
# zCFDguide Documentation

*Release v2019.10.1724-develop*

Zenotech Ltd

Oct 01, 2019

# CONTENTS

CHAPTER

# ONE

# VERSION: V2019.10.1724-DEVELOP

## 1.1 Overview

### 1.1.1 What is zCFD

zCFD is a computational fluid dynamics (CFD) solver designed to make highly effective use of modern computer architectures. The code will run on unstructured meshes from a variety of sources, and will export data in many common formats.

zCFD is a density-based finite volume solver, allowing steady-state or time-dependent, turbulent flow simulation. The numerical scheme has been selected for accuracy and robustness. The code provides automatic scaling of turbulent wall functions; automatic determination of inflow / outflow conditions and automatic 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 (IO) 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.

zCFD is a hybrid of an open-source (freely available without restriction) front-end ("ZCFD Driver") and proprietary (closed-source) back-end ("ZCFD Core"). Anyone can develop "ZCFD Driver" for their own purposes including commercial exploitation.

## 1.2 Formulation

### 1.2.1 Numerical Scheme

zCFD is a many-core accelerated Computational Fluid Dynamics solver that solves the Steady or Unsteady Reynolds Averaged Navier Stokes (RANS) equations using the finite volume technique. This is based on a cell centred approach on arbitrary polyhedral meshes with the boundary conditions applied in a strong formulation. The scheme implemented is formally second order accurate in time and third order accurate in space.

The spatial accuracy is achieved by using a slope limited MUSCL reconstruction at cell interfaces. The HLLC scheme is then used with the reconstructed values to calculate the interface flux.

The primary solver is based on a multi-stage explicit solver using local time stepping and geometric multigrid to accelerate convergence.

Turbulence modelling is provided by an implementation of Menter SST two equation model which is can be used in low Reynolds number mode (i.e. integration to the wall) or high Reynolds number mode (i.e. wall functions).

## 1.3 License

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.

## 1.4 Execution

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.

### 1.4.1 (zCFD) Command Line 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.

### 1.4.2 Smartlaunch

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
```

### 1.4.3 Input Validation

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

```
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.

## 1.5 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 \
```

(continues on next page)

(continued from previous page)

```
                --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
```

## 1.6 Control Dictionary

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

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

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

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

### 1.6.1 Reference Settings

#### Define units

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

#### Scale mesh

```
# Optional (default [1.0,1.0,1.0]):
# Scale vector
'scale' : [1.0,1.0,1.0],
```

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

Example scale function

```
# Scale function
def scale_func(point):
    # Scale y 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,
```

### 1.6.2 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']

    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

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

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,
```

### 1.6.3 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: Preconditoner factor
'preconditioner' : {
                    'minimum mach number' : 0.5,
                  },
```

### 1.6.4 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: Euler, various orders of Runge Kutta (like Euler…) and the implicit LU-SGS scheme. These can be run with dual time-stepping for both steady and unsteady simulations or with global time-stepping for unsteady simulations. The choice of scheme and time stepping algorithm affects with dictionary keys are valid.

### Runge Kutta

```
'time marching' : {
                  'scheme' : {
                              # Either 'euler' or 'runge kutta' or 'lu-sgs'
                              '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'
                              },
                   },
```

### LU-SGS

```
'time marching' : {
                   'scheme' : {
                              'name' : 'lu-sgs',
                              },
                   'lu-sgs' : {
                              # Optional (default 8)
                              'Number Of SGS Cycles' : 8,
                              # Optional (default 1)
                              'Jacobian Update Frequency' : 1,
                              # Optional (default True)
                              'Include Backward Sweep' : True,
                              # Optional (default True)
                              'Include Relaxation' : True,
                              # Optional (default 1.0e-8)
                              'Jacobian Epsilon' : 1.0e-08,
                              # Optional (default True)
                              'Use Rusanov Flux For Jacobian' : True,
                              # Optional (default False)
                              'Finite Difference Jacobian' : 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 LU-SGS is selected then the `'lu-sgs'` sub-dictionary must be provided.

### 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
```

(continues on next page)

(continued from previous page)

```
                    # 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 (from highest to lowest order)
                    'multipolycfl' : [2.0,2.0,2.0],
                    # 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 daul time-stepping:
                    # Number of pseudo time cyles
                    '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:
                   # Prologation 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,
                    },
```

### 1.6.5 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, DGeuler, DGviscous, DGLES
'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 '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 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 '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)
            '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 'HLLC'):
            # Scheme for inviscid flux: HLLC or Rusanov
            'Inviscid Flux Scheme': 'HLLC',
            # Turbulence
            'turbulence' : {
                            # turbulence model (options: 'sst', 'sa-neg')
                            '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 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,
                        },
        },
```

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,
              'turbulence' : {
                            # LES model (options 'none', 'WALE')
                            'les' : 'none',
                        },
```

(continues on next page)

(continued from previous page)

```
            },
```

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 False)
                'Approximate curved boundaries': True,
                # Optional (default 0.0)
                'c11 stability parameter': 0.0,
                # Optional (default 0.0)
                # c11 stability parameter for transported variables
                'c11 stability parameter transport': 0.0,
                # Optional (default 0.5)
                'LDG upwind parameter': 0.5,
                'LDG upwind parameter aux': 0.5,
                # Optional (default False):
                # Use MUSCL reconstruction at P=0
                'Use MUSCL Reconstruction': False,
                # Optional (default False)
                'BR2 Diffusive Flux Scheme' : False,
                # Optional (default False)
                'Use Rusanov for turbulence equations' : False,
                # 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 `'DGeuler'` are a combination of the `'euler'` keys and the `'DG...'` keys. The same logic applies to the `'DGRANS'`, `'DGviscous'` and `'DGLES'` equation sets.

```
'DGeuler' : {
                # 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 False)
                'Approximate curved boundaries': False,
                # Optional (default 0.0)
                'c11 stability parameter': 0.0,
                # Optional (default 0.0):
                # c11 stability parameter for transported variables - default 0.0
                'c11 stability parameter transport': 0.0,
                # Optional (default 0.5)
                'LDG upwind parameter': 0.5,
                'LDG upwind parameter aux': 0.5,
                # Optional (default False):
                # Use MUSCL reconstruction at P=0
                'Use MUSCL Reconstruction': False,
```

(continues on next page)

(continued from previous page)

```
                # Optional (default False)
                'BR2 Diffusive Flux Scheme' : False,
                # Optional (default False)
                'Use Rusanov for turbulence equations' : False,
                # 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',
            },
```

### 1.6.6 DG Order Specification

Regions of the mesh may be specified be be a certain order in the DG solver. This can be done by supplying a list of dictionaries containing any number of entries of the following types

```
'cell order' : [{'type' : 'wall distance',
                 'order' : 0,
                 'distance' : 1000.0},
                {'type' : 'wall distance',
                 'order' : 1,
                 'distance' : 2.0},
                 {'type' : 'sphere',
                  'order' : 0,
                  'radius' : 1.0,
                  'centre' : [0.0, 0.0, 0.0]},
                 {'type' : 'cartesian',
                  'order' : 0,
                  # xmin, xmax, ymin, ymax, zmin, zmax
                  'box' : [0.0, 1.0, 0.0, 1.0, 0.0, 1.0]}
                 ],
```

### 1.6.7 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,
                    },
```

### 1.6.8 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,
        },
```

### 1.6.9 Initial Conditions

The intial 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 ($\mu_t/\mu$) 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
'total pressure ratio' : 1.0,
# total temperature/reference static temperature
'total temperature ratio' : 1.0,
# Mach number
'mach' : 0.5,
# Direction vector
'vector' : [1.0,0.0,0.0],
```

```
'reference' : 'IC_1',
# static pressure/reference static pressure
'static pressure ratio' : 1.0,
```

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

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

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

(continues on next page)

(continued from previous page)

```
                            'axis' : [1.0,0.0,0.0],
                            # Rotation origin
                            'origin' : [0.0,0.0,0.0],
                            },
                },
          # Optional: Wall temperature specification either a constant 'saclar' 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 refering 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 overide
          'zone' : [0,1,2,3],
          # Required: Boundary condition type
          'type' : 'farfield',
          # Required: Kind of farfield options (riemann, pressure or supersonic)
          'kind' : 'riemann',
          # Required: Farfield conditions
          'condition' : 'IC_1',
          # Optional: Atmospheric Boundary Layer
          'profile' : {
                      'ABL' : {
                              # Roughness length
                              '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' : {
```

(continues on next page)

(continued from previous page)

```
                        '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 massflow ratio is required.

```
'BC_3' : {
         # Zone type tag
         'ref' : 4,
         # Optional: Specific zone boundary condition overide
         'zone' : [0,1,2,3],
         # Required: Boundary condition type
         'type' : 'inflow',
         # Required: Kind of inflow, either 'default' for presure 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 instensity' : 0.1,
         'eddy viscosity ratio' : 1.0,
         },
'IC_2' : {
         'reference' : 'IC_1',
         'total temperature ratio' : 1.0,
         'total pressure ratio; : 1.0,
         },
'BC_3' : {
         'ref' : 4,
         'zone' : [0,1,2,3],
         'type' : 'inflow',
         'kind' : 'default',
         '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 temperaure ratio, when `'massflow'` is specified a massflow ratio must be set.

```
'BC_4' : {
         # Zone type tag
         'ref' : 5,
         # Optional: Specific zone boundary condition overide
         '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 overide
         '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],
                             },
                },
```

### 1.6.11 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,
        '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'

### 1.6.12 Mesh Transformations

Radial Basis Functions (RBFs) 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' : {...},
```

```
'TR_1' : {
         # Required: list of surface zones to deform
         'zone' : [4],
         # Required: function describing surface deformation
         'func' : transWing,
         # Required: support radius of RBFs
         'alpha' : 200.0,
         # Required: type of RBF, one of c0, c2, gaussian, tps
         'basis' : 'c2',
         # Required: maximum error tolerance for greedy algorithm
         'tol' : 0.005,
         # Optional: Surface zones that remain fixed
         'fixed' : {
                   'zone' : [1],
                   },
         },
```

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}
```

### 1.6.13 Mesh Transformations

Radial Basis Functions (RBFs) 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 "greedy" method and the "multiscale" method. For the "greedy" method

```
'TR_1' : {
             'multiscale' : False,
```

(continues on next page)

(continued from previous page)

```
            'zone' : [4], # list of surface zones to be deformed
            'func' : transformFunc, # function describing surface deformation
            'alpha' : 200.0, # support radius of RBFs
            'basis' : 'c2', # type of RBF - one of c0, c2, gaussian, tps
            'tol' : 0.005, # maximum error tolerance for greedy
        },
```

For the "multiscale" method

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

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}
```

### 1.6.14 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 transform
            'forces' : {
                        # Consecutively numbered blocks
                        'FR_1' : {
```

(continues on next page)

(continued from previous page)

```
                                # 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],
                                },
                        },

            # 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]}
```

### 1.6.15 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
                    'volume interpolate' : ['mesh.vtp', 'mesh.vtu'],
                    # Optional: Paraview catalyst scripts
                    'scripts' : ['paraview_catalyst1.py','paraview_catalyst2.py'],
```

(continues on next page)

(continued from previous page)

```
                # 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' : 100,
                # Optional for DG: Output high order surface data on zones
                'high order surface list' : [1],
                # Only for time accirate simulations:
                # Start output after real time cycle
                'start output real time cycle' : 100,
                # Unsteady restart file output frequency
                'unsteady restart file output frequency' : 1000,
                # Output real time cycle frequency
                'output real time cycle frequency' : 10,
                # Optional (default False):
                # Don't output vtk volume data
                'no volume vtk' : False
            },
```

### 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 | | Different depending on turbulence model?? |
| turbulenceintensity | ti | Turbulence intensity |
| turbulencekineticenergy | | Turbulence kinetic energy ($J/kg$) |
| turbulenceeddyfrequency | | Turbulence eddy frequency ($1/s$) |
| eddy | | Eddy viscosity ($Pa/s$) |
| cell_velocity | | |
| cellzone | | |
| cellorder | | |
| centre | | Cell centre location |
| viscouslengthscale | | Defined?? Minium distance between cell and its neighbours |
| walldistance | | Wall distance ($m$) |
| sponge | | Sponge layer damping coefficient |

#### 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 |

## 1.7 Mesh Conversion

zCFD uses unstructured meshes written in an HDF5 file format. HDF5 provides a flexible, self describing specification supporting parallel I/O that can be tuned per filesystem type (e.g NFS, GPFS, Lustre etc).

Meshes from a wide variety of sources can be easily converted into this format as follows:

### 1.7.1 OpenFOAM-x.x.x

We provide a range of converters for popular mesh types via a special version of the open-source CFD code OpenFOAM, available freely for download here

Once downloaded and installed, the version of OpenFOAM will convert a range of file formats to the zCFD format. Prior to running any of these, you will need to activate your OpenFOAM environment using

```
source $FOAM_INST_DIR/OpenFOAM-x.x.x/etc/bashrc
```

### 1.7.2 OpenFOAM Mesh Converter

To convert from an OpenFOAM format mesh to zCFD, run the following command in the MESH directory.

```
foamTozCFD
```

This will create a new folder called *zInterface/* containing the HDF5 file.

### 1.7.3 Fluent Mesh Converter

The converter for ANSYS Fluent files is a utility called *fluent_convert*. This is included in the path set by activating the zCFD command line environment.

To run the utility, use

```
(zCFD): fluent_convert <filename>.msh <new_filename>.h5
```

or

```
(zCFD): fluent_convert <filename>.cas <new_filename>.h5
```

The will produce 2 files: the zCFD HDF5 file and the *name_zone.py* python file which holds the fluid and boundary zone names.

### 1.7.4 Star CCM+ Mesh Converter

There are a number of different file formats for CD-Adapco Star CCM+ meshes. The *<filename>.sim* file is in a 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.

### 1.7.5 CGNS Mesh Converter

This utility will convert an unstructured, non-chimera, hexahedral mesh in CGNS (ref) format. The converter will handle meshes with element types (HEXA_8):

```
(zCFD): cgnsconvert <filename>.cgns <new_filenam>.h5
```

This produces the zCFD HDF5 file. If you require support for additional element types please contact us: mailto:zcfd@zenotech.com.

### 1.7.6 UGRID Mesh Converter

UGRID is a NASA format for meshes (ref). You must follow the naming convention (ref) for UGRID files, which describes the format and *endian-ness* of the file data ("ascii8", "b8", "lb8", or "r8") depending upon which language (C or FORTRAN) was used to create the file. The mapbc file is optional and if not provided the boundary conditions of the zones default to wall.

```
(zCFD): ugridconvert <filename>.[ascii8, b8, lb8, r8].ugrid <new_filename>.h5  <filename>.mapbc
```

This produces the zCFD HDF5 file.

### 1.7.7 DLR Tau Mesh Converter

Tau is a CFD solver produced by the DLR (http://tau.dlr.de/startseite/). Note that Tau is a node-based solver so flow field variables are calculated for and stored at the mesh nodes (or vertices) rather than at cell centres (the approach used by Star CCM+, OpenFOAM, CFX and zCFD). Thus simply converting the file format will not guarantee a good result. In many cases, the conversion will not result in a valid cell-centred zCFD mesh.

```
(zCFD): tauconvert <filename>.XXX <new_filename>.h5
```

This produces the zCFD HDF5 file.

## 1.8 Visualisation

zCFD by default will output data in VTK (http://www.vtk.org/) format. This format can be read directly by many post-processing tools including ParaView (http://www.paraview.org/) by Kitware.

Output in other formats is also possible, including TECPLOT (http://www.tecplot.com/) and EnSight (https://www.ceisoftware.com/) as follows:

### 1.8.1 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',
```

### 1.8.2 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',
```

### 1.8.3 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',
```

### 1.8.4 ParaView Catalyst Scripts

ParaView provides a powerful yet lightweight interface that allows parallel processing of simulation data during programme execution called ParaView Catalyst (http://www.paraview.org/in-situ/). 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']
```

### 1.8.5 ParaView Client

To download the free ParaView software for visualising VTK format data on the local device, just visit the ParaView website (http://paraview.org).

### 1.8.6 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 Source

To download and install ParaView Server, you can either use the version on the main ParaView website and follow the instructions here (http://www.paraview.org/Wiki/Setting_up_a_ParaView_Server), or else use the Zenotech version (https://github.com/zenotech/ParaView.git). We currently recommend the Zenotech version as it contains a number of fixes for parallel post-processing that have not yet been added to the main release.

### 1.8.7 zPost: Post-Processing Suite

Zenotech has created an open-source post-processing code suite called *zPost*, which is available for free download (https://github.com/zenotech/zPost.git). *zPost* makes use of ParaView and iPython (http://ipython.org/). *zPost* is ideally suited to post-processing data generated by zCFD, and can exploit the full power of parallel, remote processing. The repository for *zPost* contains numerous examples to help with the generation of standard CFD outputs.

#### Installing zPost

You must have Python 2.7 (https://www.python.org/download/releases/2.7/) installed, including the *virtualenv* package. ParaView must also be installed. Once you have downloaded *zPost*, change to the *zPost/scripts* directory and run:

```
> ./create_virtualenv.bsh
```

This will install all the required python packages in *zPost/zpost-py27*. If there are aditional package that you want to include in the Python environment, just add them to the *zPost/Requirements.txt* file.

#### Running the iPython Notebook

One of the easiest ways to use Python is via an interactive notebook on your local computer. The notebook is just a locally hosted server that you can access via a web-browser. This makes saving, re-running and viewing the *zPost* output straightforward.

To automatically start up notebook server on your computer within the zPost environment, change to the *scripts* directory and run:

```
> ./start_notebook
```

This should launch a web-browser automatically.

**Note**

This will use the default version of ParaView. If you want to run a custom version of ParaView please set the PARAVIEW_HOME variable in your shell before starting the notebook server

#### Examples

http://nbviewer.ipython.org/github/zenotech/zPost/tree/master/ipynb/

## 1.9 Utilities

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

### 1.9.1 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:

```
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,
                            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 a 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,
```

(continues on next page)

(continued from previous page)

```
    '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 *turbine_vtk/<turbine>.vtp* file defining the fluid zone for each turbine is also automatically created. The *'thrust coefficient'* is 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 *'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 the VTK output files 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.

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 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():
```

(continues on next page)

(continued from previous page)

```
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 make 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.