zCFDguide Documentation

Release v2023.01.3113-1192-g97cb9e

Zenotech Ltd

Apr 24, 2023

CONTENTS

CHAPTER

ONE

VERSION: V2023.01.3113-1192-G97CB9E

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 zCAAMesh 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 Installing zCFD

1.2.2 Licence

zCFD requires a license. If you do not already have a license, please contact mailto:zcfd@zenotech.com to obtain one. The license file provided can be placed in the same directory alongside the input files or else its location can be defined using the environment variables below

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

or

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

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.

1.3 Running zCFD

zCFD is run from the command line using a customised environment that automatically sets up all of the paths and file locations correctly. To make the execution of parallel jobs easier, we also provide the free tool *MyCluster* that simplifies the interaction with a range of job schedulers.

1.3.1 Activating the environment

To run zCFD, the *zCFD command line environment* must be active. The zCFD command line environment is initialised from a terminal window by sourcing the *activate* script in the zCFD installation directory. It does not matter where this directory is - all of the executable paths and file locations will be set automatically:

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

This sets up the environment to enable execution of the specific version. When this environment is active, you should see a prefix to your command prompt, such as:

```
(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.

1.3.2 Input Files

1.3.3 Input Validation

zCFD provides an input validation script which can be used to check the solver control dictionary before runtime 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:

```
validate_input input.py [-m mesh.h5]

Validating parameters dictionary...
Parameters dictionary is valid
Checking BC_, FR_, FZ_, IC_ and TR_ numbering...
IC_ numbering is correct
FR_ numbering is correct
BC_ numbering is correct
zCFD input file input.py 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.

1.3.4 Running locally

zCFD makes very effective use of a range of processor types and computer configurations. We provide a *smartlaunch.bsh* script with zCFD to automatically detect the processor type and the presence of any accelerators (like Nvidia GPUs or Intel Xeon Phi's).

Smartlaunch uses this information to calculate the number of OpenMP threads to set per MPI task and to perform NUMA binding to specific cores and memory banks.

The optimum number of execution tasks ($NTASKS below) should match the total number of sockets (usually two per node) **not** the number of cores. This configuration will also work for systems with accelerators present.

It is also recommended to always use any computational nodes in exclusive mode to achieve the best performance.

```
# PROBLEM_NAME is the name of the hdf5 file containing the mesh
# CASE_NAME is the name of the python control

run_zcfd --ntask 10 -p $PROBLEM_NAME -c $CASE_NAME
```

Alternatively, *MyCluster* is a powerful tool for setting up, running and monitoring zCFD jobs on a range of schedulers. We recommend this tool, and it is provided as part of the zCFD environment.

```
# PROBLEM_NAME is the name of the hdf5 file containing the mesh
# CASE_NAME is the name of the python control

cluster_run -p $PROBLEM_NAME -c $CASE_NAME -j mycluster_job.job
```

1.3.5 Running on a cluster

MyCluster

MyCluster from Zenotech is used to set up, run, monitor and manage parallel jobs on a range of high performance computing environments without the user having to know the details of the queueing systems and job scheduler commands.

MyCluster is distributed as part of zCFD, and is automatically available from within the zCFD command line environment. Alternatively, MyCluster can be downloaded and installed separately.

To use MyCluster within the zCFD command line environment, you first need to configure it with your user details. This is so that MyCluster can keep track of your jobs, and email you with alerts. This step only needs to be done once per user.

First enter your details:

```
(zCFD): mycluster --firstname <Your First Name>
(zCFD): mycluster --lastname <Your Last Name>
(zCFD): mycluster --email <Your Email Address>
```

MyCluster generates and uses *job* files for zCFD (and other software packages) for each different computer system. To set up a job file for zCFD, enter (for example):

```
(zCFD): mycluster --create=my-job.job \
                  --jobname=my-job \
                  --project=my-project \
                  --jobqueue=my-job-queue \
                  --ntasks=12 \
                  --taskpernode=2 \
                  --script=mycluster-zcfd.bsh \
                  --maxtime=12
```

This will create a *job* file called *my-job.job*.

The *my-job.job* file can be called whatever you like, but avoid the use of spaces and other special characters in the title. The *project* is a billing code used on the system. If you do not know what this code should be, ask your system administrator.

The *jobqueue* is the name of the local job scheduler on the system (for example *slurm*, *moab* or *pbs*). To find out which local job scheduler is running on your system, type:

```
(zCFD): mycluster -q
```

This will also provide a list of *job queues*, and show how much resource is currently available on each.

The *ntasks* setting is the total number of processors (sockets) that you want to use. For zCFD this will be equal to the number of mesh partitions that are generated. The number of tasks per node should match the number of compute devices per node (CPU sockets, Xeon Phi devices or Nvidia GPU devices). Usually there are two sockets per node, so set *taskpernode* equal to 2. Your system administrator should advise if the local setup is different to this.

The *script* is software specific. The *mycluster-zcfd.bsh* script is included (along with scripts for a number of other CFD codes) in the MyCluster distribution in the *share/* directory, so you do not need to alter the *script* setting if you are running zCFD.

The *maxtime* setting is used to prevent failed jobs from over-running and to allow schedulers to prioritise short jobs where they can be accommodated between larger jobs. The default setting is 12 hours *(maxtime=12)* but you can specify any amount of time in hours (up to the local queue maximum) can be used. The actual limit will be shown in the *job* file, and it is worth checking whether this is adequate.

Once created, the *job* file can be re-used for similar simulation tasks on the same computer. If any of the parameters change (size of job, project, etc.) then you will need to create a new *job* file.

To run zCFD using MyCluster, make sure that the *job* file is in the directory containing the mesh and input files, change to that directory and enter (for example):

Example usage:

```
(zCFD): (export PROBLEM=my-mesh.h5;\
         export CASE=my-case;\
         mycluster --submit=my-job.job)
```

where *my-mesh.h5* is the name of the HDF5 mesh file and *my-case.py* is the name of the python input parameters file. This command will submit the job to the queue.

To track the progress of your jobs via MyCluster, enter:

```
(zCFD): mycluster -p
```

To delete a job from the queue, check the *Job ID* using *mycluster -p* and then enter:

```
(zCFD): mycluster -d <Job ID>
```

A full list of MyCluster commands can be obtained:

```
(zCFD): mycluster --help
```

1.3.6 Using the zCFD Python Environment

1.3.7 Using Juypter

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. 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_notebook
```

1.4 Getting Started

The getting started guide is here to teach the fundementals 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 Quick Start Guide

1. downloaded
2. unpack
3. Download validation case
4. Run it locally

1.4.2 More complex case

1. downloaded
2. unpack
3. Download validation case
4. Run it locally

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 Files*.

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 },
    'equations': { Equations },
    'material': { Material Specification },
    'IC_1': {Initial Conditions}
    'BC_1': {Boundary Conditions}
    'write output': {Write Output}
    'report': {Reporting}
}
```

1.5.1 Reference Settings

Reference state

Keyword	Required	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	Required	Default	Valid values
'name'	No	'IC_1'	"IC_?" where ? is an integer.
'func'	No	–	An (optional) initial conditions function.

```
# set inititial 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 provde 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	Required	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	Required	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	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	Required	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' : {
```

(continues on next page)

(continued from previous page)

```
                        # 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' : {
                '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],
```

(continues on next page)

(continued from previous page)

```
        # 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
                            'roughness length' : 0.05,
                            # Fiction velocity
                            'friction velocity' : 2.0,
                            # Surface Layer Heigh
                            '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,
         '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,
              },
```

(continues on next page)

(continued from previous page)

```
        '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
        },
'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": {
            "unsteady": {...},
            "scheme": {...},
            'cfl': 30.0,
            "multigrid": {...}
            "cycles": 3000,
        },
        ...
}
```

CFL Control

The Courant-Friedrichs-Lewy (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	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 *ramp function*.
'cfl coarse'	No	Current CFL number	Any positive floating point number.

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 with high residuals on the turbulence equations, 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
```

Solver scheme

There are three different time marching schemes available in zCFD: Explicit Euler, various orders of Runge Kutta (like Euler…) and the implicit Euler scheme. These can be run with dual time-stepping for both steady and unsteady simulations. Explicit schemes with no preconditioning can be used with global time-stepping for unsteady simulations. The choice of scheme and time stepping algorithm affects which dictionary keys are valid.

```
'time marching' : {
                  'scheme' : {
                              # Either 'euler' or 'runge kutta' or 'implicit euler'
                              'name' : 'runge kutta',
                              # Number of RK stages 'euler' or 1, 4, 5, 'rk third␣
↪order tvd'
                              'stage': 5,
                              # Time-stepping scheme: 'local timestepping' for steady␣
↪state or dual time step time accurate
                              # 'global timestepping' for time accurate
                              'kind' : 'local timestepping',
                              # With AMGX, the linearsolver can be run in single or␣
↪double precision modes
                              'double precision linear solve': True,
                              },
                  },
```

When global time-stepping is specified the keys `['unsteady']['order']` and `['unsteady']['start']` are not valid. The CFL keys in in the *CFL* section below are also not valid as the time step is explicitly specified. When the `['unsteady']['total time']` and `['unsteady']['time step']` keys are set to be equal values and `['time marching']['scheme']['kind']` is set to `'local timestepping'` the solver marches in

pseudo time towards a steady state solution. If implicit euler is selected then the `'multigrid'` keys must be removed. For large CFL numbers, it is recommended the Roe inviscid flux scheme is used.

Cycles

For steady-state simulations, the number of pseudo time cycles is the same as the number of steps that the solver should use to reach a converged solution. Note that the solver uses local pseudo time-stepping (the time-step varies according to local conditions) so any intermediate solution is not necessarily time-accurate.

For unsteady (time-accurate) simulations using 'dual time-stepping' to advance the solution in time the number of pseudo time cycles determines the number of inner iterations that the solver uses to converge each real time step. When global time-stepping is specified the cycles key is not valid.

```
'time marching' : {
                  # Required only for dual time-stepping:
                  # Number of pseudo time cycles
                  'cycles' : 5000,
},
```

Multigrid

The mesh is automatically coarsened by merging cells on successive layers in order to accelerate solver convergence. This does not alter the accuracy of the solution on the finest (original) mesh. Advanced users can control the number of geometric multigrid levels and the 'prolongation' of quantities from coarse to fine meshes.

```
'time marching' : {
                 # Maximum number of meshes (including fine mesh)
                 'multigrid' : 10,
                 # Optional:
                 # Number of multigrid cycles before solving on fine mesh only
                 'multigrid cycles' : 5000,
                 # Optional (default 0.75), Advanced:
                 # Prolongation factor
                 'prolong factor' : 0.75,
                 # Optional (default 0.3), Advanced:
                 # Prolongation factor for transported quantities
                 'prolong transport factor' : 0.3,
                },
```

Polynomial Multigrid

The polynomial basis on which the solution is computed can be successively coarsened, thus allowing the solution to be evolved quicker due to weakened stability restrictions at lower polynomial orders. This does not alter the accuracy of the solution on the highest polynomial order.

```
'time marching' : {
                   # Optional (default False):
                   # Switch on polynomial multigrid
                   'multipoly' : True,
                   },
```

1.5.4 Equations

Keyword	Required	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.

Value	Description
"euler"	Use the Euler flow equations in the simulation. These are compressible, inviscid flow equations.
"rans"	Use the RANS (Reynolds-Averaged Navier-Stokes) equations in the simulation. These are compressible, viscid 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.
"viscous"	Use the viscous flow equations. These are compressible, viscid flow equations which do not model turbulence.
"les"	
"dgviscous"	
'dgrans'	
'dgles'	
'incomp viscous'	
'incomp rans'	
'incomp les'	
'dgcaa'	

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
'limiter'	Yes	'vanalbada'	'vanalbada'
'order'	Yes	–	'first','second','euler_second'
'first order cycles'			
'linear gradients'			
'leastsq gradients'			
'inviscid flux scheme'			
'roe low dissipation sensor'			
'roe dissipation sensor minimum'			
'roe entropy fix coefficient'			
'freeze limiter cycle'			

These values are valid for all equation 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.

```
'euler' : {
            # Spatial accuracy (options: first, second)
            'order' : 'second',
            # Optional (default 0):
            # Run first order spatial accuracy for the first x cycles
            'first order cycles': 100,
            # Optional (default 'vanalbada'):
            # MUSCL limiter (options: vanalbada)
            'limiter' : 'vanalbada',
            # Optional (default off)
            # Cycle on which to freeze the limiter values
            'freeze limiter cycle': 500,
            # Optional (default False):
            # Use low speed mach preconditioner
            'precondition' : True,
            # Optional (default False):
            # Use linear gradients
            'linear gradients' : False,
            # Optional (default 'HLLC'):
            # Scheme for inviscid flux: HLLC, Rusanov or Roe
            '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 does have to resolve the boundary layer, but the solver will run faster as fewer equations are being included.

```
'viscous' : {
              # Spatial accuracy (options: first, second)
              'order' : 'second',
              # Optional (default 0):
              # Run first order spatial accuracy for the first x cycles
              'first order cycles': 100,
              # Optional (default 'vanalbada'):
              # MUSCL limiter (options: vanalbada)
              'limiter' : 'vanalbada',
              # Optional (default False):
              # Use low speed mach preconditioner
              'precondition' : True,
              # Optional (default False):
              # Use linear gradients
              'linear gradients' : False,
              # Optional (default 'HLLC'):
              # Scheme for inviscid flux: HLLC or Rusanov
              'Inviscid Flux Scheme': 'HLLC',
            },
```

RANS

The fully turbulent (Reynolds Averaged Navier-Stokes Equations)

```
'RANS' : {
            # Spatial accuracy (options: first, second, euler_second)
            'order' : 'second',
            # Optional (default 0):
            # Run first order spatial accuracy for the first x cycles
            'first order cycles': 100,
            # Optional (default 'vanalbada'):
            # MUSCL limiter (options: vanalbada)
            'limiter' : 'vanalbada',
            # Optional (default off)
            # Cycle on which to freeze the limiter values
            'freeze limiter cycle': 500,
            # Optional (default False):
            # Use low speed mach preconditioner
            'precondition' : True,
            # Optional (default False):
            # Use linear gradients
            'linear gradients' : False,
            # Optional (default False):
            # Use least squares gradients
            'leastsq gradients' : False,
            # Optional (default 'HLLC'):
            # Scheme for inviscid flux: HLLC, Rusanov, Roe or Roe low diss
            'Inviscid Flux Scheme': 'HLLC',
            # Optional (default 'NONE')
            # Roe low dissipation sensor type: 'FD' (wall distance based), 'NTS'␣
```

(continues on next page)

(continued from previous page)

```
↪(vorticity based), 'DES' (DES blending function based) or 'NONE'
            'roe low dissipation sensor': 'DES',
            # Optional (default 0.05)
            # Minimum value of the Roe low dissipation sensor
            'roe dissipation sensor minimum': 0.05,
            # Turbulence
            'turbulence' : {
                                # turbulence model (options: 'sst', 'sa-neg', 'sst-
↪transition')
                                'model' : 'sst',
                                # Optional (default 'none'):
                                # LES model (options 'none', 'DES', 'DDES', 'SAS')
                                'les' : 'none',
                                # Optional (default 0.09):
                                # betastar turbulence closure constant
                                'betastar' : 0.09,
                                # Optional (default True):
                                # turn off mu_t limiter
                                'limit mut' : False,
                                # Optional (default CDES=0.65, CDES_kw=0.78, CDES_
↪keps=0.61):
                                # DES constants
                                'CDES': 0.65
                                'CDES_kw': 0.78,
                                'CDES_keps': 0.61,
                                # Optional (default Cd1=20.0, Cd2=3, Cw= 0.15):
                                # (I)DDES constants
                                'Cd1': 20.0,
                                'Cd2': 3,
                                'Cw': 0.15,
                                # Optional when using 'sst' (default 0):
                                # Menter SST production term: 0=SST-V, 1=incompressible,
↪ 2=SST
                                'production': 0,
                                # Optional when using 'sa-neg' (default False):
                                # Use sa-neg rotation correction
                                'rotation correction': True,
                                # Optional when using sa-neg (default 0.0):
                                # Limit the sa-neg gradient based on the maximum value␣
↪between neighbouring cells.
                                # k = 0.0 corresponds to no limiting, k = 1.0 maximum␣
↪limiting.
                                'limit gradient k': 0.5
                                # Option when using 'sst-transition'
                                # Transition model constants
                                'ca1': 2.0,
                                'ca2': 0.06,
                                'ce1': 1.0,
                                'ce2': 50.0,
                                'cthetat': 0.03,
                                'sigmagamma': 1.0,
                                'sigmathetat': 2.0,
                                # Option when using 'sst-transition'
                                # Turn transition separation correction model on/off
                                'separation correction': True,
                                # Option when using 'sst' or 'sa-neg'
```

(continues on next page)

(continued from previous page)

```
                                # Use the non-linear Quadratic Constitutive Relation
                                # Spalart, P. R., "Strategies for Turbulence Modelling␣
↪and Simulation,"
                                # International Journal of Heat and Fluid Flow, Vol. 21,
↪ 2000, pp. 252-263,
                                # https://doi.org/10.1016/S0142-727X(00)00007-2
                                'qcr': True
                            },
            },
```

LES

The filtered Large Eddy Simulation (LES) equations

```
'LES' : {
                # Spatial accuracy (options: first, second)
                'order' : 'second',
                # Optional (default 'vanalbada'):
                # MUSCL limiter (options: vanalbada)
                'limiter' : 'vanalbada',
                # Optional (default False):
                # Use low speed mach preconditioner
                'precondition' : True,
                # Optional (default False):
                # Use linear gradients
                'linear gradients' : False,
                # Optional (default False):
                # Use least squares gradients
                'leastsq gradients' : False,
                # Optional (default 'HLLC'):
                # Scheme for inviscid flux: HLLC, Rusanov, Roe or Roe low diss
                'Inviscid Flux Scheme': 'HLLC',
                # Optional (default 'NONE')
                # Roe low dissipation sensor type: 'FD' (wall distance based), 'NTS'␣
↪(vorticity based), 'DES' (DES blending function based) or 'NONE'
                'roe low dissipation sensor': 'DES',
                # Optional (default 0.05)
                # Minimum value of the Roe low dissipation sensor
                'roe dissipation sensor minimum': 0.05,
                'turbulence' : {
                                # LES model (options 'none', 'WALE')
                                'les' : 'none',
                            },

            },
```

High Order

When using the high order strong form Discontinuous Galerkin/Flux Reconstruction solver the keys in the 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. These additional keys are given below

```
'DG...' : {
                # Spatial polynomial order 0,1,2,3
                'order' : 2,
                # Optional (default 0.0)
                'c11 stability parameter': 0.0,
                # Optional (default 0.5)
                'LDG upwind parameter': 0.5,
                # Optional (default False)
                'Shock Sensing': False,
                'Shock Sensing k': 1.0,
                'Shock Sensing Viscosity Scale': 1.0,
                # Variable used for shock sensing, one of: 'density', 'temperature',
↪'mach', 'turbulence'
                'Shock Sensing Variable': 'density',
            },
```

Hence all the available keys for `'DGviscous'` are a combination of the `'viscous'` keys and the `'DG...'` keys. The same logic applies to the `'DGRANS'`, and `'DGLES'` equation sets.

```
'DGviscous' : {
                # Spatial polynomial order 0,1,2,3
                'order' : 2,
                # Optional (default False)
                # Use low speed mach preconditioner
                'precondition' : True,
                # Optional (default 'HLLC')
                # scheme for inviscid flux: HLLC or Rusanov
                'Inviscid Flux Scheme': 'HLLC',
                # Optional (default 0.0)
                'c11 stability parameter': 0.0,
                # Optional (default 0.5)
                'LDG upwind parameter': 0.5,
                # Optional (default False)
                'Shock Sensing': False,
                'Shock Sensing k': 1.0,
                'Shock Sensing Viscosity Scale': 1.0,
                # Variable used for shock sensing, one of: 'density', 'temperature',
↪'mach', 'turbulence'
                'Shock Sensing Variable': 'density',
                # Optional (default False)
                'inviscid flow': False,
                # Optional (default False)
                'freeze diffusion during rk stages': False
            },
```

DG Nodal locations

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

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

To use the default values

```
'Nodal Locations' : nodal_locations_default['Nodal Locations']
```

Or select from the available types

```
'Nodal Locations' : {
                      # Options line_evenly_spaced, line_gauss_lobatto or line_gauss_
↪legendre_lobatto
                      'Line':  line_gauss_legendre_lobatto,
                      # Options tet_evenly_spaced, tet_shunn_ham
                      'Tetrahedron': tet_evenly_spaced,
                      # Options tri_evenly_spaced, tri_shunn_ham
                      'Tri' : tri_evenly_spaced,
                    },
```

DGCAA

1.5.5 Material Specification

The user can specify any fluid material properties by following the (default) scheme for 'air':

```
'material' : 'air',
```

Options

```
'air' : {
          'gamma' : 1.4,
          'gas constant' : 287.0,
          'Sutherlands const': 110.4,
          'Prandtl No' : 0.72,
          'Turbulent Prandtl No' : 0.9,
        },
```

1.5.6 Initial Conditions

Initialisation

Initial conditions are used to set the flow variable values in all cells at the start of a simulation. The initial conditions default to the reference state (above) if not specified

```
# Initial conditions
'initial' : 'IC_2',
```

See `**Initial Conditions**`_. A function which defines the initial fields may be supplied

```
# Initial conditions
'initial' : {
              # Name of initial condition
              'name' : 'IC_1',
              # Optional: User defined function
              'func' : my_initialisation,
            },
```

Example User Defined Initialisation Function

```
def my_initialisation(**kwargs):

    # Dimensional primitive variables
    pressure = kwargs['pressure']
    temperature = kwargs['temperature']
    velocity = kwargs['velocity']
    wall_distance = kwargs['wall_distance']
    location = kwargs['location']

    if location[0] > 10.0:
      velocity[0] *= 2.0

    # Return a dictionary with user defined quantity.
    # Same schema as above
    return { 'velocity' : velocity }
```

Initial Conditions

The initial condition properties are defined using consecutively numbered blocks

```
'IC_1' : {....},
'IC_2' : {....},
'IC_3' : {....},
```

Each block can contain the following options

```
# Required: Static temperature in Kelvin
'temperature': 293.0,
# Required: Static pressure in Pascals
'pressure':101325.0,
```

```
# Fluid velocity
'V': {
        # Velocity vector
        'vector' : [1.0,0.0,0.0],
        # Optional: specifies velocity magnitude
        'Mach' : 0.20,
      },
```

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,
```

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,
```

The user can also provide functions to specify a 'wall-function' - or the turbulence viscosity profile near a boundary. For example an atmospheric boundary layer (ABL) could be specified like this:

```
'profile' : {
             'ABL' : {
                       'roughness length' : 0.0003,
                       'friction velocity' : 0.4,
                       'surface layer height' : -1.0,
                       'Monin-Obukhov length' : -1.0,
                       # Non dimensional TKE/friction velocity**2
                       'TKE' : 0.928,
                       # ground level (optional if not set wall distance is used)
                       'z0'  : -0.75,
                      },
            },
```

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

Note: The conditions in the VTK file are specified by node based arrays with names 'Pressure', 'Temperature', 'Velocity', 'TI' and 'EddyViscosity'. Note the field will override the conditions specified previously therefore the user can specify only the conditions that are different from default.

Certain conditions are specified relative to a reference set of conditions

```
'reference' : 'IC_1',
# total pressure/reference static pressure
# This can be a single value or VTK field file name with array name
↪'TotalPressureRatio' or function
'total pressure ratio' : 1.0,
# total temperature/reference static temperature
# This can be a single value or VTK field file name with array name
↪'TotalTemperatureRatio' or function
'total temperature ratio' : 1.0,
# Mach number
'mach' : 0.5,
# Direction vector or function
'vector' : [1.0,0.0,0.0],
```

```
'reference' : 'IC_1',
# static pressure/reference static pressure
# This can be a single value or VTK field file name with array name
↪'StaticPressureRatio' or function
'static pressure ratio' : 1.0,
```

```
'reference' : 'IC_1',
# Mass flow ratio
'mass flow ratio' : 1.0,
# Or mass flow rate
'mass flow rate' : 10.0
```

```
# Example pressure ratio function
def total_pressure_ratio(x, y, z):

    ratio = 1.0
    if x < 10.0:
        ratio = 0.9

    return ratio
```

Note: $W^* = \frac{\dot{m}}{\rho V}$

Driven Initial Conditions

Another way of prescribing a farfield initial condition is by defining a 'driving function', 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. For a driving function `lift_target_ic_func`, the IC should be set up as follows in the parameters dictionary.

```
'IC_2' : lift_target_ic_func
```

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 acheived.

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 propogate 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 propogation of adjusted farfield conditions through the domain.

```
def lift_target_ic_func(**kwargs):
    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␣
↪flow in the _report.csv
                   },
    }

    return dict_out
```

Note: Driven initial conditions cannot be used as reference conditions, to initialise the flow or within a restarted simulation.

1.5.7 Write Output

1.5.8 Reporting

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' : {
            # Report frequency
            'frequency' : 1,
            # Extract specified variable at fixed locations
            'monitor' : {
                        # Consecutively numbered blocks
                        'MR_1' : {
                                 # Required: Output name
                                 'name' : 'monitor_1',
                                 # Required: Location
                                 'point' : [1.0, 1.0, 1.0],
                                 # Required: Variables to be extracted
                                 'variables' : ['V','ti'],
                                 },
                        },

            # Report force coefficient in grid axis as well as using user defined␣
↪transform
            'forces' : {
                       # Consecutively numbered blocks
                       'FR_1' : {
                                # Required: Output name
                                'name' : 'force_1',
                                # Required: Zones to be included
                                'zone' : [1, 2, 3],
                                # Transformation function
                                'transform' : my_transform,
                                # Optional (default 1.0):
                                'reference area' : 0.112032,
                                # Optional (default 1.0):
                                'reference length' : 1.0,
                                # Optional (default [0.0, 0.0, 0.0]):
                                'reference point' : [0.0, 0.0, 0.0],
                                # Optional (default 0.0): Calculate forces using a␣
↪specified reference pressure rather than absolute
                                'reference pressure': 0.0
                                },
                       },

            # Report massflow ratio
            'massflow' : {
                         # Consecutively numbered blocks
                         'MF_1' : {
```

(continues on next page)

(continued from previous page)

```
                                # Required: Output name
                                'name' : 'massflow_1',
                                # Required: Zones to be included
                                'zone' : [1, 2 ,3],
                                },
                        },
        },
```

Example Transformation Function

A transformation function my be supplied to the force report block to transform the force into a different coordinate system, this a particularly useful for reporting lift and drag which are often rotated from the solver coordinate system.

```
# 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 {'v1' : v[0], 'v2' : v[1], 'v3' : v[2]}
```

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 reqiured. 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	*scale_mesh*
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 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	*ramp func*
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.trasport_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,
   ...
  }
 ...
}
```

1.6.3 Initialisation Function

Description	Provide the initial flow field variables on a cell-by-cell basis
Valid for parameter	*initial*
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 localtion [X, Y, Z]
  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,
   ...
   }
 ...
}
```

1.7 Control Dictionary (old)

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 Files*.

1.7.1 Reference Settings

Define units

```
# Optional (default 'SI'):
# units for dimensional quantities
'units' : 'SI',
```

Scale mesh

Keyword	Required	Default	Valid values
'scale'	No	[1,1,1]	[scaleX,scaleY,scaleZ] : Any list of positive integers 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*)

```
# Optional
# Scale function
'scale' : scale_func,
```

Example scale function

```
# Scale function
def scale_func(point):
    # Scale z coordinate by 100
    point[2] = point[2] * 100.0
    return {'coord' : point}
```

Reference state

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

See *Initial Conditions*

Partitioner

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

```
# Optional (default 'metis'):
# partitioner type
'partitioner' : 'metis',
```

Safe Mode

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

```
# Optional (default False):
# Safe mode
'safe' : False,
```

1.7.2 Initialisation

Initial conditions are used to set the flow variable values in all cells at the start of a simulation. The initial conditions default to the reference state (above) if not specified

```
# Initial conditions
'initial' : 'IC_2',
```

See *Initial Conditions*. A function which defines the initial fields may be supplied

```
# Initial conditions
'initial' : {
              # Name of initial condition
              'name' : 'IC_1',
              # Optional: User defined function
              'func' : my_initialisation,
            },
```

Example User Defined Initialisation Function

```
def my_initialisation(**kwargs):

    # Dimensional primitive variables
    pressure = kwargs['pressure']
    temperature = kwargs['temperature']
    velocity = kwargs['velocity']
    wall_distance = kwargs['wall_distance']
    location = kwargs['location']
```

(continues on next page)

(continued from previous page)

```
    if location[0] > 10.0:
      velocity[0] *= 2.0

    # Return a dictionary with user defined quantity.
    # Same schema as above
    return { 'velocity' : velocity }
```

To restart from a previous solution there are two options. A solve can be restarted on the same mesh using the options:

```
# Restart from previous solution
'restart' : True,
# Optional: Restart from a different case
'restart casename' : 'my_case',
```

Alternatively, a solution from another mesh can be mapped on to a new mesh using the options:

```
# Restart from previous solution on a different mesh
"interpolate restart": True,
# Required: Restart from a different case
"restart casename": "different_case",
# Required: Restart from a different mesh
"restart meshname": "different_mesh",
# Optional: Number of Laplace smoothing cycles to smooth initial mapped solution␣
↪(default 0)
"solution smooth cycles": 0,
```

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,
```

1.7.3 Low Mach number preconditioner settings (Optional)

The preconditioner is a mathematical technique for improving the speed of the solver when the fluid flow is slow compared to the speed of sound. Use of the preconditioner does not alter the final converged solution produced by the solver. The preconditioner factor is used to improve the robustness of the solver in regions of very low speed. The preconditioner is controlled in the `'equations'` dictionary (see *Equations*) by setting the `'precondition'` key, for example when using the euler solver

```
'euler' : {
          # Optional (default False):
          # Use low speed mach preconditioner
          'precondition' : True,
        },

# Optional, Advanced: Preconditioner factor
'preconditioner' : {
                    'minimum mach number' : 0.5,
                   },
```

1.7.4 Time Marcher

zCFD is capable of performing both steady-state and time accurate simulations. In the first case the solution is marched in pseudo time using the dual time-stepping algorithm towards the steady-state solution. For time accurate simulations the solution can be either globally time-stepped or use dual time-stepping. The `'time marching'` dictionary controls the selection of these schemes.

```
'time marching' : {....},
```

Time accurate (unsteady) simulation control

```
'time marching' : {
                  'unsteady' : {
                                # Total time in seconds
                                'total time' : 1.0,
                                # Time step in seconds
                                'time step' : 1.0,
                                # Required only when dual time-stepping:
                                # Dual time step time accuracy (options: 'first' or
↪'second' order)
                                'order' : 'second',
                                # Required only when dual time-stepping:
                                # Number of pseudo time (steady) cycles to run␣
↪before starting dual timestep time accurate simulation
                                'start' : 3000,
                                },
                  },
```

Solver scheme

There are three different time marching schemes available in zCFD: Explicit Euler, various orders of Runge Kutta (like Euler...) and the implicit Euler scheme. These can be run with dual time-stepping for both steady and unsteady simulations. Explicit schemes with no preconditioning can be used with global time-stepping for unsteady simulations. The choice of scheme and time stepping algorithm affects which dictionary keys are valid.

Runge Kutta

```
'time marching' : {
                  'scheme' : {
                              # Either 'euler' or 'runge kutta' or 'implicit euler'
                              'name' : 'runge kutta',
                              # Number of RK stages 'euler' or 1, 4, 5, 'rk third␣
↪order tvd'
                              'stage': 5,
                              # Time-stepping scheme: 'local timestepping' for steady␣
↪state or dual time step time accurate
                              # 'global timestepping' for time accurate
                              'kind' : 'local timestepping',
                              # With AMGX, the linearsolver can be run in single or␣
↪double precision modes
                              'double precision linear solve': True,
                              },
                  },
```

When global time-stepping is specified the keys `['unsteady']['order']` and `['unsteady']['start']` are not valid. The CFL keys in in the *CFL* section below are also not valid as the time step is explicitly specified. When the `['unsteady']['total time']` and `['unsteady']['time step']` keys are set to be equal values and `['time marching']['scheme']['kind']` is set to `'local timestepping'` the solver marches in pseudo time towards a steady state solution. If implicit euler is selected then the `'multigrid'` keys must be removed. For large CFL numbers, it is recommended the Roe inviscid flux scheme is used.

CFL

The Courant-Friedrichs-Lewy (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 default values should be appropriate for most cases, but for lower-quality meshes or very complex geometries it may be necessary to use lower values (e.g. CFL of 1.0). In such cases it may also be helpful to turn off Multigrid (above) by setting the maximum number of meshes to 0. The CFL dictionary keys are only valid when using dual time-stepping.

```
'time marching' : {
                  # Default max CFL number for all equations
                  'cfl': 2.5
                  # Optional (default 'cfl' value):
                  # Override max CFL number for transported quantities
                  'cfl transport' : 1.5,
                  # Optional (default 'cfl' value):
                  # Override max CFL number for coarse meshes
                  'cfl coarse' : 2.0,
                  # Control of CFL for polynomial multigrid (highest and lowest␣
↪order)
                  'multipolycfl' : [2.0,2.0],
                  # Optional: Ramp cfl
                  'cfl ramp factor': {
                                     # cfl = cfl_ini * growth ^ cycle
                                     'growth': 1.05,
                                     # min cfl
                                     'initial': 0.1,
                                     },
                  },
```

Cycles

For steady-state simulations, the number of pseudo time cycles is the same as the number of steps that the solver should use to reach a converged solution. Note that the solver uses local pseudo time-stepping (the time-step varies according to local conditions) so any intermediate solution is not necessarily time-accurate.

For unsteady (time-accurate) simulations using 'dual time-stepping' to advance the solution in time the number of pseudo time cycles determines the number of inner iterations that the solver uses to converge each real time step. When global time-stepping is specified the cycles key is not valid.

```
'time marching' : {
                  # Required only for dual time-stepping:
                  # Number of pseudo time cycles
                  'cycles' : 5000,
},
```

Multigrid

The mesh is automatically coarsened by merging cells on successive layers in order to accelerate solver convergence. This does not alter the accuracy of the solution on the finest (original) mesh. Advanced users can control the number of geometric multigrid levels and the 'prolongation' of quantities from coarse to fine meshes.

```
'time marching' : {
                # Maximum number of meshes (including fine mesh)
                'multigrid' : 10,
                # Optional:
                # Number of multigrid cycles before solving on fine mesh only
                'multigrid cycles' : 5000,
                # Optional (default 0.75), Advanced:
                # Prolongation factor
                'prolong factor' : 0.75,
                # Optional (default 0.3), Advanced:
                # Prolongation factor for transported quantities
                'prolong transport factor' : 0.3,
                },
```

Polynomial Multigrid

The polynomial basis on which the solution is computed can be successively coarsened, thus allowing the solution to be evolved quicker due to weakened stability restrictions at lower polynomial orders. This does not alter the accuracy of the solution on the highest polynomial order.

```
'time marching' : {
                  # Optional (default False):
                  # Switch on polynomial multigrid
                  'multipoly' : True,
                  },
```

1.7.5 Equations

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. For each equation type the valid keys are listed below.

```
# Governing equations to be used
# one of: RANS, euler, viscous, LES, DGRANS, DGviscous, DGLES, DGCAA
'equations' : 'RANS',
```

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.

```
'euler' : {
            # Spatial accuracy (options: first, second)
            'order' : 'second',
            # Optional (default 0):
            # Run first order spatial accuracy for the first x cycles
            'first order cycles': 100,
            # Optional (default 'vanalbada'):
            # MUSCL limiter (options: vanalbada)
```

(continues on next page)

(continued from previous page)

```
            'limiter' : 'vanalbada',
            # Optional (default off)
            # Cycle on which to freeze the limiter values
            'freeze limiter cycle': 500,
            # Optional (default False):
            # Use low speed mach preconditioner
            'precondition' : True,
            # Optional (default False):
            # Use linear gradients
            'linear gradients' : False,
            # Optional (default 'HLLC'):
            # Scheme for inviscid flux: HLLC, Rusanov or Roe
            'Inviscid Flux Scheme': 'HLLC',
          },
```

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 does have to resolve the boundary layer, but the solver will run faster as fewer equations are being included.

```
'viscous' : {
              # Spatial accuracy (options: first, second)
              'order' : 'second',
              # Optional (default 0):
              # Run first order spatial accuracy for the first x cycles
              'first order cycles': 100,
              # Optional (default 'vanalbada'):
              # MUSCL limiter (options: vanalbada)
              'limiter' : 'vanalbada',
              # Optional (default False):
              # Use low speed mach preconditioner
              'precondition' : True,
              # Optional (default False):
              # Use linear gradients
              'linear gradients' : False,
              # Optional (default 'HLLC'):
              # Scheme for inviscid flux: HLLC or Rusanov
              'Inviscid Flux Scheme': 'HLLC',
            },
```

The fully turbulent (Reynolds Averaged Navier-Stokes Equations)

```
'RANS' : {
            # Spatial accuracy (options: first, second, euler_second)
            'order' : 'second',
            # Optional (default 0):
            # Run first order spatial accuracy for the first x cycles
            'first order cycles': 100,
            # Optional (default 'vanalbada'):
            # MUSCL limiter (options: vanalbada)
            'limiter' : 'vanalbada',
            # Optional (default off)
            # Cycle on which to freeze the limiter values
            'freeze limiter cycle': 500,
            # Optional (default False):
            # Use low speed mach preconditioner
            'precondition' : True,
```

(continues on next page)

(continued from previous page)

```
            # Optional (default False):
            # Use linear gradients
            'linear gradients' : False,
            # Optional (default False):
            # Use least squares gradients
            'leastsq gradients' : False,
            # Optional (default 'HLLC'):
            # Scheme for inviscid flux: HLLC, Rusanov, Roe or Roe low diss
            'Inviscid Flux Scheme': 'HLLC',
            # Optional (default 'NONE')
            # Roe low dissipation sensor type: 'FD' (wall distance based), 'NTS'␣
↪(vorticity based), 'DES' (DES blending function based) or 'NONE'
            'roe low dissipation sensor': 'DES',
            # Optional (default 0.05)
            # Minimum value of the Roe low dissipation sensor
            'roe dissipation sensor minimum': 0.05,
            # Turbulence
            'turbulence' : {
                                # turbulence model (options: 'sst', 'sa-neg', 'sst-
↪transition')
                                'model' : 'sst',
                                # Optional (default 'none'):
                                # LES model (options 'none', 'DES', 'DDES', 'SAS')
                                'les' : 'none',
                                # Optional (default 0.09):
                                # betastar turbulence closure constant
                                'betastar' : 0.09,
                                # Optional (default True):
                                # turn off mu_t limiter
                                'limit mut' : False,
                                # Optional (default CDES=0.65, CDES_kw=0.78, CDES_
↪keps=0.61):
                                # DES constants
                                'CDES': 0.65
                                'CDES_kw': 0.78,
                                'CDES_keps': 0.61,
                                # Optional (default Cd1=20.0, Cd2=3, Cw= 0.15):
                                # (I)DDES constants
                                'Cd1': 20.0,
                                'Cd2': 3,
                                'Cw': 0.15,
                                # Optional when using 'sst' (default 0):
                                # Menter SST production term: 0=SST-V, 1=incompressible,
↪ 2=SST
                                'production': 0,
                                # Optional when using 'sa-neg' (default False):
                                # Use sa-neg rotation correction
                                'rotation correction': True,
                                # Optional when using sa-neg (default 0.0):
                                # Limit the sa-neg gradient based on the maximum value␣
↪between neighbouring cells.
                                # k = 0.0 corresponds to no limiting, k = 1.0 maximum␣
↪limiting.
                                'limit gradient k': 0.5
                                # Option when using 'sst-transition'
                                # Transition model constants
```

(continues on next page)

(continued from previous page)

```
                            'ca1': 2.0,
                            'ca2': 0.06,
                            'ce1': 1.0,
                            'ce2': 50.0,
                            'cthetat': 0.03,
                            'sigmagamma': 1.0,
                            'sigmathetat': 2.0,
                            # Option when using 'sst-transition'
                            # Turn transition separation correction model on/off
                            'separation correction': True,
                            # Option when using 'sst' or 'sa-neg'
                            # Use the non-linear Quadratic Constitutive Relation
                            # Spalart, P. R., "Strategies for Turbulence Modelling␣
↪and Simulation,"
                            # International Journal of Heat and Fluid Flow, Vol. 21,
↪ 2000, pp. 252-263,
                            # https://doi.org/10.1016/S0142-727X(00)00007-2
                            'qcr': True
                        },
        },
```

The filtered Large Eddy Simulation (LES) equations

```
'LES' : {
            # Spatial accuracy (options: first, second)
            'order' : 'second',
            # Optional (default 'vanalbada'):
            # MUSCL limiter (options: vanalbada)
            'limiter' : 'vanalbada',
            # Optional (default False):
            # Use low speed mach preconditioner
            'precondition' : True,
            # Optional (default False):
            # Use linear gradients
            'linear gradients' : False,
            # Optional (default False):
            # Use least squares gradients
            'leastsq gradients' : False,
            # Optional (default 'HLLC'):
            # Scheme for inviscid flux: HLLC, Rusanov, Roe or Roe low diss
            'Inviscid Flux Scheme': 'HLLC',
            # Optional (default 'NONE')
            # Roe low dissipation sensor type: 'FD' (wall distance based), 'NTS'␣
↪(vorticity based), 'DES' (DES blending function based) or 'NONE'
            'roe low dissipation sensor': 'DES',
            # Optional (default 0.05)
            # Minimum value of the Roe low dissipation sensor
            'roe dissipation sensor minimum': 0.05,
            'turbulence' : {
                            # LES model (options 'none', 'WALE')
                            'les' : 'none',
                        },

        },
```

When using the high order strong form Discontinuous Galerkin/Flux Reconstruction solver the keys in the 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. These additional keys are given below

```
'DG...' : {
                # Spatial polynomial order 0,1,2,3
                'order' : 2,
                # Optional (default 0.0)
                'c11 stability parameter': 0.0,
                # Optional (default 0.5)
                'LDG upwind parameter': 0.5,
                # Optional (default False)
                'Shock Sensing': False,
                'Shock Sensing k': 1.0,
                'Shock Sensing Viscosity Scale': 1.0,
                # Variable used for shock sensing, one of: 'density', 'temperature',
↪'mach', 'turbulence'
                'Shock Sensing Variable': 'density',
            },
```

Hence all the available keys for `'DGviscous'` are a combination of the `'viscous'` keys and the `'DG...'` keys. The same logic applies to the `'DGRANS'`, and `'DGLES'` equation sets.

```
'DGviscous' : {
                # Spatial polynomial order 0,1,2,3
                'order' : 2,
                # Optional (default False)
                # Use low speed mach preconditioner
                'precondition' : True,
                # Optional (default 'HLLC')
                # scheme for inviscid flux: HLLC or Rusanov
                'Inviscid Flux Scheme': 'HLLC',
                # Optional (default 0.0)
                'c11 stability parameter': 0.0,
                # Optional (default 0.5)
                'LDG upwind parameter': 0.5,
                # Optional (default False)
                'Shock Sensing': False,
                'Shock Sensing k': 1.0,
                'Shock Sensing Viscosity Scale': 1.0,
                # Variable used for shock sensing, one of: 'density', 'temperature',
↪'mach', 'turbulence'
                'Shock Sensing Variable': 'density',
                # Optional (default False)
                'inviscid flow': False,
                # Optional (default False)
                'freeze diffusion during rk stages': False
            },
```

1.7.6 DG Nodal locations

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

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

To use the default values

```
'Nodal Locations' : nodal_locations_default['Nodal Locations']
```

Or select from the available types

```
'Nodal Locations' : {
                      # Options line_evenly_spaced, line_gauss_lobatto or line_gauss_
↪legendre_lobatto
                      'Line':  line_gauss_legendre_lobatto,
                      # Options tet_evenly_spaced, tet_shunn_ham
                      'Tetrahedron': tet_evenly_spaced,
                      # Options tri_evenly_spaced, tri_shunn_ham
                      'Tri' : tri_evenly_spaced,
                    },
```

1.7.7 Material Specification

The user can specify any fluid material properties by following the (default) scheme for 'air':

```
'material' : 'air',
```

Options

```
'air' : {
          'gamma' : 1.4,
          'gas constant' : 287.0,
          'Sutherlands const': 110.4,
          'Prandtl No' : 0.72,
          'Turbulent Prandtl No' : 0.9,
        },
```

1.7.8 Initial Conditions

The initial condition properties are defined using consecutively numbered blocks

```
'IC_1' : {....},
'IC_2' : {....},
'IC_3' : {....},
```

Each block can contain the following options

```
# Required: Static temperature in Kelvin
'temperature': 293.0,
# Required: Static pressure in Pascals
'pressure':101325.0,
```

```
# Fluid velocity
'V': {
        # Velocity vector
        'vector' : [1.0,0.0,0.0],
        # Optional: specifies velocity magnitude
        'Mach' : 0.20,
      },
```

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,
```

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,
```

The user can also provide functions to specify a 'wall-function' - or the turbulence viscosity profile near a boundary. For example an atmospheric boundary layer (ABL) could be specified like this:

```
'profile' : {
            'ABL' : {
                        'roughness length' : 0.0003,
                        'friction velocity' : 0.4,
                        'surface layer height' : -1.0,
                        'Monin-Obukhov length' : -1.0,
                        # Non dimensional TKE/friction velocity**2
                        'TKE' : 0.928,
                        # ground level (optional if not set wall distance is used)
```

(continues on next page)

(continued from previous page)

```
                    'z0'  : -0.75,
                   },
           },
```

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

Note: The conditions in the VTK file are specified by node based arrays with names 'Pressure', 'Temperature', 'Velocity', 'TI' and 'EddyViscosity'. Note the field will override the conditions specified previously therefore the user can specify only the conditions that are different from default.

Certain conditions are specified relative to a reference set of conditions

```
'reference' : 'IC_1',
# total pressure/reference static pressure
# This can be a single value or VTK field file name with array name
 ↪'TotalPressureRatio' or function
'total pressure ratio' : 1.0,
# total temperature/reference static temperature
# This can be a single value or VTK field file name with array name
 ↪'TotalTemperatureRatio' or function
'total temperature ratio' : 1.0,
# Mach number
'mach' : 0.5,
# Direction vector or function
'vector' : [1.0,0.0,0.0],
```

```
'reference' : 'IC_1',
# static pressure/reference static pressure
# This can be a single value or VTK field file name with array name
 ↪'StaticPressureRatio' or function
'static pressure ratio' : 1.0,
```

```
'reference' : 'IC_1',
# Mass flow ratio
'mass flow ratio' : 1.0,
# Or mass flow rate
'mass flow rate' : 10.0
```

```
# Example pressure ratio function
def total_pressure_ratio(x, y, z):

    ratio = 1.0
    if x < 10.0:
        ratio = 0.9

    return ratio
```

Note: $W^* = \frac{\dot{m}}{\rho V}$

Driven Initial Conditions

Another way of prescribing a farfield initial condition is by defining a 'driving function', 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. For a driving function `lift_target_ic_func`, the IC should be set up as follows in the parameters dictionary.

```
'IC_2' : lift_target_ic_func
```

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 acheived.

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 propogate 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 timestepping will likely give the fastest propogation of adjusted farfield conditions through the domain.

```
def lift_target_ic_func(**kwargs):
    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,
```

(continues on next page)

(continued from previous page)

```
            '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␣
↪flow in the _report.csv
                   },
    }

    return dict_out
```

Note: Driven initial conditions cannot be used as reference conditions, to initialise the flow or within a restarted simulation.

1.7.9 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,
```

(continues on next page)

(continued from previous page)

```
                        # Roughness field specified as a VTK file
                        'field' : 'roughness.vtp',
                        },
          # Optional: Wall velocity specification either 'linear' or 'rotating'␣
↪dictionary supplied
          'V' : {
                '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',
```

(continues on next page)

(continued from previous page)

```
        # Required: Farfield conditions
        'condition' : 'IC_1',
        # Optional: Atmospheric Boundary Layer
        'profile' : {
                    'ABL' : {
                            # Roughness length
                            'roughness length' : 0.05,
                            # Fiction velocity
                            'friction velocity' : 2.0,
                            # Surface Layer Heigh
                            '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,
         '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,
              },
```

(continues on next page)

(continued from previous page)

```
        '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
        },
'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],
                             },
                },
```

1.7.10 Fluid Structure Interaction

zCFD has the cability 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.

Modal Model

The modal model must be pre calculated and provided at run time. The total displacement of the structure is calculated from the contribution of each mode when excited by the force exerted by the fluid using the method of modal superposition. This displacement is then propagated through the mesh using the RBF or IDW transform capability detailed in *Mesh Transformations* . For both steady and unsteady simulations the structural solution is advanced using a newmark integration scheme. The dictionary keys related to the different mesh deformation algorithms are the same as described in *Mesh Transformations* . zCFD is capable of solving more than one modal model in a simulation. The modal model(s) is(are) setup using the following keys in the Parameters dictionary with one `MM_*` key per modal model:

```
'modal model': {
   'MM_1': {
       # Required: File format of the structural modes. Currently only Nastran .op2␣
↪and .bdf files are supported
       'file format': 'nastran',
       # Required: Name of the Nastran .op2 and .bdf files
       'nastran casename': 'panel_nm_0.04m',
       # Required: The wall zones in the mesh which correspond to the wetted surface␣
↪of the structural model
       'zone': [1, 2, 3],
       # Optional: Zones in the mesh which remain stationary
       'fixed zones': [4, 5, 6],
       # Required: Type of mesh transform to use: 'rbf multiscale', 'rbf' or 'idw'
       'transform type': 'rbf multiscale',
       # Support radius of the RBFs
```

(continues on next page)

(continued from previous page)

```
        'alpha': 0.02,
        # Fraction of surface points in the RBF base set (for multiscale only)
        'base point fraction': 1.0,
        # Optional: Excite the structure with a sinusoidal point force
        'forcing' : { 'frequency' : 1000.0, 'location' : [0.0, 0.0, 0.0], 'force' :␣
↪[0, 0, 10.0] },
        # Optional: Apply a scaling to the structural model
        'scale' : [1.0, 1.0, 1.0],
        # Optional: The maximum search distance when mapping the structural model to␣
↪the CFD mesh
        'mode mapping max distance' : 0.001,
        # Optional: apply a scaling to the force applied at the wetted surface
        'fluid force scaling' : 0.08 / 0.000166667,
        # Optional: Select the modes to include in the analysis
        'mode list' : [0,4,10,13]
        # Optional: Specify the modal damping for each mode
        'modal damping' : [0.0, 0.0, 0.0, 0.0],
        # Optional: Recalculate the RBF coefficients when the maximum deformation␣
↪reaches a fraction of alpha
        'recalculate rbfs alpha fraction': 0.1,
        # Required (IDW only): number of nearest neighbours to
        'n nearest' : 250,
        # Required (IDW only): power of the interpolation weight in the far field, at␣
↪the boundary and the blending function between them
        'power' : [3.0, 5.0, 8.0],
        # Required (IDW only): distance over which to decay the deformation
        'deformation distance': 100.0,
        # Required (IDW only): Stiffness of the blending function used to decay the␣
↪deformation (0.0-1.0)
        'blending stiffness': 0.5
        # Optional: Update the modal model every n cycles
        'update frequency' : 10,
        # Optional: Use only the surface nodes that influence the deformation on each␣
↪partition
        'use local rbf nodes': False
        # Required: alpha value to use for newmark scheme for structural integration␣
↪(default 0.25)
        'newmark alpha': 0.25,
        # Required: delta value to use for newmark scheme for structural integration␣
↪(default 0.5)
        'newmark delta': 0.5,
        # Optional: Relaxation factor to use instead of default newmark scheme when␣
↪performing steady state FSI
        'relaxation factor': 0.1,
    },
}
```

General FSI

The general FSI capabilities of zCFD allow the solver to expose solution variables at run time, in order to allow for a user to couple the solver to any structural model. The behaviour of the coupling scheme can be modified within the open source python layer, by editing `/bin/zcfd/utils/coupling/genericfsi.py`. The default file contains 4 hooks defininining points within the code when FSI actions can be performed- post intialisation, start of the real time cycle, post advance and post solve. In addition the file contains examples of all the functions available to the user for data communication, and how they should be executed in regards the rank of the CPU. In general any user written code for communication should be performed on the rank 0 CPU to avoid issues with parallelisation. The generic FSI scheme can be setup using the `fsi` key in the control dictionary.

```
'fsi': {
    # Required: The fsi driver to launch into, for generic FSI use the key 'generic'
    'type' : 'generic',
    # Required: The wall zones in the mesh which correspond to the wetted surface of␣
↪the structural model
    'zone' : [1, 2, 3],
    # Optional: Zones in the mesh which remain stationary
    'fixed zones' : [4, 5, 6],
    # Required: Type of mesh transform to use: 'rbf multiscale', 'rbf' or 'idw'
    'transform type' : 'rbf multiscale',
    # Required (RBF only): Support radius of the RBFs
    'alpha' : 0.02,
    # Required (RBF only): Basis function for the rbf: 'c0', 'c2', 'c4', 'tps',
↪'gaussian' (default = 'c2')
    'basis' : 'c2',
    # Required (Multiscale RBF only): Fraction of surface points in the RBF base set
    'base point fraction': 0.1,
    # Required (IDW only): number of nearest neighbours to
    'n nearest' : 250,
    # Required (IDW only): power of the interpolation weight in the far field, at the␣
↪boundary and the blending function between them
    'power' : [3.0, 5.0, 8.0],
    # Required (IDW only): distance over which to decay the deformation
    'deformation distance' : 100.0,
    # Required (IDW only): Stiffness of the blending function used to decay the␣
↪deformation (0.0-1.0)
    'blending stiffness' : 0.5
    # Optional: extra user variables to pass to solver through validation script
    'user variables' : {'data path' : '/path/to/data'}
},
```

1.7.11 Fluid Zones

The fluidic zone properties are defined using consecutively numbered blocks like

```
'FZ_1' : {....},
'FZ_2' : {....},
'FZ_3' : {....},
```

For actuator disk zones

```
'FZ_1':{
        'type':'disc',
        'def':'T38-248.75.vtp',
        'thrust coefficient':0.84,
```

(continues on next page)

(continued from previous page)

```
        'tip speed ratio':6.0,
        'centre':[-159.34009325,-2161.73165187,70.0],
        'up':[0.0,0.0,1.0],
        'normal':[-1.0,0.0,0.0],
        'inner radius':2.0,
        'outer radius':40.0,
        # Location of reference conditions used to calculate thrust from C
        'reference point': [1.0,1.0,1.0],
},
```

For tree canopies in terrain wind models

```
'FZ_1':{
        'type':'canopy',
        # Use def when the vtp file specifies a closed volumetric region defining the␣
↪canopy
        'def':'canopy_dacosta.vtp',
        # Use field when the vtp specifies a set of points on the surface (see below␣
↪for details)
        'field':'canopy_field.vtp',
        'func':lad_function,
        'cd':0.25,
        'beta_p':0.17,
        'beta_d':3.37,
        'Ceps_4':0.9,
        'Ceps_5':0.9,
},
```

where the parameters cd, beta_p,beta_d, Ceps_4 and Ceps_5 are defined in the literature (for example Desmond, 2014) and the leaf area density (LAD) is typically specified using a function

```
def lad_function(cell_centre_list):
    lai = 5.0  # for example following Da Costa 2007
    h_can = 20.0
    lad_list = []
    for cell in cell_centre_list:
        wall_distance = cell[3]
        lad = (np.interp(wall_distance, a_z[0] * h_can, a_z[1]) * lai / h_can,)
        lad_list.append(lad)

    return lad_list
```

and the variation in leaf area density is a function of height

```
a_z = (
        np.asarray([0.0,0.43,0.1,0.45,0.2,0.56,0.3,0.74,0.4,1.10,
                    0.5,1.35,0.6,1.48,0.7,1.47,0.8,1.35,0.9,1.01,1.0,0.00]).reshape(11,␣
↪2).T
      )
```

Note: When using the field specification the height of the canopy at each boundary face is set by finding the nearest point to the face centre on the supplied VTK file with the height value looked up in a node based scalar array called 'Height'

For porous media

```
'FZ_1':{
        'type':'porous',
        # Use def when the vtp file specifies a closed volumetric region defining the␣
↪canopy
        'def':'porous_region.vtp',
        # Use zone when the porous region is defined in the mesh as a cell zone
        'zone': [124, 234],
        'alpha': 0.1,
        'c2':0.0,
},
```

1.7.12 Mesh Transformations

Radial Basis Functions (RBFs) or Inverse Distance Weighted (IDW) interpolation can be used to deform the volume mesh according to the motion of some or all of the boundary nodes. zCFD can perform an arbitrary number of transforms which are specified in consecutively numbered blocks.

```
'TR_1' : {...},
'TR_2' : {...},
```

Two kinds of RBF transformation are available: the "Allen et al. greedy" method and the "multiscale" method. For the "greedy" method

```
'TR_1' : {
     # Required: 'rbf multiscale', 'rbf' or 'idw'
     'type' : 'rbf',
     # Required: list of surface zones to be deformed
     'zone' : [4],
     # Required: function describing surface deformation
     'func' : transformFunc,
     # Required: support radius of RBFs
     'alpha' : 200.0,
     # Optional: type of RBF - one of c0, c2, gaussian, tps (default: c2)
     'basis' : 'c2',
     # Required: maximum error tolerance for greedy
     'tol' : 0.005,
     },
```

For the "multiscale" method

```
'TR_1' : {
         # Required: 'rbf multiscale', 'rbf' or 'idw'
         'type' : 'rbf multiscale',
         # Required: faction of surface points to be solved in the base set
         'base point fraction' : 0.1,
         # Required: list of surface zones to be deformed
         'zone' : [4],
         # Required: function describing surface deformation
         'func' : transformFunc,
         # Required: support radius of RBFs in the base set
         'alpha' : 200.0,
         # Optional: type of RBF - one of c0, c2, gaussian, tps (default: c2)
         'basis' : 'c2', # type of RBF - one of c0, c2, gaussian, tps
         },
```

For the IDW transform

```
'TR_1' : {
         # Required: 'rbf multiscale', 'rbf' or 'idw'
         'type' : 'idw',
         # Required: number of nearest neighbours to
         'n nearest' : 250,
         # Required: list of surface zones to be deformed
         'zone' : [4],
         # Required: function describing surface deformation
         'func' : transformFunc,
         # Required: power of the interpolation weight in the far field, at the␣
↪boundary and the blending function between them
         'power' : [3.0, 5.0, 8.0],
         # Required: distance over which to decay the deformation
         'deformation distance': 100.0,
         # Required: Stiffness of the blending function used to decay the deformation␣
↪(0.0-1.0)
         'blending stiffness': 0.5
         },
```

An example of a function to deform a surface is given below

```
def transWing(x,y,z):
 ymax = 1.0
 ymin = 0.1
 yNorm =(y-ymin)/(ymax-ymin)
 x2 = x
 y2 = y
 z2 = z
 if y > 0.1 and y < 1.1 and x < 1.1 and x > -0.1 and z > -0.1 and z < 0.1:
     z2 = z - 0.5*(20.0*yNorm**3 + yNorm**2 + 10.0*yNorm)/31.0

 return {'v1' : x2, 'v2' : y2, 'v3' : z2}
```

1.7.13 Reporting

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 spcified.

```
'report' : {
            # Report frequency
            'frequency' : 1,
            # Extract specified variable at fixed locations
            'monitor' : {
                        # Consecutively numbered blocks
                        'MR_1' : {
                                 # Required: Output name
                                 'name' : 'monitor_1',
                                 # Required: Location
                                 'point' : [1.0, 1.0, 1.0],
                                 # Required: Variables to be extracted
                                 'variables' : ['V','ti'],
                                 },
                        },

            # Report force coefficient in grid axis as well as using user defined␣
```

(continues on next page)

(continued from previous page)

```
↪transform
            'forces' : {
                        # Consecutively numbered blocks
                        'FR_1' : {
                                # Required: Output name
                                'name' : 'force_1',
                                # Required: Zones to be included
                                'zone' : [1, 2, 3],
                                # Transformation function
                                'transform' : my_transform,
                                # Optional (default 1.0):
                                'reference area' : 0.112032,
                                # Optional (default 1.0):
                                'reference length' : 1.0,
                                # Optional (default [0.0, 0.0, 0.0]):
                                'reference point' : [0.0, 0.0, 0.0],
                                # Optional (default 0.0): Calculate forces using a␣
↪specified reference pressure rather than absolute
                                'reference pressure': 0.0
                                },
                        },

            # Report massflow ratio
            'massflow' : {
                         # Consecutively numbered blocks
                         'MF_1' : {
                                  # Required: Output name
                                  'name' : 'massflow_1',
                                  # Required: Zones to be included
                                  'zone' : [1, 2 ,3],
                                  },
                         },
        },
```

Example Transformation Function

A transformation function my be supplied to the force report block to transform the force into a different coordinate system, this a particularly useful for reporting lift and drag which are often rotated from the solver coordinate system.

```
# 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 {'v1' : v[0], 'v2' : v[1], 'v3' : v[2]}
```

1.7.14 Overset meshes

Within zCFD in finite volume mode it is possible to run with an arbitrary number of meshes overlapping one another. It is also possible to run a single high order DG mesh (simulation) overlapping a single finite volume background mesh. To activate the overset mesh capability, the user simply strings mesh and case names after the -p and -c options respectively. There should be a python dictionary for each mesh with the appropriate boundary conditions for that mesh defined in it. It should be noted that the ordering of the mesh and case names indicates superiority where the second mesh overlaps the first, and the third overlaps the second and first, and so on. Any mesh which overlaps another must have the boundaries defined where solution information is to be from a lower mesh. This is specified as follows.

```
'BC_1': {
          'zone': [5],
          'type': 'overset'}
        }
```

With finite volume mode, the overset solver can also perform simulations with rotating domains overlapping fixed background domains. It is assumed the geometry of the rotating domains is such that there is no new covering or uncovering of cells in the background mesh. Multiple rotating domains can also be defined. If an overset mesh is to rotate, the entire mesh must rotate with no fixed volumes and dual time-stepping time marching must be used. The rotating mesh / domain must be specified in fluid zone 1 as follows:

```
'FZ_1': {
    "type": "rotating",
    "zone": [0],
    "axis": [0, 0, -1],
    "origin": [0, 0, 0],
    "omega": 100,
},
```

To control the order of accuracy when mapping between overset meshes, the user can specify the order of accuracy of the interpolation. First (0) or second (1) order mapping can be chosen. First order mapping will use the solution in the cell containing the location requiring data, whereas second order mapping will use an inverse distance weighted interpolation using neighbouring information as well.

```
'mapping' : {
                # Order of accuracy of mapping.  Defaults to 0.
                'idw from cells order' : 0,
                },
```

1.7.15 Sliding meshes

Sliding meshes (where interfaces on each mesh slide past one another, for a moving rotor simulation for example) can be simulated in a similar fashion to overset meshes. In fact, sliding meshes are a simple extension of the overset mesh philosophy however no mesh is defined as a background mesh (with cells blanked off). For two meshes sliding across one another, we simply set the sliding boundary as `'overset'` in both python dictionaries as described above for BC_1 in the Overset meshes section. The only additional requirement is that to indicate to the solver that the first mesh and solver dictionary combination are not background overset, we must add the following key value pair to the first solver dictionary.

```
'create overset halos' : True,
```

The `'create overset halos'` keyword instructs the solver to generate an overset halo cell centre by projecting along the face normal a distance equal to the distance from the overset boundary face to the adjacent cell centre. This fictitious cell centre is used for mapping data.

1.7.16 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. 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'`.

```
'DGCAA' : {
            # Order of polynomial used to spatially represent
            # the solution within an element
            'order' : 2,

            # If refraction effects are of interest, these can be included
            # by mapping a CFD solution on to the acoustic domain.  Defaults to True.
            # (Optional)
            'Map CFD to CAA mesh' : True,

            # Option to run the FRPM solver only, without running the acoustic␣
↪propagation
            # solver.  Defaults to False. (Optional)
            'FRPM solver only' : False,

            # Specify mesh file to be used when mapping RANS simulation to CAA mesh
            # (and / or FRPM mesh)
            'RANS mesh name' : "rans_mesh.h5",

            # Specify solution file to be used when mapping RANS simulation to CAA␣
↪mesh
            # (and / or FRPM mesh)
            'RANS case name' : "rans_case.py",

            # Specify if FRPM sources are to be used in the acoustic simulation.
            # Defaults to True.  (Optional)
            'frpm sources' : True,

            # Specify after how many cycles microphone data should be extracted. ␣
↪Note this
            # is cycles rather then Hz.
            'microphone output frequency' : 10,

            # FRPM dictionary.  Specifies FRPM parameters.  Full details below.
            'FRPM' : {
            # See dictionary contents below
            }
}
```

```
'FRPM_1' : {
            # The FRPM solver can run on a subset of MPI processes
            # Choose number of MPI processes to run FRPM (Optional)
            'number of frpm mpi processes' : 1,

            # The FRPM can use freestream meanflow or flow from a RANS solution
            # Set to True (default) if RANS solution to be used to drive FRPM␣
↪(Optional)
            'Map CFD to FRPM mesh' : True,
```

(continues on next page)

(continued from previous page)

```
            # Mesh spacing to be used for implicitly defined FRPM domain
            # Ideally should be less than 0.5 the minimum turbulence integral
            # length scale of interest
            'FRPM Spacing' : 1.0,

            # Constant of proportionality for turbulence integral length scale
            # computation from RANS.  Defaults to 1.0. (Optional)
            'FRPM turbulence integral length scale parameter': 1.0,

            # The user can inspect the RANS solution and choose a turbulence␣
↪integral
            # length scale best suited to match the RANS simulation.  A uniform
            # integral length scale allows much faster source computations. ␣
↪Defaults
            # to True (recommended).  Otherwise the turbulence integral length scale
            # will be taken from the mapped RANS solution. (Optional)
            'use constant integral length scale': True,

            # Used in conjunction with 'use constant integral length scale' set to␣
↪True
            # User specified turbulence integral length scale.
            'FRPM Turbulence Integral Length Scale' : 0.5,

            # Number of cells in each direction for implicity defined FRPM mesh
            # If using parallel FRPM, x-direction should be the largest value for
            # optimum parallel efficiency
            'FRPM Cart Num Mesh Cells' : [10, 10, 10],

            # Frequency with which the FRPM sources should be updated.  Acoustic␣
↪sources
            # at time levels between updates will be linearly interpolated.  Set to -
↪1 if
            # solver should determine (recommended).  Set to 1 if FRPM should be␣
↪updated
            # each acoustic solver time step
            'FRPM March Frequency' : 1,

            # Vector the implicitly defined FRPM mesh should be translated to␣
↪correctly
            # position sources in acoustic domain
            'FRPM Domain Translate' : [0.0, 0.0, 0.0],

            # Vector the implicitly defined FRPM mesh should be rotated to correctly
            # position sources in acoustic domain
            'FRPM Domain Rotate (Deg)' : [0.0, 0.0, 0.0],

            # Physical distance sources should be blended over from the
            # [min-X, max-X, min-Y, max-Y, min-Z, max-Z] cartesian boundaries
            # Values of 0.0 indicate no blending of sources
            'FRPM Blend Sources From Side' : [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],

            # Physical distance from walls acoustic sources should be blended over␣
↪to zero.
            # Values of 0.0 or less indicates no blending using wall distance
            # (default = -1.0).  The first half of the blending distance has no␣
```

(continues on next page)

(continued from previous page)

```
↪decay, the
            # second half blends sources linearly to zero.
            'wall distance blend over': -1.0,

            # The FRPM method can reconstruct turbulence fields from many turbulence
            # spectrum models.  Typically Gauss (default) is used. It is also␣
↪possible to
            # specify Liepmann.  Liepmann is associated with greater computational␣
↪cost.
            # (Optional)
            'FRPM turbulence spectrum': "gauss",

            # Used in conjunction with the Liepmann spectrum.  Defaults to 1 (Gauss
            # spectrum).  The Liepmann spectrum is obtained by a series of Gauss␣
↪spectra,
            # the number to be used is specified here (1 to 10). (Optional)
            'FRPM number of discrete filters': 1,


            # Used in conjunction with the Liepmann spectrum.  Defines FRPM mesh␣
↪refinement
            # as a function of turbulence integral length scale from smaller scales.␣
↪ 0.5 is
            # recommended if using Liepmann. (Optional)
            'FRPM length scale resolution factor': 0.5,

            # Used in conjunction with the Liepmann spectrum.  Maximum turbulence␣
↪integral
            # length scale to be modelled (multiple of length scale from RANS).
            # Defaults to 5.0. (Optional)
            'FRPM maximum length scale factor': 1.0,

            # Used in conjunction with the Liepmann spectrum.  Minimum turbulence␣
↪integral
            # length scale to be modelled (multiple of length scale from RANS).
            # Defaults to 0.1. (Optional)
            'FRPM minimum length scale factor': 0.2,

            # Run FRPM solver a number of cycles before starting the acoustic solver.
↪
            # Defaults to 0.
            'FRPM Flush Particle Cycles' : 0,

            # Smooth the turbulence kinetic energy field.  If the RANS solution␣
↪mapped
            # on to the FRPM domain is discontinuous, that can cause problems with␣
↪the
            # computation of the acoustic sources.  Ideally a finer RANS mesh should␣
↪be used,
            # however if this is not possible, Laplace smoothing can be used.  There␣
↪is
            # potential for the accuracy of the results to deteriorate when using␣
↪this.
            # Defaults to 0.  (Optional)
                'FRPM Mapped TKE Smoothing iterations' : 5,
```

(continues on next page)

(continued from previous page)

```
            # Relaxation for turbulence kinetic energy smoothing.  Defaults to 0.2.
            # (Optional)
            'FRPM Mapped TKE Smoothing relaxation' : 0.2,

            # There are two options to map acoustic sources from the FRPM mesh to␣
↪the
            # acoustic mesh.  Firstly linearly interpolate based on the location of
            # the acoustic mesh solution points within the FRPM domain.  Secondly␣
↪use
            # in inverse distance weighted mapping based on sources in the␣
↪neighbourhood
            # of the acoustic solver solution points.  Defaults to True␣
↪(Recommended).
            # (Optional)
            'FRPM inverse distance source mapping' : True,

            # Used in conjunction with inverse distance weighted source mapping.
            # Distance to specify neighbourhood to take sources from. Recommend
            # at least twice the turbulence integral length scale.  Note, this␣
↪increases
            # memory usage.
            'frpm inverse distance source mapping distance' : 6.0,

            # List used to specify which boundaries should be used for reseeding␣
↪particled
            # once they have left the domain [XMIN, XMAX, YMIN, YMAX, ZMIN, ZMAX]
            # Defaults to all True.
            'FRPM inflow boundaries': [True, True, True, True, True, True]
}
```

1.7.17 Output

For solver efficiency, zCFD outputs the raw flow field data (plus any user-defined variables) to each parallel partition without re-combining the output files. The `'write output'` dictionary controls solver output.

```
'write output' : {
                # Required: Output format: 'vtk', 'ensight', 'native'
                'format' : 'vtk',
                # Required: Variables to output on each boundary type
                'surface variables': ['V','p'],
                # Required: Field variables to be output
                'volume variables' : ['V','p'],
                # Optional: Volume interpolation - interpolates cell values to␣
↪points in supplied meshes
                #           File types can be vtp, vtu or binary stl
                'volume interpolate' : ['mesh.vtp', 'mesh.vtu', 'mesh.stl'],
                # Optional: Paraview catalyst scripts
                'scripts' : ['paraview_catalyst1.py','paraview_catalyst2.py'],
                # Optional: If downstream processes need variables named using a␣
↪specific convention a naming alias dictionary can be supplied
                'variable_name_alias' : { "V" : "VELOCITY", },
                # Required: Output frequency
                "frequency": {
                    # Output visualisation files every 3 cycles (default = 1000000)
                    "volume data": 3,
```

(continues on next page)

(continued from previous page)

```
                    "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,
                },
                # Optional for DG: Output high order surface data on zones
                'high order surface list' : [1],
                # Optional (default False):
                # Don't output vtk volume data
                'no volume vtk' : False
                # Optional
                # Calculate mean and RMS data when running unsteady
                # Adds 'V_avg", "V_rms", "T_avg", "T_rms", "p_avg" and "p_rms" to␣
↪the available output variables.
                'calculate averge and rms': True,
                # Start time averaging at a specific time step
                'average start time cycle': 1000
            },
```

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 presure 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		

continues on next page

Table 1 – continued from previous page

Variable Name	Alias	Definition
centre		Cell centre location
viscouslengthscale		Minimum distance between cell and its neighbours
walldistance		Wall distance (m)
sponge		Sponge layer damping coefficient
menterf1		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-dinemsional wall distance
zone		Boundary zone number
cf		Skin friction coefficient
facecentre		Face centre location
frictionvelocity	ut	Friction velocity at the wall

1.8 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:

1.8.1 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
```

1.8.2 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.

1.8.3 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.

1.8.4 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 propietary 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.

1.8.5 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:zcfd@zenotech.com.

1.8.6 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.

1.8.7 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.

1.8.8 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.

1.8.9 CBA Mesh Converter

CBA is a strucutred 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.

1.9 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:

1.9.1 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',
```

1.9.2 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',
```

1.9.3 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',
```

1.9.4 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']
```

1.9.5 ParaView Client

To download the free ParaView software for visualising VTK format data on the local device, just visit the Paraview website.

1.9.6 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 remove 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.

1.9.7 Jupyter Notebooks

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. 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_notebook
```

1.10 Utilities

zCFD is bundled with a number of utility functions to streamline specific workflows:

1.10.1 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_zcfd_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_zcfd_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_zcfd_input(case_name='windfarm_name',
                            wind_direction=267.0,
                            reference_wind_speed=10.0,
                            terrain_file=None,
                            report_frequency=100,
                            update_frequency=1,
```

(continues on next page)

(continued from previous page)

```
                                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_zcfd_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://airfoiltools.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],
```

(continues on next page)

(continued from previous page)

```
    '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 or 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.

1.10.2 Propellor Modelling

Overview

Propellors 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 propellors:

```
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]],
```

(continues on next page)

(continued from previous page)

```
        '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 propellors.

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]
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.
```

(continues on next page)

(continued from previous page)

```
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 propellors 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.