD prefix:

```
(zCFD) >
```

To deactivate the command line environment, returning the environment to the previous state use:

```
(zCFD) >zdeactivate
```

Your command prompt should return to its normal appearance.

To run zCFD on the command-line:

```
(zCFD) > run_zcf -n <num_tasks> -d <device_type> -p <mesh_name> -c <case_name>
```

where:

| Parameter | Value | Description |
|--------------------|------------|---|
| num_tasks | Integer | The number of partitions (one per device socket - CPU or GPU) |
| device_type | CPU or GPU | Specifies whether or not to use GPU(s) if present |
| mesh_name | String | The name of the mesh file (<mesh>.h5) |
| case_name | String | The name of the control dictionary (<control_dict>.py) |

For example, to run zCFD from the command-line environment on a desktop computer with a single 12-core CPU processor and no GPU:

```
(zCFD) >run_zcf -n 1 -d cpu -p <mesh_name> -c <case_name>
```

By default, zCFD will use all 12 cores via OpenMP unless a specific number is required. To run the above case using only 6 cores (there will be one thread per CPU core):

```
(zCFD) >export OMP_NUM_THREADS=6; run_zcf -n 1 -d cpu -p <mesh_name> -c <case_name>
```

To run zCFD on a desktop computer with a single CPU and two GPUs:

```
(zCFD) >run_zcf -n 2 -d gpu -p <mesh_name> -c <case_name>
```

1.3.2 Input Validation

zCFD provides an input validation script which can be used to check the solver control dictionary before run time reducing the likelihood of a job spending a long time queuing only to fail at start up.

The script can be executed from the zCFD environment as follows:

```
(zCFD) >validate_input <case_name> [-m <mesh_name>]
```

```
Validating parameters dictionary...
Parameters dictionary is valid
Checking BC_, FR_, FZ_, IC_ and TR_ numbering...
IC_ numbering is correct
```

(continues on next page)

(continued from previous page)

```
FR_ numbering is correct
BC_ numbering is correct
zCFD input file <case_name> is valid
```

If the -m option is given with a mesh file the script will check whether any zones specified as lists to boundary conditions, transforms or reports in the input exist in the mesh.

1.3.3 Overriding Mesh and Case Names

For more advanced simulations with overset meshes (see *overset*) an override dictionary can be supplied as an additional argument to the command-line, in interactive mode or within a batch submission script, to specify multiple meshes and associated control dictionaries:

```
(zCFD) >run_zcfcd -n 2 -d gpu -f <override_file>
```

An example override.py file for a background domain with a single rotating turbine would be:

```
override = {'problem_case_pair': [(['background.h5', 'background.py'],
('turbine.h5', 'turbine.py')]}]
```

1.3.4 Batch Queue Submission

To run zCFD on a compute cluster with a queuing system (such as Slurm), the command-line environment activation is included within the submission script:

Example Slurm submission script “run_zcfcd.sub”:

```
#!/bin/bash
#SBATCH -J account_name
#SBATCH --output zcfcd.out
#SBATCH --nodes 2
#SBATCH --ntasks 4
#SBATCH --exclusive
#SBATCH --no-requeue
#SBATCH --time=10:00:00
#SBATCH --gres=gpu:2
#SBATCH --cpus-per-task=16
source /INSTALL_LOCATION/zCFD-version/bin/activate
run_zcfcd -n 4 -d gpu -p <mesh_name> -c <case_name>
```

In this system we have 2 nodes, each with 2 GPUs. We allocate one task per GPU, giving a total of 4 tasks. Note that num_tasks, in this case 4, on the last line should match Slurm “ntasks” on line 5. The case is, optionally, run in “exclusive” mode which means that zCFD has uncontested use of the devices. In this case we have a 64-core CPU, giving 16 CPU cores for each of the 4 tasks. The “gres” line tells zCFD that there are 2 GPUs available on each node.

The submission script would be submitted from the command-line:

```
sbatch run_zcfcd.sub
```

The job can then be managed using the standard Slurm commands.

Note: Users unfamiliar with the batch submission parameters in a specific system should consult the system administrator.

1.3.5 EPIC

For larger jobs or where resource availability is limited, **EPIC** allows users to submit zCFD jobs to a range of on-demand computing resources in an intuitive and secure way. EPIC also provides access to interactive post-processing environments without the need to download large data files.



1.3.6 EPIC Desktops

Table of Contents

- *What are EPIC Desktops?*
- *Launching a desktop*
- *Managing Data*
- *Modules*
- *Loading zCFD*
- *zCFD Licence*
- *Running Paraview*
- *Using pvserver*

What are EPIC Desktops?

EPIC provides a way of starting an interactive Linux desktop in the Cloud, these are known as EPIC Desktops. The Desktops come with a range of preinstalled software, including zCFD. Data can automatically be synchronised between EPIC and the Desktops you launch.

Launching a desktop

To launch an EPIC Desktop you need an account on EPIC, you can sign up directly on the website for free. Once you have an account, navigate to Desktops on the right hand menu to launch a new Desktop. The desktops come in a variety of sizes and all have GPUs which can be used for running zCFD in GPU mode as well as for accelerated visualisation.

If your organisation already uses EPIC then speak to your account manager for an invite to join the correct EPIC team.

Managing Data

The data you select when launching the desktop will be made available to you in the `/home/epic/data` directory on the desktop. If you selected the “Online” option when launching the desktop then any changes or new files you make in that directory will be synchronised with EPIC.

Each node also has a `/scratch/` directory, this is a local high performance drive that only exists while the node is online. It is recommended to use this directory for running any zCFD jobs on the node and then copy the results back to `/home/epic/data` when you are finished.

Note `/scratch/` will be deleted when the node terminates.

Modules

The desktops use Linux *modules* to load applications. To see what applications are available launch a new terminal and run:

```
module avail
```

You can then load the application using the *module load* command, for example to load the latest zCFD:

```
module load zcfd/latest
```

This will add zCFD to your path and enable you to start using it in that terminal.

Loading zCFD

The best way to load zCFD is to load the module and then run the *activate* script, this will activate the full zCFD environment.

For example:

```
module load zcfd/latest
. activate
```

You can then start using zCFD:

```
run_zcfd
```

zCFD Licence

After loading zCFD you need to specify which licence to use, you can do this by either copying your *zcfд.lic* file into your working directory, alongside your mesh and control dictionary, or by setting the *RLM_LICENSE_PASSWORD* environment variable. The password can be found inside your zcfд licence file.

For example:

```
module load zcfд/latest
. activate
export RLM_LICENSE_PASSWORD=mypassword
```

Running Paraview

Paraview is also available on the desktops and can be loaded using *modules* and then started from the terminal:

```
module load paraview
paraview
```

Using pvserver

To load a zCFD .h5 file you can use the Paraview server that is bundled with zCFD and then connect to it using Paraview on the desktop. To start the server run the following in the terminal:

```
module load zcfд/latest
. activate
pvserver
```

You can then launch Paraview from another terminal by running:

```
module load paraview
paraview
```

By default *pvserver* will be listening for a connection on port 11111. To connect to the zCFD *pvserver* do the following in Paraview:

1. From the *File* menu select *Connect*.
2. Click “Add server”.
3. Change the server Name to “zcfд” but leave all the other settings as the defaults, click “Configure”. On the next screen just click “Save”.
4. Click “Connect”.
5. The connection should be made and you will see a “Client connected” message output by *pvserver*.

You can then load your zCFD .h5 files via *pvserver*

Note The version of Paraview used to connect must match the version used in zCFD, if you get a client connection error please check the version numbers.

1.3.7 Using Jupyter

We also recommend the use of [Jupyter Notebooks](#) for post-processing zCFD runs. This is one of the easiest ways to use Python, via an interactive notebook on your local computer. The notebook is a locally hosted server that you can access via a web-browser. Jupyter is included in the zCFD distribution and notebooks for visualising the residual data will automatically be created when you run a zCFD case.

To automatically start up notebook server on your computer within the zCFD environment run:

```
(zCFD) >start_lab
```

1.3.8 zCFD Environment Variables

The following table gives some of the environment variables that can be set before running zCFD. These are generally for advanced users can can impact the way zCFD runs on you system.

| Variable | Default | Description |
|----------------------|---------|---|
| ZCFD_GPUDIREC | unset | Set if you are using NVIDIA GPU direct (custom builds only) with a CUDA aware mpi implementation. |
| ZCFD_PRE_LAUN | unset | This option allows a script to be run just before the code is launched so that the user can override any environment that has been automatically setup. |
| CUDA_VISIBLE_DEVICES | unset | This option restricts the list of GPU devices that zCFD will be able to use. Typically this can be used to remove an incompatible GPU or the display GPU from the list of candidate GPUs. |
| ZCFD_DISABLE_P | unset | When set this causes the GPU memory allocated to be page-able and not pinned memory. This should always be used on Windows in WSL as CUDA inside WSL has a fixed low threshold for the maximum amount of allowed pinned memory. |
| ZCFD_NCCL | unset | Set to 1 to make use of the NVIDIA NCCL library during parallel exchanges. |
| PMPI_CUDA | unset | Set if you are running with platform MPI (custom builds only) to enable GPU aware MPI memory allocations |
| MPICH_GPU_SUF | unset | Set to 1 if you are running with Cray MPI (custom builds only) to enable GPU aware MPI memory allocations |
| ZCFD_DISPATCH_ | unset | Makes all kernel launches synchronous. |

1.4 Getting Started

The getting started guide is here to teach the fundamentals of using zCFD, starting with a simple case and building up the complexity of the simulation. These should provide you a good starting point for setting up your own simulations. All of the input files required for the guide are available on line.

1.4.1 Tutorial 1: Simple Aerofoil

Table of Contents

- *What is in this tutorial?*
- *What you need?*
- *Step 1. Download the files*
 - *Time marching*
 - *Equations*
 - *IC_1*
 - *BC_1*
 - *BC_2*
 - *BC_3*
 - *Report*
 - *Write Output*
- *Step 3. Running zCFD*
- *Step 4. Review the output*
- *Step 5. Looking at residuals with notebook*
 - *Plotting residuals*
 - *Plotting forces*
 - *Plotting performance*
- *Step 6. Looking at results with Paraview*
 - *Loading your data*
 - *Visualising Variables*
 - *Pressure Coefficient Plot*

What is in this tutorial?

The NACA0012 is a symmetrical aerofoil which is commonly used for code validation. In this configuration it is a quick and simple simulation to run. By downloading this [zip file](#) and following the instructions below, you should learn to run, post-process and visualise a simple zCFD simulation. This is a small simulation which most users should be able to run on a laptop in ~5 minutes.

What you need?

To run the tutorial you will need an installation of zCFD and a valid licence (the free licence is sufficient for this tutorial). For instructions on installing zCFD please see [here](#) or if you are using an **EPIC Desktop** please see [here](#)

The tutorial assumes that you are running all commands using a bash shell on Linux.

Step 1. Download the files

The tutorial zip file can be downloaded [here](#). The zip file contains a mesh file (*naca0012.h5*) and a zCFD control file (*naca0012.py*).

This tutorial will assume you unzip the file in a directory called “`zcfдTutorial_1`” in your users home directory.

```
cd ~
mkdir zcfдTutorial_1
wget https://zcfд.zenotech.com/tutorials/1/naca0012.zip
unzip naca0012.zip
```

Step 2. Review Control file

naca0012.py is the zCFD control file. When we run zCFD, zCFD reads the control file and uses the Python dictionary **parameters** to set the solver settings. A brief overview of the parameters dictionary for the simulation is given below; for a more detailed description of the control file structure, see the [reference guide](#).

```
parameters = {
    "time marching": {
        "unsteady": {"total time": 1.0, "time step": 1.0},
        "scheme": {"name": "implicit euler", "stage": 1},
        "cfl": 50,
        "cfl ramp factor": {"growth": 1.1, "initial": 1.0},
        "cycles": 1500,
    },
    "equations": "RANS",
    "RANS": {
        "order": "euler_second",
        "turbulence": {"model": "sst"},
        "inviscid flux scheme": "Roe",
    },
    "IC_1": {
        "temperature": 300,
        "pressure": 101325.0,
        "V": {"vector": [1.0, 0.0, 0.0], "Mach": 0.15},
        "viscosity": 1.02145e-05,
        "turbulence intensity": 5.2e-2,
        "eddy viscosity ratio": 1.0,
    },
    "BC_1": {
        "zone": [1],
    }
}
```

```

    "type": "farfield",
    "condition": "IC_1",
    "kind": "riemann",
  },
  "BC_2": {
    "zone": [4],
    "type": "wall",
    "kind": "noslip",
  },
  "BC_3": {
    "zone": [2, 3],
    "type": "symmetry",
  },
  "report": {
    "frequency": 10,
    "forces": {
      "FR_1": {"name": "wall", "zone": [4]},
    }
  },
  "reference": "IC_1",
  "write output": {
    "surface variables": ["V", "p", "mach", "cp", "cf", "pressureforce",
    "frictionforce"],
    "volume variables": ["V", "p", "T", "rho", "mach", "cp", "eddy"],
  },
}

```

Time marching

The start of the parameters dictionary contains the *time marching* sub-dictionary. The *unsteady* sub-dictionary sets whether we are running an *unsteady* or a *steady* simulation. The fact that *total time* and *time step* are the same number means that we're running a steady simulation. Steady simulations assume that the flow we are simulating does not vary with time. Most real flows are not truly 'steady', but steady simulations tend to be much easier and cheaper to run than unsteady simulations. For an attached, streamlined flow like this one using a steady simulation is reasonable when time-resolved data is not required.

CFD simulations work by iteration - we start with a very rough estimate of the flow $f(x, 0)$, then during the first 'iteration cycle' the solver uses $f(x, 0)$ to create a better approximation of the flow, $f(x, 1)$. This is repeated many times - we use $f(x, 1)$ to calculate $f(x, 2)$, $f(x, 2)$ to calculate $f(x, 3)$ and so on, until eventually we reach an estimate of the flow we are happy with.

The flow will generally change very quickly in the first few iterations, and then converge asymptotically on the 'right' answer. The *cycles* parameter states how many iterations should be used in the simulation - the higher it is the more accurate the solution is likely to be, but given the convergence is asymptotic setting it too high will lead to lots of computational effort being used for very little change in the solution towards the end.

The *scheme* defines the numerical scheme used by the solver to get from $f(x, n)$ to $f(x, n + 1)$. The *implicit euler* scheme is a sensible choice to use for this type of simulation.

The *CFL* essentially sets how much the flow is allowed to change between $f(x, n)$ and $f(x, n+1)$ approximations of the flow field - if it is set too high the simulation will not converge, and if it is set too low the simulation will take many cycles reach the appropriate flow.

Equations

The *equations* sets the type of flow we are simulating - there are a number of different options depending on the assumptions we make about the flow. *RANS* is used when we are simulating a compressible flow, and want to simulate the effects of turbulence on the mean flow by use of a turbulence equation. The *order* sets how the flow is spatially discretised, and the *turbulence* sets the turbulence model used.

IC_1

Now that the numerical settings have been defined, the flow needs to have boundary conditions. Boundary conditions define what happens at exterior faces of the mesh. In this case, there are three boundary conditions.

The first boundary condition we consider is that of the freestream air, which is at a temperature of 300 K, a pressure of 101325 Pa and flows parallel to the mesh's x axis at a Mach number of 0.15. The *IC_1* dictionary defines a flow at these conditions.

Because the RANS equations include the effect of viscosity, we also need to define the viscosity at the *IC_1* flow condition (in this case, we set it to be 1.02145×10^{-5} kg m⁻¹ s⁻¹ at 300 K). This corresponds to a chord based Reynolds number of 6 million. Because the RANS equations include turbulence modelling, we also need to set the *turbulence intensity* and *eddy viscosity ratio* - these are used as boundary conditions for the SST turbulence model equations. The values given are sensible values for flows without significant freestream turbulence.

BC_1

The *BC_1* dictionary assigns the *IC_1* flow condition to all mesh faces contained in the mesh zone *1*. Mesh zones are defined during the meshing process, and in this case mesh zone *1* contains all the mesh faces which are on the farfield exterior surface of the mesh. The *farfield* type means that flow can pass freely through the specified mesh faces, and the *riemann* kind means that flow can pass both in and out of the mesh via the specified mesh faces.

The *IC_1* dictionary is, by default, also used to define the initial guess for the flow - at 0 cycles we assume that the entirety of the flow is flowing at the *IC_1* conditions.

BC_2

The *BC_2* dictionary assigns a no-slip wall boundary condition to all mesh faces contained in the mesh zone *4*. Mesh zone *4*, in this case, contains all faces on the aerofoil surface.

BC_3

The *BC_3* dictionary assigns a symmetry boundary condition to all mesh faces contained in mesh zones *2* and *3*, which are the +ve z and -ve z faces of the mesh. This boundary condition type is used here so that we can simulate an aerofoil of effectively infinite span, but with a low mesh resolution in the z axis to reduce simulation cost.

Report

The *report* dictionary defines which flow statistics zCFD should output to the user during the simulation, and how often. The ‘frequency’ key here defines that zCFD should report flow statistics to the user every 10 iteration cycles during the simulation. Exactly which flow statistics are outputted can be changed by the user, but the defaults should be enough for us to successfully monitor the convergence of the solution. The ‘forces’ sub-dictionary allows us to monitor the forces on specific mesh zones- here we are outputting the forces on mesh zone 1, which is the aerofoil surface. Some of the flow statistics (e.g. force on the aerofoil wall) are non-dimensionalised (see [here](#)) and the *reference* parameter defines the flow state they should be non-dimensionalised against. This is normally chosen to be the freestream state.

Write Output

The final section of *parameters*, *write output*, defines which variables should be outputted into VTK format for visualisation by the user at the end of the simulation.

Step 3. Running zCFD

To run zCFD you first need to source the “activate” script and then use the “run_zcf” path.

1. Activating your environment:

```
source ./<ZCFD_INSTALL_PATH>/bin/activate
```

or if using an *EPIC Desktop*:

```
module load zcf/late
source activate
```

2. Run zCFD validate input to check the control dictionary for errors:

```
cd ~/zcf_tutorial_1/
validate_input naca0012.py -m naca0012.h5
```

3. Run zCFD in the case directory:

```
run_zcf -m naca0012.h5 -c naca0012.py
```

zCFD will attempt to detect the number of CPU cores and any GPUs present and make use of them. If you want to manually configure this see [here](#).

As the solver is running it will output information about the current CFL, Multigrid level, timing, reporting information and file I/O. The output from zCFD will be written to the screen and also to a log file “naca0012.log”.

```

Total Iterations: 2
Avg Convergence Rate: 0.0017
Final Residual: 1.368765e-07
Total Reduction in Residual: 3.047538e-06
Maximum Memory Usage: 1.541 GB
-----
Total Time: 0.0204855
setup: 0.0145224 s
solve: 0.0059631 s
solve(per iteration): 0.00298155 s
Timer: Elapsed seconds: 0.112
Breakdown of time spent:
output: 0 days 0 hrs 0 mins 0 s (count: 1499 cycles, avg: 0.0 ms)
reporting: 0 days 0 hrs 0 mins 2 s (count: 1499 cycles, avg: 1.5 ms)
solving: 0 days 0 hrs 3 mins 49 s (count: 1499 cycles, avg: 152.8 ms)
update source terms: 0 days 0 hrs 0 mins 0 s (count: 1499 cycles, avg: 0.1 ms)
cycle 1500 total time: 0 days 0 hrs 0 mins 0 s
CPU 50 (80.0) - Mg 50.0 (count: 0)
iter   Mem Usage (GB)      residual      rate
-----  

 0     1.54065  1.114665e-06  

 1     1.54065  1.829795e-07  0.1642  

 2     1.54065  6.884949e-08  0.3763  

 3     1.54065  3.105055e-08  0.4518  

 4     1.54065  1.7446556e-08 0.5619  

 5     1.54065  1.028472e-08  0.5895  

 6     1.54065  6.720924e-09  0.6535  

 7     1.54065  2.735840e-09  0.5923  

 8     1.54065  1.637551e-09  0.5986  

 9     1.54065  9.747638e-10  0.5953  

-----  

Total Iterations: 18
Avg Convergence Rate: 0.4945
Final Residual: 9.747638e-10
Total Reduction in Residual: 8.744878e-04
Maximum Memory Usage: 1.541 GB
-----
Total Time: 0.072332
setup: 0.016222 s
solve: 0.0561101 s
solve(per iteration): 0.00561101 s
iter   Mem Usage (GB)      residual      rate
-----  

 0     1.54065  4.622099e-02  

 1     1.54065  3.798742e-05  0.0003  

 2     1.54065  1.755518e-07  0.0046  

-----  

Total Iterations: 2
Avg Convergence Rate: 0.0019
Final Residual: 1.755518e-07
Total Reduction in Residual: 3.798097e-06
Maximum Memory Usage: 1.541 GB
-----
Total Time: 0.020358
setup: 0.014583 s
solve: 0.00580198 s
solve(per iteration): 0.00299999 s
-----  

Report: rhoV0 0.005179
Report: rhoV1 1.17148e-13
Report: rhoV2 0.00150043
Report: rhoE 0.0181309
Report: rhok 0.000218339
Report: rhoMeng 2719.66
-----  

Checkpoint Field Output @ 1500
Writing file naca0012_results.h5
Surface Data Field Output @ 1500
Writing VTK Surface Output
Volume Data Field Output @ 1500
Writing VTK Volume Output

```

Per cycle information
(exact format varies with time stepping
and compute device settings)

Writing report

Writing output information

zCFD will run for the 1500 *cycles* specified in the control dictionary, if you want to stop the solver early then you can either press CTRL+C to kill the process or update the *cycles* keyword to be a value smaller than the current cycle number, zCFD will detect this change and stop the solver.

The solver should take a few minutes to complete on a modern GPU, and a bit longer on a CPU.

Step 4. Review the output

A successful zCFD run will result in the following files and directories being created. In this case, since our control file is called *naca0012.py* the output files will also be created with that name. The *_OUTPUT* folder also includes an indication of the number of parallel partitions used, this tutorial assumes a single partition and so the output will be “*naca0012_P1_OUTPUT*” if we ran on two partitions it would be “*naca0012_P2_OUTPUT*”

| File/Directory | Description |
|--|---|
| naca0012.log | This is simply a copy of the output to screen (see above) which happened during the run. |
| naca0012_report.csv | A .csv format table which shows the variation of flow statistics with cycle number |
| naca0012_report.ipynb | An auto-generated Jupyter notebook which can be used to read the naca0012_report.csv file and easily plot the flow statistics varying with cycle number. |
| naca0012_results.h5 | The flow solution calculated by zCFD, in zCFD format. This can be used for visualisation, but it is generally only used for restarting zCFD simulations from |
| naca0012_status.txt | A .json format file outputted by zCFD. Among other things, it records when the simulation was last run, which mesh was used and which control file was used. |
| naca0012_P1_OUTPUT/ | A directory which contains flow solution calculated by zCFD, in VTK format. This is generally more suitable for visualising the data with than the zCFD proprietary format. |
| naca0012_P1_OUTPUT/naca0012.log | A VTK file which, when opened, will load all the volume data from the simulation. |
| naca0012_P1_OUTPUT/naca0012_wall.pvd etc | A series of VTK files (one for each boundary condition type) which, when opened, will load the surface data for that boundary condition. |
| naca0012_P1_OUTPUT/VISUALISATION | Contains the raw data files which are loaded when the naca0012.pvd and naca0012_farfield.pvd, naca0012_wall.pvd etc. files are opened. |
| naca0012_P1_OUTPUT/LOGGING | Contains a separate copy of the naca0012.log file for each MPI partition. Used for debugging. |

Step 5. Looking at residuals with notebook

The naca0012_report.ipynb file can be loaded using a variety of methods, but here we will load it using a browser. The commands below should open Jupyter Notebook within a browser window. If the browser does not open automatically then the notebook URL should be visible in the command output.

1. Activate your environment:

```
. ./<ZCFD_INSTALL_PATH>/bin/activate
```

or if using an *EPIC Desktop*:

```
module load zcfd/latest
. activate
```

2. Start the Jupyter Notebook:

```
cd ~/zcfd_tutorial_1/
start_lab
```

Load the naca0012_report.ipynb in Jupyter, then click run all.

Plotting residuals

The naca0012_report.ipynb makes use of the `zutil.plot.Report` class to read and plot the zCFD report data. Calling the `plot('naca0012_report.csv')` method reads the `naca0012_report.csv` file and prints a graph of solution variable residuals against cycle number .

```
from zutil.plot import *
r = Report()
r.plot('naca0012_report.csv')
```

The residual variables essentially measure how close to a stable solution of the chosen flow equations (e.g. the RANS equations) the simulation is. There is a residual for each flow variable, and the lower the residual, the more converged the flow solution is.



As would be expected (note the logarithmic y axis), the residuals drop very quickly in the first couple of hundred iterations as the flow quickly develops from the initial condition (which in this case, is the freestream condition everywhere). After that, the flow residuals (ρ , $\rho V[0]$, $\rho V[1]$, $\rho V[2]$ and ρE) residuals continue to drop, while the turbulence residuals ($\rho \Omega$ and $\rho \Omega$) drop initially before flattening / increasing.

This behaviour is common - the turbulence equations can take a while to react to changes in the flow field, and the initial guess for the turbulence equations is often poorer than the initial conditions for the flow field. A combination of the fact that the flow residuals have dropped several orders of magnitude over the simulation history and the fact that the residuals are in effect flat from 20,000 cycles onwards mean we can be confident that the simulation is sufficiently converged.

Plotting forces

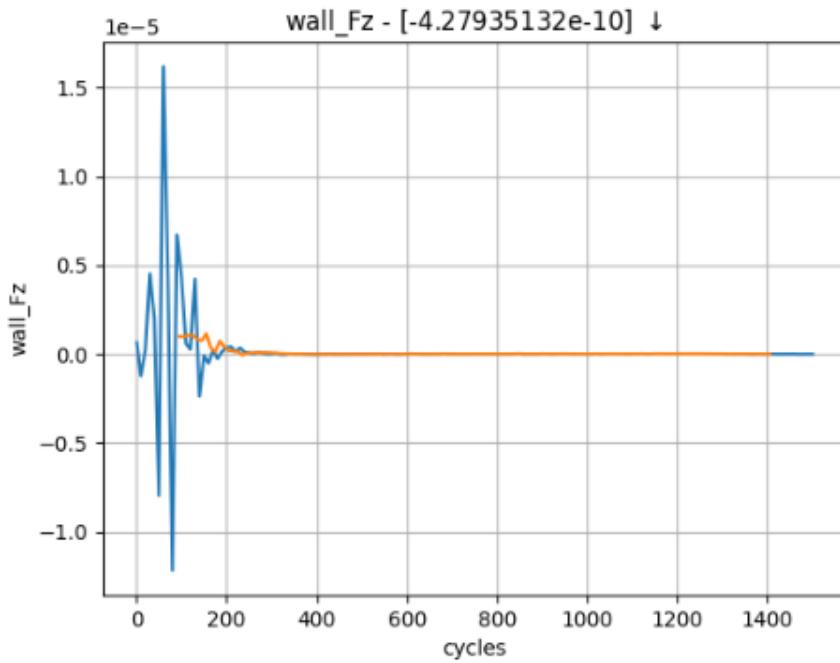
Next, we can use the Jupyter notebook to plot the variation of the force coefficients on the aerofoil surface with cycle number.

```
r.plot_forces()
```

Check the *wall_Fx* and *wall_Fz* tickboxes - given this is a symmetrical aerofoil run at 0 angle of attack, the x force corresponds to drag and the z force corresponds to lift. We would expect zero lift and a non-zero, positive drag.



Figure 6

 wall_Fx wall_Fpx wall_Ffx wall_Fy wall_Fpy wall_Ffy wall_Fz wall_Fpz wall_Ffz

The force plots tell a similar story to the residuals - there are initially large oscillations as the flow settles down, before the force traces essentially flatline from 1200 cycles onwards.

Plotting performance

```
r.plot_performance('naca0012.log')
```

Finally, we can plot the time used per cycle using the `plot_performance(naca0012.log)` method. The last cycle is particularly long because it is the only cycle where the flow solution is written to a file.

Step 6. Looking at results with Paraview

Paraview is a generally purpose visualisation tool that is the recommended way of post processing zCFD results. Please see [ParaView.org](#) for download and installation instructions.

Or if using an *EPIC Desktop* then Paraview is available as a module:

```
module load paraview
paraview
```

Loading your data

As mentioned above a “naca0012_P1_OUTPUT” folder has been generated, and contains a “.pvf” file (naca0012.pvf). This file can be seen as an of index of all the different output files, this way, your output data is imported onto Paraview using a single file, making it easier to handle.

After opening Paraview, simply click on the “Open” icon shown below or go into File>Open or press Ctrl+O. Locate the directory in which you have stored your data and open naca0012.pvf.



Once the file is correctly loaded in Paraview, the “Pipeline Browser” on the left of your screen should look like this:



Now you need to click “Apply”, which will lead to the following layout:



You may need to use the ‘set view direction to +ve y’ button:



Visualising Variables

By default, Paraview shows the “cellvolume” variable. This allows to see that the mesh is refined near the center of the mesh, where the airfoil is. This is common practice, as capturing the flow around a body is more demanding than doing so in the free-stream. Naturally, having a really fine mesh over the entire fluid domain would make the problem really computationally expensive.

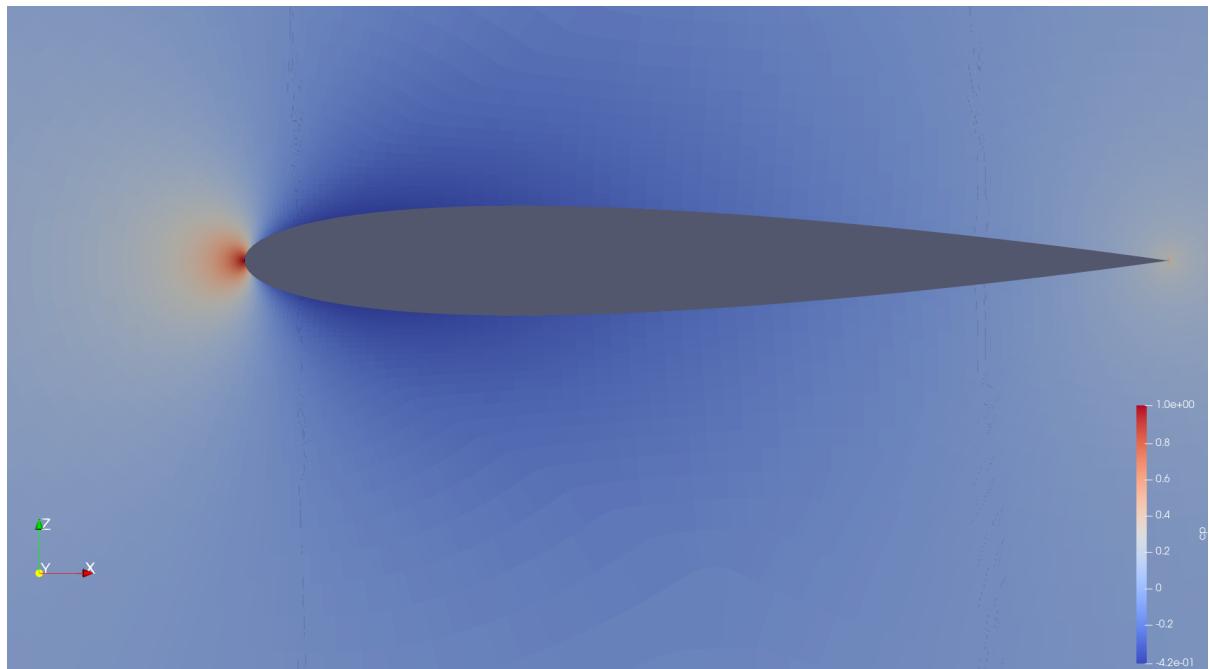
Other variables can be selected and shown using the following scrolling menu:



Let's observe the pressure coefficient around the airfoil. To do so, use the scrolling menu and select the “cp” variable.

Note: The pressure coefficient is given by the following formula $C_p = \frac{p - p_\infty}{\frac{1}{2} \rho_\infty V_\infty^2}$

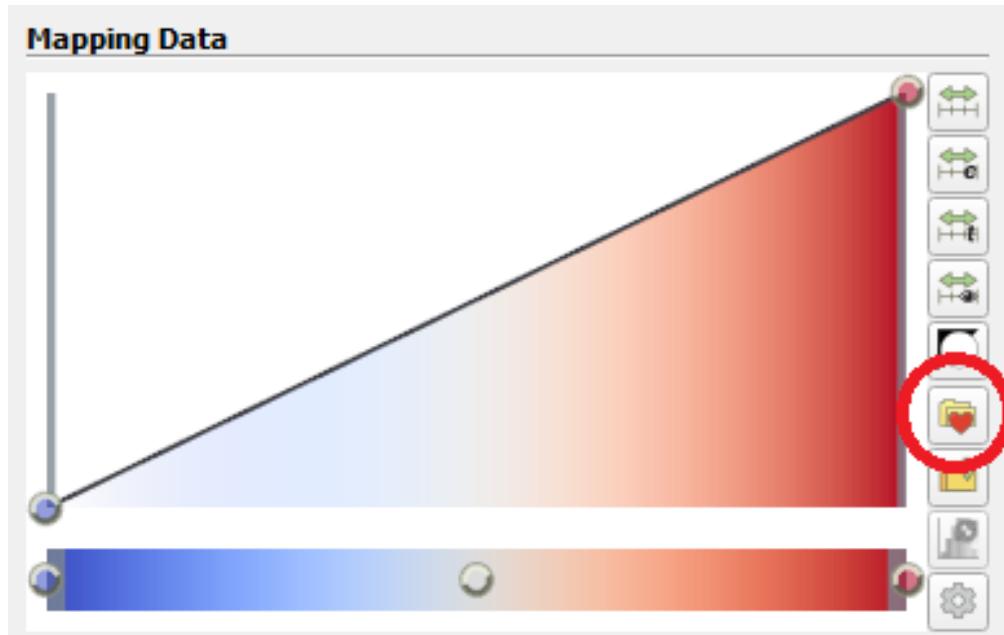
This gives the following (you will need to zoom in substantially to the centre of the mesh before you can clearly see the airfoil):



Note that you can change the colormap used by Paraview by clicking on the “Edit Color Map” icon:



and then the “Choose Preset” icon:



Here are the same results shown using different colormaps:



Those simple steps allow to qualitatively inspect the solution given by our CFD simulation. More advanced analysis is also feasible. A common result in aerodynamics, especially around an airfoil, is the plot of the pressure coefficient along the chord.

Pressure Coefficient Plot

Open the “naca0012_wall.pvd” file . Naturally, the resulting image looks different from the previous as this only contains the zones where a “wall” boundary condition has been defined, which in this case happens to just be the airfoil itself.



It is usually of interest to have a plot showing the pressure coefficient on the upper and the lower surface, therefore it is necessary to extract the data separately for each face of the airfoil. The upper and lower surfaces of the airfoil then need to be separated.

To this end, the “Clip” filter has to be used. To use it, right-click on the “naca0012_wall.pvd” object in the Pipeline Browser, select Add Filter -> Alphabetical -> Clip. The properties of the “Clip” filter are displayed on the left. Firstly, untick the “Invert” option and then you should set them to the following combination:



By default, this will “remove” the lower surface of the airfoil, leaving the following shape:



Note: To remove the upper surface instead, just tick the “Invert” option in the “Clip” filter’s properties.

Now, we want to take a “Slice” of the remaining surface. To do so, follow the same procedure as for the “Clip” filter but select the “Slice” filter instead. When prompted, set the properties of the filter to be as follows:



You should end up with this:



Then the data should be converted from “cell data” to “point data”. This is done by using the “CellDatatoPointData” filter. Access by right-clicking on the “Slice” object selecting “Add filters”.

Once this is done, a plot can be produced in Paraview using the “PlotOnSortedLines” filter. Right-click on the “CellDatatoPointData1” object created previously and add the “PlotOnSortedLines” filter. A plot should appear on the right. You can select the variables to plot in the “Properties -> Series Parameters” window, as

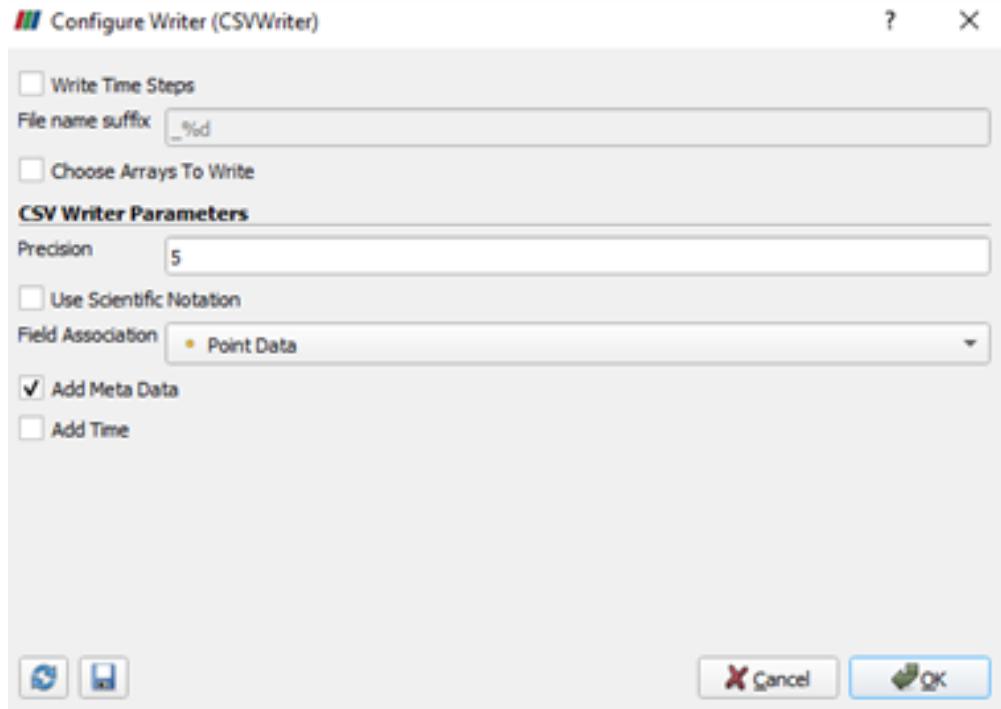
well as changing the settings of the axis. Select the “cp” variable in “Series Parameters”, and use ‘Points_X’ as the ‘X Array Name’. Untick ‘Show Legend’, set ‘Left Axis Title’ to ‘\$C_p\$’ and set the ‘Bottom Axis Title’ to ‘x’. You can also use the “Left Axis Use Custom Range” option to change the y-axis range:



Note that setting the maximum as a negative number will “invert” the y-axis and the negative numbers will be shown “at the top”.



The data can be exported as a .csv file, which can then be manipulated in Python, Excel, MATLAB or any other software. To do so, go to File -> Save data -> *select a folder and give a name to the file*. The following window should appear:



By default, Paraview writes all the variables into the .csv file. However, if only a few of them are of interest, you can use the “Choose Arrays To Write” option.

Now, this procedure can be repeated for the lower surface of the airfoil. A pressure coefficient plot can then be made for validation purposes.

1.4.2 Tutorial 2: Steady State RANS

Table of Contents

- [Control Dictionary Breakdown](#)
- [Running the case](#)
- [Monitoring convergence](#)
- [Plotting force convergence](#)
- [Paraview](#)
- [Citations](#)

In Tutorial 1, we learnt how to run a case in zCFD and visualise the results. We will now use a more complex test case to showcase some of the features of the solver. The case we will look at is the 30P30N aerofoil, which is a 2D multi element aerofoil, developed as an acoustic benchmark case [1]. The geometry of this case can be seen below:



Initially we will begin by obtaining a steady state RANS solution.

The required meshes and control dictionary can be found [here](#). The case has 64,000 cells, and will run in less than 10 minutes on an NVIDIA RTX 3060 GPU.

This case is run as a steady state simulation, using an implicit time marching scheme. The Reynolds Averaged Navier-Stokes equations are solved using Menter's SST turbulence model with wall functions.

Control Dictionary Breakdown

The setup for this case is very similar to the previous NACA0012 cases, but the control dictionary does feature some extra terms which can be useful for a more streamlined workflow.

Reference variables

At the top of the control file, the key variables for the case- Reynolds number, Mach number, temperature etc... are all hard coded, and then referred to later in the script. In cases where you might want to run many simulations with slightly different values to change, this can be an effective way to ensure consistency with variables. Additionally this allows implicit values such as the reference velocity, and freestream Mach to be calculated automatically in the script.

```
# create variables to assign main experimental parameters
reynolds = 1.71e6
mach = 0.17
T = 295.56
p = 101325
R = 287.6
gamma = 1.4
reference_length = 0.457
alpha = 5.5

# calculate implicit values
rho = p / (R * T)
U = math.sqrt(gamma * R * T) * mach

# assign each mesh zone to a boundary
wall = [6]
symmetry = [4,5]
farfield =[7]
```

Scale

In this case a scaling is applied to convert the size of the mesh. The .h5 mesh file was generated with units of inches, whereas the flow conditions are specified in SI units, therefore a scaling of 0.0254 is applied to convert the mesh into metres. The z (spanwise) direction has been scaled to be 1 metre wide. The mesh has only a single cell in the z direction so no spanwise flow is expected - making the z extent of the mesh 1 metre simply means that the forces and moments reported by zCFD will already be per unit span.

```
# scale the mesh to match the experimental data
'scale' : [0.0254,0.0254,0.0254/0.127],
```

Inflow vector

In this case we want to simulate the aerofoil operating at an angle of attack of 4 degrees. Rather than rotating the mesh, it is easier to rotate the inflow vector relative to the mesh, an example of a Galilean transformation. The zutil function `vector_from_angle()` calculates the inflow vector given an x-z angle of attack, x-y angle of attack and flow speed.

```
'IC_1' : {
    'temperature': T,
    'pressure': p,
    'V': {
        # calculate the inflow vector based on the angle of attack
        'vector' : zutil.vector_from_angle(0.0,alpha,U),
    },
    'Reynolds No' : reynolds,
    'Reference Length' : reference_length,
    'turbulence intensity': 0.01,
    'eddy viscosity ratio': 0.1,
    'ambient turbulence intensity': 1e-20,
    'ambient eddy viscosity ratio': 1e-20,
},
```

Transform

The lift and drag forces acting on a body are defined relative to the freestream flow, lift normal and drag parallel. In this case, where we have rotated the relative inflow vector to a specific angle of attack, it is also useful to rotate the force report by the same vector. The transformed forces will appear as `Ft` terms in the report file.

```
# define a function to rotate the output forces by the angle of attack
def my_transform(x,y,z):
    v = [x,y,z]
    v = zutil.rotate_vector(v,0.0,alpha)
    return {'v1' : v[0], 'v2' : v[1], 'v3' : v[2]}
```

Running the case

You can run the case as you did for Tutorial 1, but with the following **run_zcfd** command:

```
run_zcfd -m 30p30n_coarse.h5 -c 30p30n_steady.py
```

Monitoring convergence

Start by launching a Jupyter server to examine the residuals in the `30p30n_steady_report.csv` file. Running the first cell from the `30p30n_steady_report.ipynb` notebook will plot the residuals for the continuity, momentum, and energy equations, as well as the residuals from the turbulence model.

If the case is still running you can update the plot by rerunning the cell, this provides a way to monitor the progress of a running simulation.



You will notice that case is set to run for 1000 cycles but looking at the plots, it looks like the residuals are still converging. So we will now restart the solver from where we finished and run on for another 1000 cycles.

Performing a restart

To restart the solver we need to update the '`restart`' parameter in the control dictionary. If this is set to *False* the solver will always perform a fresh start, if it is *True* then a results file will be read in and the solve will start based on the data in that file. To enable the restart edit `30p30n_steady.py` and change `restart` to *True*. You will also need to update '`cycles`' to 2000.

```
"restart": True,  
  
'time marching' : {  
    ...  
    'cycles' : 2000,  
    ...  
}
```

By default zCFD will look for a results file with the same name as the current case file, appended with ‘_results.h5’. So in our case it will look for “30p30n_steady_results.h5”. When you have edited the control dictionary go ahead and restart the simulation again using the same **run_zcf**d command:

```
run_zcf d -m 30p30n_coarse.h5 -c 30p30n_steady.py
```

Once the solver has finished rerun the cell in the notebook to examine the residuals again. The residuals now look like they have converged:



Plotting force convergence

Examining the force convergence history next, we are interested in the x and y forces, but transformed by the inflow vector, therefore the wall_Ftx and wall_Fty plots are of interest.



Which show reasonable agreement with experimental results, and additionally that the forces are at least nearing convergence. The grid for this case is still extremely coarse, accounting for the differences in lift and drag coefficients.

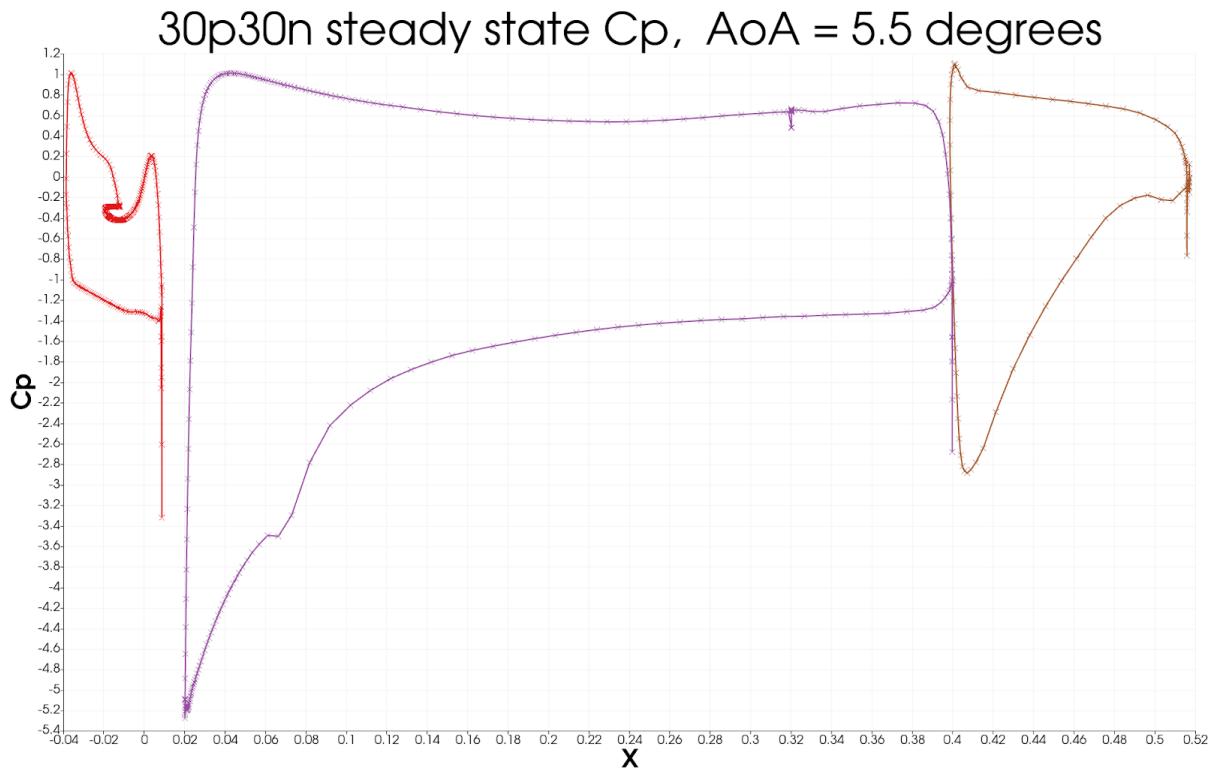
Paraview

Opening Paraview, and loading in the *30p30n_steady_wall.pvd* results will allow you to view the aerofoil surface results:



To get a continuous plot of cp against x/c you need to use a plot over sorted lines filter. To do this:

1. On the surface data apply a '*cell data to point data*' filter.
2. On the *cell data to point data* apply a '*slice*' filter, ensuring the slice uses a z normal, and is centred through the middle of the aerofoil
3. On the *slice* data apply a '*plot on sorted lines*' filter and click apply. This will then bring up a new '*line graph*' view.
4. Make sure to select all 3 segments in the composite data set dialogue box, then ensure only the cp variables are selected in the series parameters. Finally for the X Array name, select '*Points_X*'
5. Modify the style and markers until you are happy with the result.



Citations

1. Zhang, Yufei & Chen, Haixin & Wang, Kan & Wang, Meng. (2017). Aeroacoustic Prediction of a Multi-Element Airfoil Using Wall-Modeled Large-Eddy Simulation. AIAA Journal. 55. 1-15. 10.2514/1.J055853.

1.4.3 Tutorial 3: Time accurate solution

Table of Contents

- *Introduction*
- *Control Dictionary Breakdown*
- *Running the case*
- *Post Processing*

Introduction

Following on from executing the steady state simulation of [Tutorial 2](#), this tutorial will explore the generation of a time accurate result for the unsteady laminar shedding from a 2D cylinder.



You can download the required files [here](#).

Control Dictionary Breakdown

This case uses a similar control dictionary as the previous 30p30n steady state simulation, with the main difference being the time marching, and output regions of the file.

Time Marching

To switch to a time accurate simulation we need to update the “*time marching*” dictionary. In the steady state [tutorial 2](#) case the ‘*total time*’ and ‘*time step*’ parameters were both set to 1.0, this causes the solver to run a single steady state timestep for the number of cycles defined by the ‘*cycles*’ parameter.

We will now update these to run the time accurate simulation. To do this we set ‘*total time*’ to be 1.2, which means the simulation will be run for a total of 1.2 seconds. ‘*time step*’ is set to 0.002 seconds, this is the interval at which the solver will step time forwards. Which means we will have a total of $1.2 / 0.002 = 600$ total unsteady timesteps to perform.

```
'time marching' : {
    'unsteady' : {
        'total time' : 1.2,
        'time step' : 0.002,
        'order' : 'second',
        'start' : 0,
    },
    'scheme' : {
        'name' : 'implicit euler',
        'stage': 1,
    },
    "cfl": 200,
    "cfl viscous factor": 1.0E-6,
    "cfl ramp factor": {"initial": 0.1, "growth": 1.05},
    "cycles": 5,
},
```

zCFD uses dual time stepping to advance in real time by first iterating to convergence on an inner steady state ‘pseudo time’ loop, before advancing the real time cycle. In this case the inner loop is performed using implicit euler time marching, the number of cycles this inner loop performs is controlled by ‘cycles’. So the in the example above the inner loop runs at a CFL of 200 for 5 cycles.

The choice of timestep size, and number of cycles per inner loop will have a significant effect on the quality and cost of the final solution. Using too large of a real time step will mean higher frequency unsteady effects are lost, similarly too few inner cycles will result in unsteady effects being lost. A rule of thumb is to ensure the residuals converge by at least two orders of magnitude over an inner cycle. On the other hand if too many cycles or timesteps are performed, the cost of the simulation will quickly increase. Finally the number of cycles to reach convergence in a pseudo time loop also depends on the step size. Meaning in some cases it is actually beneficial to run a smaller timestep, in order to require fewer inner loop iterations.

Output

The ‘*write output*’ dictionary in the control dictionary is a bit more complicated here than in [tutorial 1](#) or [tutorial 2](#).

```
"write output": {
    "format": "vtk",
    "surface variables": ["V", "p", "T", "rho", "cp"],
    "volume variables": ["V", "p", "T", "rho", "m", "cp"],
    "frequency": {
        "volume data": 5,
        "surface data": 5,
        "checkpoint": 20,
    },
},
```

First of all, the ‘*frequency*’ parameter takes on a different meaning in a time-accurate simulation ([tutorial 1](#) and [tutorial 2](#) are both steady-state simulations).

In a steady-state simulation, the ‘*frequency*’ parameter controls how the number of pseudo-time ‘cycles’ between solver outputs. Since we have no real interest in any data apart from the converged data at the end for a steady-state simulation, every time the solver outputs data, it over-writes the previous output data.

We run time-accurate simulations when we’re interested in the variation of the flow over time (e.g. due to vortex shedding), so in the time-accurate solver data outputs are not over-written, and the ‘*frequency*’ variable relates to the number of real time steps between outputs, instead of number of pseudo-time cycles between outputs.

While we can just use a single number for ‘*frequency*’ (e.g. “frequency”: 5), we can also specify ‘*frequency*’ as a dictionary for more fine grained control. This is especially relevant in time-accurate simulations, where frequent data output can not only slow down the simulation but also quickly use up disk space. Generally the volume data takes longest to output, so for large cases it can be a good idea to output volume data output infrequently, and instead rely on surface data, [monitor points](#) and [volume interpolated data](#) for high frequency data collection.

In this, a relatively small case, we have chosen to output surface and volume data every 5 real time-steps. Since the ‘*time step*’ was set to 0.002 seconds, the volume and surface data will therefore be written out every 0.01 seconds of simulated time. We have chosen to output *checkpoint* data (see [here](#)) less frequently.

Monitor Points

In order to capture the shedding frequencies of vortices behind the cylinder, we will place a monitor point in this region to append data to the report file. A monitor point will simply report the requested output variables, at the point defined within the flow. These results can be visualised alongside the force report in the output notebook.

```
"monitor": {
    "MR_1": {
        "name": "probe",
        "point": [0.25, 1.07, 0.313],
        "variables": ["cp"]
    }
},
```

Running the case

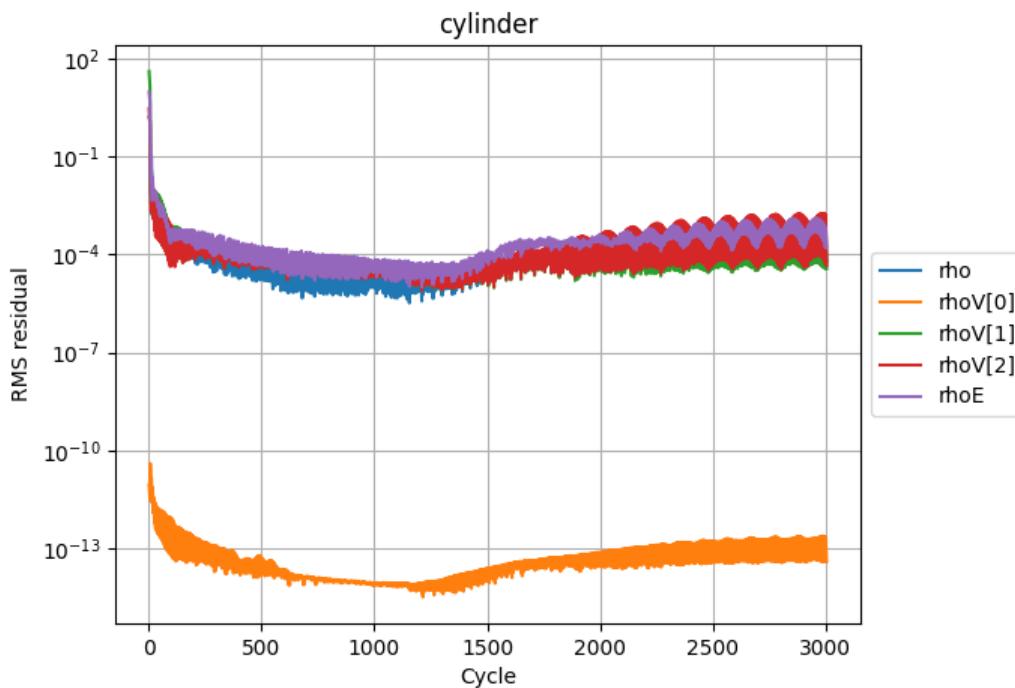
You can run the case as you did for Tutorial 1, but with the following **run_zcf** command:

```
run_zcf -m cylinder.h5 -c cylinder.py
```

Post Processing

Output report

Start and connect to a Jupyter server, using '**start_lab**' as outlined in the previous sections. Go ahead and run all cells.



In the “plot_forces” cell, you should notice additional data points available provided by the monitor point. Visualising the pressure trace [*probe_cp*] should reveal an oscillatory pattern, indicating that the predicted vortex shedding is indeed occurring. This trace can then be used to calculate the corresponding shedding frequency.



Flow field

We set the output frequency to 5 when running the case and so the solver will have written visualisation VTK files every 5 real time steps. If you look in the '*cylinder_P1_OUTPUT/VISUALISATION*' directory you will see a series of .vti and .pvti files, one for every written timestep.

You could load these individually but the easier method is to load the '*cylinder.pvd*' in '*cylinder_P1_OUTPUT*'. This .pvd file links to all of the timesteps.

Launch Paraview and load '*cylinder_P1_OUTPUT/cylinder.pvd*'. Paraview will render a single solution time step at a time, and time series data can be navigated using the icons at the top of the screen:



Animation

For unsteady data it is often useful to output an animation of the solution. This can be done in Paraview by either exporting an image per timestep and then using a third party tool to stitch the images into a video or by saving the animation directly. The advantage of the first approach is that it can easily be scripted and is more suitable for large cases. Since this case is quite small we will export an mp4 video directly from Paraview. The steps to do this are:

1. Launch Paraview and load '*cylinder_P1_OUTPUT/cylinder.pvd*'.
2. Set the view up to shade the output by "cp" and zoom/move the image until you are happy with the view:



3. Select 'File->Save Animation' in Paraview. To export as video select your preferred format, the format will depend on your Operating System but for Windows you should be able to select "MP4 files" and on Linux "FFMPEG AVI". Give the file a name and click OK.
4. On the next screen you can set the target resolution, frame rate and the timesteps to export. Set the Image resolution to 1920x1080 (FHD) and the frame rate to 24. If you just want to export the end of the simulation (where the oscillation is established) you can set the Frame Window from 300 to 600.
5. Click OK to export the animation. This will take some time.

You should now have an exported video of the unsteady simulation.

1.4.4 Tutorial 3a: FWH Solver

A *Ffowcs-Williams Hawkings (FWH) solver*, which is used for prediction of farfield noise, is included in the zCFD distribution. In this section, we will use the zCFD FWH solver to calculate the sound experienced by two 'observer microphones' as a result of the aerodynamic noise generated by the flow around the cylinder.

In order to use an FWH solver to predict noise at observer microphones, we will first rerun the CFD simulation from [Tutorial 3](#) with some additional terms to record the flow variables on an 'FWH surface' in the near-field, noise generating region. We then use the FWH solver to propagate the noise from the near-field FWH surface to the observer microphones. For more detail on the FWH solver, see [here](#).

Running zCFD with FWH output

The flow we're going to be running the FWH solver on is the shedding cylinder. To get the required inputs to our FWH solver, we need to define an *FWH surface*, and collect data on this surface during a CFD simulation.

The files required for the fwh tutorial can be downloaded [here](#). In the zip file there's a FWH surface definition file, called *fwh_surface1.stl*. Given we've already run the cylinder case in [Tutorial 3](#), we can view the FWH surface in Paraview alongside the cylinder CFD volume data (see [here](#)), as below. You may find the surface easier to see if you set the *view style* to be *Feature Edges* instead of *Surface* for the FWH surface, and increase the *Line Width* in the *Geometry Representation*.



The FWH surface encloses the noise generating near-field region, but does not intersect the wake, in order to avoid spurious pseudo-sound p[[1](#), [2](#)].

There's also a prepared control dictionary, *cylinder_fwh.py*. This is identical to tutorials 3's *cylinder.py* apart from the output section, which is detailed below.

```
"write output": {
    "format": "vtk",
    "surface variables": ["V", "p", "T", "rho", "cp"],
    "volume variables": ["V", "p", "T", "rho", "m", "cp"],
    "fwh interpolate": ["fwh_surface1.stl"],
    "frequency": {
        "fwh interpolate": 1,
        "volume data": 1000,
        "surface data": 1000,
    },
},
```

In this case, we have chosen to output *fwh interpolate* data every single timestep, using the *fwh_surface1.stl* FWH surface definition file. Volume and surface data is much less frequent, since we already have all this data from when we ran *cylinder.py* [above](#).

For the FWH tutorial we have also updated the '*time marching*' section to run the simulation for 4 seconds, rather than 1.2. We run the CFD simulation in the normal way, using the command below.

```
run_zcfd -m cylinder.h5 -c cylinder_fwh.py
```

When we've run the CFD simulation of the cylinder, we should be able to see the FWH interpolated data in '*cylinder_fwh_P1_OUTPUT/ACOUSTIC_DATA/fwh_surface1_FWHData.h5*'.

FWH Observer and surface motions

We're going to use the FWH solver to calculate noise at two observer microphones - '*Obs1*' and '*Obs2*' (see [Fig 1](#)). '*Obs1*' is an observer that is a fixed distance away from the cylinder - if we consider the cylinder in a wind tunnel, we could consider '*Obs1*' as a microphone attached to the wind tunnel wall. '*Obs2*' is different, and actually flies through the air at the same speed as the freestream air.



Fig. 1: FWH Observer and surface motions - CFD frame of reference

When the FWH equations are derived, we make an important assumption - that of a 'quiescent medium' (i.e. still air, with a free-stream velocity of zero). Clearly, with a freestream velocity of $(0, 66.8, 0)$, the flow in our CFD simulation does not satisfy this condition. In order to use the FWH solver, we therefore have to perform a co-ordinate transformation, as in [Fig 2](#). In this 'quiescent medium' co-ordinate system, the cylinder and '*Obs1*' are 'flying' through the air. In this co-ordinate system, we can also see that '*Obs2*' represents a microphone fixed to the ground, as the cylinder flies overhead.

Setting up the FWH solver inputs

Now that we know the motions of our observers and surface, we can run the FWH solver. The python code required to run the FWH solver is in the file '*run_fwh_solver.py*', and also written out below. We define the surfaces, observers and the solver settings, then use `fwh.solve()` to actually run the FWH solver.

```
from solvers import fwh, motion
import json
import math

# FWH surface motion
c = math.sqrt(1.4 * 277.7777 * 287)
v_surf = [0, -0.2 * c, 0] # = [0,-66.8,0]
t0 = 0.002 #t at first FWH data output
surfaceCentrePoint = motion.ConstantVelocityPoint([0.0, 0.0, 0.0], v_surf)
surfaceMotion = motion.NonrotatingSurface(surfaceCentrePoint)

# obs1 motion - [0,0,20] above cylinder centre
obs1_CFD_frame_posn = [0.25,0,20]
obs1_0time_posn = []
```

(continues on next page)



Fig. 2: FWH Observer and surface motions - FWH (quiescent medium) frame of reference

(continued from previous page)

```

for i in range(3):
    obs1_0time_posn.append( obs1_CFD_frame_posn[i] - t0 * v_surf[i] )
obs1Motion = motion.ConstantVelocityPoint(obs1_0time_posn, v_surf)

# obs2 motion - stationary, intersects with obs1 at t=2.5
obs2_posn = []
for i in range(3):
    obs2_posn.append( obs1_0time_posn[i] + 2.5* v_surf[i])
obs2Motion = motion.StationaryPoint(obs2_posn)

solverSettings = {
    "c": c,
    "dt": 0.002,
    "rho0": 101325.0 / (277.7777 * 287),
    "p0": 101325.0,
}

surfaces = {
    "surf1": {
        "motion": surfaceMotion,
        "fileName": "./cylinder_fwh_P1_OUTPUT/ACOUSTIC_DATA/fwh_surface1_FWHData.h5",
    }
}
observers = {"Obs1": obs1Motion, "Obs2": obs2Motion}

pOut, tOut = fwh.solve(surfaces, observers, solverSettings)

dataForJson = {"p": pOut, "t": tOut}

with open("./FWH_data.json", "w") as f:
    json.dump(dataForJson,f)

```

More details on FWH solver setup can be found [here](#), but the most important concept is that we use '*motion*

classes' to define observer and surface motions. The available *observer motions* are '*OriginPoint*', '*StationaryPoint*' and '*ConstantVelocityPoint*'. The available *surface motions* (which rely on an *observer motion* to define the motion of their centre) are '*NonrotatingSurface*' and '*RotatingSurface*'.

The velocities and fluid properties used in the FWH simulation are taken from the '*cylinder.py*' zCFD control file. `solverSettings['dt']` has been set to be the same as the timestep between outputs in the FWH surface data (0.002 seconds, see [above](#)).

In general, we can consider the motion of a point on an FWH surface to be defined by $\mathbf{x}(t) = \mathbf{x}_0 + \mathbf{V}_{surf}(t - t_0)$ in the FWH co-ordinate frame. \mathbf{V}_{surf} is defined by the user in the *surface motion class* (called *surfaceMotion* above), \mathbf{x}_0 is the position defined by the FWH surface definition file ('*fwf_surface1.stl*', see [above](#)), and t_0 is the simulation time at which the solver first outputted the FWH data. In our case, $t_0 = 0.002$ seconds (see [Output](#)).

The centre of the cylinder in the CFD mesh is at (0.25,0,0), and we want *Obs1* to always be (0,0,20) metres away from the cylinder centre during the FWH simulation. Since the cylinder 'travels with' the FWH surface (see [Fig. 2](#)), we can therefore say that the motion of the cylinder centre in the FWH co-ordinate frame is $\mathbf{x}(t) = (0.25, 0, 0) + \mathbf{V}_{surf}(t - t_0) = ((0.25, 0, 0) - 0.002 \mathbf{V}_{surf}) + \mathbf{V}_{surf} t$. We therefore set the motion of *Obs1* to be $\mathbf{x}(t) = ((0.25, 0, 20) - 0.002 \mathbf{V}_{surf}) + \mathbf{V}_{surf} t$. We set *Obs2* to be stationary in the FWH reference frame, and to be coincident with *Obs1* at $t=2.5$. We save the outputs to a json file called *FWH_data.json*.

Running the FWH solver

The python code above is replicated in '*run_fwh_solver.py*'. To run the FWH solver, we simply run the command below from within the [zCFD command-line environment](#). You may need to change the `surfaces["surf1"]["fileName"]` parameter depending on how many ranks you used in your zCFD run.

```
python3 run_fwh_solver.py
```

This will output various information to the screen, including the 'valid min and max times' for each surface / observer combination.

The valid min and max times denote the time window during which an observer receives data from all points on the FWH surface. A pressure signal emitted from a point on the FWH surface at time t takes r/c seconds to arrive at an observer, where r is the distance from the surface point to observer and c is the speed of sound. Therefore the valid observer time window will always be later than the FWH surface output time window (in our case, 0.002:1.2 seconds) due to the distance from the FWH surface to the observer. The valid observer time window will be narrower if the FWH surface is large, since the larger the surface the larger the time delay between signals from different points on the surface reaching the observer becomes.

Post-processing

The [jupyter notebook](#) `postprocess_fwh_data.ipynb` from the zip file can be used to post-process the FWH data.

Obs1 moves through the FWH medium at a constant displacement from the cylinder centre (see [Fig 1](#)). We see from our post-processing that the noise experienced by *Obs1* is dominated by a constant 10 Hz component, which is the shedding frequency of the cylinder. Various harmonics of the 10 Hz signal are present in the power spectral density plot for *Obs1*.





During the FWH simulation, the cylinder gets closer to *Obs2* from 0 to 2.5 seconds, before getting further away from *Obs2* from 2.5 seconds onwards. We can see in the *Obs2 p(t)* plot that the peaks are closer together from 0 to 2.5 seconds than they are after 2.5 seconds. This is due to Doppler shift. We can also see the effect of Doppler shift in the power spectral density, which has a more rounded 10 Hz peak than that seen in *Obs1*.



postprocess_fwh_data.ipynb also creates *.wav* files, which allow you to listen to the sound experienced by *Obs1* and *Obs2*.

Extensions

To become more comfortable with the FWH solver, various extensions to this exercise are possible. Here are some suggestions.

1. Re-run the zCFD solver, but this time also extract FWH data on the cylinder *wall*. Run the FWH solver (using the wall data as an *impermeable* FWH surface) and compare the results to those achieved using the *fwh_surface1* permeable FWH surface.
2. Re-run the zCFD solver, but this time also extract FWH data on the *fwh_surface2.stl* and *fwh_surface3.stl* surfaces. These are like *fwh_surface1.stl* but slightly larger. Run the FWH solver with these surfaces and investigate the effect of using different surfaces on the FWH results.
3. Add extra observers of your choice to the FWH solver.

Note: Since the cylinder CFD simulation is ‘2.5D’ (i.e. the mesh only has a single cell in the *x* direction), the distance scaling of *p(t)* at observers will be non-physical.

Citations

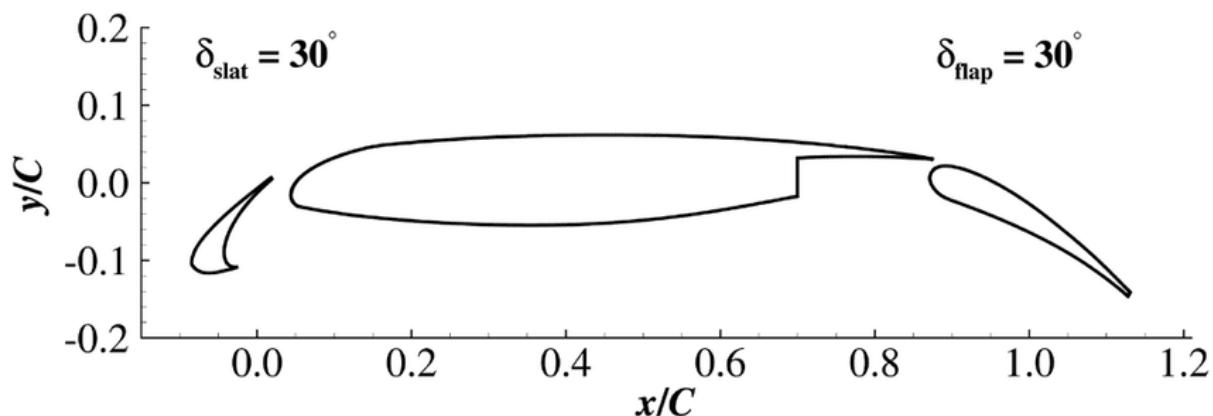
1. Mitchell, B., Lele, S., & Moin, P. (1999). Direct computation of the sound generated by vortex pairing in an axisymmetric jet. *Journal of Fluid Mechanics*, 383, 113-142. 10.1017/S0022112099003869
2. Ricciardi, T. R. , Wolf W. R, & Spalart P. R. (2022). On the Application of Incomplete Ffowcs Williams and Hawkings Surfaces for Aeroacoustic Predictions. *AIAA Journal*, 60:3, 1971-1977. 10.2514/1.J061285

1.4.5 Tutorial 4: Aero-acoustics

Table of Contents

- *Aero-acoustic mesh generation*
- *Control dictionary configuration for aero-acoustics*
- *Running the Acoustic Solver*
- *Post-Processing*

In *Tutorial 2*, we learnt how to run a steady state RANS simulation for the flow over the 2D 30P30N multi-element aerofoil geometry. In this tutorial, we will use the steady state RANS simulation to perform a broadband aero-acoustic simulation using the Fast Random Particle Mesh (FRPM) stochastic noise method. The geometry of this case is as in Tutorial 2 and can be seen below:



As we have already generated the full steady state RANS solution in Tutorial 2, we will begin by creating our aero-acoustic mesh, on which we will solve the *Acoustic Perturbation Equations* using a high order DG method. To start, we will load the wall boundary solution into Paraview and export it in STL format.

Aero-acoustic mesh generation

To generate the aero-acoustic mesh we will do the following:

1. Use Paraview to extract the wall boundary from the zCFD result generated in [tutorial 2](#) and export it as STL.
2. Generate an acoustic mesh based on the wall STL using [zM3](#).

1. Export STL with Paraview

Firstly, open the file in Paraview. The procedure to load the file is:

File > Open

then select the **30p30n_steady_wall.pvd** file in the OUTPUT directory from the steady state zCFD simulation you ran in [tutorial 2](#). In the ‘Properties’ dialog box in the bottom left of the screen, deselect any field variables as these will not be required. As we will be saving the data in STL format, we must triangulate the boundary. To do this select:

Filters > Alphabetical > Triangulate

Finally, we need to save the file in STL format, to do this:

File > Save Data

and name the file as *wall.stl* (change file to *STL*). Save the file as Binary format.

2. Create mesh with zM3

To generate the acoustic mesh we run the [zM3](#) mesh generator. zM3 will create a cartesian mesh and embed the STL file to represent the geometry and as a source of refinement. As this mesh generator is currently command line based, the commands to generate a suitable acoustic mesh is as follows:

```
# Source the zCFD environment
source <PATH_TO_ZCFD>/bin/activate

zm3 \
--meshName acoustic.h5 \
--base -2 -2 0.5 \
--dims 4 4 0.005 \
```

(continues on next page)

(continued from previous page)

```
--globalSpacing 0.005 \
--stlSpacing 0.000625 \
--seedPoints -1 -1 0.50001 \
--stlFiles wall.stl \
--rotatedTranslatedRefinement 0.15 0.075 0.01 0 0 30 -0.035 -0.085 0.5 0.000625 0.
-00625
```

The options specified (shown in more detail [here](#)) are as follows:

| Option | Description |
|---|---|
| --meshName acoustic.h5 | The name to be used for the outputted mesh file |
| --base -2 -2 0.5 | This is the minimum x,y,z coordinates that the Cartesian domain will start from |
| --dims 4 4 0.005 | Cartesian x,y,z dimensions for the full domain. In this case, this means that the domain will extend from (-2,-2,0.5) to (2,2,0.505) |
| --globalSpacing 0.005 | This is the maximum Cartesian spacing within the domain |
| --stlSpacing 0.000625 | This is the maximum spacing that will occur wherever the STL intersects the Cartesian cells |
| --stlFiles wall.stl | The STL representation of the geometry |
| --rotatedTranslatedRefinement 0.15 0.075 0.01 0 0 30 -0.035 -0.085 0.5 0.000625 0.00625 | <p>These parameters are similar to those used to specify the FRPM domain placement and are ordered as follows:</p> <ol style="list-style-type: none"> 1. x,y,z dimensions of FRPM domain (here (0.15,0.075,0.01)) 2. rotation vector of refinement box around origin in degrees (here (0,0,-30)) 3. post-rotation translation of refinement box (here (-0.035,-0.085,0.5)) 4. cell size within refinement region (here 0.000625) 5. resolution of refinement regions (typically 10x the spacing, here 0.00625) |

Once you have generated the mesh you can move onto setting up the control dictionary for the acoustics solver.

Control dictionary configuration for aero-acoustics

The acoustic solver is run in a similar fashion to the CFD solvers, where a Python control dictionary is supplied containing the relevant solver parameters. The flow conditions supplied are identical to those used for the RANS simulation in Tutorial 2. Additional sections are described below.

A copy of the acoustics control dictionary can be downloaded [here](#).

Nodal Locations

As we are using the high order DG solver to solve the Acoustic Perturbation Equations (APE-4), we can specify the nodal locations to be used when constructing our Lagrange polynomials within each element. In the current case where the equations are linear and ambient flow conditions are assumed for the acoustic medium, the main effect of this choice is where acoustic sources are sampled. In the current case we're using Hexahedral elements and assuming Gauss Lobatto nodal locations, which means that no two nodal locations are the same.

```
"Nodal Locations": {
    "Line": line_gauss_lobatto,
    "Tetrahedron": tet_shunn_ham,
    "Tri": tri_shunn_ham,
},
```

Sponge Layer Damping

In order to absorb acoustic waves and minimise reflections off farfield boundaries, we use a Sponge Layer formulation where we apply damping source terms in regions approaching the farfield. The values chosen have been selected to minimise reflections from the start of the sponge layer and the farfield. The damping source terms increase from zero at a distance of 0.25m from the farfield, to their full values in the cells adjacent to the farfield boundary.

```
"Sponge Layer Damping": {
    "Distance": 0.25, # Should be sufficiently large to prevent reflections from the
    -sponge layer
    "Damping Factor": 2.0, # Typically of order 1 to 10
},
```

DGCAA

In order to run the CAA high order solver, we simply set `equations` to `dgcaa`. We can then specify the parameters for the acoustic solver. To start with we must select the time marching options. To do this we use the following Python dictionary within our solver setup dictionary.

```
"time marching": {
    "unsteady": {
        "total time": 0.05,
    },
    "multi level": True,
    "cfl": 0.016,
},
```

Setting `multi_level` to `True` allows the use of a Low Dissipation, Low Dispersion 3rd order accurate multi-level time marching scheme. The total time should be set according to the frequency resolution of interest and ensuring there will be sufficient sample time once the signal has reached the observer. Following on from the time marching options, we now set the high order CAA solver options.

```
"DGCAA": {
    "order": 1,
    "map cfd to caa mesh": False,
    "rans mesh name": "30p30n_coarse.h5",
    "rans case name": "30p30n_steady.py",
    "frpm sources": True,
    "microphone output frequency": 10,
}
```

With these options, linear polynomials will be assumed within each cell of the acoustic mesh (there will be 8 solution points within each cell for order 1, 27 for order 2, and so on) making the spatial accuracy second order which will be sufficient for this test case. The RANS mesh and case name are those of the RANS solution the FRPM acoustic simulation should be based on. We assume a uniform background flow by not mapping the CFD solution to the CAA mesh. Finally, we will measure the acoustic pressure every 10 time steps by setting `microphone output frequency` to 10.

To set the FRPM parameters we use the following block of Python code.

```
"DGCAA": {
    "FRPM_1": {
        "map cfd to frpm mesh": True,
        "number of frpm mpi processes": 1,
        "frpm spacing": 0.0025,
        "frpm turbulence integral length scale": 0.005,
        "frpm cart num mesh cells": [int(0.15 / 0.0025), int(0.075 / 0.0025), 1],
        "frpm march frequency": -1,
        "frpm domain translate": [-0.035, -0.085, 0.5],
        "frpm domain rotate (Deg)": [0.0, 0.0, 30.0],
        "frpm blend sources from side": [ 0.01875, 5.0 * 0.01875, 0.01875, 0.01875, 0.0, 0.0],
        "frpm mapped tke smoothing iterations": 5,
        "use constant integral length scale": False,
    },
},
```

With these options, the RANS solution generated in tutorial 2 will be used to specify the turbulence parameters for the FRPM source generation. As the turbulence kinetic energy field is differentiated to provide the acoustic sources, it may be necessary to smooth the field to prevent spurious sources. This is achieved with the `frpm mapped tke smoothing iterations` keyword. The turbulence integral length scales will be computed automatically from the RANS solution if the `use constant integral length scale` keyword is set to `False` (the `frpm turbulence integral length scale` value will be ignored in this case). 1 MPI rank will be used for the FRPM source generator. The FRPM domain size, position and orientation is specified with the `frpm cart num mesh cells`, `frpm domain translate`, and `frpm domain rotate (Deg)` keywords respectively. It should be noted that the FRPM domain is first rotated about (0,0,0), then translated to the specified location. In order to prevent acoustic sources vanishing instantly at the boundary of the FRPM domain, sources are blended smoothly over the distances specified by `frpm blend sources from side`. The ordering is distance from the x-min, x-max, y-min, y-max, z-min, z-max boundaries.

Running the Acoustic Solver

Within a Linux terminal, navigate to the directory containing the `acoustic.h5` mesh file and `.py` control dictionary. To run the acoustic solver, use the `run_zcfd` command in the terminal as before in previous tutorials but instead using the acoustics mesh and control dictionary as input files:

```
run_zcfd -p acoustic.h5 -c acoustic.py
```

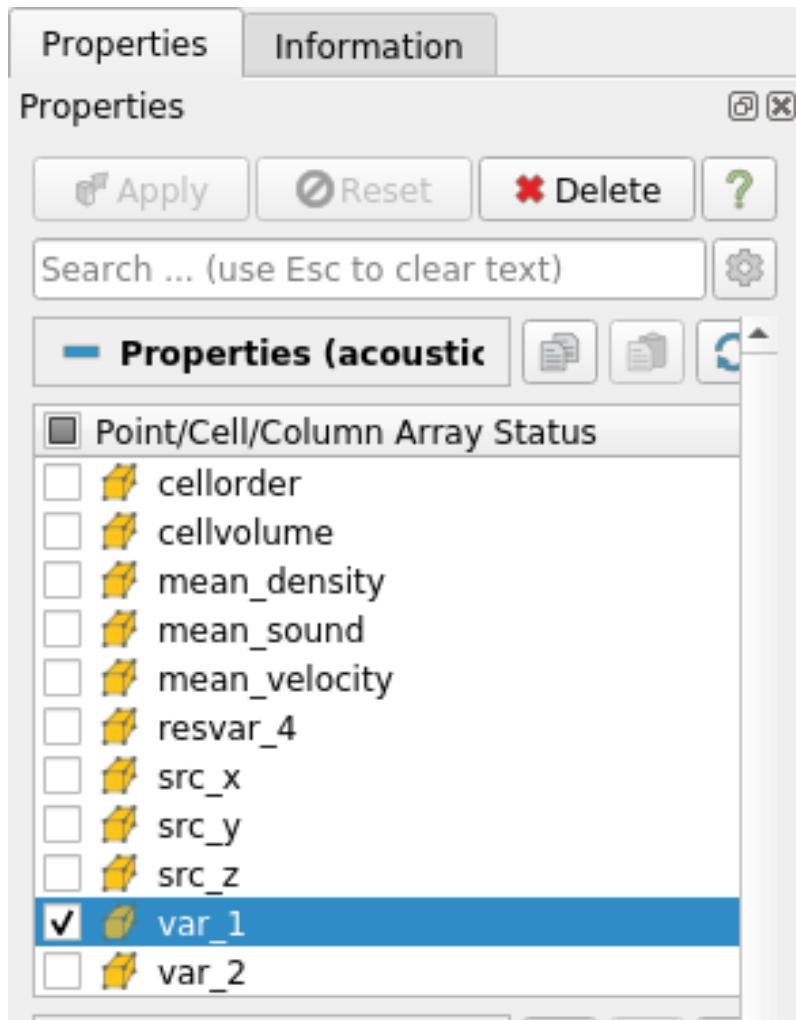
Note: you must ensure that the input files used for the steady RANS simulation (tutorial 2) and the `.h5` results file are also in the same directory before running the acoustics solver. Alternatively, you can specify the full path to the files in the `acoustics.py` python dictionary.

Post-Processing

The acoustics solver can take some time to run, if you want to look at pre-generated results then they can be downloaded [here](#) (1.5GB).

Paraview

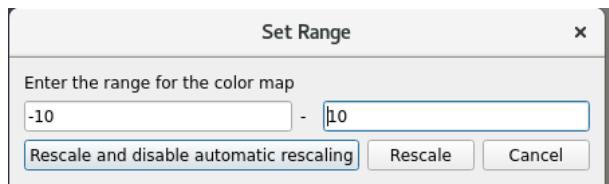
To conduct post-processing of the aeroacoustic simulation, open Paraview and load the .pvд generated in the OUTPUT folder. Since we are only interested in the acoustics, it may be useful to only have ‘var_1’ selected in the ‘Point/Cell/Column Array Status’ options as this corresponds to acoustic pressure.



To visualise the acoustics more clearly, click the ‘Rescale to Custom Data Range’ icon in the top left of the Paraview window:



and change to values -10 to 10 before clicking ‘Rescale and disable automatic rescaling’:



You can now zoom in towards the aerofoil slat region to clearly view the acoustics.



Zoomed in view:



Performing Spectral Analysis

To perform spectral analysis, ensure that you have the acoustic.ipynb file in the working directory. Launch a Jupyter server and load the acoustic.ipynb file. Once opened, run all the cells and the following plot should be produced of Power Spectral Density (PSD).



The acoustic.ipynb uses Pandas to read in the acoustic pressure data within the _MP2 file (this file contains the acoustic pressure history at monitor point 2 as specified in the Python dictionary). This signal is then checked to find where the acoustic waves meet the observer and is appropriately clipped, removing samples where there were no acoustic waves. The time signal of acoustic pressure is then processed using SciPy, and in particular, Welch's method. Finally the PSD is plotted using Matplotlib. All modules are available within the zCFD environment.

1.4.6 Running zCFD using EPIC

Table of Contents

- *What is in this tutorial?*
- *What you need?*
- *Step 1. Download the tutorial files*
- *Step 2. Upload the files to EPIC*
- *Step 3. Submit a zCFD job*
- *Step 4. Monitor the job*

- *Step 5. Look at the results*

What is in this tutorial?

This tutorial will show how to run zCFD using EPIC. EPIC provides access to a range of High Performance Computing systems allowing you to run your zCFD jobs at a large scale. This enables you to run your simulations faster or do larger simulations. EPIC also enables you to start EPIC Desktops please see [here](#), that are Linux desktop nodes running in the cloud and are great for pre or post processing simulations or even running zCFD jobs.

What you need?

You will need an account on EPIC for this tutorial, you can sign up for free [here](#). For this tutorial we will be running the 30p30n case used in [tutorial 2](#). We will use the on-demand licence version of zCFD and so there is no need to have a pre-existing zCFD licence.

Step 1. Download the tutorial files

The tutorial 2 zip file can be downloaded [here](#). The zip file contains a mesh file (*30p30n_coarse.h5*) and a zCFD control file (*30p30n_steady.py*).

Download and unzip the file to your local computer.

Step 2. Upload the files to EPIC

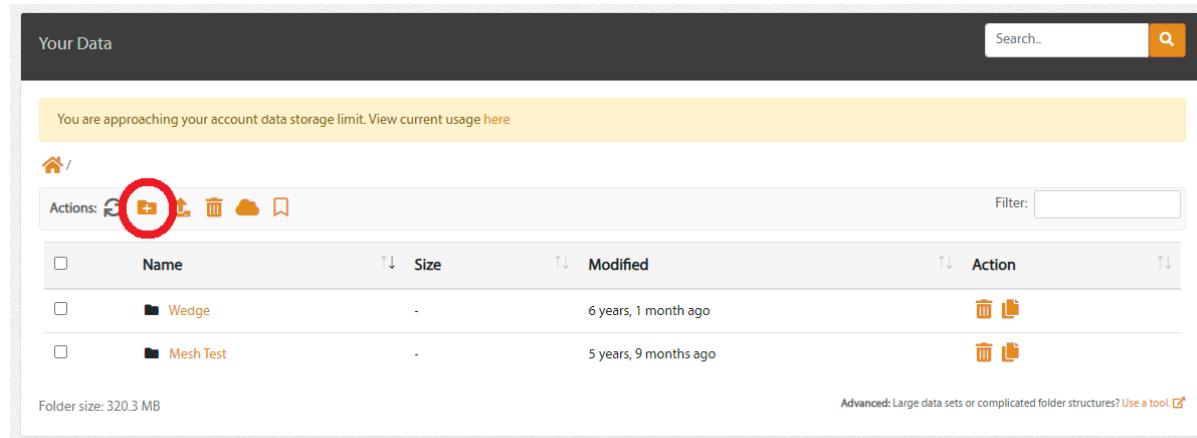
Files can be uploaded to EPIC in a variety of ways:

- Via the web portal
- Using the EPIC command line interface, [EPIC CLI](#).
- Using Python and the [PyEPIC](#) library

Since the input files are small in our case we will upload them via the browser, if the files were larger than 500MB then the other methods are recommended.

To upload the files, log in to EPIC and the select Data->Manage on the left hand menu bar. This will show the [Your Data](#) view. This lists all of the data you have stored in EPIC.

Click on the “Add folder” button, shown below, to create a new folder, and name it “30p30n”.



The screenshot shows the EPIC 'Your Data' interface. At the top, there's a dark header with 'Your Data' and a search bar. Below it is a yellow banner stating 'You are approaching your account data storage limit. View current usage here'. The main area has a table with columns: Name, Size, Modified, and Action. Two files are listed: 'Wedge' and 'Mesh Test'. At the top of the table, there's an 'Actions' row with icons for creating a folder (highlighted with a red circle), deleting, renaming, and copying. A 'Filter:' input field is also in this row. At the bottom, it says 'Folder size: 320.3 MB' and 'Advanced: Large data sets or complicated folder structures? Use a tool.'

This will create a new folder and change your view to show you the contents, it should currently be empty. To upload some data click on the “Upload Files” button, shown below.

You are approaching your account data storage limit. View current usage [here](#)

Actions:

Filter:

| <input type="checkbox"/> | Name | Size | Modified | Action |
|--------------------------|--------|------|----------|--------|
| | 30p30n | | | |

No data? Lets upload some files!

Use the Actions menu above to manage your data. You also can upload as many as you need and create folders to keep things nicely organised.

Folder size: 0 Bytes Advanced: Large data sets or complicated folder structures? [Use a tool](#)

Drag and drop the mesh and control files from your local computer into the dialog box that shows, this will trigger the upload from your local machine to the EPIC data store. Once the upload is complete the dialog should look like this:

Upload files

| | | |
|------------------|---------|--|
| 30p30n_coarse.h5 | 15.4 MB | |
| 30p30n_steady.py | 3.2 KB | |

Note: Uploading whole directories is currently only possible when using Chrome

Upload progress:

Done

| <input type="checkbox"/> | Name | Size | Modified | Action |
|--------------------------|------|------|----------|--------|
|--------------------------|------|------|----------|--------|

Click “Done” and you will go back to the data view, you should now be able to see your input files in EPIC.

Step 3. Submit a zCFD job

The next step is to create a new zCFD job. Click on “New Job” on the left hand menu bar. This will take you to the application selection screen. Select the most recent version of zCFD from the options available.

On the next screen select the folder containing the data you have just uploaded by checking the checkbox next to it:

New ZCFD Job

The wizard below will guide you through the process of launching an ZCFD job via EPIC.

1.Job 2.Application 3.Cluster

1 Staging Your Data

EPIC will take care of staging the data to the HPC cluster for you. To do this it synchronises data from "Your Data" to the remote cluster. Below please select the input data ZCFD (2023.03.3383) requires on the remote cluster, EPIC will recreate the selected directory structure exactly as you see it below on the remote cluster.

1. Job Data

Please select the folder that contains your job data. This will be copied to the remote cluster when you submit the job and deleted when the job is complete, the directory structure will replicated on the remote cluster.

2. Job Name

Please enter a unique Job Name so you can easily identify it in the future. The optional invoice reference will appear next to the job name on your EPIC invoices.

Job Name: *
30p30n

Invoice reference:
Project XYZ...

This tells EPIC where the job inputs are. You can leave the rest of the settings as they are, or update the job name if you want to.

When you click “Next” EPIC will start the zCFD Input Checker, this validates the input folder you have selected and presents you with zCFD specific choices.

Here you can set the python control dictionary, the mesh and set the number of cycles to run. You can also change the number of parallel processes to use when executing zCFD.

For our example update the cycles to 2000 and the number of parallel processes to 4. The fields should now look like this:

ZCFD Input Checker

- ZCFD check complete
- HDF5 (.h5) mesh files found
- Python control files found
- No unsafe content found in Python control files

Note: The input check performs some basic checks on your zCFD case, it does not guarantee that your case is correctly defined.

What is the zCFD Case Name?

The case name is the python file control file (*.py) - you do not need to include the file extension.

30p30n_steady.py

What is the zCFD Problem Name?

The problem name is the HDF mesh file (*.h5).

30p30n_coarse.h5

Load problem and mesh file locations from override file?

Override control file name?

The override file can set a number of meshes and control files for making use of zCFDs overset capability.

Number of time cycles?

For steady-state simulations, the number time cycles is the same as the number of steps that the solver should use to reach a converged solution.

2000

Restart zCFD from previous run?

How many parallel processes should zCFD use? ⓘ

4

Licence:

On demand licence

You can save licence information for a team, [here](#)

Mesh information

The mesh you have selected contains 63957 cells. best practice for zCFD is to run a maximum of 1m cells per 2GB of system/GPU

You will then see the cluster selection page, this lists all of the high performance computing resources available to run your job. On the previous page asked for 4 parallel processes, on the Cluster page we can say how we want these distributed and what the maximum runtime of our job will be. For zCFD lets leave the “**Tasks per node**” as “1 per device”, this will put one parallel process per CPU socket or GPU. Our job is quite small and should complete easily within 1 hour, so we can leave the Maximum Runtime as 1.

Job Size

Your jobs will be submitted as a single job to the cluster. You can specify the size and maximum runtime below.

| Name | Tasks | Tasks per node | Maximum Runtime (hours) |
|-------------|-------|----------------|-------------------------|
| ZCFD Solver | 4 | 1 per device | 1 |

Filters

CPU Cores per Node:4
Node Memory (GB):16 GB

Cluster Choice

Show 25 entries Search:

| Name | Status | Cores per Node | Memory per Node | Total Cores | Queuing time | Node Price | Job Price | Select |
|----------------------|--------|----------------|-----------------|-------------|--------------|------------|-----------|-------------------------|
| C5 - 18 Core | ✓ | 18 | 72.0 | 900 | < 5 mins | £1.26 | £8.24 | <button>Select</button> |
| C5 - 18 Core - Spot | ✓ | 18 | 72.0 | 900 | < 5 mins | £0.60 | £6.34 | <button>Select</button> |
| C5 - 36 Core | ✓ | 36 | 144.0 | 1800 | < 5 mins | £3.20 | £13.82 | <button>Select</button> |
| C5N - 36 Core - Spot | ✓ | 36 | 144.0 | 1800 | < 5 mins | £1.10 | £7.78 | <button>Select</button> |
| CFMS | ✓ | 24 | 128.0 | 2640 | < 30 mins | £1.44 | £8.76 | <button>Select</button> |
| CFMS - GPU | ✓ | 16 | 64.0 | 128 | < 5 mins | £2.02 | £7.52 | <button>Select</button> |

For our example we will select the “CFMS - GPU” cluster from the list by clicking “Select” next to it. EPIC will then show us the cost breakdown for the job, click “Submit Job” to continue.

Your job will now be submitted to the selected cluster. EPIC will transfer the data, submit the job to the cluster and monitor the progress.

Active Jobs Auto-refresh

Actions: Job states: 6 items selected Days ago: 5 Show 10 entries Search:

| Name | Application | Submitted by | Submitted | Status | Action |
|--------------------|---------------------|--------------|----------------------|-------------------------------|--------|
| Single jobs | | | | | |
| 30p30n | ZCFD (2023.03.3383) | Mike Turner | 11th Oct 2023, 14:36 | Pending submission to cluster | |

Showing 1 to 1 of 1 entries Previous **1** Next

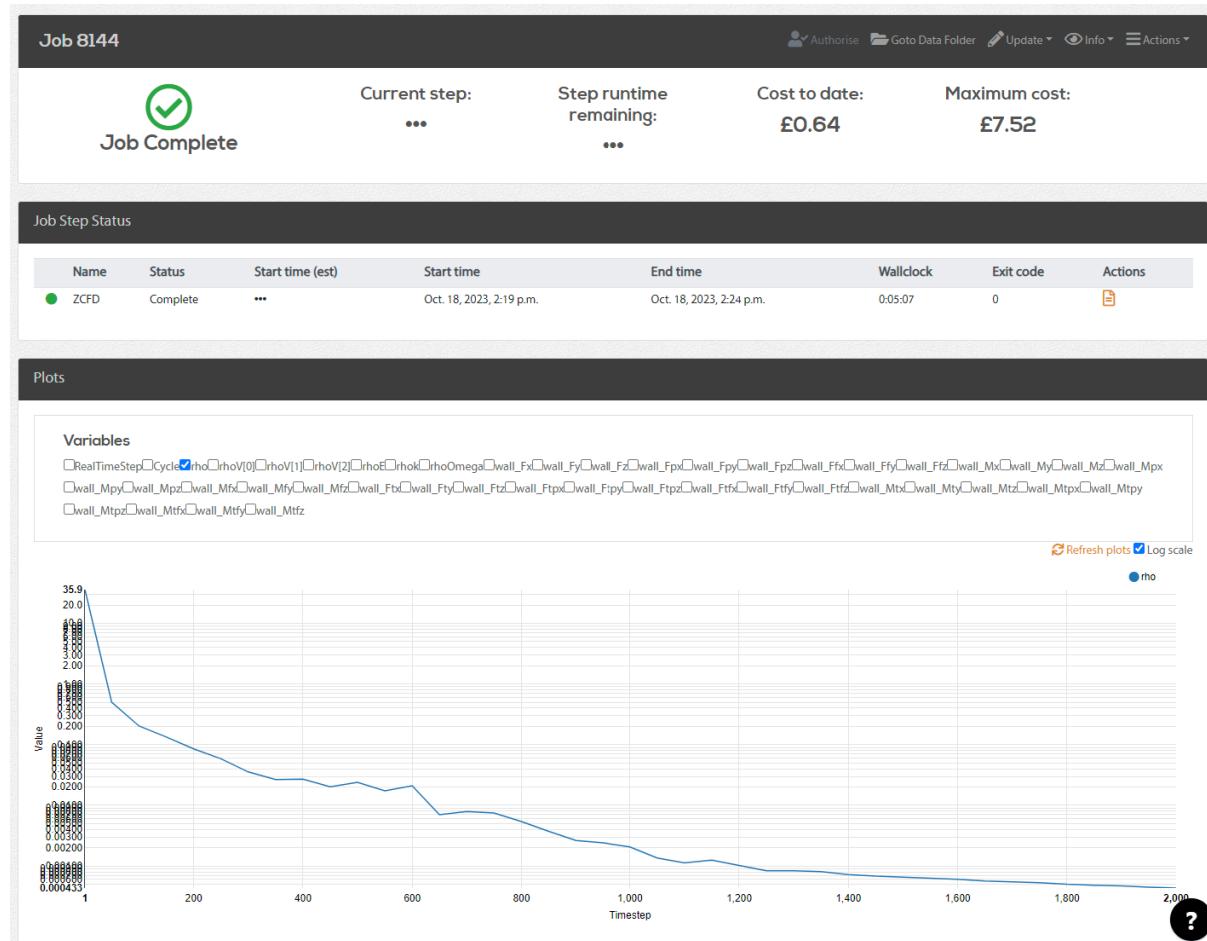
Step 4. Monitor the job

You can see your active and recent jobs by clicking on “**Active Jobs**” on the left hand side bar. Click on the “>>” next to the job to see the detailed job status.

Here you can see the current status of the job, view the job logs, view the residual plots of the job while it is running and cancel or update the job.



Step 5. Look at the results



Once the job is complete EPIC will upload the results from the cluster back to EPIC, you can click on the “**Goto data folder**” link at the top of the job status page to view the files and download them to your local machine using the browser.

An alternative method is to start an EPIC desktop node by clicking on “**New Desktop**” on the left hand tool bar. For more information about using the Desktop see [here](#).

1.5 Reference Guide

The zCFD control file is a python file that is executed at runtime. This provides a flexible way of specifying key parameters, enabling user defined functions and leveraging the rich array of python libraries available.

A python dictionary called ‘parameters’ provides the main interface to controlling zCFD

```
parameters = {....}
```

zCFD will validate the parameters dictionary specified at run time against a set of expected values for each key. In general, values that should be integers and floats will be coerced to the correct type. Values which should be strings, booleans, lists or dictionaries will cause an error if they are the incorrect type. The time marching scheme and solver/equation set chosen defines which dictionary keys are valid. A simple script is provided to allow the user to check the validity of their input parameters before submitting their job. See [Input Validation](#).

This reference guide documents the parameters available in a zCFD control dictionary and the valid values for those keywords.

```
parameters = {
```

```

'reference': IC_1,
'initial': {Initial State},
'restart': False,
'partitioner': 'metis',
'scale': [1,1,1],
'time marching': {Time Marching},
'linear solver settings': {Linear Solver Settings},
'equations': {Equations},
'material': {Material Specification},
'IC_1': {Initial Conditions}
'BC_1': {Boundary Conditions}
'TR_1': {Mesh Transformation}
'mapping': {Overset}
'write output': {Write Output}
'report': {Reporting}
}

```

1.5.1 Reference Settings

Reference state

| Keyword | Re- quired | Default | Valid values |
|-------------|---------------|---------|-------------------------------|
| 'reference' | No | 'IC_1' | "IC_?" where ? is an integer. |

```
# set reference state to IC_1
'reference' : 'IC_1',
```

Sets the *Initial Condition* which provides the reference quantities when non-dimensionalisation is applied to *force reporting*. "IC_?" must be a valid *initial conditions* dictionary defined in the parameters file

Initial State

| Keyword | Re- quired | Default | Valid values |
|---------|---------------|---------|--|
| 'name' | No | 'IC_1' | "IC_?" where ? is an integer. |
| 'func' | No | - | An (optional) <i>initial conditions function</i> . |

```
# set initial state to IC_1
"initial": {
    "name": "IC_1"
}
```

Sets which block *Initial Condition* block will be used to provide initial flow field conditions for the simulation. If 'func' is provided it will be used to provide the initial flow field variables on a cell-by-cell basis.

Note: If the 'initial' entry is missing then the conditions default to the *reference* state.

Scale mesh

| Keyword | Re- quired | Default | Valid values |
|---------|---------------|---------|--|
| 'scale' | No | [1,1,1] | [scaleX,scaleY,scaleZ] : Any list of positive floats with length 3 |

Before being loaded in to the solver, the mesh's [x,y,z] location data is multiplied by this vector.

Example usage:

```
# Scale from mm to metres
'scale' : [0.001,0.001,0.001]
```

```
# Scale from inches to metres
'scale' : [0.0254,0.0254,0.0254]
```

```
# Scale the mesh to be 10 times larger in y axis
'scale' : [1,10,1]
```

Note: The 'scale' parameter applies to the mesh only - not interpolated surface outputs etc. (see *Tab x*)

Scale Function

As an alternative to passing a fixed scaling factor the 'scale' parameter will also accept a function. See the [Scale Function](#) for details.

Partitioner

| Keyword | Re- quired | Default | Valid values |
|---------------|---------------|---------|--|
| 'partitioner' | No | 'metis' | ['metis']. The name of the partitioner to use. |

The 'metis' partitioner is used to split the computational mesh up so that the job can be run in parallel. Other partitioners will be added in future code releases

Restart

| Keyword | Re-required | Default | Valid values |
|--------------------------|-------------|---------|--|
| 'restart' | No | False | True/False |
| 'restart casename' | No | | Optionally supply the name of a case to restart from that case, if not supplied then the current casename will be used.. |
| 'interpolate restart' | No | False | True/False: True to interpolate restart results from a different mesh. |
| 'restart meshname' | No | False | The name of the mesh to interpolate the restart from. |
| 'solution smooth cycles' | No | 0 | Number of Laplace smoothing cycles to smooth initial mapped solution |

Restart allows you to load a solution from a previous run and continue the solver from that point. By default the solver will look for a <casename>_results.h5 file to load the restart, this can be overridden with the 'restart casename' parameter.

The default restart assumes that the same mesh is used. Alternatively, a solution from another mesh can be mapped on to a new mesh using the 'interpolate restart' parameter.

Example usage:

```
# Restart from previous solution
'restart' : True,
# Load solution from case name 'my_case'
'restart casename' : 'my_case',
```

```
# Restart from previous solution on a different mesh
"interpolate restart": True,
# Interpolate restart from "different_mesh.h5"
"restart meshname": "different_mesh",
# Do not smooth initial solution
"solution smooth cycles": 0,
```

Advanced Restart

It is possible to restart the SA-neg solver from results generated using the Menter SST solver and vice versa, however by default the turbulence field(s) are reset to zero. An approximate initial solution for the SA-neg model can be generated from Menter-SST results by adding the key

```
# Generate approximate initial field for SA-neg
'approximate sa from sst results' : True,
```

Safe Mode

| Keyword | Re- quired | Default | Valid values |
|---------|---------------|---------|--------------|
| 'safe' | No | False | True/False |

Safe mode turns on extra checks and provides diagnosis in the case of solver stability issues

Example usage:

```
# Enable safe mode
'safe' : True,
```

1.5.2 Boundary Conditions

Boundary condition properties are defined using consecutively numbered blocks

```
'BC_1' : {....},
'BC_2' : {....},
```

zCFD will automatically detect zone types and numbers in a number of mesh formats, and assign appropriate boundary conditions. The type tags follow the Fluent convention (wall = 3, etc), and if present no further information is required. Alternatively, the mesh format may contain explicitly numbered zones (which can be determined by inspecting the mesh). In this case, the user can specify the list of zone numbers for each boundary condition 'type' and 'kind' (see below).

Wall

Wall boundaries in zCFD can be either slip walls, no-slip walls, or use automatic wall functions. No-slip walls are appropriate when the mesh is able to fully resolve the boundary layer (i.e ($y^+ \leq 1$)) otherwise automatic wall functions should be used. This behaviour is chosen by setting 'kind' in the wall specification below. Optionally it is possible to set wall velocity, roughness and temperature.

```
'BC_1' : {
    # Zone type tag
    'ref' : 3,
    # Optional: Specific zone boundary condition override
    'zone' : [0,1,2,3],
    # Required: Boundary condition type
    'type' : 'wall',
    # Type of wall, one of 'slip', 'noslip', 'wallfunction'
    'kind' : 'slip',
    # Optional: Roughness specification
    'roughness' : {
        # Type of roughness specification (option: height or length)
        'type' : 'height',
        # Constant roughness
        'scalar' : 0.001,
        # Roughness field specified as a VTK file
        'field' : 'roughness.vtp',
    },
    # Optional: Wall velocity specification either 'linear' or 'rotating' ↴
    # dictionary supplied
    'V' : {
```

(continues on next page)

(continued from previous page)

```

'linear' : {
    # Velocity vector
    'vector' : [1.0,0.0,0.0],
    # Optional: specifies velocity magnitude
    'Mach' : 0.20,
},
'rotating' : {
    # Rotational velocity in rad/s
    'omega' : 2.0,
    # Rotation axis
    'axis' : [1.0,0.0,0.0],
    # Rotation origin
    'origin' : [0.0,0.0,0.0],
},
},
# Optional: Wall temperature specification either a constant 'scalar' value
→ or a 'profile' from file
'temperature' : {
    # Temperature in Kelvin
    'scalar' : 280.0,
    # Temperature field specified as a VTK file
    'field' : 'temperate.vtp',
},
},

```

Note: The roughness at each boundary face is set by finding the nearest point to the face centre on the supplied VTK file with the roughness value looked up in a node based scalar array called ‘Roughness’

Note: The temperature at each boundary face is set by finding the nearest point to the face centre on the supplied VTK file with the temperature value looked up in a node based scalar array called ‘Temperature’

Farfield

The farfield boundary condition can automatically determine whether the flow is locally an inflow or an outflow by solving a Riemann equation. The conditions on the farfield boundary are set by referring to an ‘IC_’ block. In addition to this an atmospheric boundary layer profile or a turbulence profile may be set.

```

'BC_2' : {
    # Zone type tag
    'ref' : 9,
    # Optional: Specific zone boundary condition override
    'zone' : [0,1,2,3],
    # Required: Boundary condition type
    'type' : 'farfield',
    # Required: Kind of farfield options (riemann, pressure or supersonic)
    'kind' : 'riemann',
    # Required: Farfield conditions
    'condition' : 'IC_1',
    # Optional: Atmospheric Boundary Layer
    'profile' : {
        'ABL' : {
            # Roughness length
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

        'roughness length' : 0.05,
        # Friction velocity
        'friction velocity' : 2.0,
        # Surface Layer Height
        'surface layer height' : 1000,
        'Monin-Obukhov length' : 2.0,
        # Turbulent kinetic energy
        'TKE' : 1.0,
        # Ground Level
        'z0' : 0.0,
    },
},
# Optional: Specify a turbulence profile
'turbulence' : {
    'length scale' : 'filename.vtp',
    'reynolds tensor' : 'filename.vtp',
},
},

```

Inflow

zCFD can specify two kinds of inflow boundary condition: pressure or massflow, selected using the 'kind' key. When using the pressure inflow the boundary condition is specified by a total pressure ratio and a total temperature ratio that needs to be defined by the condition this refers to. When using the massflow condition a total temperature ratio and mass flow ratio or a mass flow rate is required. Note the mass flow ratio is defined relative to the condition set in the 'reference' key. By default the flow direction is normal to the boundary but either a single vector or a python function can be used to specify the inflow normal vector of each face on the surface.

```

'BC_3' : {
    # Zone type tag
    'ref' : 4,
    # Optional: Specific zone boundary condition override
    'zone' : [0,1,2,3],
    # Required: Boundary condition type
    'type' : 'inflow',
    # Required: Kind of inflow, either 'default' for pressure or 'massflow'
    'kind' : 'default',
    # Required: Reference inflow conditions
    'condition' : 'IC_2',
},

```

Example Pressure Inflow

An example setup for a pressure inflow is given below:

```

'IC_1' : {
    'temperature' : 293.0,
    'pressure' : 101325.0,
    'V' : {
        'vector' : [1.0,0.0,0.0],
        'Mach' : 0.20,
    },
    'Reynolds No' : 1.0e6,
}

```

(continues on next page)

(continued from previous page)

```
'Reference Length' : 1.0,
'turbulence intensity' : 0.1,
'eddy viscosity ratio' : 1.0,
},
'IC_2' : {
'reference' : 'IC_1',
'total temperature ratio' : 1.0,
'total pressure ratio' : 1.0,
# Optional - specify mach number to set
# static pressure at inflow from total pressure
'mach' : 0.1,
},
'BC_3' : {
'ref' : 4,
'zone' : [0,1,2,3],
'type' : 'inflow',
'kind' : 'default',
'condition' : 'IC_2',
},
```

Example Massflow Inflow

An example setup for a massflow inflow is given below:

```
def inflow_direction(x, y, z):
    # x, y, and z are the coordinates of the face centre
    cen = [1.0, 0.0, 0.0]
    y1 = y - cen[1]
    z1 = z - cen[2]
    rad = math.sqrt(z1*z1 + y1*y1)
    phi = math.atan2(y1,z1)
    angle = -0.24
    x2 = math.cos(angle)
    y2 = math.sin(phi) * math.sin(angle)
    z2 = math.cos(phi) * math.sin(angle)
    return {'v1': x2, 'v2': y2, 'v3': z2}

'IC_1' : {
    'temperature' : 293.0,
    'pressure' : 101325.0,
    'V' : {
        'vector' : [1.0,0.0,0.0],
        'Mach' : 0.20,
    },
    'Reynolds No' : 1.0e6,
    'Reference Length' : 1.0,
    'turbulence intensity' : 0.1,
    'eddy viscosity ratio' : 1.0,
},
'IC_2' : {
    'reference' : 'IC_1',
    'total temperature ratio' : 1.0,
    'mass flow rate' : 10.0,
    'vector' : inflow_direction
},
```

(continues on next page)

(continued from previous page)

```
'BC_3' : {
    'ref' : 4,
    'zone' : [0,1,2,3],
    'type' : 'inflow',
    'kind' : 'massflow',
    'condition' : 'IC_2',
},
```

Outflow

Pressure or massflow outflow boundaries are specified in a similar manner to the [Inflow](#) boundary. They can be of:code` `kind`` 'pressure', 'radial pressure gradient' or 'massflow'. When either pressure boundary is set the 'IC_-' conditions it refers to must set a static temperature ratio, when 'massflow' is specified a mass flow ratio or mass flow rate must be set. Note the mass flow ratio is defined relative to the condition set in the 'reference' key.

```
'BC_4' : {
    # Zone type tag
    'ref' : 5,
    # Optional: Specific zone boundary condition override
    'zone' : [0,1,2,3],
    # Boundary condition type
    'type' : 'outflow',
    # Kind of outflow 'pressure', 'massflow' or 'radial pressure gradient'
    'kind' : 'pressure',
    # Required only when using 'radial pressure gradient'
    # Radius at which the static pressure ratio is defined
    'reference radius' : 1.0,
    # Outflow conditions
    'condition' : 'IC_2',
},
```

Symmetry

```
'BC_5' : {
    # Zone type tag
    'ref' : 7,
    # Optional: Specific zone boundary condition override
    'zone' : [0,1,2,3],
    # Required: Boundary condition type
    'type' : 'symmetry',
},
```

Periodic

This boundary condition needs to be specified in pairs with opposing transformations. The transforms specified should map each zone onto each other.

```
'BC_6' : {
    # Required: Specific zone index
    'zone' : [1],
    # Required: Type of boundary condition
    'type' : 'periodic',
    # Required: Either 'rotated' or 'linear' periodicity
    'kind' : {
        # Rotated periodic settings
        'rotated' : {
            'theta' : math.radians(120.0),
            'axis' : [1.0,0.0,0.0],
            'origin' : [-1.0,0.0,0.0],
        },
        # Linear periodic settings
        'linear' : {
            'vector' : [1.0,0.0,0.0],
        },
    },
},
```

1.5.3 Time Marching

zCFD is capable of performing both steady-state and time dependent simulations. Steady-state solutions are marched in pseudo time towards convergence. Time dependent simulations can be either globally time-stepped or use local pseudo time-stepping within a dual time-stepping iteration in physical time. The ‘time marching’ dictionary controls the selection of these schemes. Major options are what CFL number to use, whether explicit or implicit time-stepping is used and whether the simulation is time dependent.

```
parameters = {
    ...
    "time marching": {
        "cycles": 5000,
        "unsteady": {...},
        "scheme": {...},
        "cfl": 30.0,
        "multigrid": 3,
        "multigrid cycles": 20000,
        "multipoly": False,
        "moving mesh": False,
    },
    ...
}
```

Cycles

The total number of pseudo time-steps to use in a *steady-state* simulation, or the number of pseudo time-steps to use in each real time-step of an *unsteady* time-dependent simulation

Example usage:

```
# A good starting value for most explicit, steady state cases. Implicit simulations
- can use a higher CFL and therefore require fewer cycles - around 100 - 500 cycles
- is a good starting point for an implicit steady state case.
'cycles' : 5000,
```

CFL Control

The Courant-Friedrichs-Lowy (CFL) number controls the local pseudo time-step that the solver uses to reach a converged solution. The larger the CFL number, the faster the solver will run but the less stable it will be. The CFL dictionary keys are only valid when using dual time-stepping.

| Keyword | Re-required | Default | Valid values |
|----------------------|-------------|--------------------|--|
| 'cfl' | Yes | 0 | Any positive floating point number. |
| 'cfl transport' | No | - | Any positive floating point number. |
| 'cfl ramp' | No | - | Dictionary containing 'initial' and 'growth' - respectively the starting CFL number and the growth rate applied as a multiplier each successive time step until the CFL number is reached. |
| 'ramp func' | No | - | Instead of the 'cfl ramp' parameter the user can specify a <i>ramp function</i> . |
| 'cfl coarse' | No | Current CFL number | Any positive floating point number. |
| 'cfl viscous factor' | Yes | 1 | Any positive floating point number. |
| 'cfl for pmg levels' | No | - | List of 2 floating point numbers. |

cfl

Sets the CFL number used in the mass, momentum and energy equations. When using *dgcaa* this applies to the the 'rho', 'rhou_i' and 'rhoE' equations.

Example usage:

```
# Typical value used for a RANS simulation using explicit time marching
'cfl' : 1
```

```
# Typical value used for a RANS simulations using implicit time marching
'cfl' : 30
```

Note: Besides mesh quality, CFL is the most important parameter affecting simulation stability. If you find your simulation does not converge, consider reducing CFL

cfl transport

Sets the maximum CFL number used to update the turbulent transport equations (for example k and ω).

```
# Use cfl transport to promote solver stability
'cfl' : 30,
'cfl transport': 20
```

Note: If you find your simulation does not converge and reports high turbulence equation residuals which increase with successive cycles, consider setting this to a value lower than 'cfl'

cfl ramp

This allows a relatively large target CFL number to be specified, but start the simulation using a smaller one to provide stability as large non-physical transients are eliminated from the solution.

Starting a simulation with a large CFL number can cause instability to crash the solver. A more gentle start will reduce the impact of non-physical transients.

```
# Start cfl at 0.1 and grow at a rate 1.1, reaching the target cfl of 30 in 26 steps
'cfl' : 30,
'cfl ramp': {
    'initial': 0.1,
    'growth': 1.1
}
```

cfl coarse

For simulations with multigrid acceleration, the 'cfl coarse' number is used as a CFL condition on the coarse meshes. For such cases the 'cfl' number is still applied to the finest mesh.

This only applies to explicit finite volume solutions.

```
# The quality of agglomerated (multigrid) coarse meshes is typically less than the
# quality of the primary (fine) mesh and so a lower CFL number may be required for
# numerical stability.
'cfl' : 1.0,
'cfl coarse': 0.5
```

cfl viscous factor

Factor applied to the diffusive (viscous) time step when applying the CFL condition. As viscous effects are rarely the cause of solver instability, the contribution of the viscous terms to the time step evaluation may be reduced by setting 'cfl viscous factor' to a number less than 1.

```
# For the low Reynolds number cylinder case, viscous effects are dominant but do not
# contribute to instability.
# The value of 1e-6 effectively discounts viscosity in the determination of the time
# step when applying the CFL condition.
"cfl": 200,
"cfl viscous factor": 1.0E-6,
"cfl ramp factor": {"initial": 0.1, "growth": 1.05},
```

cfl for pmg levels

'cfl for pmg levels' sets the cfl number for each polynomial level during a polynomial multigrid cycle. Two values are required in the Python list, the CFL number for the highest polynomial order, and the CFL number for a P0 polynomial. The CFL number varies with polynomial order in accordance with the linear formula as $1.0/(2 * P + 1)$. Note that the order of spatial accuracy of a P0 polynomial is $P + 1$ and for a P2 polynomial the order is $P + 3$.

```
# For the high order (DG) cylinder case, the CFL for the P2 polynomial is 0.55, while
# the CFL for the P0 polynomial can be larger at 1.5.
"cfl": 0.55,
"cfl for pmg levels": [0.55, 1.5],
"cfl viscous factor": 0.5,
"cfl ramp factor": {"initial": 0.1, "growth": 1.05},
```

Solver scheme

There are two different time marching schemes available in zCFD: multistage explicit and euler implicit scheme. These can be used for both steady and unsteady (using global or dual time-stepping) simulations. Global time-stepping requires no preconditioning and excludes some of the multistage RK schemes.

| Keyword | Required | Default | Valid values | Description |
|---------|----------|----------------------|--|---|
| 'name' | Yes | "runge kutta" | "euler" "runge kutta" "implicit euler" | Name of scheme |
| 'stage' | No | "rk third order tvd" | 1 4 5 'rk third order tvd' | ("runge kutta" only) Number of RK stages |
| 'kind' | No | - | 'local timestepping' 'global timestepping' | Timestepping mode |

Note: The Euler scheme has a single stage and is the safest option for new or problematic cases.

name

The solution is integrated in time (solved) using an integration scheme that can be either explicit (variables driving the change are evaluated at the previous step) or implicit (some or all of the variables driving the change are evaluated at the current step). '**euler**' or '**runge kutta**' are the explicit schemes and '**implicit euler**' is the implicit option.

A larger time step can be taken using the implicit scheme, as the method is more stable than an explicit scheme.

Note: The trade off in using implicit time integration is the increase in memory requirement, typically 3-5 times more than for an explicit scheme.

stage

The number of Runge-Kutta stages in each step of the integration scheme affects the order of temporal convergence, and affects the maximum CFL that can be used.

Note: For most of the test cases where Runge-Kutta integration is used, 5 stages are applied. This is a typical industry standard and has been shown to be robust for most test cases.

kind

Global time stepping uses the same time step in each cell for the integration, whereas local time stepping calculates and uses the time step in each cell as determined by the local flow conditions and the prescribed CFL number. In cases where there are very small cells. it would be impractical to use global timestepping as all the cells would have to march at the pace of the slowest.

Note: While global timestepping restricts all cells to the smallest timestep and should only be used for unsteady simulations.

Unsteady

The Unsteady dictionary controls the simulation time and can be used to set up both Steady and Unsteady cases.

| Keyword | Required | Default | Valid values | Description |
|--------------|----------|----------|----------------------------|--|
| 'total time' | Yes | – | Positive float | Total time to simulate |
| 'time step' | Yes | – | Positive float | Physical time step for the simulation |
| 'order' | Yes | "second" | "first" "second" 1 2 | (Unsteady only) Order of time integration. |
| 'start' | No | – | Positive integer | (Unsteady only) For unsteady simulations, how many steady-state timesteps to run to initialise solution. |

Running steady state

For steady-state simulations, '**total time**' should be set equal to the '**time step**' value. The solver will then automatically run a steady-state simulation.

Example usage:

```
# Run an unsteady simulation for 1.7 seconds with a time step of 0.002 seconds
'time marching' : {
    ...
    "unsteady": {
        "total time": 1.0,
        "time step": 1.0,
    },
    ...
}
```

Running unsteady

If the ‘**time step**’ value is less than the ‘**total time**’ value, then an unsteady simulation is created.

Time integration for unsteady simulations is based on the solution at the current time step and the contribution of the solution in previous time steps, represented by a Backward Difference Formula. Dual time-stepping is used to update the current solution with data from previous time steps. ‘**order**’ controls the order of the Dual time-stepping.

‘**start**’ provides the option of initialising an unsteady simulation with a number of steady-state pseudo timesteps.

Example usage:

The unsteady laminar cylinder test case sheds a vortex street with a specific Strouhal number. We want to run the simulation for 1.7 seconds to allow the oscillation to begin and establish a well-formed pattern. We set the time step to a small enough value to capture the unsteady shedding phenomenon.

```
# Run an unsteady simulation for 1.7 seconds with a time step of 0.002 seconds
'time marching' : {
    ...
    "unsteady": {
        "total time": 1.7,
        "time step": 0.002,
        "order": "second",
        "start": 0
    },
    ...
}
```

The example below uses a relatively large time step and maintains accuracy with a second order integration scheme. The case starts with 2000 pseudo-timesteps before starting the unsteady simulation.

```
# Run an unsteady simulation for 1.7 seconds with a time step of 0.002 seconds
'time marching' : {
    ...
    "unsteady": {
        "total time": 0.06,
        "time step": 2.0e-5,
        "order": "second",
        "start": 2000
    },
    ...
}
```

Multigrid

| Keyword | Re-required | Default | Valid values |
|--------------------|-------------|---------|-----------------------|
| ‘multigrid’ | No | – | Any positive integer. |
| ‘multigrid cycles’ | No | – | Any positive integer. |
| ‘multipoly’ | No | False | True False |

To accelerate an explicit finite volume solution, coarse meshes are generated from the original (fine) mesh by agglomerating or combining cells together to create larger cells. The ‘**multigrid**’ value sets the number of successive times to apply this operation, each iteration resulting in a coarser mesh with fewer cells. By solving on the hierarchy of coarser meshes as well as the fine mesh, information is moved faster in the flow domain

and a converged solution is reached more efficiently. The multigrid scheme is designed so that the acceleration does not (in theory) change the result that would be obtained just by solving on the finest mesh alone, though in practice the use of multigrid does ultimately stall convergence on most meshes.

Use multigrid acceleration for a specified number of '**multigrid cycles**' and then revert to the finest mesh only. This is to make use of the more efficient scheme at the start, and avoid convergence stall due to multigrid at the end of a solution process.

For high order (Discontinuous Galerkin) simulations, '**multipoly**' may be used to accelerate convergence by using lower order polynomials in a manner analogous to finite volume multigrid acceleration. The time step limitation on the lower order polynomials is less restrictive than for the higher order polynomials and hence information corresponding to longer wavelengths may be more efficiently propagated in the fluid domain. The use of '**multipoly**' automatically uses the highest one specified and P0.

Example usage:

```
# 3 levels of multigrid is sufficient in most cases to reduce the number of cycles required by an order of magnitude.
# If this causes instability, a reduction to 2 may solve the problem.
'time marching' : {
    ...
    'multigrid': 3,
    ...
}

# After 20000 explicit cycles, multigrid stall prevents convergence to the correct drag value.
# By turning it off, the correct value is obtained on the finest mesh.
'time marching' : {
    ...
    'multigrid': 3,
    'multigrid cycles' 20000,
    ...
}
```

Note: Multigrid only applies to explicit finite volume solutions. Multipoly only applies to High Order (DG) solutions.

Moving Mesh

| Keyword | Required | Default | Valid values |
|---------------|----------|---------|--------------|
| 'moving mesh' | No | False | True False |

In any case where the mesh (or a fluid zone within the mesh) is moving (translation or rotation) the simulation can be performed in either the frame of reference of the mesh or in a fixed coordinate system. For simulations where there is no relative movement of interest, '**moving mesh**' may be set to False. For simulations where there is relative movement, setting '**moving mesh**' to True will force a recalculation of the physical location of the mesh, including the recalculation of any overset mappings.

1.5.4 Linear Solver Settings

There are several situations where the options selected in zCFD require the solution of a linear system. These are:

- When the *Time Marching* scheme is set to euler implicit.
- When RBFs are used for mesh motion.
- When the incompressible solver is selected.

When running on CPUs zCFD uses **Petsc** linear solver library and when running on NVidia GPUs zCFD uses the **AMGX** linear solver library. Both libraries offer a wide array of solver and preconditioner choices and altering the settings can have a significant impact on the performance and convergence of the solver. Options can be passed to Petsc via the control dictionary and AMGX options are set using a json file which is read at runtime.

Petsc

Details of the available options for the Petsc KSP linear system solvers used by zCFD are given [here](#). these options can be passed through to Petsc via the zCFD control dictionary using the “linear solver options” key. The default values are given below:

```
"linear solver options": {"flow": { "-ksp_type": "fgmres",
                                     "-ksp_rtol": "1.0e-3",
                                     "-ksp_monitor": "",
                                     "-ksp_converged_reason": "" },
                           "turbulence": { "-ksp_type": "fgmres",
                               "-ksp_rtol": "1.0e-5",
                               "-ksp_monitor": "",
                               "-ksp_converged_reason": "" },
                           "rbf": { "-ksp_type": "fgmres",
                                    "-ksp_rtol": "1.0e-5",
                                    "-ksp_monitor": "" }
                         }
```

Settings are provided for the mean flow, turbulence and RBF linear systems. Most of the settings detailed in the [Petsc documentation](#) can be passed through by adding them as key, value pairs to the appropriate dictionary. Where an option doesn't take a value an empty string needs to be provided.

The **Hypre** library of algebraic multigrid methods is available via **Petsc PCHYPRE** and the associated settings can be supplied via the “linear solver options” dictionary. Using Hypre's boomeramg package as a preconditioner has shown good performance with the incompressible solver:

```
"linear solver options": {"flow": { "-ksp_type": "fgmres",
                                     "-ksp_rtol": "1.0e-3",
                                     "-ksp_monitor": "",
                                     "-ksp_converged_reason": "",
                                     "-pc_type": "hypre",
                                     "-pc_hypre_type": "boomeramg"},,
                           ...
                         }
```

Further Hypre options are detailed in the [Petsc PCHYPRE](#) documentation.

AMGX

The AMGX settings for the mean flow, turbulence and RBF linear systems can be found in ZCFD_HOME/amgx.json, ZCFD_HOME/turbamgx.json and ZCFD_HOME/RBF_amgx.json. The mean flow settings are shown below:

```
"config_version": 2,
"determinism_flag": 1,
"solver": {
    "preconditioner": {
        "error_scaling": 0,
        "print_grid_stats": 0,
        "max_uncolored_percentage": 0.05,
        "algorithm": "AGGREGATION",
        "solver": "AMG",
        "smoother": "MULTICOLOR_GS",
        "presweeps": 0,
        "selector": "SIZE_8",
        "coarse_solver": "NOSOLVER",
        "max_iters": 1,
        "postsweeps": 3,
        "min_coarse_rows": 32,
        "relaxation_factor": 0.75,
        "scope": "amg",
        "max_levels": 40,
        "matrix_coloring_scheme": "PARALLEL_GREEDY",
        "cycle": "V"
    },
    "use_scalar_norm": 1,
    "solver": "FGMRES",
    "print_solve_stats": 1,
    "obtain_timings": 1,
    "max_iters": 10,
    "monitor_residual": 1,
    "gmres_n_restart": 5,
    "convergence": "RELATIVE_INI_CORE",
    "scope": "main",
    "tolerance": 1e-3,
    "norm": "L2"
}
```

In general these settings provide good performance for most cases. If issues are encountered converging the linear system reducing the “tolerance” may help. Otherwise altering the multigrid aggregation selection algorithm “selector”: “SIZE_8” to “SIZE_4” or “SIZE_2” will build smaller aggregates and hence increase the number of coarse levels - improving convergence at the expense of memory use. Increasing the number of “gmres_n_restart” may also improve convergence at the expense of increased memory use.

There are too many AMGX options to detail here but example configurations for different solvers are given [here](#).

When running on NVidia GPUs and using AMGX there is only one option available in the “linear solver options” dictionary:

```
"linear solver options": {"double precision": False}
```

The “double precision” option controls whether the mean flow linear system is solved in single or double precision. Solving in single precision brings a reduction in memory use and a speed up to the solver. However, double precision may be required for more challenging cases.

1.5.5 Equations

| Keyword | Re- quired | Default | Valid values |
|-------------|---------------|---------|--------------------------------------|
| 'equations' | Yes | - | Any of equation sets detailed below. |

The equations key selects the equation set to be solved as well as the finite volume solver or the high order strong form Discontinuous Galerkin/Flux Reconstruction solver.

The corresponding equation set options should also be defined in the control dictionary. The valid equation keys are shown in the table below.

| Solver | Description | Com- press- ible | High order | Incom- press- ible |
|---------|---|------------------------|------------------|--------------------------|
| Euler | Use the Euler flow equations in the simulation. These are compressible, inviscid flow equations. | "euler" | - | - |
| RANS | Use the RANS (Reynolds-Averaged Navier-Stokes) equations in the simulation. These are compressible, viscous flow equations with additional terms included to account for the effect of turbulence on the flow without the expense of simulating the turbulent flow structures themselves. | "rans" | "dgrans" | Coming soon |
| Viscous | Use the viscous flow equations. These are compressible, viscous flow equations which do not model turbulence. | "vis- cous" | "dgvis- cous" | Coming soon |
| LES | Use the Large Eddy Simulation (LES) equations. | "les" | "dgles" | Coming soon |

Example usage:

```
parameters = {
    ...
    # Use the euler equation set
    'equations' : 'euler',
    # Define the inputs for the 'euler' equation set
    'euler': {...}
    ...
}
```

Common Settings

The following parameters are common for all of the equation sets.

| Keyword | Required | Default | Valid values |
|----------------------------------|--|---------|---|
| 'order' | Yes | – | 'first' 'second' 'euler_second' |
| 'first order cycles' | No | – | Positive integer |
| 'linear gradients' | No | False | True False |
| 'leastsq gradients' | No | False | True False |
| 'inviscid flux scheme' | Yes | "HLLC" | "HLLC" "Rusanov" "Roe" "Roe low diss" |
| 'roe low dissipation sensor' | No | – | "FD" "NTS" "DES" "NONE" |
| 'roe dissipation sensor minimum' | No | – | Float between 0.05 and 1.0 |
| 'freeze limiter cycle' | No | – | Positive integer |
| 'relax' | Yes (for incompressible equation set) | 0.5 | Positive float |
| 'turbulence' | Yes (for 'les' or 'rans' equation set) | {} | See " turbulence " |

order

Sets the order of the spatial discretisation. '**euler_second**' is second order for the flow variables and first order for the turbulence.

first order cycles

Specifies the number of cycles the solver will run with first order spatial discretisation. This can be used to improve stability at start up.

Example usage: Run 500 cycles first order.

```
parameters = {
    ...
    'equations' : 'euler',
    'euler': {
        ...
        'first order cycles': 500,
        ...
    }
}
```

linear gradients & leastsq gradients

By default zCFD uses a Weighted Green-Gauss gradient calculation. This can be switched to a standard Green-Gauss by setting '**linear gradients**' to True. Setting '**leastsq gradients**' to True will set the gradient calculation to Least Squares.

inviscid flux scheme

Picks the scheme to use for the inviscid flux. **HLLC** is good when running explicit, **Roe** is best for implicit and **Rusanov** is good for getting an implicit simulation going. The **Roe low diss** scheme can be used with LES or Hybrid RANS/LES

roe low dissipation sensor

The sensor used to decide whether to use (and by how much) the Roe low dissipation scheme.

| Setting | Notes |
|---------|--|
| 'DES' | Uses the blending function from the selected DES model to turn on the Roe low dissipation scheme in LES regions |
| 'NONE' | Uses the value set on ' roe dissipation sensor minimum ' everywhere |
| 'FD' | See Johnsen et al. JCP 229 (2010) pag. 1234 |
| 'NTS' | See Xiao et al. INT J HEAT FLUID FL 51 (2015) pag. 141 https://doi.org/10.1016/j.ijheatfluidflow.2014.10.007 |

roe dissipation sensor minimum

Sets the minimum value that the dissipation term in the Roe scheme is multiplied by when using the '**Roe low diss**' scheme. This allows you control the stability of the low diss scheme (smaller values reduce the dissipation and hence the stability of the scheme)

freeze limiter cycle

If set the values of the limiter used in the MUSCL reconstruction are frozen at the specified cycle. This can be useful in the latter stages of a simulation if noise from the limiter is causing the residual convergence to stall.

relax

The relaxation factor used on the flow equations in the incompressible solver. Increasing this will speed up convergence at the expense of stability and vice versa.

turbulence

The turbulence dictionary is required for the "*rans*" and "*les*" equations sets.

Euler

Compressible Euler flow is inviscid (no viscosity and hence no turbulence). The compressible Euler equations are appropriate when modelling flows where momentum significantly dominates viscosity - for example at very high speed. The computational mesh used for Euler flow does not need to resolve the flow detail in the boundary layer and hence will generally have far fewer cells than the corresponding viscous mesh would have.

As well as the “*common*” settings, the following settings are available:

| Keyword | Required | Default | Valid values | Description |
|----------------|----------|---------|--------------|-----------------------------------|
| ‘precondition’ | Yes | False | True False | Use low speed mach preconditioner |

Example usage:

```
parameters = {
    ...
    # Use the euler equation set
    'equations' : 'euler',
    'euler' : {
        'order' : 'second',
        'first order cycles': 100,
        # Cycle on which to freeze the limiter values
        'freeze limiter cycle': 500,
        # Use low speed mach preconditioner
        'precondition' : True,
        'Inviscid Flux Scheme': 'HLLC',
    },
}
```

Viscous

The viscous (laminar) equations model flow that is viscous but not turbulent. The [Reynolds number](#) of a flow regime determines whether or not the flow will be turbulent.

The computational mesh for a viscous flow has to resolve the boundary layer so will generally be larger than that used for an Euler simulation. A viscous simulation will run faster than a RANS simulation using the same mesh since fewer equations are being modelled.

As well as the “*common*” settings, the following settings are available:

| Keyword | Required | Default | Valid values | Description |
|----------------|----------|---------|--------------|-----------------------------------|
| ‘precondition’ | Yes | False | True False | Use low speed mach preconditioner |

Example usage:

```
parameters = {
    ...
    # Use the viscous equation set
    'equations' : 'viscous',
    'viscous' : {
        'order' : 'second',
        'first order cycles': 100,
        # Use low speed mach preconditioner
    },
}
```

(continues on next page)

(continued from previous page)

```

    'precondition' : True,
    'linear gradients' : False,
    'Inviscid Flux Scheme': 'HLLC',
},
..
}

```

RANS

The fully turbulent (Reynolds Averaged Navier-Stokes Equations)

As well as the “**common**” settings, the ‘RANS’ equation set requires the additional ‘turbulence’ dictionary, which can have the following parameters:

| Keyword | Required | Default | Valid values | Description |
|-----------------------------------|----------|---------|---|--|
| ‘les’ | Yes | ‘none’ | ‘none’ ‘DES’ ‘DDES’ ‘IDDES’ ‘SAS’ | Selects the hybrid RANS/LES model. |
| ‘model’ | Yes | – | ‘sst’ ‘sas’ ‘sa-neg’ ‘sst-transition’ | Selects the RANS turbulence model |
| ‘approximate sa from sst results’ | No | | True False | Restart an SA-neg calculation from an SST solution |
| ‘rotation correction’ | No | False | True False | Adds the rotation correction term to SST or SA-neg |
| ‘limit gradient k’ | No | | Float from 0.01 to 1 | Activates the \tilde{v} gradient limiter in the SA-neg model |
| ‘qcr’ | No | False | True False | Activates the non-linear QCR correction to the turbulent stress. Applicable to both SST and SA-neg |

les

Selects the hybrid RANS/LES model in the RANS solver which is applied to the base model selected with the “**model**” key word.

| Setting | Notes |
|---------|--|
| ‘none’ | Solve the RANS equations with no hybrid RANS/LES model |
| ‘DES’ | Detached Eddy Simulation |
| ‘DDES’ | Delayed Detached Eddy Simulation |
| ‘IDDES’ | Improved Delayed Detached Eddy Simulation |
| ‘SAS’ | Scale Adaptive Simulation |

model

Selects the RANS turbulence model.

| Setting | Notes |
|------------------|--|
| 'sst' | The Menter Shear Stress Transport Turbulence Model (https://turbmodels.larc.nasa.gov/sst.html) |
| 'sa' | Spalart-Allmaras turbulence model (https://turbmodels.larc.nasa.gov/spalart.html) |
| 'sa-neg' | Negative Spalart-Allmaras turbulence model (https://turbmodels.larc.nasa.gov/spalart.html) |
| 'sst-transition' | The Langtry-Menter 4-equation Transitional SST Model (https://turbmodels.larc.nasa.gov/langtrymenter_4eqn.html) |

limit gradient k

Activates the \tilde{v} gradient limiter in the SA-neg model. This helps eliminate gradient under/over shoot in regions of poor mesh quality.

The value should be between 0 and 1. **0** is off and **1** is maximum limiting.

qcr

Activates the non-linear QCR correction to the turbulent stress. Applicable to both SST and SA-neg. QCR has been shown to improve the prediction of separated corner flows, particularly on highly loaded wings.

Example RANS usage:

```
parameters = {
    ...
    # Use the viscous equation set
    'equations' : 'RANS',
    'RANS': {
        'order' : 'second',
        'first order cycles': 100,
        'freeze limiter cycle': 500,
        'linear gradients' : False,
        'leastsq gradients' : False,
        'Inviscid Flux Scheme': 'HLLC',
        'roe low dissipation sensor': 'DES',
        'roe dissipation sensor minimum': 0.05,
        'turbulence' : {
            'model' : 'sst',
            # No Hybrid RANS/LES
            'les' : 'none',
            'rotation correction': False,
            'limit gradient k': 0.5
            'qcr': True
        },
    },
    ...
}
```

Advanced parameters for turbulence models

The following parameters can be used in the ‘turbulence’ dictionary to allow tuning of the turbulence models in zCFD. These are advanced parameters and shouldn’t normally need to be adjusted.

| Keyword | Required | Default | Valid values | Description |
|--------------|----------|---------|------------------|---|
| ‘betastar’ | No | 0.09 | Positive number | Constant in the dissipation term on the k equation in the SST model |
| ‘cdes_kw’ | No | 0.78 | Positive number | The $k-\omega$ part of the blended C_{DES} in SST |
| ‘cdes_keps’ | No | 0.21 | Positive number | The $k-\epsilon$ part of the blended C_{DES} in SST |
| ‘cd1’ | No | 20 | Positive number | Constant in the f_{dt} blending function in IDDES |
| ‘cd2’ | No | 3 | Positive integer | Constant in the f_{dt} blending function in IDDES |
| ‘production’ | No | 0 | Positive integer | Selects the model for the production term in the SST model. |
| ‘cdes’ | No | 0.65 | Positive number | C_{DES} constant used for SA-neg based hybrid RANS/LES |

‘production’ can have the following values:

| Setting | Notes |
|---------|--|
| 0 | Selects the vorticity based production term (SST-V) $P_m = \mu_t \Omega_v^2 - \frac{2}{3} \rho k \delta_{ij} \frac{\partial u_i}{\partial x_j}$ |
| 1 | Selects the strain based production term for incompressible flows $P_m = \mu_t S^2$ |
| 2 | Selects the strain based production term for compressible flows $P_m = \mu_t S^2 - \frac{2}{3} \rho k \delta_{ij} \frac{\partial u_i}{\partial x_j}$ |
| 3 | Selects the Kato-Launder source term $P_m = \mu_t S \Omega_v - \frac{2}{3} \rho k \delta_{ij} \frac{\partial u_i}{\partial x_j}$ |

LES

As well as the “common” settings, the ‘LES’ equation set requires the additional ‘turbulence’ dictionary, which can have the following parameters:

| Keyword | Required | Default | Valid values | Description |
|---------|----------|---------|-----------------|------------------------------------|
| ‘les’ | Yes | ‘none’ | ‘none’ ‘WALE’ | Selects the hybrid RANS/LES model. |

les

Selects the model for the subgrid eddy viscosity in the LES solver: either “**none**” for implicit LES or “**WALE**” to use the Wall Adapted Local Eddy Viscosity model.

Example usage:

```
parameters = {
    ...
    # Use the viscous equation set
    'equations' : 'LES',
    'LES': {
        'order' : 'second',
        'first order cycles': 100,
        'freeze limiter cycle': 500,
        'linear gradients' : False,
        'leastsq gradients' : False,
        'Inviscid Flux Scheme': 'HLLC',
        'roe low dissipation sensor': 'DES',
        'roe dissipation sensor minimum': 0.05,
        'turbulence' : {
            'les' : 'none',
        },
    },
    ...
}
```

High Order

When using the high order strong form Discontinuous Galerkin/Flux Reconstruction solver the keys in the “**common**” equations dictionary are the same for each equation set except that the ‘**linear gradients**’ and ‘**limiter**’ keys are not valid and additional keys are available in the equations dictionary. For example all the available keys for ‘**dviscous**’ are a combination of the ‘**viscous**’ keys and the keys shows below. The same logic applies to the ‘**dtrans**’, and ‘**dges**’ equation sets.

These additional DG keys are:

| Keyword | Required | Default | Valid values | Description |
|-------------------------------------|----------|---------|--|--|
| 'order' | Yes | | 0 1 2 3 4 | This parameter sets the order of the polynomials within each element used to define the numerical solution |
| 'freeze diffusion during rk stages' | Yes | True | True False | Fixes diffusion terms after first RK stage |
| 'inviscid flow' | Yes | False | True False | Defines if only advective terms should be computed |
| 'c11 stability parameter' | Yes | 0 | Number | |
| 'ldg upwind parameter' | Yes | 0.5 | Number between 0 & 1 | |
| 'shock sensing' | Yes | False | True False | |
| 'shock sensing variable' | Yes | density | "density" "temperature" "mach" "turbulence" | |
| 'shock k' | No | 1.0 | Number | |
| 'shock sensing viscosity scale' | No | 1.0 | Number | |

DG Nodal locations

In addition to the equation set the location of the DG solution points must be specified in the parameters dictionary. The definitions first need to be imported by including this import statement at the start of the control dictionary. The nodal locations are used in the generation of Lagrange polynomials used to define the solution within an element.

There are various sets of nodal locations defined in literature for use with high order methods. Varying the nodal locations can minimise aliasing errors within the DG method. It is recommended to use the default values.

| Key-word | Re-quired | Default | Valid values | Description |
|----------------|-----------|---------------|--|--|
| 'line' | Yes | line_gaus | line_evenly_spaced line_gauss_lobatto line_gauss_legendre | Specifies 1D points defining nodal locations. |
| 'tri' | Yes | tri_shunn | tri_shunn_ham tri_evenly_spaced, | Specifies 2D triangle points defining nodal locations. |
| 'tetra-hedron' | Yes | tet_shunn | tet_shunn_ham tet_evenly_spaced | Specifies 3D tetra points defining nodal locations. |
| 'pyra-mid' | Yes | pyra-mid_gaus | pyra-mid_evenly_spaced pyra-mid_gauss_legendre pyra-mid_gauss_legendre | Specifies 3D pyramid points defining nodal locations. |

Before you use the nodal locations the dictionaries must be imported by adding the following to your control file:

```
from zcfd.solvers.utils.DGNodalLocations import *
```

To use the default values:

```
from zcfd.solvers.utils.DGNodalLocations import *

parameters = {
..
    'Nodal Locations' : nodal_locations_default['Nodal Locations']
..
}
```

Or select from the available types

```
from zcfd.solvers.utils.DGNodalLocations import *

parameters = {
..
    'Nodal Locations' : {
        'line': line_gauss_legendre_lobatto,
        'tetrahedron': tet_evenly_spaced,
        'tri' : tri_evenly_spaced,
        'pyramid': pyramid_evenly_spaced
    },
..
}
```

Example “dgviscous”

```
from zcfd.solvers.utils.DGNodalLocations import *

parameters = {
..
    'equation': "dgviscous",
    'dgviscous' : {
        # Keys for viscous
        'precondition' : True,
        'Inviscid Flux Scheme': 'HLLC',

        # Additional DG Keys
        'order' : 2,
        'inviscid flow': False,
        'freeze diffusion during rk stages': False
        'c11 stability parameter': 0.0,
        'LDG upwind parameter': 0.5,
        'Shock Sensing': False,
        'Shock Sensing k': 1.0,
        'Shock Sensing Viscosity Scale': 1.0,
        'Shock Sensing Variable': 'density',
    },
    'Nodal Locations' : nodal_locations_default['Nodal Locations']
..
}
```

1.5.6 Aero-Acoustics

Broadband aero-acoustic simulations can be performed by selecting the DGCAA solver. Aero-acoustic sources are currently limited to stochastic noise sources generated by the FRPM method. These sources are typically driven by a steady state RANS simulation.

The CAA conditions can be mapped from a CFD solution or specified as a set of initial conditions to give freestream density and sound speed.

The configuration can make use of multiple “[FRPM](#)” domains and also include a [sponge](#) layer.

DGCAA Solver

The DGCAA is configured in the same way as the other solvers via an equations set with the name “[dgcaa](#)”.

| Keyword | Required | Default | Valid values |
|--|----------|---------|-------------------------------|
| ‘order’ | Yes | | 0 1 2 3 4 |
| ‘map cfd to caa mesh’ | Yes | True | True False |
| ‘frpm solver only’ | Yes | False | True False |
| ‘microphone output frequency’ | Yes | 1 | Positive integer |
| ‘frpm sources’ | Yes | True | True False |
| ‘number of smallest timestep locations to display’ | Yes | 1 | Positive integer |
| ‘rans case name’ | Yes | | String; python case file name |
| ‘rans mesh name’ | No | | String; mesh file name |
| ‘frpm’ | No | {} | See “ frpm ” |

Example usage

```
parameters = {
    ...
    'equations' : 'DGCAA',
    'DGCAA' : {
        'order' : 2,
        'map cfd to caa mesh' : True,
        'frpm solver only' : False,
        'rans mesh name' : "rans_mesh.h5",
        'rans case name' : "rans_case.py",
        'frpm sources' : True,
        'microphone output frequency' : 10,
        'frpm' : {...}
    }
    ...
}
```

order

This parameter sets the order of the polynomials within each element used to define the numerical solution.

Map CFD to CAA mesh

If there are large variations in the freestream flow properties, it may be desirable to include refraction effects within the acoustic simulation. This can be done by supplying a CFD flow field where the variations in temperature and velocity are mapped onto the acoustic mesh and are used within the CAA propagation. If this is set to FALSE, the freestream properties supplied within the IC_1 dictionary are used.

frpm solver only

If this is set to TRUE, there will be no solution of the acoustic (APE-4) equations, only the FRPM solver will be run. This allows you to examine the nearfield FRPM predicted flow field, without the necessity of running a full CAA simulation.

microphone output frequency

This is after how many time steps the next solution at the microphone locations should be written to file.

Note: Note this is number of cycles rather than Hz.

frpm sources

If this is set to TRUE, the acoustic sources driving the acoustic simulation will come from the FRPM method. If this is set to FALSE, sources will be supplied by the user elsewhere, for example, velocity fluctuations on a surface.

number of smallest timestep locations to display

The solver will report this number of cells with the smallest time steps.

Note: The time step utilised by the solver is limited by the smallest (or poorest quality) cells in the mesh. This keyword allows the user to see the X,Y,Z location of the cells which are restricting the time step and so can be useful for mesh improvement purposes.

rans case name

Specifies the zCFD results file to provide turbulent mean flow data to both the FRPM and CAA solver if required.

rans mesh name

Specifies the zCFD mesh file to provide turbulent mean flow data to both the FRPM and CAA solver if required.

FRPM Domains

Several FRPM domains can be defined within the aero-acoustic simulation by adding additional 'FRPM_2', 'FRPM_3', etc., dictionaries. With a single FRPM domain, the dictionary must be labelled 'FRPM_1'. Each dictionary can contain the following parameters:

| Keyword | Required | Default | Valid values |
|---|----------|--------------------------------------|--|
| 'map cfd to frpm mesh' | Yes | True | True False |
| 'number of frpm mpi processes' | Yes | 1 | Positive integer |
| 'frpm spacing' | Yes | | Positive float |
| 'frpm turbulence integral length scale' | Yes | | Positive float |
| 'frpm maximum variable turbulence integral length dx scale' | Yes | 7 | Positive float |
| 'frpm turbulence integral length scale parameter' | Yes | 1 | Positive float |
| 'frpm cart num mesh cells' | Yes | | [x, y, z]: vector of integers or function |
| 'frpm domain translate' | Yes | | [x, y, z]: vector of integers or function |
| 'frpm domain rotate (deg)' | Yes | | [x, y, z]: vector of integers or function |
| 'frpm blend sources from side' | Yes | | []: list of floats |
| 'frpm monitor points' | No | | [x1, y1, z1, x2, y2, z2...]: list of X, Y, Z coordinates |
| 'frpm flush particles cycles' | Yes | 0 | Positive integer |
| 'frpm mapped tke smoothing iterations' | Yes | 0 | Positive integer |
| 'frpm mapped tke smoothing relaxation' | Yes | 0.2 | Float between 0 & 1 |
| 'frpm inverse distance source mapping' | Yes | True | True False |
| 'frpm inverse distance source mapping' | Yes | True | True False |
| 'minimum particle speed' | Yes | 0.0000000001 | Positive float |
| 'use constant integral length scale' | Yes | True | True False |
| 'wall distance blend over' | Yes | -1 | Float |
| 'frpm anisotropy' | Yes | -1 | Float |
| 'frpm inverse distance source mapping distance' | No | | Float |
| 'frpm inflow boundaries' | No | [True, True, True, True, True, True] | List of booleans |
| 'frpm minimum turbulence integral length scale' | No | | Float |

Example usage

```
parameters = {
    ...
    'equations' : 'DGCAA',
    'DGCAA': {
        ...
        "FRPM_1": {
            "number of frpm mpi processes": 1,
            "map cfd to frpm mesh": True,
            "frpm Spacing": 0.1,
            "frpm turbulence integral length scale": 0.2,
            "frpm cart num mesh cells": [100, 100, 100],
            "frpm march frequency": [0, 0, 0],
            "frpm domain translate": [0, 0, 0],
            "frpm domain rotate (Deg)": [0, 0, 0],
            "frpm blend sources From side": [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1],
            "frpm monitor points": [0.0, 0.0, 0.0],
            "frpm flush particle cycles": 2500,
            "frpm mapped tke smoothing iterations": 5,
            "frpm mapped tke smoothing relaxation": 0.2,
            "frpm inverse distance source mapping": True,
            "use constant integral length scale": False,
            "frpm turbulence integral length scale parameter": 0.61,
            "frpm anisotropy": 3.0,
        },
        ...
    },
    ...
}
```

map cfd to frpm mesh

Load turbulence information from a CFD RANS solution (see ‘[Map CFD to CAA mesh](#)’), if this is set to FALSE, the freestream properties supplied within the IC_1 dictionary are used.

number of frpm mpi processes

Specifies the number of host CPUs to be used by the FRPM solver.

If the FRPM solver is slowing down the overall CAA computation, the user may wish to increase the parallelism by adding more host CPUs to the FRPM computation. Ideally this should be kept as low as possible as the FRPM method does not naturally scale well with increased CPUs.

frpm spacing

Specifies the cartesian cell dimensions for the FRPM mesh which is automatically generated by the FRPM solver.

This is required to specify the FRPM mesh used in the FRPM computations. This dimension will limit the resolved turbulence integral length scale to $l_{t_{resolved}} = \sqrt{l_t^2 + (0.75 * \Delta)^2}$ where Δ is the frpm spacing.

frpm turbulence integral length scale

If the user chooses to use a constant integral length scale for the FRPM source predictions, this value will be used.

This value currently requires being set, however is only used if '**use constant integral length scale**' is set to TRUE. A constant turbulence integral length scale permits the use of numerically efficient (fast) filtering operations and considerably reduces the FRPM run times. If a constant turbulence length scale provides a reasonable estimate of the sound producing length scales, this parameter should be used in conjunction with '**use constant integral length scale**' set to TRUE.

frpm maximum variable turbulence integral length dx scale

Limits the CFD predicted turbulence integral length scale to this value multiplied by the '**frpm spacing**'.

If the FRPM solver is to use the turbulence integral length scales provided by a CFD solution, it may be necessary to limit the maximum integral length scale within the FRPM solver. This is to prevent filtering operations which extend over the entire FRPM domain, for each FRPM node, which rapidly becomes expensive.

frpm turbulence integral length scale parameter

Specifies the constant of proportionality for the turbulence length scale calculation from the CFD simulation. The turbulence integral length scale is specified from turbulence kinetic energy and turbulence eddy frequency using the following expression $l_t = C_l * \sqrt{k}/(\beta * \omega)$. This value specifies C_l .

Note: 0.61 to 1 are the values specified within the literature

frpm cart num mesh cells

Specifies the number of Cartesian cells in each direction.

This vector of integers specifies how many cells are in each Cartesian direction, effectively specifying the extent of the FRPM domain.

frpm march frequency

Specifies after how many time steps the FRPM solver should be updated.

The CAA solver time step may be much smaller than that necessary for the FRPM solver, therefore to reduce the FRPM overhead, we can perform fewer FRPM updates and interpolate at time levels in between FRPM updates.

Note: If this is set to -1 the FRPM solver will compute this update frequency based on particles not traversing a cell.

frpm domain translate

Specifies the X, Y, Z translation to place the FRPM domain in the desired location.

The FRPM domain by default is cartesian aligned and placed at the origin. This parameter allows the user to place the FRPM domain anywhere in the domain. Note that translation happens AFTER rotation of the domain.

frpm domain rotate (deg)

Specifies the X, Y, Z rotations to place the FRPM domain in the desired location.

The FRPM domain by default is cartesian aligned and placed at the origin. This parameter allows the user to rotate the FRPM domain into the desired orientation PRIOR to translation of the domain.

frpm blend sources from side

Acoustic sources will be blended to zero strength across this distance from the interior of the FRPM domain, towards each of the cartesian boundaries.

It is undesirable to have acoustic sources appearing to vanish within the CAA solver, therefore it is necessary to blend sources smoothly to zero strength. This is done via a shifted Cosine function where the value is zero at the boundaries and 1 the specified distance into the FRPM domain (when considered in its original Cartesian alignment). The ordering is distance from [XMIN, XMAX, YMIN, YMAX, ZMIN, ZMAX]. If any of XMIN, XMAX, etc are set to 0, no blending is done.

frpm monitor points

Contiguous list of X,Y,Z locations specifying monitor points where data such as velocity and source fluctuations should be output from the FRPM solver.

It may be useful to the user to interrogate the data being produced by the FRPM solver to monitor statistics such as mean turbulence kinetic energy, turbulence spectra (using the frozen turbulence approximation), etc.

frpm flush particles cycles

Specifies number of FRPM cycles to run prior to starting an aero-acoustic calculation.

During initialisation, particles are placed randomly within active areas in the FRPM domain. The user can perform a specified number of particle marches to essentially flush particles through the domain such that the distribution of particles is as would be achieved after running for a period of time. No filtering operations are applied, simply particle tracking.

frpm mapped tke smoothing iterations

Number of smoothing iterations for turbulence kinetic energy field.

When a CFD solution is mapped to the FRPM domain, there may be a considerable difference in local cell sizes. As such, the mapped solution may show mild discontinuities. In order to prevent this causing an issue with numerical differentiations during the FRPM computations, the mapped solution can be smoothed using a relaxed Lagrangian smoothing method. This parameter specifies the number of smoothing cycles.

It should be noted that peak turbulence levels may reduce and smear our as a result of the smoothing process so the user should take care to ensure the smoothed turbulence field is as representative of the actual CFD solution as possible.

frpm mapped tke smoothing relaxation

Relaxation during smoothing iterations for turbulence kinetic energy field. This parameter specifies the relaxation during the Lagrange smoothing steps specified by '**frpm mapped tke smoothing iterations**'

frpm inverse distance source mapping

Use inverse distance weighing mapping to map acoustic sources from the FRPM domain to the CAA mesh.

There are two options to map acoustic sources from the FRPM domain to the CAA mesh. The first is using a modified Shepard inverse distance weighted mapping (recommended) or simply locate the CAA mesh point in the FRPM domain and interpolate the solution to that point ('**frpm inverse distance source mapping**' set to FALSE). The inverse distance mapping has been observed to exhibit minimal spurious numerical noise at high frequencies.

minimum particle speed

The minimum local flow speed of a particle before it is deleted and reseeded at an inflow boundary.

It is possible, as a result of numerical integration, that a particle may enter an almost dead zone with very negligible flow velocity. It may be desirable to delete the particle and reseed the particle at an inflow boundary to prevent the build up of particles in stagnant areas (or areas which are inactive). This should only be used if issues with large numbers of stationary particles are encountered in regions of the FRPM domain.

use constant integral length scale

Defines whether or not a constant, user specified, turbulence integral length scale should be used in the FRPM computations.

A constant turbulence integral length scale permits the use of numerically efficient (fast) filtering operations and considerably reduces the FRPM run times. If a constant turbulence length scale provides a reasonable estimate of the sound producing length scales, this parameter should be set to True. If this parameter is set to False, the turbulence integral length scale varies throughout the domain and is calculated from the CFD solution.

wall distance blend over

Defines the distance from walls the acoustic sources should blend to zero.

It is undesirable to have acoustic sources appearing to vanish within the CAA solver, therefore it is necessary to blend sources smoothly to zero strength. This is done via a shifted Cosine function where the value is 1 at a distance of this value from a wall, then blends to zero over this value such that it is zero two times this value from the wall.

frpm anisotropy

Defines anisotropy factor for acoustic sources generated by the FRPM method.

If the turbulence field to be modelled is anisotropic, it is possible to specify anisotropy with this parameter. The streamwise integral length scale will be scaled by $\sqrt{anisotropy}$ and the length scale normal to the streamwise direction will be scaled by $1.0/anisotropy$.

frpm inverse distance source mapping distance

Specifies the radial distance from a target CAA mesh point that source data from the FRPM domain should be taken from.

This parameter defines a sphere around the CAA mesh point where we wish to map acoustic data to, where any FRPM point within will contribute to the inverse distance weighted mapping. By default this is set to 3 times to FRPM mesh spacing, however can be increased or decreased if necessary (due to memory limitations for example).

frpm inflow boundaries

List of booleans defining the Cartesian FRPM domain boundaries in which deleted particles can be reseeded into the domain.

If an FRPM particle exits the domain, it is reseeded at one of the FRPM domain boundaries. This list of booleans allows the user to set which boundaries particles can be reseeded. By default, reseeding can happen at all boundaries, however some can be turned off if desired. The booleans are ordered XMIN, XMAX, YMIN, YMAX, ZMIN, ZMAX.

For example, *[True, False, True, False, False, False]* will have particles reseeded at only the XMIN and YMIN boundaries.

frpm minimum turbulence integral length scale

Specifies the minimum turbulence integral length scale permitted when using the turbulence integral length scales predicted by a CFD simulation.

This parameter specifies the minimum turbulence integral length scale permitted, when the turbulence integral length scales being used are those from a CFD simulation. It is highly recommended to set this value to twice the FRPM spacing. Any value below this will likely result in the sources being poorly resolved in the FRPM domain and result in spurious high frequency noise.

Sponge Layers

This option can be used to permit non reflecting farfield boundaries. When set, damping terms will be linearly ramped from zero at this distance from farfield boundaries, to the highest value (defined by ‘damping factor’) at the farfield boundary.

| Keyword | Required | Default | Valid values |
|------------------|----------|---------|--------------|
| ‘distance’ | No | – | Number |
| ‘damping factor’ | No | – | Number |

‘distance’ specifies the distance from farfield boundaries in which the solution should be damped towards freestream values. ‘damping factor’ specifies the amount of damping to be applied at cells close to farfield boundaries.

Example usage

```
parameters = {
    ...
    'equations' : 'DGCAA',
    'DGCAA': {...},
    "sponge layer damping": {
        "distance": 1.0,
        "damping factor": 1.0,
    },
    ...
}
```

1.5.7 Fluid-structure interaction

zCFD has the capability of simulating Fluid Structure Interaction (FSI) using both a modal model, as well as any generic full FEA program through the use of our generic coupling scheme.

| Keyword | Re- quire | Default | Valid values | Description |
|------------------------|--------------|----------------|--|--|
| 'type' | Yes | | 'abaqus' 'generic' | FSI driver to be used when coupling to an FEA code |
| 'zone' | Yes | | List of zone ID integers | Which boundaries within the aerodynamic mesh are moving structural boundaries |
| 'fixed zones' | Yes | | List of zone ID integers | Which boundaries should not be moved by any mesh deformation |
| 'transform type' | Yes | | 'rbf multiscale' 'rbf' 'IDW' | Interpolation method to be used. |
| 'alpha' | Yes | 1 | Float | Support radius used by RBF |
| 'basis' | Yes | 'c2' | 'c2' 'c4' 'c0' 'tps' 'gaussian' | Which basis function to use for RBF |
| 'base point fraction' | Yes | 0.1 | Float | Number of points to use in the reduced base set for rbf multiscale interpolation |
| 'tol' | Yes | 0.01 | Float | Set the target error tolerance for the rbf interpolation |
| 'max error' | Yes | 0.1 | Float | Set the maximum initial error tolerance for rbf interpolation |
| 'n nearest' | Yes | 100 | Positive integer | Number of points to be used in IDW interpolation |
| 'deformation distance' | Yes | 1 | Float | Set the maximum support radius for IDW interpolation |
| 'blending stiffness' | Yes | 0.5 | Float | Set the blending stiffness for the IDW interpolation |
| 'power' | Yes | [3.0,5.0,10.0] | Set the IDW power for the nearfield and farfield boundaries, and the power between them. | List of Float |
| 'user variables' | No | {} | Dict of variables to values | Custom parameters passed into the generic FSI code |

type

Specifies which FSI driver should be used when coupling to an FEA code, ‘**abaqus**’ or ‘**generic**’. This tells zCFD which FSI python file should be used to execute the FSI coupling:

‘**generic**’ will use the **genericFSI.py** interface that should be implemented to meet your requirements. Out the box **genericFSI.py** will not perform any changes.

‘**abaqus**’ is an extension of generic interface that allows the solver to run in tandem with the SIMULIA CSE engine.

Note: Abaqus requires the CSE engine to be built with zCFD

zone

zone should be a list of zone IDs, these specify which surface is the moving FSI surface. This provides methods to extract surface normals, pressure forces, and cell centres from the specified cells, as well as allowing these surface points to drive mesh deformations in the volume mesh.

For RBF, this must include any static zones, where the motion of a static zone is set to 0.

fixed zones

fixed zones should be a list of zone IDs, these specify surfaces which must remain fixed, such as an interface, periodic boundary, or symmetry plane, these surfaces will remain untouched by the IDW mesh motion interpolation.

Note: IDW only

transform type

In FSI simulations the active surface defined in the ‘zone’ block, will move relative to the rest of the mesh. In order to maintain a conformal volume mesh, these displacements must be interpolated into the volume mesh. **transform type** defines which interpolation scheme should be used.

| Setting | Notes |
|---------------------------|---|
| ‘ rbf ’ | Radial Basis Functions require the solving of a linear system, where a continuous function is fitted between all points which defines the motion. This function can then be evaluated at other points within the domain to get the interpolated values. |
| ‘ rbf multiscale ’ | RBF multiscale uses the a multiscale algorithm using a varying support radius to reduce the number of nodes in the linear system, whereas RBF uses a Greedy algorithm. |
| ‘ IDW ’ | Inverse Distance Weighting does not require the solving of a linear system, and instead represents a function using a weighted sum of nearby function values. |

Note: ‘**rbf multiscale**’ has generally superseded the greedy algorithm, used in ‘**rbf**’, in terms of both speed performance and accuracy, so is the preferred RBF option.

alpha

Only applies to '**rbf**' and '**rbf multiscale**' interpolation schemes.

'**alpha**' sets the support radius used by the RBF. The Radial Basis Function used in RBF interpolation will have a set support radius defining the 1D shape of the function as a function of r . This determines the region of influence for the function. Compact functions such as C0, C2, C4 have a maximum radius, outside of which the influence of the RBF is 0. For functions with global influence such as Gaussian and TPS, alpha becomes the normalisation factor - i.e. when $r = \text{alpha}$, $e = 1$ if $\text{rbf} = f(e)$.

'**alpha**' should be larger than the maximum anticipated deformation.

basis

Only applies to '**rbf**' and '**rbf multiscale**' interpolation schemes.

When performing RBF interpolation a specific function needs to be defined to create the interpolant. Compact functions such as C0, C2, C4 have a maximum region of influence, Gaussian and TPS have global influence.

Note: For 99% of applications C2 works best

base point fraction

Only applies to '**rbf**' and '**rbf multiscale**' interpolation schemes.

Solving the RBF system requires solving an $N \times N$ sized linear system, where N is the number of moving nodes-i.e. nodes on the surface [4] defined earlier. For large meshes this quickly becomes prohibitively expensive, therefor it is desirable to solve a reduced set of points. The Multiscale method is a 'reduced base set method' where the linear system is only solved on the base set of nodes, with the remaining nodes solved iteratively. '**base point fraction**' dictates how many nodes of the full set should be included in the base set. Setting the base point fraction to 1.0 returns a standard full RBF.

tol

Only applies to '**rbf**' interpolation scheme.

At each point selection step during Greedy selection, the interpolation problem is solved on the current base set of data. From this the error between the exact solution and interpolated solution for all points in the mesh is identified, this value is used as the initial guess for the 'worst' error within the mesh. Once the total error is below this tolerance, the greedy point selection is complete.

max error

Only applies to '**rbf**' interpolation scheme.

At each point selection step during Greedy selection, the interpolation problem is solved on the current base set of data. From this the error between the exact solution and interpolated solution for all points in the mesh is identified. If this error is greater than the max error specified here, an error message is thrown.

n nearest

Only applies to ‘**IDW**’ interpolation scheme.

When using IDW mapping, the deformation of each surface point is mapped onto a set number of volume points, with the value determined by the weighted inverse distance between the donor and the recipient.

deformation distance

Only applies to ‘**IDW**’ interpolation scheme.

Sets the maximum region of influence for the inverse distance weighting mapping. If there are fewer than “n” neighbours within a distance “d”, then only neighbours within that distance are used.

blending stiffness

Only applies to ‘**IDW**’ interpolation scheme.

Set the blending stiffness for the IDW interpolation.

power

Only applies to ‘**IDW**’ interpolation scheme.

Set the IDW power for the nearfield and farfield boundaries, and the power between them.

user variables

If you had additional variables for time marching scheme or FSI control (e.g. file names) you may want to pass as variables to the **genericFSI.py** script, these should be included in this dictionary.

1.5.8 Material Specification

This dictionary sets the material properties of the fluid being modelled. The default values are those of air, but the dictionary can be modified to model any ideal, compressible, perfect gas.

| Keyword | Re-required | Default | Valid values |
|------------------------|-------------|---------|--------------|
| ‘sutherlands const’ | Yes | 110.4 | Float |
| ‘prandtl no’ | Yes | 0.72 | Float |
| ‘gas constant’ | Yes | 287 | Float |
| ‘gamma’ | Yes | 1.4 | Float |
| ‘turbulent prandtl no’ | Yes | 0.9 | Float |
| ‘gravity’ | No | | Vector |

Example usage:

```
parameters = {
..
'material' : 'air',
'air' : {
    'gamma' : 1.4,
    'gas constant' : 287.0,
    'Sutherlands const': 110.4,
```

(continues on next page)

(continued from previous page)

```
'Prandtl No' : 0.72,
'Turbulent Prandtl No' : 0.9,
},
..
}
```

sutherlands const

This option controls the variation of the viscosity of the fluid with temperature.

Sutherland's law: $\mu = \mu_{ref} \left(\frac{T}{T_{ref}} \right)^{3/2} \frac{T_{ref} + S}{T + S}$, where S is Sutherland's constant.

In zCFD, T_{ref} and μ_{ref} are given by the [reference](#) initial condition dictionary, where T is specified directly and μ is specified either directly via 'viscosity' or indirectly via 'reynolds no'.

Typically for air $\mu_{ref} = 1.716e-5$ and $T_{ref} = 273.15$

For more information: https://www.cfd-online.com/Wiki/Sutherland%27s_law

prandtl no

This option sets the Prandtl number, a non-dimensional number which is the ratio of momentum and thermal diffusivities in a fluid. In zCFD, where μ and c_p are already set, setting the Prandtl number has the effect of setting the thermal conductivity, k , of the fluid.

$$\text{Pr} = \frac{\text{momentum diffusivity}}{\text{thermal diffusivity}} = \frac{c_p \mu}{k}$$

This can also be considered as $\text{Pr} = \frac{\text{Thickness of thermal boundary layer}}{\text{Thickness of fluid boundary layer}}$

gas constant

This changes R , the gas constant of the fluid. Different fluids have different molecular masses, and therefore different gas constants. 287.1 is appropriate for air.

Constant pressure and constant volume specific heats, c_p and c_v , are set by γ and R according to the following relationships:

$$c_p = \frac{\gamma - 1}{\gamma} R$$

$$\gamma = \frac{c_p}{c_v}$$

gamma

This changes γ , the ratio of specific heats of the fluid. 1.4 is appropriate for air, but a value of 1.33 is often used for combustion exhaust gases.

turbulent prandtl no

This changes the turbulent Prandtl number of the fluid being modelled.

In simulations which model turbulence viscosity (RANS, LES and other scale resolving simulations), the eddy viscosity, μ_t , accounts for the additional diffusivity of momentum due to turbulence relative to the equivalent laminar flow. There is likewise additional diffusivity of heat due to turbulence relative to the equivalent laminar flow, and the turbulent prandtl number accounts for this.

In an eddy viscosity based simulation, $Pr_t = \frac{c_p \mu_t}{k_t}$

gravity

This sets the direction in which gravity acts in your simulation. For most industrial and aerospace flows, buoyancy effects are not important and therefore buoyancy should not be set. However, certain flows (e.g. plumes) do require buoyancy effects for accurate modelling.

gravity is specified as x,y,z vector, for example “[0,0,-9.81]”

1.5.9 Initial Conditions

An initial condition dictionary defines a fluid state. They are used to provide a reference initial state for the solution as well as specific conditions that can be assigned to *boundary conditions*.

Initial condition properties are defined using consecutively numbered blocks

```
'IC_1' : {....},
'IC_2' : {....},
```

By default zCFD will use the “IC_1” block to provide initial flow field conditions for the simulation but this can be changed with the *initial* parameter.

In addition to fully describing a set of conditions, zCFD allows you to specify *reference conditions* to be specified relative to a base set. This is typically used to set conditions for flow entering (inflow) or leaving (outflow) the domain.

Another way of prescribing an initial condition is by defining a dynamic *driving function*, which on evaluation returns an initial condition dictionary.

| Keyword | Re-required | Default | Valid values | Description |
|-------------------------------------|--------------------------------------|---------|--------------|---|
| 'pressure' | Yes | | Float | Static pressure in Pascals. Used to compute density and set Total Energy in the Compressible solver. |
| 'temperature' | Yes | | Float | Static temperature in Kelvin. Used to compute density and for dynamic viscosity scaling using Sutherlands law. |
| 'V' | No | | Dict | See Velocity |
| 'reynolds no' | When 'viscosity' not set. | | Float | Used to calculate viscosity at supplied temperature |
| 'reference length' | No | 1 | Float | Reference length to be used to calculate viscosity from the Reynolds number |
| 'viscosity' | When 'reynolds no' not set. | | Float | Sets the dynamic viscosity at the temperature with scaling provided by Sutherlands law defined in material section. |
| 'turbulence intensity' | No | | Float | Turbulence intensity (%) |
| 'eddy viscosity ratio' | No | | Float | Ratio of dynamic viscosity to turbulent viscosity used to set the turbulence quantities |
| 'ambient turbulence in- tensity' | No | | Float | Value of background turbulence intensity (%) to maintain in fluid |
| 'ambient eddy viscosity ra- tio' | No | | Float | Value of background turbulence eddy viscosity ratio to maintain in fluid |
| 'profile' | No | | Dict | See Velocity Profile |

Example usage:

```
parameters = {
  ...
  "IC_1": {
    "V": {"vector": [0.0, 2.5, 0.0]},
    "ambient eddy viscosity ratio": 1e-20,
    "ambient turbulence intensity": 1e-20,
    "eddy viscosity ratio": 0.1,
    "pressure": 101325.0,
    "temperature": 273.15,
    "turbulence intensity": 0.01,
    "viscosity": 1.79e-05,
  },
}
```

Velocity

Velocity can be specified in two ways, either a vector or a vector and a Mach number. When Mach number is specified the velocity direction is provided by the vector and the velocity magnitude is computed from the Mach number and speed of sound at the pressure and temperature provided.

For example to set a velocity of 50 mesh units/sec in the X direction:

```
...
"V" : {[50.0,0.0,0.0]},  
...
```

Or to set a velocity of mach 0.2 in the X direction:

```
...
"V" : {[1.0,0.0,0.0]}, "mach": 0.2}  
...
```

Viscosity

Dynamic (shear, absolute or molecular) viscosity should be defined at the static temperature previously specified. This can be specified either as a dimensional quantity or by a Reynolds number and reference length

```
# Dynamic viscosity in dimensional units  
'viscosity' : 1.83e-5,
```

or

```
# Reynolds number  
'reynolds No' : 5.0e6,  
# Reference length  
'reference length' : 1.0,
```

Note: Reynolds number is defined as: $Re = \frac{\text{density} * V * \text{referencelength}}{\text{viscosity}}$

Turbulence intensity & Eddy viscosity

Turbulence intensity is defined as the ratio of velocity fluctuations u' to the mean flow velocity. A turbulence intensity of 1% is considered low and greater than 10% is considered high.

```
# Turbulence intensity %  
'turbulence intensity': 0.01,  
# Turbulence intensity for sustainment %  
'ambient turbulence intensity': 0.01,
```

The eddy viscosity ratio (μ_t/μ) varies depending type of flow. For external flows this ratio varies from 0.1 to 1 (wind tunnel 1 to 10)

For internal flows there is greater dependence on Reynolds number as the largest eddies in the flow are limited by the characteristic lengths of the geometry (e.g. The height of the channel or diameter of the pipe). Typical values are:

| Re | 3000 | 5000 | 10,000 | 15,000 | 20,000 | > 100,000 |
|------|------|------|--------|--------|--------|-----------|
| eddy | 11.6 | 16.5 | 26.7 | 34.0 | 50.1 | 100 |

```
# Eddy viscosity ratio
'eddy viscosity ratio': 0.1,
# Eddy viscosity ratio for sustainment
'ambient eddy viscosity ratio': 0.1,
```

Velocity Profile

The user can also provide functions to specify a ‘wall-function’ - or the turbulence viscosity profile near a boundary.

| Keyword | Re-required | Default | Valid values | Description |
|---------------------|-------------|-----------|--------------|--|
| ‘abl’ | No | Dict | | Specify an <i>ABL</i> (Atmospheric Boundary Layer) |
| <i>field</i> | No | File name | | Specify a field of data that is used as a lookup for setting the flow conditions. |
| ‘use wall distance’ | No | False | True/False | Use wall distance in place of z coordinate in field during interpolation process. This can be used in setting the profile of flows on a terrain mesh with a variable ground location at the boundary |

field

The file specified using the ‘field’ parameter should be in the ParaView/VTK VTP format. It should contain a node array with one or more of the following array names: ‘Pressure’, ‘Temperature’, ‘Velocity’, ‘TI’ and ‘EddyViscosity’. zCFD will then look up those values in the solution by looking up the nearest point in the VTP file.

```
..
'profile' : {
    'field' : 'inflow_field.vtp',
    # Localise field using wall distance rather than z coordinate
    'use wall distance' : True,
},
..
```

Note: Note the field will override the conditions specified previously therefore the user can specify only the conditions that are different from default.

ABL

| Keyword | Re-required | Default | Valid values | Description |
|------------------------|-------------|---------|--------------|---|
| 'roughness length' | Yes | | Float | It is a measure of the height at which the mean velocity of the wind becomes zero, due to the frictional effects between the air and the surface. |
| 'friction velocity' | Yes | | Float | Specify a field of data that is used as a lookup for setting the flow conditions. |
| 'surface layer height' | No | | Float | The surface layer height, also known as the aerodynamic or boundary layer height. |
| 'monin-obukhov length' | No | | Float | The Monin-Obukhov length. |
| 'tke' | No | | Float | Set the turbulent kinetic energy. |
| 'z0' | No | | Float | Set the z location of the ground rather than using the z coordinate in the mesh. |
| 'up' | No | [0,0,1] | Vector | Direction of gravity |

roughness length

Is a measure of the height at which the mean velocity of the wind becomes zero, due to the frictional effects between the air and the surface.

The roughness length is typically denoted by the symbol “ z_o ” and is defined as the vertical distance between the Earth’s surface and the height at which the logarithmic wind profile intersects the surface. The logarithmic wind profile is a mathematical relationship that describes the variation of wind speed with height in the atmospheric boundary layer.

The roughness length depends on the surface characteristics of the terrain or objects present on the Earth’s surface. Different surfaces, such as forests, urban areas, or bodies of water, have different roughness lengths. For example, a smooth surface like open water would have a small roughness length, while a densely forested area would have a larger roughness length.

friction velocity

Friction velocity, often denoted as “ u_* ” (pronounced “u-star”), is another important parameter in the study of atmospheric boundary layers. It represents the characteristic velocity of turbulent motions in the surface layer of the atmosphere, generated by the shear stress between the air and the Earth’s surface.

Friction velocity is defined as the square root of the shear stress divided by the air density:

$$u_* = \sqrt{\frac{\tau}{\rho}}$$

where:

- u_* is the friction velocity,
- τ is the shear stress between the air and the surface, and
- ρ is the density of the air.

The shear stress, τ , arises from the momentum transfer between the moving air and the surface roughness elements. It is related to the wind speed and the roughness length through a logarithmic wind profile, described by the following equation:

$$u(z) = (\frac{u_*}{\kappa}) * \ln \frac{z}{z_0}$$

where:

- $u(z)$ is the wind speed at height z above the surface,
- κ is the von Kármán constant (approximately 0.4), and
- z_0 is the roughness length.

The friction velocity provides a measure of the intensity of turbulence near the Earth's surface. It influences several atmospheric processes, including heat and moisture exchange, pollutant dispersion, and the formation of atmospheric boundary layer structures. The magnitude of the friction velocity depends on surface roughness, wind speed, and atmospheric stability, among other factors.

surface layer height

The surface layer height, also known as the aerodynamic or boundary layer height, refers to the vertical extent of the atmospheric boundary layer near the Earth's surface. It represents the region where the physical properties, such as wind speed, temperature, humidity, and turbulence, are influenced by the underlying surface. The height of the surface layer is often estimated based on empirical relationships or atmospheric stability conditions. It can vary from a few meters to a few hundred meters above the ground, depending on factors such as surface roughness length, wind speed, and atmospheric stability. Additionally, the surface layer height can change diurnally, with variations due to daytime heating and nighttime cooling effects.

monin-obukhov length

The Monin-Obukhov length, often denoted as "L", is a parameter used in the study of atmospheric boundary layers to characterise the stability of the air near the Earth's surface. It is named after the Russian scientists A. S. Monin and A. M. Obukhov, who made significant contributions to the understanding of turbulence and boundary layer physics.

The Monin-Obukhov length is defined as the ratio of the potential temperature difference to the buoyancy flux. It is given by the following equation:

$$L = -\frac{u^*^3 * \theta_v}{\kappa * g * w\theta}$$

where:

- L is the Monin-Obukhov length,
- u^* is the friction velocity,
- θ_v is the virtual potential temperature,
- κ is the von Kármán constant (approximately 0.4),
- g is the acceleration due to gravity, and
- $w\theta$ is the vertical flux of virtual potential temperature.

The Monin-Obukhov length is an indicator of atmospheric stability. It quantifies the relationship between the mechanical production of turbulence due to wind shear and the buoyancy effects caused by vertical temperature gradients. Positive values of L indicate stable atmospheric conditions, while negative values indicate unstable conditions. A neutral atmosphere corresponds to $L = 0$.

The Monin-Obukhov length influences various atmospheric processes, such as turbulence structure, heat and moisture exchange, and dispersion of pollutants. It plays a significant role in determining the behaviour of the atmospheric boundary layer, including the vertical mixing of air masses and the development of turbulence. The value of L is influenced by factors such as surface properties, wind speed, temperature, and humidity gradients.

1.5.10 Reference Conditions

Reference Conditions enable a set of conditions to be specified relative to a base set of *initial conditions*. This is typically used to set conditions for flow entering (inflow) or leaving (outflow) the domain.

Reference condition properties are defined in the same way as initial conditions using consecutively numbered blocks.

```
'IC_2' : {....},
'IC_3' : {....},
```

Note: Reference conditions can not be used for "IC_1"

| Keyword | Re-required | Default | Valid values | Description |
|---------------------------|-------------|---------|---|--|
| 'reference' | Yes | | Float | Specify the base condition as the reference for this set of conditions |
| 'static pressure ratio' | No | | Float or <i>Function</i> or <i>File</i> | Set the static pressure using a ratio relative to the reference static pressure. Where static pressure = reference static pressure * static pressure ratio |
| 'total pressure ratio' | No | | Float or <i>Function</i> or <i>File</i> | Set the total pressure using a ratio relative to the reference total pressure. Where total pressure = reference total pressure * total pressure ratio |
| 'total temperature ratio' | No | | Float or <i>Function</i> or <i>File</i> | Set the total temperature using a ratio relative to the reference total temperature. Where total temperature = reference total temperature * total temperature ratio |
| 'mass flow ratio' | No | | Float | Specify a <i>mass flow</i> ratio. |
| 'mass flow rate' | No | | Float | Specify a <i>mass flow</i> rate. |
| 'vector' | No | | [X, Y, Z] or <i>Function</i> | Specify direction vector used to specify the direction of the flow for specific boundary conditions. |
| 'mach' | No | | Float | Specify a mach number used for specific boundary conditions like inflows/jets. |

Example usage:

```
parameters = {
..
'IC_2' : {
    'reference' : 'IC_1',
    'total temperature ratio' : 1.0,
    'mass flow rate' : 10.0,
    'vector' : [1,0,0],
    'mach' : 0.5,
},
```

(continues on next page)

(continued from previous page)

```
..  
},
```

Mass Flow

Mass flow ratio is defined as:

$$ratio = (density * area * V) / (density_{ref} * area * V_{ref}) = (density * V) / (density_{ref} * V_{ref})$$

Mass flow rate is defined as:

$$\dot{m} = density * area * V$$

Note: Note area is in mesh units squared

Using a file

A file can be used to provide a lookup for the pressure and temperature ratios. This file should be in the Paraview/VTK VTP format. The file should contain a surface with specific values as node arrays data. zCFD then looks up the values by finding the nearest point to the mesh location.

The node array data in the VTP should be named specifically for each keyword:

| Keyword | Node Array Name |
|---------------------------|-----------------------|
| 'static pressure ratio' | StaticPressureRatio |
| 'total pressure ratio' | TotalPressureRatio |
| 'total temperature ratio' | TotalTemperatureRatio |

1.5.11 Mesh Transformation

This dictionary is used to setup an initial deformation of the mesh. Typically a known deformation of a surface in the mesh is propagated out into the volume using either radial basis function or inverse distance weighted interpolation. Deforming an aircraft wing or wind turbine blade to a static load shape are two uses of this functionality.

| Keyword | Re- quire | Default | Valid values | Description |
|------------------------|--------------|----------|--|--|
| 'type' | Yes | | 'rbf' 'rbf multiscale' 'idw' | Selects the <i>interpolation</i> method used. |
| 'base point fraction' | No | | Float | Fraction of points included in the base set when using ' rbf multiscale ' |
| 'alpha' | No | | Float | Sets the support radius (the radius of influence) of the rbf surface points. |
| 'zone' | Yes | | List of zone ID integers | Sets a list of surface zones to deform |
| 'fixed zones' | No | | List of zone ID integers | Sets the surface zones which remain fixed in position when using ' idw ', useful if you have surfaces that are within the influence of the interpolation that should remain fixed. |
| 'func' | Yes | | <i>Transform Function</i> | A python function that describes the surface deformation to be propagated into the volume mesh. The function is called for each node on the surfaces specified in the 'zones' key. |
| 'basis' | No | | <i>c0</i> <i>c2</i> <i>tps</i> <i>gaussian</i> | Selects the type of radial basis function used. Can be Welland's <i>c0</i> or <i>c2</i> function, thin plate spline (<i>tps</i>) or Gaussian. |
| 'tol' | No | | Float | The maximum permissible distance error between the deformed surface shape and that obtained using the greedy ' rbf ' method. No more points will be added to the system once this tolerance is reached. |
| 'power' | No | [3,5,10] | List of int | A list of three exponents used in the ' idw ' interpolation scheme. |
| 'n nearest' | No | 100 | Int | The number of nodes closest to the current point used to determine the ' idw ' interpolation weight. |
| 'deformation distance' | No | 1 | Float | The distance over which the surface deformation is blended into the volume mesh. |
| 'blending stiffness' | No | 0.5 | Float | The stiffness of the blending function used to propagate the ' idw ' interpolation of the surface deformation into the volume mesh. |

Interpolation method

Selects the interpolation method used to propagate the surface deformation into the volume mesh. Three methods are available:

'rbf' uses the 'greedy' method of [Allan et al.](#) where the deformation is started with a subset of points and additional points are added until the surface deformation reaches the requested tolerance.

'rbf multiscale' uses the [multiscale](#) rbf method where a subset of surface points with a fixed support radius are solved implicitly and the remaining points are solved explicitly with a support radius determined as part of the solution process. The fraction of points included in the base set can be set with '**base point fraction**'. The base set of points are assigned the support radius '**alpha**' and hence can have a wider sphere of influence than the

remaining points which are used in the multiscale process.

Note: Increasing the number of points in the base set and/or increasing alpha generally makes the surface deformation propagate further into the volume mesh reducing the likelihood of generating poor quality cells

'idw' uses a simple inverse distance weighted interpolation. The `power` parameter can provide a list of three exponents used in the IDW scheme. The first (p1) is applied to the interpolation weight of points close to the surface, the second (p2) to points which are the deformation distance away from the surface and the third (p3) is used to blend between the two extremes.

$$wt_1 = (\text{deformationdistance}/\text{distance})^{p1} \quad wt_2 = (\text{deformationdistance}/\text{distance})^{p2} \quad \text{blend} = ((\text{deformationdistance} - \text{distance})/\text{deformationdistance})^{p3}$$

Then the weight is:

$$wt = (wt1 * (1.0 - \text{blend}) + \text{blend} * wt2)$$

Examples

RBF Example

```
import zutil

rotation = 10.0

def my_transform(x, y, z):
    v = [x, y, z]
    v = zutil.rotate_vector(v, rotation, 0.0)
    return {"v1": v[0], "v2": v[1], "v3": v[2]}

parameters = {
    ...
    "TR_1": {
        "type": "rbf",
        "zone": [4],
        "func": my_transform,
        "alpha": 200.0,
        "basis": "c2",
        "tol": 0.01,
    },
    ...
}
```

RBF Multiscale Example

```
import zutil

rotation = 10.0

def my_transform(x, y, z):
    v = [x, y, z]
    v = zutil.rotate_vector(v, rotation, 0.0)
    return {"v1": v[0], "v2": v[1], "v3": v[2]}
```

(continues on next page)

(continued from previous page)

```
parameters = {
    ...
    "TR_1": {
        "type": "rbf_multiscale",
        "base point fraction": 0.1,
        "zone": [4],
        "func": my_transform,
        "alpha": 200.0,
        "basis": "c2",
    },
    ...
}
```

IDW Example

```
import zutil

rotation = 10.0

def my_transform(x, y, z):
    v = [x, y, z]
    v = zutil.rotate_vector(v, rotation, 0.0)
    return {"v1": v[0], "v2": v[1], "v3": v[2]}

parameters = {
    ...
    "TR_1": {
        "type": "idw",
        "power": [4.0, 8.0, 10.0],
        "fixed zones": [1],
        "n nearest": 500,
        "deformation distance": 100.0,
        "blending stiffness": 0.5,
        "zone": [4],
        "func": my_transform,
    },
    ...
}
```

1.5.12 Overset

zCFD is capable of automatically oversetting multiple meshes into a single simulation. To run an overset case you need to provide an *override file*.

Override file

The override file tells zCFD which combinations of mesh and control dictionary to load and overset. The first entry in the list should be the background mesh, this is typically the largest of the meshes that the other meshes overlap with. The following entries in the list are then overset onto the background mesh when the solver starts. The meshes are overlapped in the order they appear in the list, starting with the background mesh.

The list should be defined in a separate python file containing an **override** dictionary.

Example **override** dictionary (e.g. `override.py`):

```
override = {
    'problem_case_pair': [
        ("background_mesh.h5", "background.py"),
        ("overset_1.h5", "overset_1.py"),
        ("overset_2.h5", "overset_2.py"),
    ]
}
```

Running an overset case:

```
(zCFD)> run_zcfд -f override.py
```

Mapping

The **mapping** dictionary sets additional mapping options for overset methods and should be defined in the background control file.

| Keyword | Re- quired | Default | Valid values | Description |
|--|---------------|---------|------------------|---|
| 'idw from cells order' | 0 | - | 0 or 1 | Sets the order of accuracy for IDW mapping |
| 'number of additional overset active layers' | Yes | 0 | Positive integer | Adds additional active cells to overlapped mesh |

Example usage:

```
parameters = {
    ..
    'mapping': {
        'idw from cells order': 0,
        'number of additional overset active layers': 0,
    }
    ..
}
```

idw from cells order

If ‘**idw from cells order**’ is set to 0, this provides a 0th order interpolation where we only utilise data from cell a mapped point is located within. If set to 1, a first order interpolation is achieved using an IDW interpolation, using the cell a point is found in, and its neighbours.

number of additional overset active layers

Adds additional active cells which may have been made inactive by an overlapping mesh. If for example, we have a rotating domain, the resolution of the rotating boundary may result in rotated overset mesh boundary points, entering cells made inactive. This will result in no mapping between background and overset meshes at these points. To resolve this issue, it is possible to add additional active cells (for safety) to ensure a mapping between overset and background meshes. Whilst not necessary, this parameter may be useful if issues with mapping between meshes occur.

1.5.13 Write Output

write output

Controls the frequency and contents of the files output into the the **output** directory and the **VISUALISATION** subdirectory.

The **output** directory will be created in the working directory from which the solver is run and is named using the following format:

```
<case_name>_P<number of mpi ranks>_OUTPUT
```

| Keyword | Required | Default | Valid values |
|----------------------------|----------|---------------------------------|-----------------------------------|
| ‘frequency’ | Yes | Positive integer or <i>None</i> | See “ <i>frequency</i> ” |
| ‘format’ | Yes | ‘vtk’ | ‘none’ ‘vtk’ |
| ‘scripts’ | No | [] | List of catalyst script filenames |
| ‘variable_name_alias’ | No | {} | Dict |
| ‘surface variables’ | Yes | [“p”, “T”, “rho”] | List of “ <i>variable</i> ” names |
| ‘volume variables’ | Yes | [...] | List of “ <i>variable</i> ” names |
| ‘volume interpolate’ | No | | List of surface file filenames |
| ‘fwh interpolate’ | No | | List of .stl file filenames |
| ‘fwh wall data’ | No | False | True False |
| ‘compute average and rms’ | No | False | True False |
| ‘average start time cycle’ | No | | Positive integer |

Example usage:

```
parameters = {
    ...
    'write output' : {
        'format' : 'vtk',
        'surface variables': ['V','p'],
        'volume variables' : ['V','p'],
        'volume interpolate' : ['mesh.stl'],
        'scripts' : ['paraview_catalyst1.py'],
        'variable_name_alias' : { "V" : "VELOCITY", },
        "frequency": 500,
        'calculate average and rms': True,
```

(continues on next page)

(continued from previous page)

```
'average start time cycle': 1000
},
..
}
```

format

Selects the file format to output the data in. Using ‘none’ as format will output no files. ‘vtk’ outputs in legacy vtk format for reading into [ParaView](#).

scripts

zCFD is capable of calling ParaView catalyst scripts while running, this allows you to generate “in situ” visualisation output while the solver is running.

This allows you to extract data as your simulation progresses without needing to store the full solution at every timestep. For instance a video of a slice through the dataset could be generated for an unsteady solution.

For more information about Catalyst see: <https://www.paraview.org/hpc-insitu/>

variable_name_alias

Maps the zCFD variable names to a different variable name in the output file. If downstream processes need variables named using a specific convention a naming alias dictionary can be supplied.

surface variables

A list of *variables* to output on the boundaries of the mesh. This is useful for visualisation of the flow variables on the wall of the mesh for post processing. For the ‘vtk’ format the surface data is written to the *output* directory with the file name:

```
<case_name>_<boundary>.pv
```

volume variables

A list of *variables* to output in a volumetric format.

For the ‘vtk’ format the volume data is written to the *output* directory with the file name:

```
<case_name>.pv
```

volume interpolate

zCFD can interpolate volume data to a supplied surface. This parameter lets you list the filenames of surfaces to interpolate to. The supported file formats are VTK (vtu or vtp files) and STL.

fwh interpolate

In addition to the ‘volume interpolate’ option (which outputs VTK files), zCFD is capable of writing volume interpolated data to HDF5 files for use in the *zCFD Ffowcs-Williams Hawkins (FWH) solver*. This parameters lets you list the filenames of surfaces to interpolate to for HDF5 file output. The files specifying the surfaces must be .STLs. The outputted files contain density, velocity and pressure data only, and are suitable for use as permeable surfaces in the zCFD FWH solver. The data is outputted to the *ACOUSTIC_DATA* sub-directory within the *output* directory.

fwh wall data

This parameter controls whether wall data is outputted to a HDF5 file for use in the *zCFD FWH solver*. The outputted file contains pressure data only, and is suitable for use as an impermeable surface in the zCFD FWH solver. The data is outputted to the *ACOUSTIC_DATA* sub-directory within the *output* directory.

compute average and rms

Enables computation of time averaged and RMS values for pressure, temperature and velocity. Enabling this adds these variables to the list of variables to be output automatically.

This options only applies to *unsteady* cases.

average start time cycle

If **compute average and rms** is True, this controls first real time cycle to start calculating the running average. This allows you to run the solver for a number of time cycles before you start calculating the average and RMS values.

Frequency

Sets how frequently the solver should output the flow solution. A single integer value will control output of both the volume and surface data output. A dictionary value here allows for fine grained control over the frequency with which each individual output is done. If the solver is running an unsteady simulation then this is the frequency of the real time step otherwise the pseudo cycle.

| Keyword | Required | Default | Valid values |
|----------------------------|----------|----------|------------------|
| ‘volume data’ | Yes | 1.00E+08 | Positive integer |
| ‘surface data’ | Yes | 1.00E+08 | Positive integer |
| ‘volume interpolate’ | Yes | 1.00E+08 | Positive integer |
| ‘fwh interpolate’ | Yes | 1.00E+08 | Positive integer |
| ‘fwh wall data’ | Yes | 1.00E+08 | Positive integer |
| ‘checkpoint’ | Yes | 1.00E+08 | Positive integer |
| ‘volume data start’ | Yes | 1 | Positive integer |
| ‘surface data start’ | Yes | 1 | Positive integer |
| ‘volume interpolate start’ | Yes | 1 | Positive integer |
| ‘fwh interpolate start’ | Yes | 1 | Positive integer |
| ‘fwh wall data start’ | Yes | 1 | Positive integer |
| ‘checkpoint start’ | Yes | 1 | Positive integer |

volume data, **surface data**, **volume interpolate**, **fwh interpolate**, **fwh wall data** and **checkpoint** control the frequency at which the solution data of the given type is written. The frequency is specified as the solver cycle number. The **checkpoint** option writes a *restartable* checkpoint of the flow solution at that point.

volume data start, **surface data start**, **volume interpolate start**, **fwf interpolate start**, **fwf wall data start** and **checkpoint start** specifies the first cycle that the solver should start writing that particular output.

Note: On systems with slower file I/O the overhead of writing data can slow the overall solve time, reducing the output frequency lets you address this.

Example usage:

```
parameters = {
    ...
    'write output' : {
        'format' : 'vtk',
        'surface variables': ['V','p'],
        'volume variables' : ['V','p'],
        "frequency": {
            # Output visualisation files every 3 cycles
            "volume data": 3,
            "surface data": 3,
            "volume interpolate": 3,
            "checkpoint": 3,
            # Start output every 3 cycles after 1 cycle (default = 1)
            "volume data start": 1
            "surface data start": 1,
            "volume interpolate start": 1,
            "checkpoint start": 1,
        },
    },
    ...
}
```

1.5.14 Reporting

zCFD has the capability to output certain parameters to a <casename>_report.csv file as the solver progresses. These can be used to monitor the solver's convergence. By default, the solver outputs volume integrated residuals for each solution variable to the <casename>_report.csv file, but adding to the report dictionary means other quantities of interest can also be monitored as the simulation progresses.

| Keyword | Re-required | Default | Valid values |
|-------------|-------------|---------|--|
| 'frequency' | Yes | 1 | Positive integer |
| 'monitor' | No | - | Dictionary of monitor point configs (MR_?) |
| 'forces' | No | - | Dictionary of force configs (FR_?) |
| 'massflow' | No | - | Dictionary of massflow configs (MF_?) |

'frequency' sets how frequently a new line is appended to the <casename>_report.csv dictionary. If frequency = x, the solver will output to the <casename>_report.csv every x solver cycles. For a dual time-stepping simulation, this is the inner (pseudo-time), not outer, cycles.

'monitor', 'forces' and 'massflow' should be dictionaries of the required format. The dictionary's keys should be consecutively numbered blocks prefixed with the correct string:

| Reporting on | Key name |
|--------------|----------|
| 'monitor' | MR_? |
| 'forces' | FR_? |
| 'massflow' | MF_? |

Example usage:

For a simulation with 1000's of cycles, it can be unnecessary to report every time cycle, which slows the solver down. For a dual timestepping simulation with 20 inner time cycles, outputting every 10 cycles generally gives enough information without substantially slowing the solver down

```
# Report every 10 cycles
'report' : {
    ...
    'frequency': 10
    ...
}
```

A simulation with two monitor points and one forces block.

In addition to standard flow field outputs (see below), zCFD can provide information at monitor points in the flow domain, and integrated forces across all parallel partitions, any number of 'MR_', 'FR_' or 'MF_' blocks may be specified.

```
# Report every 10 cycles
'report' : {
    ...
    'frequency': 10
    'monitor' : {
        'MR_1' : {
            ...
        },
        'MR_2' : {
            ...
        }
    }
    'forces' : {
        'FR_1' : {
            ...
        }
    }
    ...
}
```

Monitor Points

Monitor points report the specified variable at fixed mesh locations.

| Keyword | Re-required | Default | Valid values |
|-------------|-------------|---------|---|
| 'name' | Yes | - | String; Name of the monitor point |
| 'point' | No | - | [x,y,z] position vector specifying a valid location in the mesh |
| 'variables' | No | - | List of variables to report (see Output Variables) |

'name' sets the name that appears in the <casename>_report.csv column heading for variables from this monitor point.

'point' should be the x,y,z location of the point of interest in the mesh as it appears in the solver (i.e. after any mesh scaling due to parameters *scaling* parameters)

'variables' controls which variables are reported at this point. In the <casename>_report.csv the column name will be a combination of the **'name'** and the variable name.

Example usage:

Tracking pressure and velocity is useful for determining shedding cycle behaviour, or in the case of a scale resolving simulation, to determine the turbulence spectrum which is being resolved. It may be important to track other variables (e.g. turbulence intensity) may also be appropriate depending on the simulation setup.

```
'report' : {
    'frequency' : 10,
    'monitor' : {
        'MR_1' : {
            'name' : 'monitor_1',
            'point' : [180.0, 100.0, 50.0],
            'variables' : ['V', 'p', 'T', "Ti"],
        },
    },
},
```

Forces

Force reporting outputs the x,y,z total forces and moments on a surface or set of surfaces. These are presented as force coefficients (using the non-dimensionalisation convention used for aerofoil lift, drag and moment coefficients), as opposed to dimensional values. The force and moment coefficients resulting exclusively from viscous forces and exclusively from pressure forces are also given.

| Keyword | Required | Default | Valid values | Description |
|----------------------|----------|---------|--|--|
| 'name' | Yes | – | String | Sets the name that appears in the <casename>_report.csv column headings for variables from this force monitor. |
| 'zone' | Yes | – | List of valid mesh zone IDs (integers) | All the mesh zone IDs for the surfaces used in this force report. The forces and moments over all the zones are summed to give the forces and moments outputted in the force report. |
| 'transform' | No | – | An (optional) transform function. | See Transform function . A transformation function may be supplied to the force report block to transform the force into a different coordinate system. |
| 'reference area' | No | – | Positive number | Sets the dimensional reference area which is used to non-dimensionalise the reported force and moment coefficients |
| 'reference length' | No | – | Positive number | Sets the dimensional reference length which is used to non-dimensionalise the moment coefficients |
| 'reference point' | No | – | [x,y,z] position vector | Sets the reference point which is which is used to non-dimensionalise the moment coefficients |
| 'reference pressure' | No | – | Positive number | For a wetted surface, this sets the pressure on the non-wetted side of the surface. Pressure forces on the surface are calculated by integrating ($p - p_{\text{Ref}}$) over the surface |

Example usage:

```
'report' : {
    'frequency' : 10,
    # Report force coefficient in grid axis as well as using user defined
    ↵transform
    'forces' : {
        'FR_1' : {
            'name' : 'force_1',
            # Required: Zones to be included
            'zone' : [1, 2, 3],
            # Transformation function
            'transform' : my_transform,
            'reference area' : 0.112032,
            'reference length' : 1.0,
            'reference point' : [0.0, 0.0, 0.0],
            # Calculate forces using a specified reference pressure
            ↵rather than absolute
            'reference pressure' : 0.0
        },
    },
},
```

Output names

The forces will be written into the <casename>_report.csv prefixed with the supplied ‘name’. The forces will be written with the following names:

| Item | Reporting output name |
|---|---|
| Total forces and moments | ‘name’_F[x,y,z] and ‘name’_M[x,y,z] |
| Forces and moments due only to pressure forces | ‘name’_Fp[x,y,z] and ‘name’_Mp[x,y,z] |
| Forces and moments due only to viscous forces | ‘name’_Fv[x,y,z] and ‘name’_Mv[x,y,z] |
| Transformed total forces and moments (if transform function is set) | ‘name’_Ft[x,y,z] and ‘name’_Mt[x,y,z] |
| Transformed forces and moments due only to pressure forces (if transform function is set) | ‘name’_Ftp[x,y,z] and ‘name’_Mtp[x,y,z] |
| Transformed Forces and moments due only to viscous forces (if transform function is set) | ‘name’_Ftv[x,y,z] and ‘name’_Mtv[x,y,z] |
| Total forces and moments in static frame of reference (if specified zone is rotating) | ‘name’_Fif[x,y,z] and ‘name’_Mif[x,y,z] |
| Forces and moments due only to pressure forces in static frame of reference (if specified zone is rotating) | ‘name’_Fifp[x,y,z] and ‘name’_Mifp[x,y,z] |
| Forces and moments due only to viscous forces in static frame of reference (if specified zone is rotating) | ‘name’_Fifv[x,y,z] and ‘name’_Mifv[x,y,z] |

Additional information

Total force

Total force, \mathbf{F} , is simply, $\mathbf{F}_p + \mathbf{F}_v$, and likewise for moments.

Non-dimensionalisation

Forces are non-dimensionalised into force coefficients as $C_x = \frac{\mathbf{F}_x}{q_{inf}S}$ Where q_{inf} = reference dynamic pressure and S = reference area

Moments are non-dimensionalised into force coefficients as $C_{Mx} = \frac{\mathbf{M}_x}{q_{inf}Sc}$ Where c = reference length

Dimensional pressure force

$$\mathbf{F}_p = \oint - (p - p_{ref}) \hat{\mathbf{n}} dA$$

Where p is surface pressure and $\hat{\mathbf{n}}$ is the surface normal (pointing towards the wetted area).

Dimensional viscous force

$$\mathbf{F}_v = \oint \mu \nabla \cdot \mathbf{V} \, dA$$

Where μ is dynamic viscosity and \mathbf{V} is the velocity vector at the wall.

Dimensional pressure moment

$$\mathbf{M}_p = \oint -(p - p_{ref}) \hat{\mathbf{n}} \cdot (\mathbf{x} - \mathbf{x}_{ref}) \, dA$$

Dimensional viscous moment

$$\mathbf{M}_p = \oint \mu \nabla V \cdot (\mathbf{x} - \mathbf{x}_{ref}) \, dA$$

Note: Reference pressure p_{ref} is dynamic pressure ($\frac{1}{2}\rho V^2$), taken from the [reference state](#).

Mass flow

| Keyword | Required | Default | Valid values | Description |
|---------|----------|---------|--|--|
| 'name' | Yes | - | String | Sets the name that appears in the <casename>_report.csv column headings for variables from this force monitor. |
| 'zone' | Yes | - | List of valid mesh zone IDs (integers) | All the mesh zone IDs used in this mass flow monitor |

'name' Sets the name that appears in the <casename>_report.csv column heading for variables from this mass flow monitor. For example, setting this to 'pipe_outflow' would mean the massflow would appear as 'pipe_outflow_massflow' in the <casename>_report.csv file

'zones' The mass flows through all the zones are summed to give a single output value for the mass flow monitor. Flow out of the domain is considered positive massflow, flow into the domain is considered positive massflow and the massflows are dimensional (typically kg per second).

Example usage:

```
'report' : {
    'frequency' : 10,
    # Report massflow ratio
    'massflow' : {
        'MF_1' : {
            # Output name
            'name' : 'massflow_1',
            # Zones to be included
            'zone' : [1, 2, 3],
        },
    },
},
```

1.5.15 Output variables

Output Variables

| Variable Name | Alias | Definition |
|-------------------------|-------|---|
| temperature | T, t | Temperature in (K) |
| temperaturegrad | | Temperature gradient vector |
| potentialtemperature | | Potential temperature in (K) |
| pressure | p | Pressure (Pa) |
| gauge_pressure | | Pressure relative to the reference value (Pa) |
| density | rho | Density (kg/m^3) |
| velocity | V, v | Velocity vector (m/s) |
| velocitygradient | | Velocity gradient tensor |
| cp | | Pressure coefficient |
| totalcp | | Total pressure coefficient |
| mach | m | Mach number |
| viscosity | mu | Dynamic viscosity (Pa/s) |
| kinematicviscosity | nu | Kinematic viscosity (m^2/s) |
| vorticity | | Vorticity vector ($1/s$) |
| Qcriterion | | Q criterion |
| reynoldstress | | Reynolds stress tensor |
| helicity | | Helicity |
| enstrophy | | |
| ek | | |
| lesregion | | |
| turbulenceintensity | ti | Turbulence intensity |
| turbulencekineticenergy | | Turbulence kinetic energy (J/kg) |
| turbulenceeddyfrequency | | Turbulence eddy frequency ($1/s$) |
| eddy | | Eddy viscosity (Pa/s) |
| cell_velocity | | |
| cellzone | | |
| cellorder | | |
| centre | | Cell centre location |
| viscouslengthscale | | Minimum distance between cell and its neighbours |
| walldistance | | Wall distance (m) |
| sponge | | Sponge layer damping coefficient |
| menterfl | | Menter SST blending function |

Surface Only Quantities

| Variable Name | Alias | Definition |
|------------------|-------|---|
| pressureforce | | Pressure component of force on a face |
| pressuremoment | | Pressure component of moment on a face |
| pressuremomentx | | Component of pressure moment around x axis |
| pressuremomenty | | Component of pressure moment around y axis |
| pressuremomentz | | Component of pressure moment around z axis |
| frictionforce | | Skin friction component of force on a face |
| frictionmoment | | Skin friction component of moment on a face |
| frictionmomentx | | Component of skin friction moment around x axis |
| frictionmomenty | | Component of skin friction moment around y axis |
| frictionmomentz | | Component of skin friction moment around z axis |
| roughness | | Wall roughness height |
| yplus | | Non-dimensional wall distance |
| zone | | Boundary zone number |
| cf | | Skin friction coefficient |
| facecentre | | Face centre location |
| frictionvelocity | ut | Friction velocity at the wall |

1.6 zCFD Functions

Several parameters in the zCFD control dictionary will accept functions as arguments. These functions are then evaluated by the solver when required. This allows you to customise & tune to solver as it runs.

1.6.1 Scale Mesh Function

| | |
|----------------------------|---|
| Description | Transform the location of a supplied point. |
| Valid for parameter | <i>scale_mesh</i> |
| Parameters | [x, y, z]: Point to transform |
| Returns | {'coord': [x, y, z]}: Transformed coordinates |

Example:

```
# Custom function to scale z coordinate by 100
def my_scale_func(point):
    point[2] = point[2] * 100.0
    return {'coord' : point}
```

```
parameters = {
    ...
    'scale': my_scale_func
    ...
}
```

1.6.2 Transform Mesh Function

| | |
|----------------------------|--|
| Description | Transform the location of a supplied point. |
| Valid for parameter | <i>mesh transform func</i> |
| Parameters | [x, y, z]: Point to transform |
| Returns | {"v1": x, "v2": y, "v3": z}: Transformed coordinates |

Example:

```
rotation = 10.0

def my_transform(x, y, z):
    v = [x, y, z]
    v = zutil.rotate_vector(v, rotation, 0.0)
    return {"v1": v[0], "v2": v[1], "v3": v[2]}
```

1.6.3 Ramp CFL Function

There may be cases where more detailed control of the CFL number is desired - for example in some aerodynamic simulations where a specific physical event occurs at a given time. The CFL Ramp function lets you do this.

The ramp function returns a single floating point number (the CFL number) and can also be used to update elements of the 'cfl' Python object such as cfl.min_cfl, cfl.max_cfl, cfl.current_cfl, cfl.coarse_cfl, cfl.transport_cfl, cfl.cfl_ramp, cfl.cfl_pmg, or cfl.transport_cfl_pmg.

| | |
|----------------------------|-----------------------------------|
| Description | Provide a dynamic CFL number |
| Valid for parameter | <i>ramp func</i> |
| Parameters | solve_cycle, real_time_cycle, cfl |
| Returns | cfl: float |

Example:

```
# Custom function to ramp CFL based on cycle, updates the cfl object variables and
# returns a CFL
def my_cfl_ramp(solve_cycle, real_time_cycle, cfl):
    cfl.min_cfl = 1.0
    cfl.max_cfl = 10.0
    cfl.cfl_ramp = 1.005
    cfl_transport = 5.0
    if solve_cycle < 500:
        cfl_new = min(cfl.min_cfl * cfl.cfl_ramp ** (max(0, solve_cycle - 1)), cfl.max_
        cfl)
        cfl.transport_cfl = cfl_transport * cfl_new / cfl.max_cfl
        return cfl_new

    elif solve_cycle < 1000:
        cfl.transport_cfl = 7.5
        cfl.max_cfl = 15.0
        return 15.0

    else:
        cfl.transport_cfl = 10.0
        cfl.max_cfl = 30.0
        return 30.0
```

```
parameters = {
...
'time marching': {
...
'ramp func': my_cfl_ramp,
...
}
...
}
```

1.6.4 Transform Function

A transformation function may be supplied to the force report block to transform the force into a different coordinate system. This is particularly useful for reporting lift and drag, which are often rotated from the solver coordinate system.

| | |
|----------------------------|--|
| Description | Transform a supplied point - rotate, scale or translate. |
| Valid for parameter | <i>Forces transform</i> |
| Parameters | (x, y, z): floats |
| Returns | (x, y, z): floats |

Example:

```
#zutil is a Zenotech python library providing various utilities, available within a zCFD installation
import zutil

# Angle of attack
alpha = 10.0
# Transform into wind axis
def my_transform(x,y,z):
    v = [x,y,z]
    v = zutil.rotate_vector(v,alpha,0.0)
    return {v[0], v[1], v[2]}
```

1.6.5 Initialisation Function

| | |
|----------------------------|---|
| Description | Provide the initial flow field variables on a cell-by-cell basis |
| Valid for parameter | <i>initial</i> |
| Parameters | kwargs dictionary containing “pressure”, “temperature”, “velocity”, “wall_distance”, “location” |
| Returns | dictionary containing one or more of “pressure”, “temperature” or “velocity” |

```
def my_initialisation(**kwargs):
    # Dimensional primitive variables
    pressure = kwargs['pressure']
    temperature = kwargs['temperature']
    velocity = kwargs['velocity']
    wall_distance = kwargs['wall_distance']
    # Cell centre location [X, Y, Z]
```

(continues on next page)

(continued from previous page)

```

location = kwargs['location']

if location[0] > 10.0:
    velocity[0] *= 2.0

# Return a dictionary with user defined quantity.
return { 'velocity' : velocity }

```

```

parameters = {
...
'initial': {
...
'func': my_initialisation,
...
}
...
}

```

1.6.6 Driven initial condition

A ‘driving function’ provides another way of prescribing a farfield initial condition, which on evaluation returns an initial condition dictionary.

The driving function is re-evaluated at zCFD’s reporting frequency, meaning that the condition (e.g. inlet velocity) can be programmed to change as the simulation progresses.

At startup, the function is only provided with the key word arguments RealTimeStep=0, Cycle=0. However, the driving function is subsequently provided with key word arguments containing all the data in the ‘_report.csv’ file, evaluated at the previous timestep. For any driven initial condition, every value in the initial condition dictionary is also appended to the _report.csv file.

This means that the initial condition can be programmed to respond to the flow. An example is shown below, where the driven IC will continually adjust inlet angle of attack until a specified lift coefficient is achieved.

If the driven initial condition has been programmed to respond to changes in the flow, it is important to ensure that the flow has had enough time to propagate from the farfield through to the region of interest and settle before readjusting the driven initial condition. For simulations where the farfield is distant from the geometry, local and implicit time-stepping will likely give the fastest propagation of adjusted farfield conditions through the domain.

Note: Driven initial conditions **cannot** be used as reference conditions, to initialise the flow (e.g ‘IC_1’) or within a restarted simulation.

| | |
|----------------------------|--|
| Description | Provides a farfield initial condition that can respond to changes in the flow. |
| Valid for parameter | <i>initial condition</i> |
| Parameters | kwargs dictionary containing “RealTimeStep”, “Cycle” and all reporting variables |
| Returns | dictionary containing valid keys for an <i>initial condition</i> |

```

def example_ic_func(**kwargs):
"""
Example driver function to target a lift coefficient by varying
angle of attack
"""

```

(continues on next page)

(continued from previous page)

```

alpha_init = 0 # Initial angle of attack
lift_target= 0.5 # Targeted lift coefficient
update_period = 1000 # How often alpha should be updated
flow_settling_period = 1500

assumed_d_cL_d_alpha = 0.1 # Used to estimate alpha which would give required lift
relaxation_factor = 1.15 # <1: under-relaxation, >1: over-relaxation

if 'lift_target_alpha' in kwargs.keys(): # If timestep > 1
    alpha_current = kwargs['lift_target_alpha']
    F_xyz = [kwargs['wall_Fx'], kwargs['wall_Fy'], kwargs['wall_Fz']]
    F_LDS = zutil.rotate_vector(F_xyz, alpha_current, 0.0)
    lift_current = F_LDS[2] # Lift coefficient, having rotated to account for alpha
else:
    lift_current = 0 # placeholder value

if kwargs['Cycle'] < flow_settling_period:
    alpha_new = alpha_init
else:
    if (kwargs['Cycle'] % update_period) == 0:
        d_cL_required = lift_target - lift_current
        d_alpha = relaxation_factor / assumed_d_cL_d_alpha * d_cL_required
        alpha_new = alpha_current + d_alpha
        print("Alpha updated from {} to {}".format(alpha_current, alpha_new), flush=True)
    else:
        alpha_new = alpha_current

dict_out = {
    "temperature": 300,
    "pressure": 101325.0,
    'v': {
        'mach' : 0.15,
        'vector' : zutil.vector_from_angle(alpha_new, 0.0)
    },
    "reynolds no": 6.0e6,
    "eddy viscosity ratio": 1.0,
    "monitor": { # Monitor entries can be floats or lists of floats
        "prefix": "lift_target",
        "alpha": alpha_new, # Extra keys can be added to a driven IC dictionary,
        "lift": lift_current, # allowing monitoring of key parameters in the _report.csv
    },
}

return dict_out

```

```

parameters = {
...
'IC_2' : example_ic_func
...
}

```

1.6.7 Static Pressure Ratio

The python function takes in three values (x, y, z) and returns the static pressure ratio.

| | |
|----------------------------|--|
| Description | Returns a static pressure ratio for a given point. |
| Valid for parameter | <i>Static Pressure Ratio</i> |
| Parameters | (x, y, z): floats |
| Returns | float |

1.6.8 Total Pressure Ratio

The python function takes in three values (x, y, z) and returns the total pressure ratio.

| | |
|----------------------------|---|
| Description | Returns a total pressure ratio for a given point. |
| Valid for parameter | <i>Total Pressure Ratio</i> |
| Parameters | (x, y, z): floats |
| Returns | float |

1.6.9 Total Temperature Ratio

The python function takes in three values (x, y, z) and returns the temperature pressure ratio.

| | |
|----------------------------|--|
| Description | Returns a total temperature ratio for a given point. |
| Valid for parameter | <i>Total Temperature Ratio</i> |
| Parameters | (x, y, z): floats |
| Returns | float |

1.6.10 Direction Vector

The python function takes in three values (x, y, z) and returns a tuple (x, y, z).

| | |
|----------------------------|---|
| Description | Returns a direction vector for a given point. |
| Valid for parameter | <i>vector</i> |
| Parameters | (x, y, z): floats |
| Returns | (x, y, z): floats |

```
def inflow_direction(x, y, z):
    # x, y, and z are the coordinates of the face centre
    cen = [1.0, 0.0, 0.0]
    y1 = y - cen[1]
    z1 = z - cen[2]
    rad = math.sqrt(z1*z1 + y1*y1)
    phi = math.atan2(y1,z1)
    angle = -0.24
    x2 = math.cos(angle)
    y2 = math.sin(phi) * math.sin(angle)
    z2 = math.cos(phi) * math.sin(angle)
    return (x2, y2, z2)
```

```
parameters = {
...
'IC_2' : {
    'reference' : 'IC_1',
    'vector' : inflow_direction
},
...
}
```

1.7 Mesh Conversion

zCFD uses unstructured meshes written in an [HDF5](#) file format. HDF5 provides a flexible, self describing specification supporting parallel I/O that can be tuned per filesystem type (e.g NFS, GPFS, Lustre etc).

Meshes from a wide variety of sources can be easily converted into this format as follows:

1.7.1 OpenFOAM-x.x.x

We provide a range of converters for popular mesh types via a special version of the open-source CFD code OpenFOAM, available freely for download [here](#)

Once downloaded and installed, the version of OpenFOAM will convert a range of file formats to the zCFD format. Prior to running any of these, you will need to activate your OpenFOAM environment using

```
source $FOAM_INST_DIR/OpenFOAM-x.x.x/etc/bashrc
```

1.7.2 OpenFOAM Mesh Converter

To convert from an OpenFOAM format mesh to zCFD, run the following command in the MESH directory.

```
foamTozCFD
```

This will create a new folder called *zInterface*/ containing the HDF5 file.

1.7.3 Fluent Mesh Converter

The converter for [ANSYS Fluent](#) files is a utility called *fluent_convert*. This is included in the path set by activating the zCFD command line environment.

To run the utility, use

```
(zCFD): fluent_convert <filename>.msh <new_filename>.h5
```

or

```
(zCFD): fluent_convert <filename>.cas <new_filename>.h5
```

The will produce 2 files: the zCFD HDF5 file and the *name_zone.py* python file which holds the fluid and boundary zone names.

1.7.4 Star CCM+ Mesh Converter

There are a number of different file formats for **CD-Adapco Star CCM+** meshes. The `<filename>.sim` file is in a proprietary format and at this stage cannot be converted. The `<filename>.ccm` can be converted using

```
(zCFD): ccmconvert <filename>.ccm <new_filenam>.h5
```

This produces the zCFD HDF5 file.

1.7.5 CGNS Mesh Converter

This utility will convert an unstructured, non-chimera, hexahedral mesh in CGNS (ref) format. The converter will handle meshes with element types (HEXA_8):

```
(zCFD): cgnsconvert <filename>.cgns <new_filenam>.h5
```

This produces the zCFD HDF5 file. If you require support for additional element types please contact us: <mailto:zcfdb@zenotech.com>.

1.7.6 UGRID Mesh Converter

UGRID is a NASA format for meshes. You must follow the naming convention for UGRID files, which describes the format and *endian-ness* of the file data (“ascii8”, “b8”, “lb8”, or “r8”) depending upon which language (C or FORTRAN) was used to create the file. The mapbc file is optional and if not provided the boundary conditions of the zones default to wall.

```
(zCFD): ugridconvert <filename>.[ascii8, b8, lb8, r8].ugrid <new_filename>.h5  
-<filename>.mapbc
```

This produces the zCFD HDF5 file.

1.7.7 DLR Tau Mesh Converter

Tau is a CFD solver produced by the DLR (<http://tau.dlr.de/startseite/>). Note that Tau is a node-based solver so flow field variables are calculated for and stored at the mesh nodes (or vertices) rather than at cell centres (the approach used by Star CCM+, OpenFOAM, CFX and zCFD). Thus simply converting the file format will not guarantee a good result. In many cases, the conversion will not result in a valid cell-centred zCFD mesh.

```
(zCFD): tauconvert <filename>.XXX <new_filename>.h5
```

This produces the zCFD HDF5 file.

1.7.8 SU2 Mesh Converter

SU2 is an open source CFD solver (<https://su2code.github.io/>). Note that conversion of 2D meshes is not supported as they are not handled in zCFD.

```
(zCFD): su2convert <filename>.su2 <new_filename>.h5
```

This produces the zCFD HDF5 file.

1.7.9 CBA Mesh Converter

CBA is a structured multiblock mesh format used within the University of Bristol and HMB projects. Note 2D meshes will have a symmetry plane added to extend them into 3D for use in zCFD.

```
(zCFD): cba_convert <filename>.cba, blk <new_filename>.h5
```

This produces the zCFD HDF5 file.

1.8 FWH solver

A Ffowcs-Williams Hawkings (FWH) solver, which is used for prediction of farfield noise, is included in the zCFD distribution. The zCFD FWH solver uses the [Farrasat 1A formulation](#), and can accept both permeable and impermeable FWH surfaces.

1.8.1 What is an FWH solver for?

A common problem in CFD is the prediction of the farfield aerodynamic noise which results from a particular airflow. Two issues make prediction of farfield aerodynamic noise challenging. First of all, many of the sources of aerodynamic noise (turbulence, vortex shedding etc.) require high fidelity, time accurate simulations and fine CFD meshes to be properly captured. Secondly, a mesh must be very fine and the simulation must have very low numerical dissipation in order to propagate aerodynamic noise any distance within a CFD mesh.

The first issue of noise simulation can be made tractable by the use of techniques like FRPM, but the second issue of far-field mesh requirements for noise propagation means that e.g. predicting the noise from a helicopter main rotor a kilometre away is simply not feasible using a conventional CFD simulation.

This issue of noise propagation in the far-field is solved by the use of the Ffowcs-Williams Hawkings (FWH) equations. These equations allow us to propagate sound to an observer an arbitrary distance away without the use of a mesh which extends all the way to the observer.

1.8.2 How does an FWH solver work?

In order to use the FWH equations to propagate noise to a far-field observer, we first run a CFD simulation and record the flow variables ($p(\mathbf{x}, t)$, $\rho(\mathbf{x}, t)$ and $\mathbf{u}(\mathbf{x}, t)$) on an ‘FWH surface’ in the near-field, noise generating region. [Fig. 1](#) shows an FWH surface enclosing the nearfield for a simulation of an aerofoil in a wind tunnel.

The FWH surface should enclose the noise generating region, and the CFD mesh should be sufficiently fine in the near-field region to propagate noise to the FWH surface without excessive dissipation. Ideally, the FWH surface should not intersect the nearfield region of any wakes.

In some situations, it is sufficient to collect FWH surface data on the wall instead of using a separate FWH surface inside the fluid domain. FWH surfaces coincident with the wall are called *impermeable* (as opposed to *permeable*) FWH surfaces, and only require p (not ρ , \mathbf{u} and \mathbf{p}) to be recorded.

Once the flow variables have been recorded on the FWH surface, we can propagate the noise to our observers using an FWH solver. In order to do so we must define the motion of our FWH surface and our observers in the FWH simulation. The FWH equations assume a quiescent medium (i.e. still air, with a free-stream velocity of zero), which means that the FWH surface and observer motions are frequently different to the motions used in the CFD simulation used to produce the FWH surface data.

In our example of an aerofoil in a wind tunnel, the CFD simulation has a freestream velocity, which does not satisfy the ‘quiescent medium’ condition. When using an FWH simulation to calculate the noise at the farfield observer microphone in [Fig. 1](#), we therefore have to perform a co-ordinate transformation and ‘fly’ both the FWH surface and the observer microphone forwards through the quiescent medium, as in [Fig. 2](#).

Once we have collected the FWH surface data from a CFD simulation and correctly defined the FWH surface and observer microphone settings, we simply use the FWH equations to propagate the pressure and velocity



Fig. 3: CFD simulation of an aerofoil, including an FWH surface



Fig. 4: FWH simulation of an aerofoil, with quiescent medium co-ordinate transformation applied

signals from the FWH surface to the our farfield observer locations. The equations used in the zCFD FWH solver are the [Farrasat 1A equations](#), with a retarded time formulation and a Newton-Raphson algorithm for calculating the retarded time. The retarded time is the time τ_* at which a sound wave must be emitted by a point on the FWH surface in order for it to reach an observer at time t , given the prescribed motions of both surface and observer.

Note: The Newton-Raphson solver which calculates retarded time in the FWH solver will struggle to converge as the closing speed between an FWH surface and an observer approaches or exceeds Mach 1.

1.8.3 Using the zCFD FWH solver

To use the zCFD FWH solver, we must first generate FWH surface data during a zCFD simulation (see [write output](#) for more details). The FWH surface data is stored in a HDF5 format. While no mesh conversion tools are provided, the zCFD FWH file format can easily be interrogated and replicated with tools like *h5ls* and *h5py* should you wish to convert data from other simulation tools to this format.

The zCFD FWH solver consists of a series of python modules, which are accessible from within the [zCFD command-line environment](#). Three python dictionaries, controlling the surface, observer, and solver settings, are required as inputs to the *fwh.solve* function. This function then returns time histories of the pressures at the specified observer points, in the nested dictionaries *pOut* and *tOut*.

```
from solvers import fwh, motions

mySurfaces = { surfaces }
myObservers = { observers }
mySolverSettings = { solverSettings }

pOut,tOut = fwh.solve(mySurfaces, myObservers, solverSettings)
```

For an aerofoil in a wind tunnel (as in [Fig. 1](#) and [Fig. 2](#)), an appropriate python script (including a small amount of post-processing) might look like this:

```
from solvers import fwh, motions
from matplotlib import pyplot as plt
import json

v = [-50,0,0]
surfaceCentreMotion = motion.ConstantVelocityPoint([0,0,0],v)
surfaceMotion = motion.NonrotatingSurface(surfaceCentreMotion)
obs1Motion = motion.ConstantVelocityPoint([1,0,0],v)
obs2Motion = motion.ConstantVelocityPoint([2,0,0],v)

mySurfaces = {
    "fwhSurf1": {
        "motion": surfaceMotion,
        "fileName": "./zcfdsim_P2_OUTPUT/ACOUSTIC_DATA/fwhsurf1.h5"
    }
}
myObservers = {"Obs1": obs1Motion, "Ob2": obs2Motion}
mySolverSettings = { "c": 340, "rho0": 1.2, "p0": 1e5}

pOut,tOut = fwh.solve(mySurfaces,myObservers,solverSettings)

fig,ax = plt.subplots()
ax.plot(tOut["fwhSurf1"]["Obs1"],pOut["fwhSurf1"]["Obs1"],label="Obs1")
ax.plot(tOut["fwhSurf1"]["Obs2"],pOut["fwhSurf1"]["Obs2"],label="Obs2")
ax.set_xlabel("t (s)")
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel("p (Pa)")
fig.savefig("./FWH_figure.pdf")

dataForJson = {"p": pOut, "t": tOut}
with open("./FWH_data.json", "w") as f:
    json.dump(dataForJson, f)
```

Surfaces dictionary

The zCFD FWH solver is capable of using multiple FWH surfaces in the same FWH calculation. The Surfaces dictionary therefore contains a key-value pair for each FWH surface to be used, where the key is the surface's name (to be used in the FWH solver) and the value is an *individual surface definition dictionary*.

Example usage:

```
from solvers import motion
v = [-50,0,0]
surfaceCentreMotion = motion.ConstantVelocityPoint([0,0,0],v)
surfaceMotion = motion.NonrotatingSurface(surfaceCentreMotion)

mySurfaces = {
    "fwhSurf1": {
        "motion": surfaceMotion,
        "fileName": "./zcfdsim_P2_OUTPUT/ACOUSTIC_DATA/fwhsurf1.h5"
    }
}
```

Individual surface definition dictionary

| Keyword | Required | Default | Valid values | Description |
|----------------------------|----------|---------|-------------------------------|--|
| 'motion' | Yes | | A <i>surface motion class</i> | The motion of the surface through the FWH medium |
| 'fileName' | Yes | | A readable file path | The FWH surface data, in the FWH HDF5 format outputted by the zCFD solver |
| 'permeable' | No | True | True False | Whether or not the surface is permeable or not (see <i>above</i>). Set to False for FWH wall data. |
| 'flipSurfaceNormals' | No | False | True False | Whether or not to invert the FWH surface normals before running the FWH solver. The surface normals should always point outwards towards the farfield. For FWH surface data generated using the <i>fwf interpolate</i> keyword in the zCFD solver, the surface normal direction depends on the .stl file used to define the FWH surface, and can be checked using a visualisation tool (e.g. Paraview). |
| 'transformSurfaceVelocity' | No | True | True False | Relevant for permeable surfaces only, and should normally be kept <i>True</i> . Setting this to <i>False</i> means that the FWH surface fluid velocity used in the FWH solver does not include the velocity resulting from the motion of the surface (as defined my the 'motion' keyword above). This option may be required when using data from other solvers, in the case where the FWH surface moves through a quiescent medium during the simulation used to create the FWH surface data. |

Observers dictionary

The Observers dictionary defines a list of observer microphones at which farfield noise should be calculated. In the observers dictionary, keys are used to set the observer's name in the FWH solver output, and the value sets the motion of the observer. Observer motions must be defined using the *point motion classes*.

Example usage:

```
from solvers import motion
v = [-50,0,0]
obs1Motion = motion.ConstantVelocityPoint([1,0,0],v)
obs2Motion = motion.ConstantVelocityPoint([2,0,0],v)

myObservers = {"Obs1": obs1Motion, "Ob2": obs2Motion}
```

Solver settings dictionary

The solver settings dictionary sets the properties of the FWH medium, and the numerical settings used in the FWH solver.

Example usage:

```
mySolverSettings = { "c": 340, "rho0": 1.2, "p0": 1e5}
```

| Keyword | Required | Default | Valid values | Description |
|--------------------|----------|---------|------------------|--|
| 'c' | Yes | | Positive number | speed of sound in the far-field medium, m/s |
| 'rho0' | Yes | | Positive number | density of the far-field medium, kg/s |
| 'p0' | Yes | | Positive number | static pressure of the far-field medium, Pa |
| 'dt' | Yes | | Positive number | timestep between observer microphone recording times, s |
| 'breakTo- lAbs' | No | 1.0e-5 | Positive number | the time accuracy (s) with which we would like to predict the retarded time τ_* in the retarded time calculation. |
| 'maxIts' | No | 50 | Positive integer | The maximum number of iterations to use in the Newton Raphson algorithm used to predict retarded time |

Point (observer) motion classes

FWH Observers are points which move through the FWH quiescent medium according to the function $x = f(t)$. In the zCFD FWH solver, we define the motion of observers using one of the following classes, which are available inside the *solvers.motion* module inside the zCFD command line environment (see [above](#)).

| Motion Class | Inputs | Description |
|---------------------------------------|---|--|
| <i>OriginPoint0</i> | | A point which is always on the origin, i.e. $f(t) = [0, 0, 0]$. |
| <i>StationaryPoint(X0)</i> | X0: a 3-d vector $[x_0, y_0, z_0]$ | A point which is stationary, i.e. $f(t) = [x_0, y_0, z_0]$. |
| <i>ConstantVelocityPoint(X0,dXdt)</i> | X0: a 3-d vector $[x_0, y_0, z_0]$ dXdt: a 3-d vector $[dxdt, dydt, dzdt]$ | A point with constant velocity, i.e. $f(t) = [x_0 + t * dxdt, y_0 + t * dydt, z_0 + t * dzdt]$ |

Surface motion classes

Just like FWH Observers, FWH Surfaces require their motion to be defined. There are two surface motion classes - NonrotatingSurface and RotatingSurface. Both use a [point motion class](#) to define the motion of their 'centre point'. In this way, complex motions like that of a propellor which is both rotating and translating through the FWH medium can be defined.

| Motion Class | Inputs | Description |
|--|---|---|
| <i>NonrotatingSurface(centrePoint)</i> | centrePoint: a <i>point motion class</i> | A surface which does not rotate, and translates at the same velocity as the ‘centrePoint’. |
| <i>RotatingSurface(centrePoint, axis, omega)</i> | centrePoint: a <i>point motion class</i> axis: a 3-d vector [ax_x,ax_y,ax_z] omega: a number. | A surface which translates at the same velocity as the <i>centrePoint</i> , and also rotates about the centre-Point with rotation axis <i>axis</i> and rotation angular velocity (rad / s) <i>omega</i> |

1.9 zM3

zM3 or “zMesh Cubed” is a mesh generator that is included in the zCFD distribution. It is a general purpose meshing tool that can rapidly create cartesian meshes for zCFD’s acoustic solver or finite volume solver. As a user you define the domain and can then include geometry in the form of STL files to include in the domain. Mesh refinement can be controlled in a number of ways including explicit refinement regions as well as refinement based on the size of the STL triangulation.

1.9.1 Execution

zM3 is included in the standard zCFD installation, to run it simply activate your zCFD environment in the normal way and the *zm3* executable will be added to your path.

```
zm3
```

To see the command line options run:

```
zm3 -help
```


1.9.2 Options

| Option | De-fault | Description | Example |
|---|------------------|---|---|
| <code>-meshName</code> | mesh | String defining the output mesh name | <code>-meshName output.h5</code> |
| <code>-base</code> | 0 0 0 | Three real numbers defining the (min_x, min_y, min_z) corner | <code>-base 0.0 0.0 0.0</code> |
| <code>-dims</code> | 1.0 1.0 1.0 | Three real numbers defining the x, y, z extent of the domain | <code>-dims 100.0 100.0 100.0</code> |
| <code>-globalSpacing</code> | 1.0 | Real number defining largest cell size in the mesh | <code>-globalSpacing 5.0</code> |
| <code>-stlFiles</code> | | List of STL files containing surfaces for meshing | <code>-stlFiles file1.stl file2.stl</code> |
| <code>-stlSpacing</code> | | Target cell size for each STL surface file loaded, the the same order as specified in <code>stlFiles</code> | <code>-stlSpacing 1.0 0.5</code> |
| <code>-relativeSpacings</code> | 0 (for each STL) | Integer for each STL file specified, 1 = True, 0 = False. If True then the spacings defined in <code>stlSpacing</code> are treated as a factor multiplying the local STL size to get the spacing value at that location. | <code>-relativeSpacings 0 1</code> |
| <code>-minSurfaceSpacing</code> | | Minimum surface spacing when using <code>relativeSpacings</code> | <code>-minSurfaceSpacing 0.5</code> |
| <code>-maxSurfaceSpacing</code> | | Maximum surface spacing when using <code>relativeSpacings</code> | <code>-maxSurfaceSpacing 0.5</code> |
| <code>-includeProximitySpacings</code> | 0 (for each STL) | Integer for each STL file specified, 1 = True, 0 = False. If True reduce cell sizes to account for the proximity of other triangles nearby. | <code>-includeProximitySpacings 0 1</code> |
| <code>-refineAnisotropicSTLTris</code> | 0 (for each STL) | Integer for each STL file specified, 1 = True, 0 = False. If True triangles will be sub-divided until all edges approach the smallest edge length for that STL. | <code>-refineAnisotropicSTLTris 0 1</code> |
| <code>-includeCutCells</code> | 0 | Integer specifying if cells intersected by STL are to remain in mesh, 1 = True, 0 = False | <code>-includeCutCells 1</code> |
| <code>-seedPoints</code> | | List of three real numbers defining a point in the mesh to be kept, from which flood filling occurs. Multiple seed points can be defined. | <code>-seedPoints 50.0 50.0 50.0</code> |
| <code>-solidPoints</code> | | Three real numbers defining a point in the mesh, within a region which must be deleted. Defines non-fluid regions. Multiple solid points can be defined. | <code>-solidPoints 10.0 5.0 3.0</code> |
| <code>-boxRefinement</code> | | Lists of 7 real numbers defining a refinement box in the order x_min y_min z_min x_max y_max z_max spacing. | <code>-boxRefinement 20.0 20.0 20.0 50.0 50.0 50.0 7.5</code> |
| <code>-surfaceBoxRefinement</code> | | Similar to <code>boxRefinement</code> but only STL elements within the box will be given the target refinement. | <code>-surfaceBoxRefinement 20.0 20.0 20.0 50.0 50.0 50.0 7.5</code> |
| <code>-rotatedTranslatedRefinement</code> | | A rotated and translated box refinement region. The box is rotated around 0,0,0 and then translated. The rotated/translated box is represented as a group of cartesian aligned boxes with the resolution of those boxes controlled by | <code>-rotatedTranslatedRefinement 10.0 10.0 10.0 0.0 0.0 1.0 10.0 10.0 10.0 7.5</code> |

1.10 Visualisation

zCFD by default will output data in VTK format. This format can be read directly by many post-processing tools including ParaView by Kitware.

Output in other formats is also possible, including TECPLOT and EnSight as follows:

1.10.1 VTK Output

By default, zCFD will output data in VTK format. To explicitly force VTK output, use the following option in the ‘*write output*’ part of the control dictionary:

```
'write output' : {  
    # Output format  
    'format' : 'vtk',
```

1.10.2 EnSight Output

To output in EnSight format, use the following option in the ‘*write output*’ part of the control dictionary:

```
'write output' : {  
    # Output format  
    'format' : 'ensight',
```

1.10.3 Native HDF5 Output

To output in HDF5 format (the native format for zCFD), use the following option in the ‘*write output*’ part of the control dictionary:

```
'write output' : {  
    # Output format  
    'format' : 'native',
```

1.10.4 ParaView Catalyst Scripts

ParaView provides a powerful yet lightweight interface that allows parallel processing of simulation data during programme execution called ParaView Catalyst. This is driven by scripts (Python files) that can be called when data is output:

```
'write output' : {  
    # Output format MUST be VTK  
    'format' : 'vtk',  
    # The scripts should be in the same directory as the input files  
    'scripts' : ['paraview_catalyst1.py','paraview_catalyst2.py']}
```

1.10.5 ParaView Client

To download the free ParaView software for visualising VTK format data on the local device, just visit the [ParaView website](#).

1.10.6 ParaView Client-Server

In addition to running entirely locally on a device, ParaView can interface to a remote server. This allows users to run ParaView to access data that is held (and processed) on a different computer, potentially running larger simulation tasks that would be possible on the local machine. This also means that there is no need to transfer the large output files back to a local machine for post-processing. The remote machine can easily be set up to run a ParaView Server in parallel - meaning that very large post-processing jobs can be run quickly.

ParaView Server

To download and install ParaView Server, you can either use the version on the main ParaView website and follow the instructions [here](#).

Alternatively or else use the version bundled with zCFD. The ‘pvserver’ included in the zCFD distribution includes plugins for reading the zCFD HDF5 mesh format directly.

To start a ParaView Server on your computer within the zCFD environment run:

```
> pvserver
```

ParaViewConnect

[ParaviewConnect](#) is a helper utility for connecting to remote ParaView servers. It reduces the complexity of setting up the required ssh connections and works well with zCFD. More information and installation instructions can be found on the [Github page](#).

1.10.7 Jupyter Notebooks

We also recommend the use of [Jupyter Lab](#) for post-processing zCFD runs. This is one of the easiest ways to use Python, via an interactive notebook on your local computer. The notebook is a locally hosted server that you can access via a web-browser. Jupyter is included in the zCFD distribution and notebooks for visualising the residual data will automatically be created when you run a zCFD case. In addition there are several example notebooks available in the [zPost](#).

To automatically start up notebook server on your computer within the zCFD environment run:

```
> start_lab
```

1.11 Utilities

zCFD is bundled with a number of utility functions to streamline specific workflows:

1.11.1 Windfarm Modelling

Overview

Windfarms consist of a set of specific turbine types at geographic locations. zCFD allows users to specify the turbine characteristics and locations, and will automatically create turbine models within an existing mesh and update the local flow conditions to match the thrust coefficient and tip speed ratio for each turbine.

The locations and TRBX turbine definition

zCFD provides a utility function *create_trbx_zcfld_data* that can be called from within the zCFD environment and Python interactive shell. To start the zCFD command line environment, type:

```
> source /INSTALL_LOCATION/zCFD-version/bin/activate
```

The command prompt will now appear with a (zCFD) prefix. To run the *create_trbx_zcfld_data* utility, start the interactive Python shell:

```
> (zCFD) > python
```

The utility takes several arguments, depending on the turbine model to be used. For *simple* and *induction* models (where a reference velocity point is used to set the power, thrust and torque for the turbine) the function is used to look up data in a *TRBX* file format:

```
import zutil.farm as farm
farm.create_trbx_zcfld_input(case_name='windfarm_name',
                             wind_direction=267.0,
                             reference_wind_speed=10.0,
                             terrain_file=None,
                             report_frequency=100,
                             update_frequency=1,
                             reference_point_offset=1.0,
                             turbine_zone_length_factor=2.5,
                             model='simple',
                             turbine_files=[['xy_file_1.txt','turbine_type_1.trbx'],
                                            ['xy_file_2.txt','turbine_type_2.trbx']])
```

The utility will read each *<xy_file>*, *<trbx_file>* pair provided. The terrain file is any file format that can be read by ParaView (STL, VTK) to use as the basis for the ground level at any given location. The default 'None' sets ground level to 0. The model uses a reference velocity at a point upstream of the turbine. The point is offset upstream by a factor applied to the rotor diameter. The fluid zone associated with the turbine is a cylinder with diameter equal to the rotor diameter and length given by the turbine zone length factor. The model is either an *induction* model where the induction factor is automatically calculated from the thrust coefficient and 1D momentum theory, or the *simple* model where the reference velocity is equal to the velocity at the rotor. The *<xy_file>* is a 3-column data file in ASCII format with *<turbine name>*, *<easting>* and *<northing>* data such as:

```
T001 251865.0 646486.0
T002 252240.0 646200.0
...
T015 253308.0 647985.0
```

The TRBX file is a data file with turbine information provided by manufacturers including a description of the turbine geometry and aerodynamic performance characteristics.

The utility extracts key information from the TRBX file to automatically create a turbine definition (*<case_name>_zones.py*) with an entry for each turbine in the array:

```
turb_zone = {
    'FZ_1':{
        'type':'disc',
        'def':'./turbine_vtp/T001-267.0.vtp',
        'thrust coefficient':0.821,
        'thrust coefficient curve':[[0.0,0.0],[1.0,0.0],...,[51.0,0.0],],
        'tip speed ratio':8.79645943005,
        'tip speed ratio curve':[[0.0,0.0],[1.0,0.0],...,[51.0,0.0],],
        'centre':[251865.0,646486.0,338.5],
        'up':[0.0,0.0,1.0],
        'normal':[-0.998629534755,-0.0523359562429,-0.0],
        'inner radius':1.95500004292,
        'outer radius':60.0,
        'reference point':[251745.164456,646479.719685,338.5],
    },
    ...
}
```

The '*thrust coefficient*' may be defined as a single value, and if a data point curve is present in the TRBX file this data is also supplied as the tuple array '*thrust coefficient curve*', where the wind speed in metres per second is the index. If this curve is provided, the utility will interpolate the curve at the reference speed to create the single value otherwise a default value (10 m/s) is used.

The same approach is taken for the '*tip speed ratio*' single value and curve - which is automatically calculated from the rotor speed array (revolutions per minute) in the TRBX file.

The '*reference point*' defines the location in the flow domain that is used as the reference value of wind velocity for this turbine. This velocity is used in combination with the thrust coefficient and the tip speed ratio for zCFD to calculate the momentum sources associated with the turbine. By default the reference point is automatically located 1.0 turbine diameters upstream of the disc centre, assuming that the reference wind direction is also the local wind direction. This is easy to modify.

The '*centre*' is the centre of the disc, which is automatically determined from the nominal hub height in the TRBX file as an offset to the ground height at the specified location. The local ground height is automatically determined from a supplied VTK files, or from a previous solver run. Note that the VTK ground data can be created with a single cycle of the solver, and does not need to include any turbines.

The vertical orientation is defined by the '*up*' vector - normally this will be the unit vector in the *z*-direction. The '*normal*' defines the vector perpendicular to the disc. The inner and outer radii are based on the TRBX definition of the size of the disc. No account is made of the hub or tower geometry.

For the *blade element model* the TRBX file is replaced with a dictionary that contains information about the blades and aerofoil sections used on the rotor, plus information that can be specified to inform the controller how to adjust the rotation rate and blade pitch each cycle:

```
farm.create_trbx_zcfд_input(
    case_name='windfarm_name',
    wind_direction=280.0,
    reference_wind_speed=8.0,
    terrain_file=None,
    report_frequency=100,
    update_frequency=1,
    turbine_zone_length_factor=2.5,
    model='blade element theory',
    turbine_files=[['data/xy_file_1.txt', bet_dict_1],
                  ['data/xy_file_2.txt', bet_dict_2]],
)
```

The *xy_file* is as above and the *bet_dict* is a Python dictionary with details of the rotor and blade, including sectional aerodynamic characteristics and dimensions:

```

bet_dict = {
    'name': 'T001',
    'status': 'on',
    'number of blades': 3,
    'rotation direction': 'clockwise',
    'hub height': 70.0,
    'inner radius': 4.0,
    'outer radius': 40.0,
    'rated power': 2000000.0,
    'tip speed limit': 80.0,
    'verbose': False,
    'tilt': 6.0,
    'yaw': 0.0,
    'auto yaw': True,
    'cut in speed': 4.0,
    'cut out speed': 25.0,
    'number of segments': 12,
    'up': [0.0, 0.0, 1.0],
    'aerofoil profile':
        {'source': 'http://airfoilttools.com/airfoil/details?airfoil=n63415-il',
         'upper surface': [[0.0, 0.0], [0.003, 0.01287], ..., [0.95028, 0.00931], [1.0, 0.0]],
         'lower surface': [[0.0, 0.0], [0.007, -0.01087], ..., [0.94972, 0.00333], [1.0, 0.0]]},
        'aerofoil cl': [[-90.0, -0.0], [-49.869, -0.202], ..., [49.907, 1.823], [90.0, 0.0]],
        'aerofoil cd': [[-90.0, 2.0], [-70.099, 1.796], ..., [60.11, 1.645], [90.087, 1.996]],
        'blade chord': [[0.0, 0.1], [0.1, 0.1], ..., [0.7, 0.034], [1.0, 0.018]],
        'blade twist': [[0.0, 21.924], [0.1, 21.924], ..., [0.7, 2.461], [1.0, 0.023]],
        'blade pitch range': [-10.0, 10.0],
        'blade pitch': 1.0,
        'blade pitch tol': 0.01,
        'blade pitch step': 0.1,
        'mean blade material density': 200.0,
        'blade pitch tol': 0.01,
        'dt': 0.2,
        'inertia': True,
        'damage ti': 0.15,
        'damage speed': 10.0,
        'friction loss': 0.01,
        'thrust factor': 1.10,
        'tip loss correction': 'acos-fit',
        'rpm ramp': [[4.0, 6.0], [13.0, 16.0]]},
}

```

The ‘name’ key is used to tag results associated with the turbine.

The ‘status’ key indicates whether the turbine should be active or not. This makes it straightforward to turn a turbine off without regenerating the model. The ‘number of blades’ is typically 3 (which is also the default value) but some turbines have 2 blades. The ‘rotation direction’ is typically clockwise (when viewed from the front) for modern turbines, but the keyword values allow ‘clockwise’ or ‘anticlockwise’. This alters the rotation of the wake.

The ‘hub height’, ‘inner radius’ and ‘outer radius’ define the vertical offset from the ground in metres, and the rotor dimensions respectively as above. The ‘rated power’ (Watts) of the turbine defines an operating limit of power production that will not be exceeded.

The ‘tip speed limit’ is an environmental limit on the rate of rotor rotation, and is typically set to 80m/s. Note

that the tip speed is based on the absolute rotation rate of the rotor, and the actual tip speed relative to the air may be increased by angular induction.

The model can produce various diagnostics. Setting '*verbose*' to *True* will output these.

The rotor '*tilt*' is an upward rotation of the entire rotor, to allow for tower clearance under blade deflection. The value is typically a few degrees. The '*yaw*' is a rotation angle in degrees from the prescribed wind direction, with the option to specify '*auto yaw*' which will automatically over-ride the '*yaw*' value with a rotation into the local (disc average) wind direction. A warning is issued if '*auto yaw*' requests a '*yaw*' value exceeding 10 degrees in either direction.

The manufacturer typically specifies a '*cut in speed*' and '*cut out speed*' range for the local reference wind speed (for blade elements this is the average windspeed on the rotor disc). The rotor is automatically stowed outside these limits.

The '*number of segments*' defines the resolution of the actuator disc discretisation (not the computational domain mesh) with a default value of 12.

The '*aerofoil profile*' is used to integrate the blade unit sectional area between the '*upper surface*' and '*lower surface*' over the range [0,1]. Once integrated, the value is stored in the turbine zone dictionary. The optional '*source*' can be provided as a reference for the data. The sectional area multiplied by local radius is integrated over the length of each blade and multiplied by the '*mean blade material density*' (kilograms per cubic metre, typically 200.0) to provide a value for the '*rotor moment*'.

The turbine controller assumes that the physical time between iterations is '*dt*'. This does not have to be equal to the CFD timestep. If '*inertia*' is *True* then the available torque is divided by the '*rotor moment*' to determine the rotor acceleration. Otherwise the rotor is accelerated with a 10% increase in rotational rate capped per iteration. When the rotor is positively accelerated, half of the available torque is used to accelerate the rotor and half is supplied to the generator (unless doing so would exceed the rated power). This is an arbitrary value and it is expected that users will modify this behaviour when modelling more sophisticated control laws.

The aerodynamic characteristics of the aerofoil section are critical to the blade element theory model and are not calculated automatically. These should be specified as points on the '*aerofoil cl*' (local lift coefficient) and '*aerofoil cd*' (local drag coefficient) curves over a range of angles of attack in degrees covering at least (-30,30). Values beyond this range will be extrapolated as needed.

The '*blade chord*' is the local chord length in metres used to scale the aerofoil along the normalised blade length [0,1]. The '*blade twist*' is the twist angle in degrees of the blade over the normalised blade length [0,1].

The turbine controller will automatically optimise the power production (up to the limit) by pitching the blades over the range '*blade pitch range*' in degrees using a Golden Section Search to a tolerance of '*blade pitch tol*' degrees. All blades are pitched equally and the optimum pitch value accounts for the contribution of all blades averaged over the disc. The '*blade pitch*' can be set initially and a step size limit '*blade pitch step*' can be set to the range over which the blade pitch is optimised to within '*blade pitch tol*' on each iteration. Typically the blades are pitched to their maximum setting on startup and then the pitch value is reduced as the rotor accelerates. A negative value of '*blade pitch*' may be achieved at high rates of rotation as the local angle of attack of the blade will still be positive. A warning is issued if the aerofoil angle of attack is less than zero anywhere on the rotor.

The turbine controller will determine whether (anywhere on the disc) the '*damage ti*' (turbulence intensity) and '*damage speed*' are simultaneously exceeded. If so, a warning is issued and the rotor is stowed. It is intended that this function may be modified by users to provide more sophisticated controller rules and wind farm behaviour.

To account for frictional losses in the system, 1% (or other value of '*friction loss*') is applied to the rate of rotor rotation each iteration.

The '*blade element theory*' fluid zones follow the '*simple*' and '*induction*' model example above, with inclusion of the '*blade element theory*' parameters.

A '*thrust factor*' can be applied to the rotor to include an approximation for the effect of the tower and the hub on the obstruction to airflow past the turbine.

The blade element theory model assumed 2D flow over each aerofoil section, and a 3D correction can be applied at the rotor tips. Options for the '*tip loss correction*' keyword are '*elliptic*' (which is not recommended

for most modern turbines), ‘acos-fit’, ‘acos shift-fit’ and ‘f-fit’. Definitions for these functions are provided in the Reference Guide.

In addition to the tip speed limit imposed by environmental considerations, some manufacturers also apply a calibrated ramp factor to the revolutions per minute that the rotor can achieve for any given freestream hub-height wind speed. This is to avoid overspeeds at low wind speeds. The ‘rpm ramp’ keyword takes a pair of points [[cut in speed, lowest rpm],[full speed, rpm at full speed]] defining the ends of the linear limit curve applied.

For all models, the *turbine_vtk/<turbine>.vtp* file defining the fluid zone for each turbine is also automatically created.

The utility also automatically creates a set of monitor points for each turbine, all in a single file (*<case_name>_probes.py*):

```
turb_probe = {
    'report' : {
        'frequency' : 100,
        'monitor' : [
            {
                'MR_1' : {
                    'name' : 'probe1@MHH@87',
                    'point' : [251865.0, 646486.0, 338.5],
                    'variables' : ['V', 'ti'],
                },
                ...
            }
        ]
}
```

The ‘frequency’ is the number of solver cycles between outputs, and the ‘monitor’ defines the name of the probe using the WindFarmer standard notation.

Because the zone and probe files are automatically created, the following lines must be added to the end of the standard zCFD parameter definition file *<case_name>.py* to insert the data:

```
z = zutil.get_zone_info('<case_name>_zones')
for key,value in z.turb_zone.items():
    parameters[key]=value

p = zutil.get_zone_info('<case_name>_probes')
for key,value in p.turb_probe.items():
    parameters[key]=value
```

When run, zCFD will include the probe data in the *<case_name>_report.csv* file. Note that this utility may take a few seconds to run, especially if there are large numbers of turbines, or points on the mesh boundary.

Writing WindFarmer Data Files

In order to export the data from a zCFD run in a format that can be read by WindFarmer, we provide the utility *write_windfarmer_data*, with usage:

```
import zutil.farm as farm
farm.write_windfarmer_data(case_name='windfarm_name',
                           num_processes=48,
                           up = [0,0,1])
```

The *up* vector is used to check that the orientation expected by WindFarmer is that same as the orientation used in the simulation. In most cases this will be the *z*-axis.

The utility will output the probe information plus additional fields, calculated automatically.

1.11.2 Propellor Modelling

Overview

Propellers are similar to wind turbines except that they are designed to produce thrust rather than torque from lift. zCFD includes a blade element fluid zone model based on an actuator disc, for propellers:

```
propellor_zones = {
    'FZ_1': {
        'name': 'R01',
        'type': 'disc',
        'model': 'blade element propellor',
        'number of blades': 2,
        'status': 'on',
        'outer radius': 0.175,
        'inner radius': 0.039025,
        'def': './propellor_vtp/R01.vtp',
        'rotation direction': 'clockwise',
        'verbose': True,
        'update frequency': 1,
        'centre': [0.0, 0.0, 0.0],
        'reference plane': True,
        'normal': [0.0, 0.0, 1.0],
        'up': [1.0, 0.0, 0.0],
        'blade chord': [[0.2333, 0.1468], [0.2667, 0.1681], ..., [1.0, 0.0581]],
        'blade twist': [[0.2333, 28.6202], [0.2667, 25.5228], ..., [1.0, 0.0]],
        'aerofoil cd': [[-60.0, 1.4552], [-50.0, 1.0992], ..., [60.0, 2.3542]],
        'aerofoil cl': [[-60.0, -0.7978], [-50.0, -0.8195], ..., [60.0, 1.3818]],
        'number of segments': 24,
        'tip loss correction': 'elliptic',
        'tip loss correction radius': 0.8,
    }
}
```

The ‘*name*’ key is used to tag results associated with the propellor.

The ‘*type*’ defines the fluid zone type for propellers.

The ‘*model*’ should be set to ‘blade element propellor’.

The ‘*number of blades*’ can be any positive integer - typically 2, 3 or 4.

The ‘*status*’ can be set to on or off, as a quick way to disable a propellor without re-generating the model.

The ‘*inner radius*’ and ‘*outer radius*’ refer to the rotor disc.

The ‘*def*’ is a path to a file containing the VTP file defining the disc fluid zone geometry. This can easily be created interactively in ParaView or else using the *ParaView Simple* library in Python:

```
line = Line()
line.Resolution = 10
line.Point1 = (0.0, 0.0, -0.5 * 0.175)
line.Point2 = (0.0, 0.0, 0.5 * 0.175)
tube = Tube(Input=line)
tube.NumberofSides = 128
tube.Radius = 0.175
transform = Transform(Input=tube)
transform.Transform = "Transform"
transform.Transform.Rotate = [0.0, 0.0, 0.0]
transform.Transform.Translate = [0.0, 0.0, 0.0]
```

(continues on next page)

(continued from previous page)

```
writer = CreateWriter('./propellor_vtp/R01.vtp')
writer.Input = transform
writer.UpdatePipeline()
```

where the above code block would produce a fluid zone for a propellor with normal [0.0,0.0,1.0]. Note that general use of the ParaView rotate command should follow the (default) ZXY convention, for which the following is a reasonable estimate:

```
def rotation_matrix(axis, theta):
    # Rotation matrix for counterclockwise rotation about axis by theta (radians)
    axis = np.asarray(axis)
    axis = axis / math.sqrt(np.dot(axis, axis))
    a = math.cos(theta / 2.0)
    b, c, d = -axis * math.sin(theta / 2.0)
    aa, bb, cc, dd = a * a, b * b, c * c, d * d
    bc, ad, ac, ab, bd, cd = b * c, a * d, a * c, a * b, b * d, c * d
    return np.array([[aa + bb - cc - dd, 2 * (bc + ad), 2 * (bd - ac)],
                    [2 * (bc - ad), aa + cc - bb - dd, 2 * (cd + ab)],
                    [2 * (bd + ac), 2 * (cd - ab), aa + dd - bb - cc]])
```

```
def trans_rot(v, vec, ang):
    v2 = np.dot(rotation_matrix(vec, ang), v)
    return [v2[0], v2[1], v2[2]]
```

```
# ParaView rotation emulator - note that ParaView (i.e. VTK) uses the order ZXY for rotation.
```

```
def pv_rotate(pt,args):
    pt1 = trans_rot(pt,[0.0,0.0,1.0],math.radians(args[2]))
    x1 = trans_rot([1.0,0.0,0.0],[0.0,0.0,1.0],math.radians(args[2]))
    y1 = trans_rot([0.0,1.0,0.0],[0.0,0.0,1.0],math.radians(args[2]))
    pt2 = trans_rot(pt1,x1,math.radians(args[0]))
    y2 = trans_rot(y1,x1,math.radians(args[0]))
    pt3 = trans_rot(pt2,y2,math.radians(args[1]))
    return pt3
```

The '*rotation direction*' can be clockwise or anticlockwise. zCFD will apply the necessary orientations to the forces and wake rotation.

The '*verbose*' keyword (True or False) specifies the level of output from the model.

The '*update frequency*' for propellers should be 1 - meaning that the model is evaluated on every solver cycle.

The '*centre*' is the centre of the disc rotor in 3D, and the '*normal*' defines the orientation.

The '*up*' vector is used to define \$theta_0\$ on the rotor: the radial vector from the centre to the rotor tip at \$theta_0\$ is defined by the cross product of the up vector and disc normal. All forces on the disc are evaluated by incrementing \$theta\$ over the range \$[0,2pi]\$.

The '*blade chord*' and '*blade twist*' are each defined as a list of points over the domain [0,1]. Note that the twist is measured in degrees and the chord is scaled by the outer radius in zCFD, so the input has to be scaled for a radius of 1.

The '*aerofoil cl*' and '*aerofoil cd*' curves are lists of points defining the aerofoil characteristics. zCFD interpolates values between these points and will extrapolate if necessary.

The '*number of segments*' used in the model should be chosen to match the mesh spacing in the CFD mesh around the propellor. zCFD does not generate a new mesh in order to apply the propellor model.

A '*tip loss correction*' can be applied to the blade element model, and the only option is '*elliptic*'. The '*tip loss correction radius*' is the radius from which the tip correction is applied and has a default value of 0.8.