CFD

zCFD documentation

VERSION

v2026.11.16174

BUILT

4 November 2026

Contents

Welcome to the zCFD documentation! 6

REFERENCE 7

Input deck 8
- solver 10
  - solver settings 11
    - linear solver options 13
    - cell reordering 16
  - equations 18
    - turbulence 22
    - base state 31
    - theta perturbation 33
    - velocity perturbation 35
    - sponge layer 36
  - fluid properties 37
  - reference conditions 39
    - v 42
    - profile 43
      - abl 44
  - initialisation 47
  - time settings 51
    - scheme 54
  - convergence control 56
    - inner convergence 58
    - scheme 61
    - cfl 67
      - cfl ramp 69
    - viscosity ramp 72
    - relaxation ramp 73
  - numerical scheme 74
    - multigrid 81
    - preconditioner 83
  - output settings 84
    - solution 85
      - frequency 88
    - report 90
      - monitor 92
      - forces 93
      - mass flow 95
- model 96
  - boundary conditions 98

- wall 99
  - most 102
  - temperature 104
  - heat flux 105
  - velocity 106
  - roughness 108
  - fsi 110
    - harmonic initial state 116
- farfield 117
  - sponge layer damping 119
  - target 120
    - alpha 121
      - advanced 123
    - axes 125
- symmetry 127
- inflow 128
- outflow 131
- interface 134
  - connection 139
  - transform 141
- fluid zones 143
  - momentum 144
  - rotating 147
  - disc 149
    - controller 151
    - geometry 153
    - discretisation 154
    - model 155
  - porous 159
  - canopy 161
  - abl 163
  - idw transform 164
  - rbf transform 166
- transforms 169
  - motion 171

Output variables 173

COMMAND LINE

Command line invocation and flags 179
Environment variables 181
Parallel invocation 182

FILE FORMATS

Mesh file 185

GUIDE 187

CONCEPTS

What is zCFD? 188
How the deck works 189

Converting an older control file 192

SETTING UP…

Installation and licensing 197
Preparing a mesh 201
Running zCFD 212
Python functions in the deck 218

CHOOSING…

Choosing an equation set 222
Choosing a turbulence model 226
Choosing a wall model 231
Choosing a time-marching scheme 237
Choosing a spatial scheme 254
Choosing a limiter 258
Choosing a linear solver 262
Choosing a fluid-zone model 273
Choosing boundary conditions 285
Choosing initialisation and reference state 293

WORKING WITH…

Aeroacoustics (FWH) 305
Fluid–structure interaction 317
Monitoring a run 324
Overset meshes 328
Post-processing and visualisation 335

TROUBLESHOOTING

Troubleshooting 349

WORKING WITH

Trimming to a target 360
Placing and moving a mesh 365

TUTORIALS 371

Tutorials 372
Simple aerofoil 373
Steady-state RANS 400
Time-accurate solution 414
Aeroacoustics (FWH) 426
Overset meshes 442
Incompressible flow 462
Rotor in hover 473
Polars and trimming to a target 489

RELEASE NOTES 500

Unreleased 501
2026.11.16174 502
2026.08.14860 503

2026.07.14548 504
2026.06.14412 505
2025.11.13331 506
2025.09.12996 507
2025.07.12237 508
2025.06.11828 509
2025.05.11605 510
2025.03.10415 511
2023.next 512
2025.02.10039 513
2025.01.9934 514
2024.11.9510 515
2024.10.9337 516
2024.09.9288 517
2024.06.8798 518
2024.05.8491 519
2024.04.8259 520
2023.11.6459 521
2023.10.5890 522
2023.06.5001 523
2023.03.3383 524
2023.01.3113 525
2022.09.2514 526
2022.05.1788 527
2022.03.1467 528
2022.02.1380 529
2021.11.765 530
2021.08.269 531
2021.07.123 532
2021.05.230 533
2021.04.122 534
2021.02.120 535
2020.06.110 536
2020.04.108 537
2020.03.106 538
2019.11.95 539
2019.06.86 540
2019.01.68 541
2018.09.62 542
2018.07.60 543
2018.05.58 544
2018.02.2 545
2017.2.3 546
2017.2.2 547
2017.2.1 548
2017.2.0 549
2017.1.1 551

2017.1.0 552
2016.3.0 554
2016.2.0 555
2016.1.1 556

Welcome to the zCFD documentation!

zCFD is a GPU-accelerated finite-volume CFD solver. These pages cover installing it, setting up and running a case, and every setting the input deck accepts.

New to zCFD

- What is zCFD? (p. 188): what the solver does and how it is put together.
- Installation and licensing (p. 197): installing zCFD and obtaining a licence.
- Tutorials (p. 372): five worked cases, from a simple aerofoil to overset meshes.

Guide

How zCFD works and how to use it.

- Concepts (p. 189): the solver and its input deck.
- Setting up… (p. 201): meshes, Python functions in the deck, and running a case.
- Choosing… (p. 285): boundary conditions, turbulence models, time-marching schemes and the other choices a case needs.
- Working with… (p. 324): monitoring, post-processing, overset meshes, fluid–structure interaction and aeroacoustics.
- Troubleshooting (p. 349): when a run fails or will not converge.

Reference

Every setting, output and command, in full.

- Input deck (p. 8): every key the control dictionary accepts.
- Output variables (p. 173): the fields zCFD can write.
- Command line (p. 179): invocation, flags and environment variables.

Release notes

- Release notes (p. 501): what has changed in each release.

PART 1

Reference

The input reference: every key of the zCFD control dictionary, generated from the input schema, followed by the output variables, the command line and the mesh file format.

Input deck

A whole deck: how to solve (`solver`) and what to solve (`model`).

A minimal deck

```
parameters = {
    "config version": 2,
    "solver": {
        "solver settings": {...},
        "equations": {...},
        "fluid properties": {...},
        "initialisation": {...},
        "time settings": {...},
        "numerical scheme": {...},
        "output settings": {...},
        ...
    },
    "model": {
        "<mesh>": {
            "mesh": <str>,
            "boundary conditions": {
                "<name>": {
                    "type": "wall",
                    "zones": [...],
                    ...
                },
            },
            "fluid zones": {
                "<name>": {
                    "type": "momentum",
                    ...
                },
            },
            ...
        },
    },
}
```

- config version: The deck format version. The only value accepted is `2`.
- solver (p. 10): Settings shared by every mesh. It must set `solver settings`, `equations`, `fluid properties`, `initialisation`, `time settings`, `numerical scheme` and `output settings`. `reference conditions` and `convergence control` are optional.
- model (p. 96): One block per mesh, under a name of your choosing. Each must set `mesh` and `boundary conditions`, and may set `fluid zones` and `transforms`. Its `initialisation`, `convergence control`, `numerical scheme` and `output settings` override the solver's blocks of the same name for that mesh.

KEY	TYPE	DEFAULT	DESCRIPTION
`config version`	choice	`2`	Configuration schema version.
`solver` (p. 10)	settings block		Global solver configuration and overrideable defaults

KEY	TYPE	DEFAULT	DESCRIPTION
`model` (p. 96)	named settings blocks		The meshes to solve, each under a name of the engineer's choosing.

Input deck

```
parameters = {
    "config version": 2,
    "solver": {...},
    "model": {...},
}
```

config version

Configuration schema version. Only version 2 is supported.

Type choice · **Default** 2

VALUE	WHAT IT DOES	WHEN TO USE IT
2 (default)	The current input format, against which the deck is validated.	Every deck sets it. A deck without `config_version` is read as a legacy deck and rejected.

solver

Global solver configuration and overrideable defaults

KEY	TYPE	DEFAULT	DESCRIPTION
`solver settings` (p. 11)	choice of settings blocks		Which solver runs the case, and how it is partitioned and parallelised
`equations` (p. 18)	choice of settings blocks		The equations solved, and their turbulence and transport models
`fluid properties` (p. 37)	choice of settings blocks		Fluid model properties and definition
`reference conditions` (p. 39)	settings block		Dictionary of reference conditions used throughout the simulation
`initialisation` (p. 47)	settings block		Default initialisation and restart settings for all meshes. A mesh may set its own.
`time settings` (p. 51)	choice of settings blocks		Physical time layout (steady, global or dual timestepping)
`convergence control` (p. 56)	choice of settings blocks		Convergence pacing for the inner pseudo-time loop (required for steady/DTS) A mesh may set its own.
`numerical scheme` (p. 74)	choice of settings blocks		Default spatial numerical scheme for all meshes. A mesh may set its own.
`output settings` (p. 84)	settings block		Default output variables and monitors for all meshes. A mesh may set its own.

solver

```
parameters = {
    "solver": {
        "solver settings": {...},
        "equations": {...},
        "fluid properties": {...},
        "reference conditions": {...},
        "initialisation": {...},
        "time settings": {...},
        "convergence control": {...},
        "numerical scheme": {...},
        "output settings": {...},
    },
    ...
}
```

solver settings

Which solver runs the case, and how it is partitioned and parallelised

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	not set	The zCFD solver to be used
`linear solver options` (p. 13)	settings block		Linear solver options passed to C++.
`solution smooth cycles`	int	`0`	Number of Laplace smoothing cycles for mapped solutions.
`safe`	bool	`False`	Enable safe mode for extra checks and diagnostics.
`cell reordering` (p. 16)	settings block		Cell and face reordering options for cache locality and colouring balance.
`mesh quality remediation angle threshold`	float, > 0 and < = 180	not set	Non-orthogonality angle in degrees above which an interior face uses a first-order convective flux.

solver settings

```
parameters = {
    "solver": {
        "solver settings": {
            "type": <choice>,
            "linear solver options": {...},
            "solution smooth cycles": 0,
            "safe": False,
            "cell reordering": {...},
            "mesh quality remediation angle threshold": <float>,
        },
        ...
    },
    ...
}
```

`type`

The zCFD solver to be used

Type choice · **Default** not set

VALUE	WHAT IT DOES	WHEN TO USE IT
`compressible`	The density-based solver for the compressible equations: `euler`, `viscous`, `rans` or `les` with an ideal gas.	Where density varies with pressure and temperature, including transonic and supersonic flow.
`incompressible`	The pressure-based solver for incompressible flow, coupling momentum and pressure with either the `coupled` or the `simple` scheme.	Low-speed flow in which density does not depend on pressure. Buoyancy and anelastic stratification are available through the energy equation.

`solution smooth cycles`

Number of Laplace smoothing cycles for mapped solutions.

Type int · **Default** `0`

Guidance: Choosing initialisation and reference state › Restart (p. 299)

`mesh quality remediation angle threshold`

Non-orthogonality angle in degrees above which an interior face uses a first-order convective flux. Non-orthogonality is the angle between the line joining the two cell centres and the face normal. Unset leaves every face at the scheme's order.

Type float, > 0 and < = 180 · **Default** not set · **Applies when** solver.solver settings.type is 'compressible'

linear solver options

Linear solver options passed to C++.

KEY	TYPE	DEFAULT	DESCRIPTION
`flow`	settings block	shown below	PETSc options for the mean flow linear solver
`turbulence`	settings block	shown below	PETSc options for the turbulence linear solver
`rbf`	settings block	shown below	PETSc options for the RBF linear solver
`double precision`	bool	`False`	Enable double precision for linear solver
`pressure correction`	settings block	shown below	*incompressible only* · PETSc options for the SIMPLE pressure-correction (block-1 Poisson) solver. simpleFoam's p entry: algebraic multigrid, solved to one or two orders
`momentum`	settings block	shown below	*incompressible only* · PETSc options for the SIMPLE momentum predictor (block-3) solver. simpleFoam's U entry: BiCGStab + ILU to relTol 0.1.

linear solver options

```
parameters = {
    "solver": {
        "solver settings": {
            "linear solver options": {
                "flow": {...},
                "turbulence": {...},
                "rbf": {...},
                "double precision": False,
                "pressure correction": {...},
                "momentum": {...},
            },
            ...
        },
        ...
    },
    ...
}
```

`flow`

PETSc options for the mean flow linear solver

Type settings block

Default

```
{
    "-ksp_min_it": "2",
    "-ksp_type": "fgmres",
    "-ksp_rtol": "1.0e-3",
    "-ksp_converged_reason": "",
}
```

Guidance: Choosing a linear solver › Petsc (p. 263)

`turbulence`

PETSc options for the turbulence linear solver

Type settings block

Default

```
{
    "-ksp_min_it": "2",
    "-ksp_type": "fgmres",
    "-ksp_rtol": "1.0e-5",
    "-ksp_converged_reason": "",
}
```

Guidance: Choosing a linear solver › Petsc (p. 263)

`rbf`

PETSc options for the RBF linear solver

Type settings block

Default

```
{
    "-ksp_type": "fgmres",
    "-ksp_rtol": "1.0e-5",
}
```

Guidance: Choosing a linear solver › Petsc (p. 263)

`double precision`

Enable double precision for linear solver

Type bool · **Default** `False`

Guidance: Choosing a linear solver › AMGX (p. 267)

`pressure correction`

PETSc options for the SIMPLE pressure-correction (block-1 Poisson) solver. simpleFoam's p entry: algebraic multigrid, solved to one or two orders

Type settings block · **Applies when** solver.convergence control.scheme.name is 'simple'

Default

```
{
    "-ksp_min_it": "1",
    "-ksp_type": "cg",
    "-ksp_rtol": "0.01",
    "-ksp_atol": "1.0e-12",
    "-ksp_max_it": "100",
    "-pc_type": "hypre",
    "-pc_hypre_type": "boomeramg",
    "-zcfd_pc_reuse_cycles": "20",
    "-zcfd_constant_nullspace": "1",
    "-ksp_converged_reason": "",
}
```

Guidance: Choosing a linear solver › Incompressible solver (p. 264)

`momentum`

PETSc options for the SIMPLE momentum predictor (block-3) solver. simpleFoam's U entry: BiCGStab + ILU to relTol 0.1. The under-relaxed momentum matrix is strongly diagonally dominant, so this is a few iterations

Type settings block · **Applies when** solver.convergence control.scheme.name is 'simple'

Default

```
{
    "-ksp_min_it": "1",
    "-ksp_type": "bcgs",
    "-ksp_rtol": "0.1",
    "-ksp_atol": "1.0e-12",
    "-ksp_max_it": "50",
    "-pc_type": "bjacobi",
    "-sub_pc_type": "ilu",
    "-ksp_converged_reason": "",
}
```

Guidance: Choosing a linear solver › Incompressible solver (p. 264)

cell reordering

Cell and face reordering options for cache locality and colouring balance.

KEY	TYPE	DEFAULT	DESCRIPTION
`rcm`	bool	`False`	Reverse Cuthill-McKee bandwidth-reducing cell reordering
`sloan`	bool	`False`	Sloan profile-reducing cell reordering
`sfc`	bool	`True`	Space-filling-curve cell reordering
`sfc curve`	choice	`hilbert`	Space-filling curve used when sfc is enabled
`face sort`	bool	`True`	Sort faces within each colour by their lowest cell index

cell reordering

```
parameters = {
    "solver": {
        "solver settings": {
            "cell reordering": {
                "rcm": False,
                "sloan": False,
                "sfc": True,
                "sfc curve": "hilbert",
                "face sort": True,
            },
            ...
        },
        ...
    },
    ...
}
```

`sfc curve`

Space-filling curve used when sfc is enabled

Type choice · **Default** `hilbert`

VALUE	WHAT IT DOES	WHEN TO USE IT
`hilbert` (default)	Orders each partition's interior cells along a Hilbert curve through their centroids. The curve has no jumps, so cells close in the ordering are close in space.	The default. The ordering is kept only if it reduces the cell-adjacency profile.

VALUE	WHAT IT DOES	WHEN TO USE IT
`morton`	Orders each partition's interior cells along a Morton (Z-order) curve, which interleaves the bits of the centroid coordinates. The curve jumps between distant regions.	An alternative to `hilbert`. It too is kept only if it reduces the cell-adjacency profile.

equations

The equations solved, and their turbulence and transport models

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	not set	The form of the flow equations to solve: inviscid Euler, viscous Navier–Stokes with no turbulence model, Reynolds-averaged Navier–Stokes (RANS) or large-eddy simulation (LES).
`precondition`	bool	`False`	Enable low mach number preconditioning
`turbulence` (p. 22)	settings block		*rans / les only* · Turbulence model configuration
`correction`	choice	`over_relaxed`	*incompressible only* · Non-orthogonal correction method
`correction limiter`	float, 0.0 to 1.0	`0.5`	*incompressible only* · Non-orthogonal correction limiter coefficient
`gradient limiter type`	choice	`cellmd`	*incompressible only* · Gradient limiter type
`gradient limiter`	float, 0.0 to 1.0	`0.5`	*incompressible only* · Gradient limiter coefficient
`max second order non orthogonal angle`	float, 0.0 to 90.0	`90`	*incompressible only* · Maximum angle for second order non-orthogonal correction
`central blend`	float	`0.8`	*incompressible only* · Central differencing blend factor
`solve energy`	bool	`False`	*incompressible only* · Solve the energy (temperature) transport equation
`energy variable`	choice	`temperature`	*incompressible only* · Variable transported by the energy equation: temperature, or potential temperature theta = T (p0/p)^(R/cp) for atmospheric cases
`base state` (p. 31)	choice of settings blocks		*incompressible only* · Hydrostatic base state for a stratified (anelastic) atmosphere.

KEY	TYPE	DEFAULT	DESCRIPTION
`theta perturbation` (p. 33)	settings block		*incompressible only* · Initial potential-temperature perturbation added to the base state (gravity-wave and bubble benchmarks)
`buoyancy diagnostics`	bool	`False`	*incompressible only* · Log the buoyancy source against the momentum diagonal every cycle.
`velocity perturbation` (p. 35)	settings block		*incompressible only* · Initial random velocity seed for shear-driven turbulence (LES spin-up)
`geostrophic wind`	list of float, in m/s	not set	*incompressible only* · Large-scale forcing u_g: the Coriolis source becomes -f k x (u - u_g), standing in for the synoptic pressure gradient.
`sponge layer` (p. 36)	settings block		*incompressible only* · Height-based wave-absorbing layer below the domain lid.

equations

```
parameters = {
    "solver": {
        "equations": {
            "type": <choice>,
            "precondition": False,
            "turbulence": {...},
            "correction": "over_relaxed",
            "correction limiter": 0.5,
            "gradient limiter type": "cellmd",
            "gradient limiter": 0.5,
            "max second order non orthogonal angle": 90.0,
            "central blend": 0.8,
            "solve energy": False,
            "energy variable": "temperature",
            "base state": {...},
            "theta perturbation": {...},
            "buoyancy diagnostics": False,
            "velocity perturbation": {...},
            "geostrophic wind": [...],
            "sponge layer": {...},
        },
        ...
    },
    ...
}
```

type

The form of the flow equations to solve: inviscid Euler, viscous Navier–Stokes with no turbulence model, Reynolds-averaged Navier–Stokes (RANS) or large-eddy simulation (LES).

Type choice · **Default** not set

VALUE	WHAT IT DOES
`euler`	Use the Euler flow equations in the simulation. These are compressible, inviscid flow equations.
`viscous`	Use the viscous flow equations. These are compressible, viscid flow equations which do not model turbulence.
`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.
`les`	Use the Large Eddy Simulation (LES) equations.

Guidance: Choosing an equation set › Equation sets (p. 222)

precondition

Enable low mach number preconditioning

Type bool · **Default** `False` · **Applies when** solver.time settings.kind is one of unset, 'dual time stepping'

Guidance: Choosing an equation set › precondition (p. 223)

correction

Non-orthogonal correction method

Type choice · **Default** `over_relaxed`

VALUE	WHAT IT DOES	WHEN TO USE IT
`none`	No non-orthogonal correction. The implicit face-normal gradient uses the full face area vector and the explicit correction is zero.	Only for meshes that are close to orthogonal.
`orthogonal`	Splits the face area vector $\mathbf{S}$ into an implicit part $\Delta = \lVert\mathbf{S}\rVert\,\hat{\mathbf{d}}$ along the line between cell centres, with the remainder treated explicitly.	
`minimum`	Takes the implicit part as the projection of $\mathbf{S}$ on the line between cell centres, $\Delta = (\mathbf{d}\cdot\mathbf{S}/\mathbf{d}\cdot\mathbf{d})\,\mathbf{d}$, which keeps the explicit remainder smallest.	
`over_relaxed` (default)	Takes the implicit part as $\Delta = \mathbf{d}\,\lVert\mathbf{S}\rVert^2/(\mathbf{d}\cdot\mathbf{S})$, which grows as the mesh becomes more non-orthogonal.	The default.

Guidance: Choosing an equation set › Incompressible equation sets (p. 224)

gradient limiter type

Gradient limiter type

Type choice · **Default** `cellmd`

VALUE	WHAT IT DOES
`cellmd` (default)	More robust on highly non-orthogonal meshes.
`cell`	Less dissipative.

energy variable

Variable transported by the energy equation: temperature, or potential temperature theta = T (p0/p)^(R/cp) for atmospheric cases

Type choice · **Default** `temperature`

VALUE	WHAT IT DOES	WHEN TO USE IT
`temperature` (default)	The energy equation transports static temperature T.	The default, for heat transfer without stratification.
`potential temperature`	Potential temperature $\theta = T(p_0/p)^{R/c_p}$.	The natural choice for atmospheric cases.

buoyancy diagnostics

Log the buoyancy source against the momentum diagonal every cycle. Four host syncs per cycle: debugging only

Type bool · **Default** `False` · **Applies when** base state is set

geostrophic wind

Large-scale forcing u_g: the Coriolis source becomes -f k x (u - u_g), standing in for the synoptic pressure gradient. Needs a non-zero latitude

Type list of float, in m/s · **Default** not set · **Applies when** solver.fluid properties.latitude is set and non-zero

turbulence

Turbulence model configuration

KEY	TYPE	DEFAULT	DESCRIPTION
`model`	choice	required	*turbulence only* · RANS turbulence model (sst = Menter SST k-ω, sas = Scale-Adaptive Simulation, sa-neg = Spalart-Allmaras Negative, sst-transition = γ-Reθ transition)
`hybrid`	choice	not set	*turbulence only* · Hybrid RANS-LES approach (DES = Detached Eddy Simulation, DDES = Delayed DES, IDDES = Improved Delayed DES, SAS = Scale-Adaptive Simulation)
`limit mut`	bool	`True`	*turbulence only* · Limit eddy viscosity in SST model
`limit gradient k`	float, 0.01 to 1.0	not set	*turbulence only* · Gradient limiter coefficient for turbulence kinetic energy k
`qcr`	choice	not set	*turbulence only* · QCR variant year (2000, 2020, or 2021)
`rotation correction`	bool	`False`	*turbulence only* · Enable the rotation correction.
`production`	choice	not set	*turbulence only* · Production term model for SST (vorticity = SST-V, strain-incompressible = strain-based for incompressible, strain-compressible = strain-based for compressible, kato-launder = Kato-Launder).
`betastar`	float	`0.09`	*turbulence only* · Constant in the dissipation term on the k equation in the SST model
`cdes kw`	float	`0.78`	*turbulence only* · The k-omega part of the blended C_DES in SST
`cdes keps`	float	`0.61`	*turbulence only* · The k-epsilon part of the blended C_DES in SST
`cd1`	float	`20`	*turbulence only* · Constant in the f_dt blending function in DDES and IDDES
`cd2`	int	`3`	*turbulence only* · Constant in the f_dt blending function in DDES and IDDES
`cw`	float	`0.15`	Constant in the length scale calculation in IDDES
`a1`	float	`0.31`	*turbulence only* · Constant in the eddy viscosity limiter in SST
`cdes`	float	`0.65`	*turbulence only* · C_DES constant used for SA-neg based hybrid RANS/LES models
`ca1`	float	`2`	*turbulence only* · Constant in the production of the gamma equation in the transition model

KEY	TYPE	DEFAULT	DESCRIPTION
`ca2`	float	`0.06`	*turbulence only* · Constant in the destruction term for gamma equation in the transition model
`ce1`	float	`1`	*turbulence only* · Constant in the production term of the gamma equation in the transition model
`ce2`	float	`50`	*turbulence only* · Constant in the destruction term for gamma equation in the transition model
`cthetat`	float	`0.03`	*turbulence only* · Constant in the Re-theta transition correlation in the transition model
`sigmagamma`	float	`1`	*turbulence only* · Constant in the transition model in the compressible solver
`sigmathetat`	float	`2`	*turbulence only* · Constant in the transition model in the compressible solver
`separation correction`	bool	`True`	*turbulence only* · Activates the separation correction for the transition model
`les cfl limit`	float, >= 0.0	not set	*turbulence only* · Restrict the LES branch of the DES/DDES/IDDES length scale to cells whose nominal CFL is at or below this limit. 0 (the default) disables the shield, which is also always inactive unless the run is time accurate
`les cfl width`	float, > 0.0	not set	*turbulence only* · Width of the blend above les_cfl_limit.
`les turnover coefficient`	float, >= 0.0	not set	*turbulence only* · Weight on the eddy turnover contribution to the shielded CFL.
`les`	choice	required	*les only* · LES subgrid-scale model (wale = Wall-Adapting Local Eddy-viscosity, deardorff = 1.5-order prognostic SGS-TKE with a stratification-limited length scale, none = no SGS model)
`wall damping`	bool	not set	*les only* · Enable wall damping for LES model
`relax`	float, 0.1 to 1.0	`0.5`	*les only* · Under-relaxation factor for the SGS turbulence equation.
`eps`	float	`0.41`	*les only* · von Kármán constant for WALE near-wall scaling

turbulence

```
parameters = {
    "solver": {
        "equations": {
            "turbulence": {
                "model": <choice>,
                "hybrid": <choice>,
                "limit mut": True,
                "limit gradient k": <float>,
                "qcr": <choice>,
```

```
                "rotation correction": False,
                "production": <choice>,
                "betastar": 0.09,
                "cdes kw": 0.78,
                "cdes keps": 0.61,
                "cd1": 20.0,
                "cd2": 3,
                "cw": 0.15,
                "a1": 0.31,
                "cdes": 0.65,
                "ca1": 2.0,
                "ca2": 0.06,
                "ce1": 1.0,
                "ce2": 50.0,
                "cthetat": 0.03,
                "sigmagamma": 1.0,
                "sigmathetat": 2.0,
                "separation correction": True,
                "les cfl limit": <float>,
                "les cfl width": <float>,
                "les turnover coefficient": <float>,
                "les": <choice>,
                "wall damping": <bool>,
                "relax": 0.5,
                "eps": 0.41,
            },
            ...
        },
        ...
    },
    ...
}
```

`model`

RANS turbulence model (sst = Menter SST k-ω, sas = Scale-Adaptive Simulation, sa-neg = Spalart-Allmaras Negative, sst-transition = γ-Reθ transition)

Type choice · **Required** yes

VALUE	WHAT IT DOES
`sst`	The Menter Shear Stress Transport Turbulence Model (https://turbmodels.larc.nasa.gov/sst.html)
`sas`	Scale-Adaptive Simulation, run as a standalone RANS model
`sa-neg`	Negative Spalart-Allmaras turbulence model (https://turbmodels.larc.nasa.gov/spalart.html)
`sst-transition`	The Langtry-Menter 4-equation Transitional SST Model (https://turbmodels.larc.nasa.gov/langtrymenter_4eqn.html)

VALUE	WHAT IT DOES
`spalart-allmaras`	Spalart-Allmaras turbulence model (https://turbmodels.larc.nasa.gov/spalart.html)

Guidance: Choosing a turbulence model (p. 226)

`hybrid`

Hybrid RANS-LES approach (DES = Detached Eddy Simulation, DDES = Delayed DES, IDDES = Improved Delayed DES, SAS = Scale-Adaptive Simulation)

Type choice · **Default** not set

VALUE	WHAT IT DOES
`DES`	Detached Eddy Simulation
`DDES`	Delayed Detached Eddy Simulation
`IDDES`	Improved Delayed Detached Eddy Simulation
`SAS`	Scale-Adaptive Simulation run as the hybrid branch of `model`

Guidance: Choosing a turbulence model › Hybrid RANS–LES (p. 226)

`qcr`

QCR variant year (2000, 2020, or 2021)

Type choice · **Default** not set

VALUE	WHAT IT DOES
`2000`	QCR 2000
`2020`	QCR 2020
`2021`	QCR 2021

Guidance: Choosing a turbulence model › qcr (p. 227)

`rotation correction`

Enable the rotation correction. Read by both turbulence models, in their own forms: SST applies Hellsten's f4 = 1/(1 + Crc*Ri) to the omega destruction term, and SA adds Crot*min(0, S - Omega) to production (Dacles-Mariani / Spalart-Shur).

Type bool · **Default** `False`

production

Production term model for SST (vorticity = SST-V, strain-incompressible = strain-based for incompressible, strain-compressible = strain-based for compressible, kato-launder = Kato-Launder). When omitted the solver build chooses: kato-launder for sst-transition, vorticity otherwise (the solver's Menter SST defaults) — a schema default here would silently override the transition build's choice.

Type choice · **Default** not set

VALUE	WHAT IT DOES
`vorticity`	The vorticity based production term (SST-V) $P_m = \mu_t \Omega_v^2 - \frac{2}{3}\rho k \delta_{ij} \frac{\partial u_i}{\partial x_j}$
`strain-incompressible`	The strain based production term for incompressible flows $P_m = \mu_t S^2$
`strain-compressible`	The strain based production term for compressible flows $P_m = \mu_t S^2 - \frac{2}{3}\rho k \delta_{ij} \frac{\partial u_i}{\partial x_j}$
`kato-launder`	The Kato-Launder source term $P_m = \mu_t S \Omega_v - \frac{2}{3}\rho k \delta_{ij} \frac{\partial u_i}{\partial x_j}$

Guidance: Choosing a turbulence model › production (p. 227)

betastar

Constant in the dissipation term on the k equation in the SST model

Type float · **Default** `0.09`

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

cdes kw

The k-omega part of the blended C_DES in SST

Type float · **Default** `0.78`

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

cdes keps

The k-epsilon part of the blended C_DES in SST

Type float · **Default** `0.61`

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

cd1

Constant in the f_dt blending function in DDES and IDDES

Type float · Default 20

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

cd2

Constant in the f_dt blending function in DDES and IDDES

Type int · Default 3

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

cw

Constant in the length scale calculation in IDDES

Type float · Default 0.15

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

a1

Constant in the eddy viscosity limiter in SST

Type float · Default 0.31

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

cdes

C_DES constant used for SA-neg based hybrid RANS/LES models

Type float · Default 0.65

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

ca1

Constant in the production of the gamma equation in the transition model

Type float · Default 2

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

ca2

Constant in the destruction term for gamma equation in the transition model

Type float · Default 0.06

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

ce1

Constant in the production term of the gamma equation in the transition model
Type float · **Default** `1`

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

ce2

Constant in the destruction term for gamma equation in the transition model
Type float · **Default** `50`

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

cthetat

Constant in the Re-theta transition correlation in the transition model
Type float · **Default** `0.03`

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

sigmagamma

Constant in the transition model in the compressible solver
Type float · **Default** `1`

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

sigmathetat

Constant in the transition model in the compressible solver
Type float · **Default** `2`

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

separation correction

Activates the separation correction for the transition model
Type bool · **Default** `True`

Guidance: Choosing a turbulence model › Advanced parameters for turbulence models (p. 227)

les cfl limit

Restrict the LES branch of the DES/DDES/IDDES length scale to cells whose nominal CFL is at or below this limit. 0 (the default) disables the shield, which is also always inactive unless the run is time accurate

Type float, >= 0.0 · **Default** not set · **Applies when** solver.time settings.type is 'unsteady'

Guidance: Choosing a turbulence model > Shielding the LES branch by CFL (p. 228)

`les cfl width`

Width of the blend above les_cfl_limit. The shield ramps from LES at les_cfl_limit to full RANS at les_cfl_limit plus this width.

Type float, > 0.0 · **Default** not set · **Applies when** solver.time settings.type is 'unsteady'

Guidance: Choosing a turbulence model > Shielding the LES branch by CFL (p. 228)

`les turnover coefficient`

Weight on the eddy turnover contribution to the shielded CFL. The time step has to resolve both the transport of an eddy across the cell and the eddy's own turnover, and the shield takes whichever is larger. 0 leaves the transport only criterion, which cannot see a cell whose speed relative to the mesh is near zero. Defaults to 1, and has no effect unless les_cfl_limit is set

Type float, >= 0.0 · **Default** not set · **Applies when** solver.time settings.type is 'unsteady'

Guidance: Choosing a turbulence model > les_turnover_coefficient (p. 228)

`les`

LES subgrid-scale model (wale = Wall-Adapting Local Eddy-viscosity, deardorff = 1.5-order prognostic SGS-TKE with a stratification-limited length scale, none = no SGS model)

Type choice · **Required** yes

VALUE	WHAT IT DOES	WHEN TO USE IT
`none`	Implicit LES (no explicit SGS model).	
`wale`	The Wall-Adapting Local Eddy-viscosity model.	
`deardorff`	The Deardorff 1.5-order closure, which transports the subgrid turbulent kinetic energy e and limits the mixing length under stable stratification. The eddy viscosity is $0.1\, l\sqrt{e}$ and the heat diffusivity is $(1 + 2l/\Delta)$ times it.	For stratified atmospheric boundary layers with the incompressible solver. The compressible solver does not implement it.

Guidance: Choosing a turbulence model > LES sub-grid models (p. 227)

`relax`

Under-relaxation factor for the SGS turbulence equation. Read UNCONDITIONALLY by the C++ (IncompSolverData.h initKernels, turbulence.relax), so it must always be present in the emitted dict – it carried a default of 0.5 in the legacy schema and was lost when the schemas moved, leaving the deardorff path with no way to supply it.

Type float, 0.1 to 1.0 · **Default** `0.5`

base state

Hydrostatic base state for a stratified (anelastic) atmosphere. Its presence turns stratification on: density becomes the base-state rho(z) and buoyancy is g (theta - theta_base)/theta_base. Needs a non-zero gravity

KEY	TYPE	DEFAULT	DESCRIPTION
`reference`	str	not set	Reference condition supplying the surface temperature and pressure (e.g.
`surface temperature`	float, > 0.0 K	not set	Temperature at the reference location; overrides the reference condition's value
`surface pressure`	float, > 0.0 Pa	not set	Pressure at the reference location; overrides the reference condition's value
`kind`	choice	required	Uniform temperature; theta grows as exp(g z / (cp T0))
`lapse rate`	float, in K/m	`0.0065`	*constant lapse rate only* · -dT/dz; g/cp (about 0.0098) is the neutral dry adiabat
`brunt vaisala frequency`	float, >= 0.0 1/s	`0.01`	*constant brunt vaisala only* · N
`mixed layer depth`	float, > 0.0 m	`100`	*two layer only* · Depth h of the neutral mixed layer
`upper lapse rate`	float, >= 0.0 K/m	`0.01`	*two layer only* · dtheta/dz above the mixed layer

```
base state

parameters = {
    "solver": {
        "equations": {
            "base state": {
                "reference": <str>,
                "surface temperature": <float>,
                "surface pressure": <float>,
                "kind": <choice>,
                "lapse rate": 0.0065,
                "brunt vaisala frequency": 0.01,
                "mixed layer depth": 100.0,
                "upper lapse rate": 0.01,
            },
            ...
        },
        ...
    },
    ...
```

```
}
```

reference

Reference condition supplying the surface temperature and pressure (e.g. 'IC_1'); defaults to the initialisation reference

Type str · **Default** not set

kind

Uniform temperature; theta grows as exp(g z / (cp T0))

Type choice · **Required** yes

VALUE	WHAT IT DOES	WHEN TO USE IT
`isothermal`	Uniform temperature T_0, so potential temperature grows with height as $\theta = \theta_0 \exp(gz/(c_p T_0))$.	
`constant lapse rate`	Temperature falls linearly with height, $T = T_0 - \Gamma z$, with Γ set by `lapse_rate`.	
`constant brunt vaisala`	Uniform stratification $\theta = \theta_0 \exp(N^2 z/g)$.	
`two layer`	A neutral mixed layer of uniform θ_0 up to `mixed_layer_depth`, capped by stable stratification in which θ rises at `upper_lapse_rate`.	The GABLS1 stable boundary layer set-up.

theta perturbation

Initial potential-temperature perturbation added to the base state (gravity-wave and bubble benchmarks)

KEY	TYPE	DEFAULT	DESCRIPTION
`kind`	choice	required	inertia gravity wave: A sin(pi z/H) / (1 + ((x - xc)/a)^2) (Skamarock & Klemp 1994); bubble: A cos^2(pi r/2) inside the ellipse r = |((x - xc)/rx, (z - zc)/rz)| < = 1; random: position-hashed +/-A turbulence seed below 'depth' (GABLS1)
`amplitude`	float, in K	required	A; negative for a cold bubble
`centre`	list of float, in m	`[0, 0]`	[xc, zc]; zc is used by the bubble only
`half width`	float, > 0.0 m	`5000`	a for the gravity wave, or the bubble x radius rx
`half height`	float, > 0.0 m	`2000`	Bubble z radius rz
`depth`	float, > 0.0 m	`10000`	H in sin(pi z/H) for the gravity wave

theta perturbation

```
parameters = {
    "solver": {
        "equations": {
            "theta perturbation": {
                "kind": <choice>,
                "amplitude": <float>,
                "centre": [0.0, 0.0],
                "half width": 5000.0,
                "half height": 2000.0,
                "depth": 10000.0,
            },
            ...
        },
        ...
    },
    ...
}
```

`kind`

inertia gravity wave: A sin(pi z/H) / (1 + ((x - xc)/a)^2) (Skamarock & Klemp 1994); bubble: A cos^2(pi r/2) inside the ellipse r = |((x - xc)/rx, (z - zc)/rz)| < = 1; random: position-hashed +/-A turbulence seed

below 'depth' (GABLS1)

Type choice · **Required** yes

VALUE	WHAT IT DOES	WHEN TO USE IT
`inertia gravity wave`	$\theta' = A\sin(\pi z/H)/(1+((x-x_c)/a)^2)$ (Skamarock & Klemp 1994).	
`bubble`	$\theta' = A\cos^2(\pi r/2)$ inside the ellipse $r = \lVert((x-x_c)/r_x, (z-z_c)/r_z)\rVert \le 1$, zero outside.	
`random`	A pseudo-random θ' of up to `amplitude` either side of zero, hashed from each cell's position and applied only below `depth`. The seed is reproducible and independent of the partitioning.	Seeding turbulence in a stable boundary layer, as in GABLS1.

velocity perturbation

Initial random velocity seed for shear-driven turbulence (LES spin-up)

KEY	TYPE	DEFAULT	DESCRIPTION
`kind`	choice	required	Position-hashed +/-amplitude seed on all three components
`amplitude`	float, > 0.0 m/s	required	Peak perturbation on each velocity component
`depth`	float, > 0.0 m	`50`	Height below which the seed is applied; zero above

velocity perturbation

```
parameters = {
    "solver": {
        "equations": {
            "velocity perturbation": {
                "kind": <choice>,
                "amplitude": <float>,
                "depth": 50.0,
            },
            ...
        },
        ...
    },
    ...
}
```

`kind`

Position-hashed +/-amplitude seed on all three components

Type choice · **Required** yes

VALUE	WHAT IT DOES
`random`	Independent pseudo-random perturbations of up to `amplitude` either side of zero on each velocity component, hashed from each cell's position and applied only below `depth`. The seed is reproducible and independent of the partitioning.

sponge layer

Height-based wave-absorbing layer below the domain lid. Requires base_state

KEY	TYPE	DEFAULT	DESCRIPTION
`sigma max`	float, > 0.0 1/s	required	Peak damping rate at the domain top
`start height`	float, >= 0.0 m	required	Height where the damping ramp begins
`relax theta`	bool	`True`	Relax potential temperature toward the base state inside the sponge, as well as velocity.
`target velocity`	list of float, in m/s	not set	Velocity the sponge damps toward; unset damps toward rest

sponge layer

```
parameters = {
    "solver": {
        "equations": {
            "sponge layer": {
                "sigma max": <float>,
                "start height": <float>,
                "relax theta": True,
                "target velocity": [...],
            },
            ...
        },
        ...
    },
    ...
}
```

`relax theta`

Relax potential temperature toward the base state inside the sponge, as well as velocity. TRUE reproduces the existing behaviour: EnergyEqn.cu applies the sponge to theta whenever a sponge and a base state are present, which the deck's 'target velocity' wording does not convey. Set FALSE for a velocity-only sponge – the upper boundary then stops being a heat sink, which is what isolates its effect on a stably-stratified case without also letting gravity waves reflect off the lid.

Type bool · **Default** `True`

fluid properties

Fluid model properties and definition

KEY	TYPE	DEFAULT	DESCRIPTION
`material`	str	`ideal gas`	Fluid material model
`gamma`	float, > 1.0	`1.4`	Ratio of specific heats Cp/Cv
`gas constant`	float, > 0	`287`	Specific gas constant (J/(kg·K))
`sutherlands const`	float, > 0	`110.4`	Sutherland's reference temperature (K)
`prandtl no`	float, > 0	`0.72`	Molecular Prandtl number
`gravity`	vector (3 floats), in m/s^2	not set	Gravitational acceleration vector, for example [0, 0, -9.81].
`latitude`	float, -90 to 90 deg	not set	Latitude for the f-plane Coriolis force, f = 2 Omega sin(latitude), positive north.
`planet rotation`	float, > 0 rad/s	not set	Planetary rotation rate for the Coriolis parameter f = 2 Omega sin(latitude).
`thermal expansion coeff`	float, > 0 1/K	not set	Thermal expansion coefficient beta for Boussinesq buoyancy.
`turbulent prandtl no`	float, > 0	`0.9`	*viscous only* · Turbulent Prandtl number

fluid properties

```
parameters = {
    "solver": {
        "fluid properties": {
            "material": "ideal gas",
            "gamma": 1.4,
            "gas constant": 287.0,
            "sutherlands const": 110.4,
            "prandtl no": 0.72,
            "gravity": <[x, y, z]>,
            "latitude": <float>,
            "planet rotation": <float>,
            "thermal expansion coeff": <float>,
            "turbulent prandtl no": 0.9,
        },
        ...
    },
```

```
    ...
}
```

gravity

Gravitational acceleration vector, for example [0, 0, -9.81]. Drives Boussinesq and anelastic buoyancy, sets the vertical for the Coriolis force and the stratified turbulence terms, and enters the Monin-Obukhov wall law. Omitted means no gravity.

Type vector (3 floats), in m/s^2 · **Default** not set · **Applies when** solver.solver settings.type is 'incompressible'

latitude

Latitude for the f-plane Coriolis force, f = 2 Omega sin(latitude), positive north. Needs a non-zero gravity, which sets the local vertical. Omitted means no Coriolis force.

Type float, -90 to 90 deg · **Default** not set · **Applies when** solver.solver settings.type is 'incompressible' and solver.fluid properties.gravity is set and non-zero

planet rotation

Planetary rotation rate for the Coriolis parameter f = 2 Omega sin(latitude). Needs a non-zero latitude. Defaults to Earth's sidereal rate, 7.292115e-5

Type float, > 0 rad/s · **Default** not set · **Applies when** solver.solver settings.type is 'incompressible' and solver.fluid properties.latitude is set and non-zero

thermal expansion coeff

Thermal expansion coefficient beta for Boussinesq buoyancy. The solver stores the dimensionless group beta*Tref, so the buoyancy source becomes -g_nondim* (beta*Tref)* (theta - 1). Needs a non-zero gravity. Ignored when base_state is set.

Type float, > 0 1/K · **Default** not set · **Applies when** solver.solver settings.type is 'incompressible' and solver.fluid properties.gravity is set and non-zero and solver.equations.solve energy is True

reference conditions

Dictionary of reference conditions used throughout the simulation

KEY	TYPE	DEFAULT	DESCRIPTION
`temperature`	float, >= 0	required	Static temperature (K)
`pressure`	float, >= 0	required	Static pressure (Pa)
`v` (p. 42)	vector (3 floats) or settings block		Velocity vector [Vx, Vy, Vz] (m/s) or inflow vector specification
`reference length`	float, >= 0	`1`	Reference length (m) used in Reynolds number calculation
`profile` (p. 43)	settings block		Atmospheric profile parameters for specialised flow conditions
`total pressure ratio`	float, >= 0	not set	Total pressure ratio for inflow boundary conditions
`total temperature ratio`	float, >= 0	not set	Total temperature ratio for inflow boundary conditions
`static pressure ratio`	float, >= 0	not set	Static pressure ratio for outflow boundary conditions
`reynolds no`	float, >= 0	not set	*viscous / rans only* · Reynolds number.
`viscosity`	float, >= 0	not set	*viscous / rans only* · Viscosity.
`turbulence intensity`	float, >= 0	`0.01`	*rans only* · Freestream turbulence intensity (fraction, e.g. 0.01 = 1%)
`eddy viscosity ratio`	float, >= 0	`0.1`	*rans only* · Ratio of eddy viscosity to molecular viscosity (μt/μ)
`ambient turbulence intensity`	float, >= 0	`1e-20`	*rans only* · Ambient turbulence intensity for far-field conditions
`ambient eddy viscosity ratio`	float, >= 0	`1e-20`	*rans only* · Ambient eddy viscosity ratio for far-field conditions

reference conditions

```
parameters = {
    "solver": {
        "reference conditions": {
            "<name>": {
                "temperature": <float>,
                "pressure": <float>,
```

```
                "v": {"vector": [0.0, 0.0, 0.0]},
                "reference length": 1.0,
                "profile": {...},
                "total pressure ratio": <float>,
                "total temperature ratio": <float>,
                "static pressure ratio": <float>,
                "reynolds no": <float>,
                "viscosity": <float>,
                "turbulence intensity": 0.01,
                "eddy viscosity ratio": 0.1,
                "ambient turbulence intensity": 1e-20,
                "ambient eddy viscosity ratio": 1e-20,
            },
        },
        ...
    },
    ...
}
```

`reference length`

Reference length (m) used in Reynolds number calculation

Type float, >= 0 · **Default** 1

Guidance: Choosing initialisation and reference state > Viscosity (p. 295)

`total pressure ratio`

Total pressure ratio for inflow boundary conditions

Type float, >= 0 · **Default** not set

Guidance: Choosing initialisation and reference state > Reference conditions (p. 293)

`total temperature ratio`

Total temperature ratio for inflow boundary conditions

Type float, >= 0 · **Default** not set

Guidance: Choosing initialisation and reference state > Reference conditions (p. 293)

`static pressure ratio`

Static pressure ratio for outflow boundary conditions

Type float, >= 0 · **Default** not set

Guidance: Choosing initialisation and reference state > Reference conditions (p. 293)

`reynolds no`

Reynolds number. Only valid for viscous or RANS equations.

Type float, >= 0 · **Default** not set

Guidance: Choosing initialisation and reference state > Viscosity (p. 295)

`viscosity`

Viscosity. Only valid for viscous or RANS equations.

Type float, >= 0 · **Default** not set

Guidance: Choosing initialisation and reference state > Viscosity (p. 295)

`turbulence intensity`

Freestream turbulence intensity (fraction, e.g. 0.01 = 1%)

Type float, >= 0 · **Default** `0.01`

Guidance: Choosing initialisation and reference state > Turbulence intensity and eddy viscosity (p. 296)

`eddy viscosity ratio`

Ratio of eddy viscosity to molecular viscosity (μt/μ)

Type float, >= 0 · **Default** `0.1`

Guidance: Choosing initialisation and reference state > Turbulence intensity and eddy viscosity (p. 296)

`ambient turbulence intensity`

Ambient turbulence intensity for far-field conditions

Type float, >= 0 · **Default** `1e-20`

Guidance: Choosing initialisation and reference state > Turbulence intensity and eddy viscosity (p. 296)

`ambient eddy viscosity ratio`

Ambient eddy viscosity ratio for far-field conditions

Type float, >= 0 · **Default** `1e-20`

Guidance: Choosing initialisation and reference state > Turbulence intensity and eddy viscosity (p. 296)

v

Velocity vector [Vx, Vy, Vz] (m/s) or inflow vector specification

KEY	TYPE	DEFAULT	DESCRIPTION
vector	vector (3 floats)	required	Velocity [Vx, Vy, Vz] in m/s.
mach	float, >= 0	not set	Mach number to scale magnitude of inflow vector by

v

```
parameters = {
    "solver": {
        "reference conditions": {
            "<name>": {
                "v": {
                    "vector": <[x, y, z]>,
                    "mach": <float>,
                },
                ...
            },
        },
        ...
    },
    ...
}
```

vector

Velocity [Vx, Vy, Vz] in m/s. When mach is set, the compressible solver keeps only its direction and takes the speed from the Mach number; the incompressible solver ignores mach.

Type vector (3 floats) · **Required** yes

Guidance: Choosing initialisation and reference state › Velocity (p. 295)

mach

Mach number to scale magnitude of inflow vector by

Type float, >= 0 · **Default** not set

Guidance: Choosing initialisation and reference state › Velocity (p. 295)

profile

Atmospheric profile parameters for specialised flow conditions

KEY	TYPE	DEFAULT	DESCRIPTION
`abl` (p. 44)	settings block		Atmospheric Boundary Layer profile parameters
`field`	str	not set	Profile file name (e.g. .vtp)
`use wall distance`	bool	not set	Use wall distance for profile

profile

```
parameters = {
    "solver": {
        "reference conditions": {
            "<name>": {
                "profile": {
                    "abl": {...},
                    "field": <str>,
                    "use wall distance": <bool>,
                },
                ...
            },
        },
        ...
    },
    ...
}
```

`field`

Profile file name (e.g. .vtp)

Type str · **Default** not set

Guidance: Choosing initialisation and reference state › field (p. 297)

`use wall distance`

Use wall distance for profile

Type bool · **Default** not set

Guidance: Choosing initialisation and reference state › field (p. 297)

abl

Atmospheric Boundary Layer profile parameters

KEY	TYPE	DEFAULT	DESCRIPTION
`roughness length`	float, > 0	required	Surface roughness length in meters.
`friction velocity`	float, > 0	not set	Friction velocity in m/s.
`surface layer height`	float, > 0	not set	Height of the surface layer (m) that the log-law profile applies within; above this height the profile is clamped.
`monin obukhov length`	float	not set	Monin-Obukhov stability length L (m).
`tke`	float, >= 0	not set	Reference turbulent kinetic energy ($m^{2/s}2$), scaled by friction_velocity**2 to give the profile's TKE.
`up`	vector (3 floats)	not set	Unit vector [x, y, z] defining the vertical ('up') direction used with ground_level to compute height above ground as dot(position, up) - ground_level.
`ground level`	float, >= 0	`0`	Datum elevation of the ground (m) in the mesh's own coordinate system, measured along 'up'.

abl

```
parameters = {
    "solver": {
        "reference conditions": {
            "<name>": {
                "profile": {
                    "abl": {
                        "roughness length": <float>,
                        "friction velocity": <float>,
                        "surface layer height": <float>,
                        "monin obukhov length": <float>,
                        "tke": <float>,
                        "up": <[x, y, z]>,
                        "ground level": 0.0,
                    },
                    ...
                },
                ...
            },
        },
        ...
    },
```

```
    ...
}
```

`roughness length`

Surface roughness length in meters. Must be positive. This is the aerodynamic roughness z_0 in the log-law profile (not to be confused with 'ground_level', which is a ground datum elevation, not a roughness).

Type float, > 0 · **Required** yes

Guidance: Choosing initialisation and reference state > roughness_length (p. 298)

`friction velocity`

Friction velocity in m/s. Must be positive when set. The compressible solver needs friction_velocity, surface_layer_height or both.

Type float, > 0 · **Default** not set · **Applies when** 'friction velocity' is required when solver.solver settings.type is 'compressible' and surface layer height is unset

Guidance: Choosing initialisation and reference state > friction_velocity (p. 298)

`surface layer height`

Height of the surface layer (m) that the log-law profile applies within; above this height the profile is clamped. C++ sentinel default is -1.0 (unset): the compressible read sites then derive it from roughness_length and friction_velocity when both are set.

Type float, > 0 · **Default** not set · **Applies when** solver.solver settings.type is one of unset, 'compressible'

Guidance: Choosing initialisation and reference state > surface_layer_height (p. 299)

`monin obukhov length`

Monin-Obukhov stability length L (m). Negative values are read as unstable and add a stability correction to the log-law velocity/TKE/epsilon (only when L < 0 — the current C++ never applies a correction for L > 0, i.e. stable stratification is not represented here). Zero is not a valid value. The solver's local default is +1.0 (a positive, 'no correction' value) when the key is absent — this is the effective neutral default. Leave unset for neutral stability.

Type float · **Default** not set · **Applies when** solver.solver settings.type is one of unset, 'compressible'

Guidance: Choosing initialisation and reference state > monin_obukhov_length (p. 299)

`tke`

Reference turbulent kinetic energy ($m^{2/s}2$), scaled by friction_velocity**2 to give the profile's TKE. Only applied when friction_velocity is also set — if friction_velocity is absent, the solver discards this value with a warning and derives TKE from the wall distance instead.

Type float, >= 0 · **Default** not set · **Applies when** solver.solver settings.type is one of unset, 'compressible'

Guidance: Choosing initialisation and reference state > ABL (p. 297)

`up`

Unit vector [x, y, z] defining the vertical ('up') direction used with ground_level to compute height above ground as dot(position, up) - ground_level. Default when unset is (0, 0, 1).

Type vector (3 floats) · **Default** not set · **Applies when** solver.solver settings.type is one of unset, 'compressible'

Guidance: Choosing initialisation and reference state > ABL (p. 297)

`ground level`

Datum elevation of the ground (m) in the mesh's own coordinate system, measured along 'up'. Subtracted from dot(position, up) before the log-law height is computed, as height = min(dot(p, up) - ground_level, surface_layer_height) — it shifts where the profile's height origin sits, it is NOT a measurement or reference height. This is unrelated to the aerodynamic roughness length z_0 (that's 'roughness_length' above) and unrelated to ABLZone.reference_height. NOTE: the initial-condition profile reads this key but never applies it to the initial field — only the farfield BC profile honours it — so a mesh with a non-zero ground datum gets inconsistent IC/BC profiles.

Type float, >= 0 · **Default** `0` · **Applies when** solver.solver settings.type is one of unset, 'compressible'

Guidance: Choosing initialisation and reference state > ABL (p. 297)

initialisation

Default initialisation and restart settings for all meshes.

KEY	TYPE	DEFAULT	DESCRIPTION
`reference`	str	not set	Reference condition used for non-dimensionalization (e.g.
`initial conditions`	str or Python function, or a Python function	not set	Initial conditions for the simulation: a named condition from reference_ conditions, or a function of each cell's dimensional state (m, m, Pa, K, m/s), called as f(*, location, wall_distance, pressure, temperature, velocity); it returns a dict of any of 'pressure', 'temperature' and 'velocity'.
`restart`	bool	`False`	Restart from a previous solution.
`restart casename`	str	not set	Name of the case to restart from.
`restart meshname`	str	not set	Mesh the restart case ran on, for an interpolated restart from a results file that has no case record beside it.
`restart ignore history`	bool	`False`	Ignore cycle history from restart file.
`interpolate restart`	bool	`False`	Start from the restart case's solution interpolated onto this case's mesh, which may differ in resolution, cell type or geometry.
`interpolation`	choice	`linear`	How an interpolated restart forms each new cell's value from the old cell containing its centre: 'linear' adds the old cell's least-squares gradient times the offset between the centres, held within the values of that cell and its face neighbours; 'injection' copies the old cell's value.
`restart models`	settings block	not set	For an interpolated restart, the restart case's model each of this case's models starts from, where the names differ, as {this case's model: restart case's model}.
`interpolate outside limit`	float, 0.0 to 1.0	`0.001`	Largest fraction of this case's cells that may lie outside the restart case's mesh, so 0.001 is 0.1%.

initialisation

```
parameters = {
    "solver": {
        "initialisation": {
            "reference": <str>,
            "initial conditions": <str or Python function>,
            "restart": False,
            "restart casename": <str>,
            "restart meshname": <str>,
            "restart ignore history": False,
            "interpolate restart": False,
            "interpolation": "linear",
            "restart models": {...},
            "interpolate outside limit": 0.001,
        },
        ...
    },
    ...
}
```

reference

Reference condition used for non-dimensionalization (e.g. 'IC_1'). When the initial_conditions IC has zero velocity (e.g. a quiescent cavity), set this to a reference condition with the physical velocity/Reynolds number so the solver can non-dimensionalize correctly.

Type str · **Default** not set · **Applies when** 'reference' is required when restart is True and initial conditions is unset

Guidance: Choosing initialisation and reference state › Reference state (p. 295)

initial conditions

Initial conditions for the simulation: a named condition from reference_conditions, or a function of each cell's dimensional state (m, m, Pa, K, m/s), called as f(*, location, wall_distance, pressure, temperature, velocity); it returns a dict of any of 'pressure', 'temperature' and 'velocity'. Required unless restart=True.

Type str or Python function, or a Python function · **Default** not set

Guidance: Choosing initialisation and reference state › Initial state (p. 294)

restart

Restart from a previous solution. Default is False.

Type bool · **Default** `False`

Guidance: Choosing initialisation and reference state › Restart (p. 299)

`restart casename`

Name of the case to restart from. Defaults to the current casename.

Type str · **Default** not set · **Applies when** restart is True

Guidance: Choosing initialisation and reference state › Restart (p. 299)

`restart meshname`

Mesh the restart case ran on, for an interpolated restart from a results file that has no case record beside it. A case run by this version of zCFD records its own mesh.

Type str · **Default** not set · **Applies when** restart is True

Guidance: Choosing initialisation and reference state › Restart (p. 299)

`restart ignore history`

Ignore cycle history from restart file. Default is False.

Type bool · **Default** `False` · **Applies when** restart is True

Guidance: Choosing initialisation and reference state › Restart (p. 299)

`interpolate restart`

Start from the restart case's solution interpolated onto this case's mesh, which may differ in resolution, cell type or geometry. Variables the restart case does not solve come from the initial conditions.

Type bool · **Default** `False` · **Applies when** restart is True

Guidance: Choosing initialisation and reference state › Restart (p. 299)

`interpolation`

How an interpolated restart forms each new cell's value from the old cell containing its centre: 'linear' adds the old cell's least-squares gradient times the offset between the centres, held within the values of that cell and its face neighbours; 'injection' copies the old cell's value.

Type choice · **Default** `linear` · **Applies when** restart is True

VALUE	WHAT IT DOES
`linear` (default)	Linear, bounded by the neighbouring cells
`injection`	Copy the containing cell

Guidance: Choosing initialisation and reference state › Restart (p. 299)

`restart models`

For an interpolated restart, the restart case's model each of this case's models starts from, where the names differ, as {this case's model: restart case's model}. Models not listed start from the model of the same name.

Type settings block · **Default** not set · **Applies when** restart is True

---

`interpolate outside limit`

Largest fraction of this case's cells that may lie outside the restart case's mesh, so 0.001 is 0.1%. Those cells take the nearest restart cell's value; above the limit the run stops.

Type float, 0.0 to 1.0 · **Default** `0.001` · **Applies when** restart is True

time settings

Physical time layout (steady, global or dual timestepping)

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	not set	Whether the run is time-accurate.
`kind`	choice	required	*unsteady only* · How each physical time step is taken: `global timestepping` advances in physical time alone, `dual time stepping` converges a pseudo-time loop within every step.
`total time`	float, > 0	required	*unsteady only* · Total physical simulation time
`time step`	float, > 0	required	*unsteady only* · Physical time step size
`scheme` (p. 54)	settings block		*global timestepping only* · Time integration scheme.
`order`	choice	`second`	*dual time stepping only* · Time integration order
`start`	int, >= 0	`0`	*dual time stepping only* · Steady pseudo cycles run at real-time cycle 0 before time-accurate stepping begins; the same solve as a steady run of that length.

time settings

```
parameters = {
    "solver": {
        "time settings": {
            "type": <choice>,
            "kind": <choice>,
            "total time": <float>,
            "time step": <float>,
            "scheme": {...},
            "order": "second",
            "start": 0,
        },
        ...
    },
    ...
}
```

`type`

Whether the run is time-accurate. 'steady' converges a single steady state in pseudo-time; 'unsteady' marches in physical time over total_time in steps of time_step.

Type choice · **Default** not set

VALUE	WHAT IT DOES	WHEN TO USE IT
`steady`	Converges one steady state by marching in pseudo-time, with a local time step in each cell and no physical time derivative.	For flows that settle to a steady state. `convergence_control` sets the scheme, CFL and number of cycles.
`unsteady`	Marches the solution in physical time to `total_time` in steps of `time_step`. `kind` selects dual time stepping or global time stepping.	For flows that vary in time. Dual time stepping converges each physical step in pseudo-time, so `time_step` is set by the time scale to be resolved. Global time stepping is explicit, so `time_step` must also be stable in the smallest cell.

Guidance: Choosing a time-marching scheme › Running steady state (p. 238)

`kind`

How each physical time step is taken: `global timestepping` advances in physical time alone, `dual time stepping` converges a pseudo-time loop within every step.

Type choice · **Required** yes

VALUE	WHAT IT DOES
`global timestepping`	Advances the solution in physical time only, using the same physical time step in every cell — there is no pseudo-time convergence loop.
`dual time stepping`	Advances the solution in physical time, with an inner pseudo-time convergence loop run at each physical time step to converge that step before advancing.

Guidance: Choosing a time-marching scheme › Global time stepping (p. 240)

`order`

Time integration order

Type choice · **Default** `second`

VALUE	WHAT IT DOES	WHEN TO USE IT
`first`	First-order backward difference (BDF1, implicit Euler) for the physical time derivative.	More dissipative in time than `second`.
`second` (default)	Second-order backward difference (BDF2) for the physical time derivative. Until two earlier time levels are held, the step is taken at first order.	The default, for time-accurate results.
`1`	The same as `first`.	
`2`	The same as `second`.	

Guidance: Choosing a time-marching scheme › Dual time stepping (p. 238)

`start`

Steady pseudo cycles run at real-time cycle 0 before time-accurate stepping begins; the same solve as a steady run of that length. A settling phase for an impulsive start. 0 (the default) runs none, so real-time cycle 0 is itself time-accurate and a supplied initial condition is marched from as given

Type int, >= 0 · **Default** `0`

Guidance: Choosing a time-marching scheme › Dual time stepping (p. 238)

scheme

Time integration scheme. GTS marches physical time with this scheme itself, so the scheme sets the run's order of accuracy — only the three-stage TVD Runge-Kutta (SSP-RK3, third order) is admissible.

KEY	TYPE	DEFAULT	DESCRIPTION
`name`	choice	`runge kutta`	Physical time integration scheme.
`stage`	choice	`3`	Runge-Kutta stage configuration: the three-stage TVD scheme (SSP-RK3), third order in physical time.

scheme

```
parameters = {
    "solver": {
        "time settings": {
            "scheme": {
                "name": "runge kutta",
                "stage": 3,
            },
            ...
        },
        ...
    },
    ...
}
```

`name`

Physical time integration scheme. Global time stepping admits only the three-stage TVD Runge-Kutta

Type choice · **Default** `runge kutta`

VALUE	WHAT IT DOES
`runge kutta` (default)	The three-stage TVD Runge-Kutta (SSP-RK3), third order in physical time.

Guidance: Choosing a time-marching scheme › Global time stepping (p. 240)

`stage`

Runge-Kutta stage configuration: the three-stage TVD scheme (SSP-RK3), third order in physical time. Spelled either 3 or 'rk third order tvd' — the same scheme either way

Type choice · **Default** 3

VALUE	WHAT IT DOES
3 (default)	The three-stage TVD Runge-Kutta (SSP-RK3), third order in physical time.
`rk third order tvd`	The same as 3.

Guidance: Choosing a time-marching scheme › Global time stepping (p. 240)

convergence control

Convergence pacing for the inner pseudo-time loop (required for steady/DTS)

KEY	TYPE	DEFAULT	DESCRIPTION
`cycles`	int, > 0	required	Number of convergence cycles (steady) or max pseudo-time cycles per step (unsteady)
`inner convergence` (p. 58)	settings block		Let the solver decide how many inner (pseudo-time) cycles each real time step needs, with 'cycles' acting as an upper bound.
`scheme` (p. 61)	choice of settings blocks		*compressible / incompressible only* · Pseudo-time/Convergence integration scheme
`cfl` (p. 67)	settings block		*compressible only* · CFL configuration.
`viscosity ramp` (p. 72)	settings block		*incompressible only* · Scales the effective viscosity, decaying log-linearly from initial to 1, lowering the effective Reynolds number early so the start is more diffusive. initial <= 1 disables it
`relaxation ramp` (p. 73)	settings block		*incompressible only* · Scales the under-relaxation factors (alpha_u, alpha_p, turbulence relax), rising linearly from initial to 1, so the solver starts more relaxed. initial >= 1 (the default) disables it

convergence control

```
parameters = {
    "solver": {
        "convergence control": {
            "cycles": <int>,
            "inner convergence": {...},
            "scheme": {...},
            "cfl": {...},
            "viscosity ramp": {...},
            "relaxation ramp": {...},
        },
        ...
    },
    ...
}
```

`cycles`

Number of convergence cycles (steady) or max pseudo-time cycles per step (unsteady)

Type int, > 0 · **Required** yes

Guidance: Choosing a time-marching scheme > Cycles (p. 241)

inner convergence

Let the solver decide how many inner (pseudo-time) cycles each real time step needs, with 'cycles' acting as an upper bound. Unset runs the full 'cycles' every real time step.

KEY	TYPE	DEFAULT	DESCRIPTION
`target orders`	float	`1`	Orders of magnitude the worst selected residual measure must fall within the current real time step before it is called converged: the InnerRatio diagnostic for the compressible solver, the reported residual itself (InnerDrop) for the incompressible solver.
`target ratio`	float, >= 0.0	not set	Absolute floor on the InnerRatio diagnostic itself, ORed with 'target_orders'.
`min cycles`	int, >= 1	`5`	Inner cycles that always run before the criterion is tested.
`check frequency`	int, >= 1	`5`	Inner cycles between evaluations of the criterion.
`variables`	choice or list of str	`mean flow`	Residuals the criterion is judged on: 'mean flow', 'all' (mean flow plus the turbulence transport equations), or an explicit list of residual names.
`stall window`	int, >= 0	`10`	Give up when the reduction stops improving over this many checks: limiter switching can floor a residual norm no number of cycles will get below. 0 disables stall detection.
`stall tolerance`	float	`0.05`	Orders that must be gained across 'stall_window' checks for the inner iteration to count as still making progress.
`start real time step`	int, >= 0	`0`	Real time step from which the criterion may fire.

inner convergence

```
parameters = {
    "solver": {
        "convergence control": {
            "inner convergence": {
                "target orders": 1.0,
                "target ratio": <float>,
                "min cycles": 5,
                "check frequency": 5,
                "variables": "mean flow",
                "stall window": 10,
                "stall tolerance": 0.05,
                "start real time step": 0,
            },
```

```
            ...
        },
        ...
    },
    ...
}
```

target orders

Orders of magnitude the worst selected residual measure must fall within the current real time step before it is called converged: the InnerRatio diagnostic for the compressible solver, the reported residual itself (InnerDrop) for the incompressible solver. Scale free in both, so one value carries between cases. 0 or less disables this test.

Type float · **Default** 1

Guidance: Choosing a time-marching scheme › Choosing a target (p. 249)

target ratio

Absolute floor on the InnerRatio diagnostic itself, ORed with 'target_orders'. Compressible solver only. Not portable between cases; measure it from a run first. Unset or 0 disables this test.

Type float, >= 0.0 · **Default** not set · **Applies when** solver.solver settings.type is one of unset, 'compressible'

Guidance: Choosing a time-marching scheme › Choosing a target (p. 249)

min cycles

Inner cycles that always run before the criterion is tested.

Type int, >= 1 · **Default** 5

Guidance: Choosing a time-marching scheme › Choosing a target (p. 249)

check frequency

Inner cycles between evaluations of the criterion. A check cycle is also a reporting cycle: the criterion is judged on the reported residuals.

Type int, >= 1 · **Default** 5

variables

Residuals the criterion is judged on: 'mean flow', 'all' (mean flow plus the turbulence transport equations), or an explicit list of residual names. What 'mean flow' selects follows the solver: compressible takes continuity, momentum and energy (rho, rhoV[0..2], rhoE); the incompressible coupled scheme takes momentum and pressure (rhoV[0..2], p) — pressure being the continuity

constraint that governs its outer iteration; SIMPLE takes momentum alone, its reported pressure being the pressure-correction solve's own residual rather than a flow one. The incompressible solver reports no energy residual, so an energy-solving case cannot select one.

Type choice or list of str · **Default** `mean flow`

VALUE	WHAT IT DOES
`mean flow` (default)	Density, momentum and energy for the compressible solver, momentum and pressure for the incompressible coupled scheme, and momentum alone for SIMPLE.
`all`	Adds the turbulence transport equations.

Guidance: Choosing a time-marching scheme › The incompressible measure (p. 249)

`start real time step`

Real time step from which the criterion may fire. A solution restarted from a steady state and left to develop — RANS into DES — looks converged per real time step while the unsteadiness is still small, so an early exit through that phase would cut the inner iteration short exactly while the instability that has to grow is being resolved.

Type int, >= 0 · **Default** `0`

Guidance: Choosing a time-marching scheme › Choosing a target (p. 249)

scheme

Pseudo-time/Convergence integration scheme

KEY	TYPE	DEFAULT	DESCRIPTION
`name`	choice	not set	Pseudo-time integration scheme: euler = first-order explicit
`stage`	choice	not set	*euler / runge kutta / implicit euler only* · Has no effect under 'euler' (single-stage by construction).
`sub iterations`	int, > 0	not set	*lu-sgs only* · Number of LU-SGS/DP-LUR sweeps per pseudo-time step
`over relaxation`	float, > 0.0	not set	*lu-sgs only* · Diagonal over-relaxation factor for the LU-SGS/DP-LUR sweeps
`turbulence solver`	choice	not set	*lu-sgs / mf-gmres only* · Linear solver for the turbulence equations under LU-SGS.
`krylov subspace`	int, > 0	not set	*mf-gmres only* · Restart length m.
`linear tolerance`	float, > 0.0	not set	*mf-gmres only* · Relative residual drop required of the linear solve each implicit step
`max iterations`	int, > 0	not set	*mf-gmres only* · Hard cap on total Krylov iterations (across restarts) per implicit step
`preconditioner`	choice	not set	*mf-gmres only* · 'block jacobi' inverts the exact analytic 5x5 block diagonal.
`preconditioner sweeps`	int, > 0	not set	*mf-gmres only* · Symmetric Gauss-Seidel sweeps per preconditioner application (lu-sgs only)
`orthogonalisation`	choice	not set	*mf-gmres only* · How the Krylov basis is orthogonalised.
`preconditioner over relaxation`	float, > 0.0	not set	*mf-gmres only* · Diagonal over-relaxation for the lu-sgs preconditioner sweeps.
`turbulence sweeps`	int, > 0	not set	*mf-gmres only* · Sweeps for the point-implicit turbulence solve.
`flexible`	bool	not set	*mf-gmres only* · Use FGMRES instead of right-preconditioned GMRES.
`implicit relax`	float, 0.1 to 1.0	`0.5`	*coupled / simple only* · Under-relaxation factor (alpha_u) on the implicit momentum, pressure and energy updates

KEY	TYPE	DEFAULT	DESCRIPTION
`turbulence relax`	float, > 0 and <= 1.0	`0.5`	*coupled / simple only* · Under-relaxation factor on the turbulence transport update.
`pin pressure`	bool	not set	*coupled / simple only* · Pin the pressure at a reference cell with a large diagonal instead of the default constant-null-space + mean-removal gauge fix.
`pressure relax`	float, > 0.0 and <= 1.0	not set	*simple only* · Explicit pressure under-relaxation factor (alpha_p) for p += alpha_p * p'.
`pressure regularisation`	float, >= 0.0	not set	*simple only* · Diagonal regularisation (diag += beta*Idiagl) of the pressure-correction Poisson operator.
`pressure correction limit`	float, > 0.0	not set	*simple only* · Limit factor applied to the pressure correction

scheme

```
parameters = {
    "solver": {
        "convergence control": {
            "scheme": {
                "name": <choice>,
                "stage": <choice>,
                "sub iterations": <int>,
                "over relaxation": <float>,
                "turbulence solver": <choice>,
                "krylov subspace": <int>,
                "linear tolerance": <float>,
                "max iterations": <int>,
                "preconditioner": <choice>,
                "preconditioner sweeps": <int>,
                "orthogonalisation": <choice>,
                "preconditioner over relaxation": <float>,
                "turbulence sweeps": <int>,
                "flexible": <bool>,
                "implicit relax": 0.5,
                "turbulence relax": 0.5,
                "pin pressure": <bool>,
                "pressure relax": <float>,
                "pressure regularisation": <float>,
                "pressure correction limit": <float>,
            },
            ...
        },
        ...
    },
    ...
}
```

name

Pseudo-time integration scheme: euler = first-order explicit

Type choice · **Default** not set

VALUE	WHAT IT DOES	WHEN TO USE IT
`euler` *compressible only*	Explicit forward Euler in pseudo-time: one residual evaluation and update per cycle.	The `"euler"` scheme has a single stage and is the safest option for new or problematic cases.
`runge kutta` *compressible only*	Explicit multistage Runge-Kutta in pseudo-time. `stage` selects the number of stages and their coefficients.	
`implicit euler` *compressible only*	The default implicit scheme, solving the full linearised system with the Petsc or AMGX linear solver each pseudo-time step.	
`lu-sgs` *compressible only*	A matrix-free, point-implicit LU-SGS/DP-LUR scheme. It avoids assembling and inverting the full linear system, trading some convergence rate for lower memory use and, often, faster wall-clock time per cycle than `"implicit euler"`.	
`mf-gmres` *compressible only*	Restarted GMRES that applies the exact analytic flux Jacobian matrix-free, with no assembled mean-flow matrix.	Not available with the SAS turbulence model or the SAS hybrid.
`coupled` *incompressible only*	Solves a single block-4 [u,v,w,p] system per pseudo-time step.	
`simple` *incompressible only*	Uses a segregated SIMPLE momentum predictor followed by a pressure correction.	

Guidance: Choosing a time-marching scheme › name (p. 242)

stage

Has no effect under 'euler' (single-stage by construction). Accepted for backward compatibility with decks that carry this key as boilerplate; prefer omitting it.

Type choice · **Default** not set

VALUE	WHAT IT DOES
`1`	Under `runge kutta`, a single stage, the same update as `euler`. Other schemes ignore `stage`.
`3`	Under `runge kutta`, the three-stage TVD Runge-Kutta (SSP-RK3). Other schemes ignore `stage`.
`4`	Under `runge kutta`, a four-stage scheme with stage coefficients 1/4, 1/2, 0.55 and 1. Other schemes ignore `stage`.

VALUE	WHAT IT DOES
`5`	Under `runge kutta`, a five-stage scheme with stage coefficients 0.0695, 0.1602, 0.2898, 0.506 and 1. Other schemes ignore `stage`.
`rk third order tvd`	The same as `3`.

`turbulence solver`

Linear solver for the turbulence equations under LU-SGS. 'amg' keeps the assembled AMG solve; 'point-implicit' uses the matrix-free sweep. The C++ default when unset is 'point-implicit'.

Type choice · **Default** not set

VALUE	WHAT IT DOES
`amg`	Keeps the assembled AMG solve for the turbulence equations.
`point-implicit`	Uses the matrix-free sweep instead.

`krylov subspace`

Restart length m. The solver stores m+1 Krylov vectors, or 2m+1 with flexible, so this is the dominant storage cost; size it from the observed iteration count rather than padding it.

Type int, > 0 · **Default** not set

Guidance: Choosing a time-marching scheme › mf-gmres (p. 243)

`preconditioner`

'block jacobi' inverts the exact analytic 5x5 block diagonal. 'lu-sgs' uses linearised symmetric Gauss-Seidel sweeps. 'lu-sgs nonlinear' uses the stock unlinearised sweep and REQUIRES flexible=true, since it is not a linear operator and plain GMRES cannot converge with it.

Type choice · **Default** not set · **Applies when** flexible is True

VALUE	WHAT IT DOES	WHEN TO USE IT
`block jacobi`	Inverts the exact analytic 5x5 block diagonal. The same operator on the host and on a GPU.	
`lu-sgs`	Linearised symmetric Gauss-Seidel sweeps.	
`lu-sgs nonlinear`	The unlinearised LU-SGS sweep.	Requires `flexible: True`: the sweep is not a linear operator, and plain GMRES cannot converge with it.
`none`	No preconditioner.	

Guidance: Choosing a time-marching scheme › mf-gmres (p. 243)

orthogonalisation

How the Krylov basis is orthogonalised. 'mgs' (default) is modified Gram-Schmidt, matching AMGX's gmres/fgmres and so directly comparable with the implicit euler benchmark. 'cgs2' is classical Gram-Schmidt with one reorthogonalisation pass and batched reductions: it reduces the reduction round trips per restart cycle from O(m^2) to O(m) at the cost of twice the vector work, which is only expected to pay at high rank counts where each reduction is a real collective.

Type choice · **Default** not set

VALUE	WHAT IT DOES	WHEN TO USE IT
`mgs`	Modified Gram-Schmidt.	The default.
`cgs2`	Classical Gram-Schmidt with one reorthogonalisation pass and batched reductions: fewer reductions per restart cycle, at twice the vector work.	Only expected to pay at high rank counts, where each reduction is a real collective.

preconditioner over relaxation

Diagonal over-relaxation for the lu-sgs preconditioner sweeps. Named separately from the lu-sgs solver's own 'over relaxation' because here it tunes a PRECONDITIONER, not the solve.

Type float, > 0.0 · **Default** not set

turbulence sweeps

Sweeps for the point-implicit turbulence solve. Deliberately NOT called 'sub iterations': that name belongs to the lu-sgs solver, and confusing it with 'preconditioner sweeps' (which tunes the MEAN-FLOW preconditioner) would be easy.

Type int, > 0 · **Default** not set

flexible

Use FGMRES instead of right-preconditioned GMRES. Permits a preconditioner that varies between iterations, at the cost of a second set of m Krylov vectors.

Type bool · **Default** not set

Guidance: Choosing a time-marching scheme > mf-gmres (p. 243)

turbulence relax

Under-relaxation factor on the turbulence transport update. Has no effect without a turbulence model (laminar viscous, non-hybrid LES)

Type float, > 0 and <= 1.0 · **Default** `0.5`

pin pressure

Pin the pressure at a reference cell with a large diagonal instead of the default constant-null-space + mean-removal gauge fix. Needed on a fully closed domain (no fixed-pressure boundary)

Type bool · **Default** not set

pressure relax

Explicit pressure under-relaxation factor (alpha_p) for p += alpha_p * p'. The solver default is 0.3

Type float, > 0.0 and <= 1.0 · **Default** not set

pressure regularisation

Diagonal regularisation (diag += beta*Idiagl) of the pressure-correction Poisson operator. Vanishes at convergence, so the steady solution is unbiased

Type float, >= 0.0 · **Default** not set

cfl

CFL configuration. A bare number is shorthand for the working CFL: `cfl: 2.0` means `cfl: {cfl: 2.0}`.

KEY	TYPE	DEFAULT	DESCRIPTION
`cfl`	float, > 0	required	CFL number
`cfl turbulence`	float, > 0	not set	Working CFL for the turbulence transport equations (the conserved-variable slots beyond the 5 mean-flow equations — SA: 1, SST: 2).
`cfl coarse`	float, > 0	not set	CFL number for multigrid coarse levels
`cfl ramp` (p. 69)	list of settings blocks		How the CFL climbs toward 'cfl'.
`cfl viscous factor`	float, > 0	not set	CFL viscous factor

cfl

```
parameters = {
    "solver": {
        "convergence control": {
            "cfl": {
                "cfl": <float>,
                "cfl turbulence": <float>,
                "cfl coarse": <float>,
                "cfl ramp": [...],
                "cfl viscous factor": <float>,
            },
            ...
        },
        ...
    },
    ...
}
```

`cfl`

CFL number

Type float, > 0 · **Required** yes

Guidance: Choosing a time-marching scheme › cfl (p. 245)

`cfl turbulence`

Working CFL for the turbulence transport equations (the conserved-variable slots beyond the 5 mean-flow equations — SA: 1, SST: 2). The solver uses it directly as the pseudo-time CFL for those equations, not as a cap. Has no effect when no turbulence model is active (euler, laminar viscous, non-hybrid LES).

Type float, > 0 · **Default** not set

Guidance: Choosing a time-marching scheme › cfl_turbulence (p. 245)

`cfl coarse`

CFL number for multigrid coarse levels

Type float, > 0 · **Default** not set · **Applies when** solver.numerical scheme.multigrid is set

Guidance: Choosing a time-marching scheme › cfl_coarse (p. 245)

`cfl viscous factor`

CFL viscous factor

Type float, > 0 · **Default** not set

Guidance: Choosing a time-marching scheme › cfl_viscous_factor (p. 247)

cfl ramp

How the CFL climbs toward 'cfl'. A list of ramps, each naming its shape — one entry for a single ramp, several to run in order. Alternatively a function, called as f(*, cycle, current_cfl); it returns the new CFL as a number, or a dict of any of 'cfl', 'cfl_turbulence' and 'cfl_coarse'. Keys a returned dict leaves out keep their static value.

KEY	TYPE	DEFAULT	DESCRIPTION
`initial`	float, > 0	not set	CFL this ramp starts from, returned at its own start cycle.
`start cycle`	int, >= 1	`1`	First solve cycle this ramp acts on.
`end cycle`	int, >= 1	not set	Last solve cycle this ramp acts on.
`type`	choice	required	How the ramp changes the CFL in its window: `exponential` multiplies it, `linear` and `stepped_linear` add to it.
`factor`	float, >= 1.0	required	*exponential only* · Multiplied into the CFL every cycle.
`max allowed`	float, > 0	`1000000`	*exponential only* · Ceiling on the ramp itself, guarding against overflow.
`increment`	float	required	*linear / stepped_linear only* · Added to the CFL every cycle in the window.
`step cycles`	int, > 0	required	*stepped_linear only* · Cycles between steps.

```
cfl ramp

parameters = {
    "solver": {
        "convergence control": {
            "cfl": {
                "cfl ramp": [
                    {
                        "initial": <float>,
                        "start cycle": 1,
                        "end cycle": <int>,
                        "type": <choice>,
                        "factor": <float>,
                        "max allowed": 1000000.0,
                        "increment": <float>,
                        "step cycles": <int>,
                    },
                ],
                ...
            },
```

```
            ...
        },
        ...
    },
    ...
}
```

initial

CFL this ramp starts from, returned at its own start cycle. Leave unset to carry on from wherever the previous ramp left off.

Type float, > 0 · **Default** not set

Guidance: Choosing a time-marching scheme › cfl_ramp (p. 246)

start cycle

First solve cycle this ramp acts on.

Type int, >= 1 · **Default** 1

Guidance: Choosing a time-marching scheme › cfl_ramp (p. 246)

end cycle

Last solve cycle this ramp acts on. Unset runs to the end of the run.

Type int, >= 1 · **Default** not set

Guidance: Choosing a time-marching scheme › cfl_ramp (p. 246)

type

How the ramp changes the CFL in its window: `exponential` multiplies it, `linear` and `stepped_linear` add to it.

Type choice · **Required** yes

VALUE	WHAT IT DOES
`exponential`	Multiplies the CFL by `factor` (>= 1.0) each cycle, capped at `max_allowed`.
`linear`	Adds `increment` to the CFL every cycle between `start_cycle` and `end_cycle`.
`stepped_linear`	Adds `increment` every `step_cycles` cycles.

Guidance: Choosing a time-marching scheme › cfl_ramp (p. 246)

`factor`

Multiplied into the CFL every cycle. Must be at least 1.

Type float, >= 1.0 · **Required** yes

`increment`

Added to the CFL every cycle in the window. An increment of 0 holds the CFL steady.

Type float · **Required** yes

viscosity ramp

Scales the effective viscosity, decaying log-linearly from initial to 1, lowering the effective Reynolds number early so the start is more diffusive. initial <= 1 disables it

KEY	TYPE	DEFAULT	DESCRIPTION
initial	float, > 0	required	Ramp value on the first pseudo cycle; the ramp eases to 1 from here
cycles	float, > 0	300	Number of pseudo cycles over which the ramp reaches 1

viscosity ramp

```
parameters = {
    "solver": {
        "convergence control": {
            "viscosity ramp": {
                "initial": <float>,
                "cycles": 300.0,
            },
            ...
        },
        ...
    },
    ...
}
```

initial

Ramp value on the first pseudo cycle; the ramp eases to 1 from here

Type float, > 0 · **Required** yes

Guidance: Choosing a time-marching scheme › Startup ramps (p. 252)

cycles

Number of pseudo cycles over which the ramp reaches 1

Type float, > 0 · **Default** 300

Guidance: Choosing a time-marching scheme › Startup ramps (p. 252)

relaxation ramp

Scales the under-relaxation factors (alpha_u, alpha_p, turbulence relax), rising linearly from initial to 1, so the solver starts more relaxed. initial >= 1 (the default) disables it

KEY	TYPE	DEFAULT	DESCRIPTION
initial	float, > 0	required	Ramp value on the first pseudo cycle; the ramp eases to 1 from here
cycles	float, > 0	300	Number of pseudo cycles over which the ramp reaches 1

relaxation ramp

```
parameters = {
    "solver": {
        "convergence control": {
            "relaxation ramp": {
                "initial": <float>,
                "cycles": 300.0,
            },
            ...
        },
        ...
    },
    ...
}
```

initial

Ramp value on the first pseudo cycle; the ramp eases to 1 from here

Type float, > 0 · **Required** yes

Guidance: Choosing a time-marching scheme > Startup ramps (p. 252)

cycles

Number of pseudo cycles over which the ramp reaches 1

Type float, > 0 · **Default** 300

Guidance: Choosing a time-marching scheme > Startup ramps (p. 252)

numerical scheme

Default spatial numerical scheme for all meshes.

KEY	TYPE	DEFAULT	DESCRIPTION
`order`	choice	`second`	Spatial order of accuracy
`limiter`	choice	`van albada`	Slope limiter for higher-order schemes
`limiter coefficient`	float, >= 0.0	`5`	K in the grid-scaled smoothness threshold eps^2 = (K h / reference_ length)^3 used by the 'michalak gooch' and 'venkatakrishnan' cell-based limiters to deactivate limiting in smooth flow.
`first order cycles`	int, >= 0	`0`	Number of first-order cycles before switching to higher order
`freeze limiter cycle`	int	`-1`	Cycle at which to freeze the limiter (-1 = never freeze)
`gradient`	choice	`green-gauss`	Gradient calculation method (green-gauss or least-squares)
`reconstruction`	choice	`muscl`	Reconstruction method for MUSCL-type schemes
`permit backward difference during reconstruction`	bool	`False`	Allow backward difference stencils during MUSCL reconstruction
`inviscid flux scheme`	choice	`roe`	Inviscid flux calculation scheme
`linear gradients`	bool	`False`	*compressible only* · Use an unweighted (linear) Green-Gauss gradient instead of the default inverse-distance-weighted one.
`multigrid` (p. 81)	settings block		*compressible only* · Multigrid acceleration for explicit schemes
`preconditioner` (p. 83)	settings block		*compressible only* · Low Mach number preconditioner settings
`entropy fix`	float, 0 to 0.2	`0`	*compressible only* · Entropy fix coefficient for Roe scheme (limits maximum dissipation)
`roe low dissipation sensor`	choice	`NONE`	*compressible only* · Sensor type for Roe low-dissipation scheme
`roe entropy fix coefficient`	float, >= 0	`0.001`	*compressible only* · Roe entropy fix coefficient for low-dissipation scheme

KEY	TYPE	DEFAULT	DESCRIPTION
`roe dissipation sensor minimum`	float, 0.05 to 1.0	not set	*compressible only* · Minimum value for Roe dissipation sensor (prevents excessive dissipation reduction)
`roe low dissipation sensor parameter`	choice	not set	*compressible only* · Field used to compute the Roe low-dissipation sensor
`viscous time step factor`	float, > 0	`1`	*viscous / rans / les only* · Time step scaling for viscous terms
`turbulence limiter`	choice	not set	*rans only* · Treatment of invalid turbulence variables after the solution update: 'zero' clamps them to their floor, 'no update' reverts the cell's turbulence state (default when unset)

numerical scheme

```
parameters = {
    "solver": {
        "numerical scheme": {
            "order": "second",
            "limiter": "van albada",
            "limiter coefficient": 5.0,
            "first order cycles": 0,
            "freeze limiter cycle": -1,
            "gradient": "green-gauss",
            "reconstruction": "muscl",
            "permit backward difference during reconstruction": False,
            "inviscid flux scheme": "roe",
            "linear gradients": False,
            "multigrid": {...},
            "preconditioner": {...},
            "entropy fix": 0.0,
            "roe low dissipation sensor": "NONE",
            "roe entropy fix coefficient": 0.001,
            "roe dissipation sensor minimum": <float>,
            "roe low dissipation sensor parameter": <choice>,
            "viscous time step factor": 1.0,
            "turbulence limiter": <choice>,
        },
        ...
    },
    ...
}
```

`order`

Spatial order of accuracy

Type choice · **Default** `second`

VALUE	WHAT IT DOES	WHEN TO USE IT
`first`	First-order upwind: face values are the upwind cell values, with no reconstruction.	
`second` (default)	Second order. The compressible solver reconstructs every variable to the face with the limited scheme set by `reconstruction`; the incompressible solver uses linear-upwind interpolation for momentum.	The default.
`euler_second`	The compressible solver reconstructs the mean-flow variables to second order and keeps the turbulence variables first order. The incompressible solver blends 75% linear and 25% linear-upwind interpolation for momentum.	
`central` *incompressible only*	Incompressible solver only. Momentum is interpolated by central differencing blended with linear-upwind, in the proportion set by `central_blend`.	

Guidance: Choosing a spatial scheme › Spatial order (p. 254)

`limiter`

Slope limiter for higher-order schemes

Type choice · **Default** `van albada`

VALUE	WHAT IT DOES	WHEN TO USE IT
`van albada` (default)	Default. One-dimensional: $\psi = \max\left(0, \frac{2ab + \varepsilon^2}{a^2 + b^2 + \varepsilon^2}\right)$. Smooth, symmetric, bounded in $[0, 1]$.	The recommended general purpose choice.
`van leer`	One-dimensional: $\psi = \max\left(0, \frac{4ab + \varepsilon^2}{(a+b)^2 + \varepsilon^2}\right)$. Slightly more compressive than van Albada.	
`symmetric ratio`	One-dimensional: $\psi = \max\left(0, \min\left(1, \frac{4\min(\sigma a, \sigma b) + \varepsilon}{\|a + b\| + \varepsilon}\right)\right)$ with $\sigma = \text{sign}(a + b)$. The least dissipative of the three.	
`none`	$\psi = 1$. Unlimited reconstruction: second order everywhere, but not monotonicity preserving.	Appropriate only for smooth flows with no shocks or strong gradients.

VALUE	WHAT IT DOES	WHEN TO USE IT
`barth jespersen`	Cell-based: $P(y) = \min(1, y)$. Barth & Jespersen (1989) as published. Bound preserving, but only C^0: the kink at $y = 1$ makes the limiter value flicker between cycles and can stall the residual. It has no smoothness deactivation, so it limits wherever the bound binds. ::: {.warning} `"barth jespersen"` is the multidimensional, cell-based Barth & Jespersen limiter. It is not the one-dimensional `"symmetric ratio"` function, which takes no minimum or maximum over the cell's neighbours. A deck whose results depend on the one-dimensional behaviour must set `"symmetric ratio"`; `"barth jespersen"` gives different results. See the Unreleased (p. 501) release notes for when this changed. :::	
`michalak gooch`	Cell-based: $P(y) = y - \frac{4}{27}y^3$ for $y < 3/2$, else 1. Michalak & Ollivier-Gooch. C^1, and $P(y) \leq \min(1, y)$, so never less monotonicity preserving than Barth-Jespersen. Deactivates in smooth flow via a grid-scaled threshold, so formal order survives refinement.	The recommended choice.
`venkatakrishnan`	Cell-based: $P(y) = \min\left(1, (y^2 + 2y)/(y^2 + y + 2)\right)$. Smooth, but exceeds 1 for $y > 2$, so it needs the clip, which reintroduces a C^0 kink at $y = 2$.	Provided as a familiar cross-check.

Guidance: Choosing a limiter (p. 258)

`limiter coefficient`

K in the grid-scaled smoothness threshold eps^2 = (K h / reference_length)^3 used by the 'michalak gooch' and 'venkatakrishnan' cell-based limiters to deactivate limiting in smooth flow. Larger values treat more of the flow as smooth, so limit less. Ignored by the other limiters.

Type float, >= 0.0 · **Default** 5

`freeze limiter cycle`

Cycle at which to freeze the limiter (-1 = never freeze)

Type int · **Default** -1

Guidance: Troubleshooting › Limiter switching (p. 356)

gradient

Gradient calculation method (green-gauss or least-squares)

Type choice · **Default** `green-gauss`

VALUE	WHAT IT DOES	WHEN TO USE IT
`green-gauss` (default)	Green-Gauss gradients from face values interpolated by inverse distance, or by a plain average when `linear_gradients` is set.	The default.
`least-squares`	A least-squares fit to the differences between each cell and its face neighbours, weighted by face area.	

reconstruction

Reconstruction method for MUSCL-type schemes

Type choice · **Default** `muscl`

VALUE	WHAT IT DOES	WHEN TO USE IT
`muscl` (default)	The MUSCL κ-scheme with $\kappa = 1/3$, blending the limited cell-gradient extrapolation with the central difference across the face.	The default.
`umuscl`	The unstructured MUSCL of Burg, $u_f = u_L + (1-\kappa)\,\mathbf{r}\cdot\nabla u_L + \frac{\kappa}{2}(u_R - u_L)$ with $\kappa = 1/3$, scaled by the limiter.	
`lpumuscl`	Accepted, but the solver currently treats it as `muscl`.	

inviscid flux scheme

Inviscid flux calculation scheme

Type choice · **Default** `roe`

VALUE	WHAT IT DOES	WHEN TO USE IT
`roe` (default)	Roe's approximate Riemann solver. `entropy_fix` floors each wave speed at that fraction of the largest. Read by the compressible solver only.	The default.
`hllc`	The HLLC approximate Riemann solver, which restores the contact wave missing from HLL.	
`roe low-diss`	The Roe flux with its upwind dissipation scaled by a factor between `roe_dissipation_sensor_minimum` and 1, set per face by `roe_low_dissipation_sensor`.	Scale-resolving simulations, where full upwind dissipation would damp the resolved turbulence. With the sensor `NONE` it is the same as `roe`.
`hllc low-diss`	The HLLC flux with its dissipation scaled in the same way as `roe low-diss`.	
`rusanov`	The Rusanov (local Lax-Friedrichs) flux, which scales the dissipation by the largest local wave speed.	The most dissipative of the schemes.

Guidance: Choosing a spatial scheme › Inviscid flux scheme (p. 256)

linear gradients

Use an unweighted (linear) Green-Gauss gradient instead of the default inverse-distance-weighted one. Ignored when 'gradient' is 'least-squares' — the least-squares branch never reads the flag.

Type bool · **Default** `False`

roe low dissipation sensor

Sensor type for Roe low-dissipation scheme

Type choice · **Default** `NONE`

VALUE	WHAT IT DOES	WHEN TO USE IT
`FD`	Scales the dissipation by $1 - f_d$, where f_d is the DDES shielding function of the turbulence model: full dissipation in RANS regions, the minimum in LES regions. Faces the shock sensor flags keep full dissipation.	With a DES-family hybrid. Without a RANS turbulence model every cell is treated as LES.
`NTS`	The blending function of Xiao et al. (2012), built from vorticity, strain rate, the turbulent length scale and the cell size, and combined with the shock sensor.	
`DES`	Scales the dissipation by the turbulence model's length-scale ratio, capped at 1, which is near 0 in LES regions and 1 in RANS regions. Faces the shock sensor flags keep full dissipation.	With a DES-family hybrid. Without a RANS turbulence model every cell is treated as LES.
`NONE` (default)	No scaling: the low-dissipation fluxes apply full upwind dissipation.	The default.

roe low dissipation sensor parameter

Field used to compute the Roe low-dissipation sensor

Type choice · **Default** not set

VALUE	WHAT IT DOES	WHEN TO USE IT
`density`	The shock sensor is the normalised density jump between each cell and its face neighbours.	The default when unset.
`velocity`	The shock sensor is the Ducros ratio $\|\nabla \cdot \mathbf{u}\| / (\|\nabla \cdot \mathbf{u}\| + \|\omega\|)$.	
`velocity divergence/vorticity`	The same as `velocity`.	

turbulence limiter

Treatment of invalid turbulence variables after the solution update: 'zero' clamps them to their floor, 'no update' reverts the cell's turbulence state (default when unset)

Type choice · **Default** not set

VALUE	WHAT IT DOES	WHEN TO USE IT
`zero`	When an update leaves a cell's turbulence variables invalid, they are clipped to their lower limits.	
`no update`	When an update leaves a cell's turbulence variables invalid, the cell keeps its turbulence values from before the update.	The default when unset.

multigrid

Multigrid acceleration for explicit schemes

KEY	TYPE	DEFAULT	DESCRIPTION
`levels`	int, 1 to 10	required	Number of multigrid levels
`cycles`	int, >= 1	`100000000`	Number of multigrid cycles
`multigrid ramp`	int, >= 0	not set	Solve cycle after which the coarse multigrid levels are used
`prolong factor`	float	not set	Prolongation factor
`prolong transport factor`	float	not set	Prolongation factor for the turbulence/transport equations

multigrid

```
parameters = {
    "solver": {
        "numerical scheme": {
            "multigrid": {
                "levels": <int>,
                "cycles": 100000000,
                "multigrid ramp": <int>,
                "prolong factor": <float>,
                "prolong transport factor": <float>,
            },
            ...
        },
        ...
    },
    ...
}
```

`levels`

Number of multigrid levels

Type int, 1 to 10 · **Required** yes

Guidance: Choosing a time-marching scheme › Multigrid (p. 252)

`cycles`

Number of multigrid cycles

Type int, >= 1 · **Default** `100000000`

Guidance: Choosing a time-marching scheme > Multigrid (p. 252)

preconditioner

Low Mach number preconditioner settings

KEY	TYPE	DEFAULT	DESCRIPTION
`minimum mach number`	float, > 0.0	required	Cut-off Mach number for low-Mach preconditioning: floors the preconditioning reference velocity so it cannot collapse at a stagnation point.

preconditioner

```
parameters = {
    "solver": {
        "numerical scheme": {
            "preconditioner": {
                "minimum mach number": <float>,
            },
            ...
        },
        ...
    },
    ...
}
```

`minimum mach number`

Cut-off Mach number for low-Mach preconditioning: floors the preconditioning reference velocity so it cannot collapse at a stagnation point. Overrides the Mach-dependent default.

Type float, > 0.0 · **Required** yes

output settings

Default output variables and monitors for all meshes.

KEY	TYPE	DEFAULT	DESCRIPTION
`solution` (p. 85)	settings block		Solution file output settings
`report` (p. 90)	settings block		Report and monitoring settings
`compute average and rms`	bool	`False`	Compute time-averaged and RMS statistics
`average start time cycle`	int, >= 0	`0`	Cycle to start averaging
`output directory`	str	not set	Output directory path

output settings

```
parameters = {
    "solver": {
        "output settings": {
            "solution": {...},
            "report": {...},
            "compute average and rms": False,
            "average start time cycle": 0,
            "output directory": <str>,
        },
        ...
    },
    ...
}
```

solution

Solution file output settings

KEY	TYPE	DEFAULT	DESCRIPTION
`frequency` (p. 88)	settings block		Output frequency (every N cycles)
`format`	choice	`vtk`	Output file format
`surface variables`	list of str	not set	Variables to output on surfaces
`volume variables`	list of str	not set	Variables to output in volume
`scripts`	list of str	not set	ParaView Catalyst Python scripts, run on the volume solution each time it is written.
`fwh interpolate`	list of str	not set	List of stl files for FWH surface interpolation
`fwh wall data`	bool	`False`	Record impermeable FWH data on every wall: the pressure and, for viscous equations, the wall shear stress.
`write restart`	bool	`True`	Write restart files
`binary`	bool	`True`	Use binary format
`precision`	choice	`single`	Output precision
`global node ids`	bool	`True`	Retain the local→global node map so VTKHDF output can stitch partitions together without a clean-to-grid filter
`output solver cycle`	bool	`False`	Stamp VTKHDF output steps with the solver cycle number instead of physical time.
`ghost cells`	bool	`True`	Emit a one-cell-deep ghost layer in the VTKHDF volume output: the partition halo cells the solver already exchanges, marked DUPLICATECELL, so filters that interpolate cell data to points (isosurfaces, for instance) are correct across partition interfaces instead of showing a seam

solution

```
parameters = {
    "solver": {
        "output settings": {
            "solution": {
                "frequency": {...},
                "format": "vtk",
                "surface variables": [...],
                "volume variables": [...],
                "scripts": [...],
```

```
                "fwh interpolate": [...],
                "fwh wall data": False,
                "write restart": True,
                "binary": True,
                "precision": "single",
                "global node ids": True,
                "output solver cycle": False,
                "ghost cells": True,
            },
            ...
        },
        ...
    },
    ...
}
```

`format`

Output file format

Type choice · **Default** `vtk`

VALUE	WHAT IT DOES	WHEN TO USE IT
`none`	Outputs no files.	
`vtk` (default)	Outputs in VTK HDF format (.vtkhdf) for reading into ParaView. The .vtkhdf output format requires ParaView 6.1.1 or later.	
`ensight`	Writes the volume solution as EnSight files with VTK's EnSight writer, one set per output cycle. Surface output is still written as VTK HDF.	For post-processing in EnSight.
`fwensight`	Writes the same VTK HDF output as `vtk`, and in addition an EnSight Gold copy (`.encas` and `.geo` files) of the volume and of the boundaries.	
`native`	Writes each volume variable to its own HDF5 file per output cycle, with an index file of external links to them. Surface output is still written as VTK HDF.	

Guidance: Post-processing and visualisation (p. 335)

`scripts`

ParaView Catalyst Python scripts, run on the volume solution each time it is written. Paths are passed to Catalyst as written, so a relative path is taken from the directory the run is started in.

Type list of str · **Default** not set

`fwh interpolate`

List of stl files for FWH surface interpolation

Type list of str · **Default** not set

Guidance: Aeroacoustics (FWH) (p. 305)

`fwh wall data`

Record impermeable FWH data on every wall: the pressure and, for viscous equations, the wall shear stress. On a mesh that overset hole cutting can blank, each face is also flagged active or inactive, so a wall covered by another mesh does not radiate.

Type bool · **Default** `False`

Guidance: Aeroacoustics (FWH) (p. 305)

`write restart`

Write restart files

Type bool · **Default** `True`

Guidance: Choosing initialisation and reference state › Restart (p. 299)

`precision`

Output precision

Type choice · **Default** `single`

VALUE	WHAT IT DOES	WHEN TO USE IT
`single` (default)	32-bit floating-point field values. The solver writes field data at this precision.	The default.
`double`	Accepted, but not read by the solver: field data is written in single precision whatever is set.	

`output solver cycle`

Stamp VTKHDF output steps with the solver cycle number instead of physical time. Every step of a steady run carries the same physical time, so the default leaves the appended steps indistinguishable to a viewer; the cycle number makes them steppable

Type bool · **Default** `False`

frequency

Output frequency (every N cycles)

KEY	TYPE	DEFAULT	DESCRIPTION
surface data	int, >= 1	100000000	Frequency for surface variable output
volume data	int, >= 1	100000000	Frequency for volume variable output
volume interpolate	int, >= 1	100000000	Frequency for volume data interpolation
fwh interpolate	int, >= 1	100000000	Frequency for FWH surface interpolation
fwh wall data	int, >= 1	100000000	Frequency for FWH wall data output
checkpoint	int, >= 1	100000000	Frequency for writing checkpoint/restart files
volume data start	int, >= 0	1	Cycle to start volume data output
surface data start	int, >= 0	1	Cycle to start surface data output
volume interpolate start	int, >= 0	1	Cycle to start volume data interpolation
checkpoint start	int, >= 0	1	Cycle to start writing checkpoint/restart files
fwh interpolate start	int, >= 0	1	Cycle to start FWH surface interpolation
fwh wall data start	int, >= 0	1	Cycle to start FWH wall data output

frequency

```
parameters = {
    "solver": {
        "output settings": {
            "solution": {
                "frequency": {
                    "surface data": 100000000,
                    "volume data": 100000000,
                    "volume interpolate": 100000000,
                    "fwh interpolate": 100000000,
                    "fwh wall data": 100000000,
                    "checkpoint": 100000000,
                    "volume data start": 1,
                    "surface data start": 1,
                    "volume interpolate start": 1,
                    "checkpoint start": 1,
                    "fwh interpolate start": 1,
                    "fwh wall data start": 1,
                },
```

```
                ...
            },
            ...
        },
        ...
    },
    ...
}
```

report

Report and monitoring settings

KEY	TYPE	DEFAULT	DESCRIPTION
`frequency`	int, >= 1	required	Report frequency (every N cycles)
`variables`	list of str	not set	Variables to include in reports
`monitor` (p. 92)	named settings blocks		Monitor point configuration
`forces` (p. 93)	named settings blocks		Force calculation zones
`mass flow` (p. 95)	named settings blocks		Mass flow calculation zones
`scale residuals by volume`	bool	`False`	Scale residuals by cell volume in reports
`residual statistics`	bool	`True`	Add the infinity norm of the true residual, its RMS, the participation ratio Neff and the worst cell's share of the total to the report, four columns per equation.

report

```
parameters = {
    "solver": {
        "output settings": {
            "report": {
                "frequency": <int>,
                "variables": [...],
                "monitor": {...},
                "forces": {...},
                "mass flow": {...},
                "scale residuals by volume": False,
                "residual statistics": True,
            },
            ...
        },
        ...
    },
    ...
}
```

residual statistics

Add the infinity norm of the true residual, its RMS, the participation ratio Neff and the worst cell's share of the total to the report, four columns per equation. Neff is the number of cells the residual is effectively spread over, so a small value against a large mesh means the field is converged everywhere except a handful of cells; the resshare_* output variables show which. On by default; set it False to save an extra pass over the residual field and two MPI reductions per report.

Type bool · **Default** `True`

monitor

Monitor point configuration

KEY	TYPE	DEFAULT	DESCRIPTION
point	vector (3 floats)	required	Monitor point coordinates [x, y, z]
variables	list of str	required	Variables to monitor at this point
name	str	not set	Monitor point name

monitor

```
parameters = {
    "solver": {
        "output settings": {
            "report": {
                "monitor": {
                    "<name>": {
                        "point": <[x, y, z]>,
                        "variables": [...],
                        "name": <str>,
                    },
                },
                ...
            },
            ...
        },
        ...
    },
    ...
}
```

forces

Force calculation zones

KEY	TYPE	DEFAULT	DESCRIPTION
`zones`	list of zone ids	required	Zone IDs for force calculation
`name`	str	not set	Name of the force component
`reference area`	float, > 0	`1`	Reference area for force coefficients
`reference length`	float, > 0	`1`	Reference length for moment coefficients
`reference pressure`	float, >= 0	`0`	Reference pressure for coefficients
`transform`	Python function, or a Python function	not set	Function applied to each force and moment vector before it is reported, called as f(*, vector); it returns the transformed vector as three numbers
`reference point`	vector (3 floats)	`[0, 0, 0]`	Reference point for moment calculations
`dimensional`	bool	`False`	Report dimensional forces (not coefficients)
`inertial frame`	bool	`True`	Report forces in inertial frame

forces

```
parameters = {
    "solver": {
        "output settings": {
            "report": {
                "forces": {
                    "<name>": {
                        "zones": [...],
                        "name": <str>,
                        "reference area": 1.0,
                        "reference length": 1.0,
                        "reference pressure": 0.0,
                        "transform": <function>,
                        "reference point": [0.0, 0.0, 0.0],
                        "dimensional": False,
                        "inertial frame": True,
                    },
                },
                ...
            },
            ...
```

```
        },
        ...
    },
    ...
}
```

`transform`

Function applied to each force and moment vector before it is reported, called as f(*, vector); it returns the transformed vector as three numbers

Type Python function, or a Python function · **Default** not set

Guidance: Python functions in the deck > Transform Function (p. 219)

mass flow

Mass flow calculation zones

KEY	TYPE	DEFAULT	DESCRIPTION
zones	list of zone ids	required	Zone IDs for mass flow calculation

mass flow

```
parameters = {
    "solver": {
        "output settings": {
            "report": {
                "mass flow": {
                    "<name>": {
                        "zones": [...],
                    },
                },
                ...
            },
            ...
        },
        ...
    },
    ...
}
```

model

The meshes to solve, each under a name of the engineer's choosing. Each entry names its mesh file and carries that mesh's boundary conditions, fluid zones, motion and any overrides of the solver defaults. Several entries run together in one simulation, coupled through overset or sliding interfaces.

KEY	TYPE	DEFAULT	DESCRIPTION
`mesh`	str	required	Path to the HDF5 mesh file.
`boundary conditions` (p. 98)	settings block		Boundary conditions mapped to surface zones.
`fluid zones` (p. 143)	settings block		Fluid zone models (e.g. porous, actuator disc).
`transforms` (p. 169)	settings block		Placement and motion of the mesh.
`initialisation` (p. 47)	settings block		The initialisation settings that differ on this mesh. A mesh may set its own.
`convergence control` (p. 56)	choice of settings blocks		The convergence control settings that differ on this mesh. A mesh may set its own.
`numerical scheme` (p. 74)	choice of settings blocks		The numerical scheme settings that differ on this mesh. A mesh may set its own.
`output settings` (p. 84)	settings block		The output settings that differ on this mesh. A mesh may set its own.

model

```
parameters = {
    "model": {
        "<mesh>": {
            "mesh": <str>,
            "boundary conditions": {...},
            "fluid zones": {...},
            "transforms": {...},
            "initialisation": {...},
            "convergence control": {...},
            "numerical scheme": {...},
            "output settings": {...},
        },
    },
    ...
}
```

mesh

Path to the HDF5 mesh file.

Type str · **Required** yes

Guidance: How the deck works (p. 189)

boundary conditions

Boundary conditions mapped to surface zones.

TYPE	WHAT IT IS
wall (p. 99)	Wall boundary condition.
farfield (p. 117)	Farfield boundary condition.
symmetry (p. 123)	Symmetry boundary condition.
inflow (p. 128)	Pressure-based inflow boundary condition.
outflow (p. 131)	Pressure-based outflow boundary condition.
interface (p. 134)	Overset interface: this mesh overlaps another and takes its fringe values from it.

wall

Wall boundary condition.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	always `wall`	Boundary condition applied on these zones; it determines which further settings the block takes.
`zones`	list of zone ids	required	Mesh zone ids this boundary condition applies to.
`kind`	choice	`no slip`	Wall model applied on these zones.
`most` (p. 102)	settings block		Monin-Obukhov surface-layer settings.
`temperature` (p. 104)	float or settings block, > 0.0 K		Wall temperature, as a number or a {'field': '<map>.vtp'} map; unset, the wall is adiabatic.
`heat flux` (p. 105)	float or settings block, in W/m^2		Wall heat flux, positive into the fluid, as a number or a {'field': '<map>.vtp'} map.
`compressible wall function`	bool	`False`	Applies the compressible law of the wall on an adiabatic wall-function wall; a 'temperature' or a 'heat flux' selects it anyway.
`velocity` (p. 106)	choice of settings blocks		Velocity of the wall surface, for surfaces in relative motion such as a moving belt or a rotating wheel.
`roughness` (p. 108)	settings block		Surface roughness; unset, the wall is smooth.
`fsi` (p. 110)	choice of settings blocks		Fluid-structure interaction model that couples this wall's motion to a structural solve.

wall

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "type": "wall",
                    "zones": [...],
                    "kind": "no slip",
                    "most": {...},
                    "temperature": <float or settings block>,
                    "heat flux": <float or settings block>,
                    "compressible wall function": False,
                    "velocity": {...},
                    "roughness": {...},
```

```
                    "fsi": {...},
                },
            },
            ...
        },
    },
    ...
}
```

type

Boundary condition applied on these zones; it determines which further settings the block takes.

Type choice · **Default** always `wall`

Guidance: Choosing boundary conditions (p. 285)

zones

Mesh zone ids this boundary condition applies to.

Type list of zone ids · **Required** yes

Guidance: Choosing boundary conditions › Zones (p. 286)

kind

Wall model applied on these zones.

Type choice · **Default** `no slip`

VALUE	WHAT IT DOES	WHEN TO USE IT
`slip`	Zero velocity normal to the wall and no shear stress along it.	
`no slip` (default)	Applies the no-slip condition directly at the wall.	Viscous wall with the mesh resolved to $y^+ \sim 1$.
`wall function`	Takes the wall stress from a law of the wall instead of resolving the near-wall layer: Spalding's law with SST, the Allmaras fit with SA-neg.	Viscous wall with the first cell between $y^+ = 30$ and 250; it tolerates a first cell down to $y^+ \sim 1$. The law of the wall (Spalding's with SST, the Allmaras fit with SA-neg) represents a flat plate in zero pressure gradient, so accuracy falls on highly curved surfaces and in strong pressure gradients.
`most`	Takes the wall stress and surface heat flux from Monin-Obukhov similarity theory on a roughness length.	

Guidance: Choosing a wall model › Slip, no slip or wall function (p. 231)

`compressible wall function`

Applies the compressible law of the wall on an adiabatic wall-function wall; a 'temperature' or a 'heat flux' selects it anyway.

Type bool · **Default** `False` · **Applies when** kind is 'wall function'

Guidance: Choosing a wall model › Wall function (p. 232)

most

Monin-Obukhov surface-layer settings.

KEY	TYPE	DEFAULT	DESCRIPTION
`roughness length`	float, > 0.0 m	required	Aerodynamic roughness length z0
`heat roughness length`	float, > 0.0 m	not set	Thermal roughness length z0h; defaults to roughness_length
`surface potential temperature`	float, > 0.0 K	not set	Prescribed surface theta_s; unset leaves the wall thermally passive
`surface cooling rate`	float, in K/h	not set	d(theta_s)/dt applied to surface_potential_temperature (GABLS1 uses -0.25)
`stable form`	choice	`hogstrom`	Stability function family for stable stratification
`von karman`	float, > 0.0	`0.4`	Von Karman constant (atmospheric convention 0.4)
`beta m`	float, > 0.0	`4.8`	Stable momentum coefficient, phi_m = 1 + beta_m zeta
`beta h`	float, > 0.0	`7.8`	Stable heat coefficient, phi_h = 1 + beta_h zeta

most

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "most": {
                        "roughness length": <float>,
                        "heat roughness length": <float>,
                        "surface potential temperature": <float>,
                        "surface cooling rate": <float>,
                        "stable form": "hogstrom",
                        "von karman": 0.4,
                        "beta m": 4.8,
                        "beta h": 7.8,
                    },
                    ...
                },
            },
            ...
        },
    },
```

```
    ...
}
```

`stable form`

Stability function family for stable stratification

Type choice · **Default** `hogstrom`

VALUE	WHAT IT DOES
`hogstrom` (default)	Linear functions, phi = 1 + beta zeta, with coefficients 'beta m' and 'beta h' (Högström 1988); the surface flux collapses as stability grows.
`beljaars-holtslag`	Beljaars and Holtslag (1991) functions, which keep a finite surface flux under strong stability.

temperature

Wall temperature, as a number or a {'field': '<map>.vtp'} map; unset, the wall is adiabatic.

KEY	TYPE	DEFAULT	DESCRIPTION
field	str	required	VTP file with a point array named after the quantity ('Temperature' or 'HeatFlux'); each face takes the nearest point's value

temperature

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "temperature": {
                        "field": <str>,
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

heat flux

Wall heat flux, positive into the fluid, as a number or a {'field': '<map>.vtp'} map.

KEY	TYPE	DEFAULT	DESCRIPTION
field	str	required	VTP file with a point array named after the quantity ('Temperature' or 'HeatFlux'); each face takes the nearest point's value

heat flux

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "heat flux": {
                        "field": <str>,
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

velocity

Velocity of the wall surface, for surfaces in relative motion such as a moving belt or a rotating wheel.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	required	How the wall moves: `linear` translates it along a vector, `rotating` turns it about an axis.
`vector`	vector (3 floats)	required	*linear only* · Direction of linear velocity
`mach`	float	not set	*linear only* · Mach number for linear velocity - overrides magnitude of the translation velocity
`axis`	unit vector (3 floats)	required	*rotating only* · Axis of rotation.
`origin`	vector (3 floats), in m	required	*rotating only* · A point on the axis of rotation.
`omega`	float, in rad/s	required	*rotating only* · Angular velocity, positive right-handed about 'axis'.

velocity

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "velocity": {
                        "type": <choice>,
                        "vector": <[x, y, z]>,
                        "mach": <float>,
                        "axis": <[x, y, z]>,
                        "origin": <[x, y, z]>,
                        "omega": <float>,
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

type

How the wall moves: `linear` translates it along a vector, `rotating` turns it about an axis.

Type choice · **Required** yes

VALUE	WHAT IT DOES
`linear`	Selects a linear (translational) wall velocity.
`rotating`	Selects a rotational wall velocity.

Guidance: Choosing a wall model › Moving walls (p. 236)

roughness

Surface roughness; unset, the wall is smooth.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	required	Which roughness measure the value gives.
`scalar`	float, >= 0 m	not set	Uniform roughness over the wall, as the measure 'type' names.
`field`	str	not set	VTP file with a point array named 'Roughness'; each wall face takes the nearest point's value.
`thermal model`	choice	`owen thomson`	How roughness affects wall heat transfer under the compressible wall function.

roughness

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "roughness": {
                        "type": <choice>,
                        "scalar": <float>,
                        "field": <str>,
                        "thermal model": "owen thomson",
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

type

Which roughness measure the value gives.

Type choice · **Required** yes

VALUE	WHAT IT DOES
`height`	The value is the equivalent sand-grain height of the roughness elements.
`length`	The value is the aerodynamic roughness length, converted internally to a sand-grain height of length / 0.03.

Guidance: Choosing a wall model › Roughness (p. 235)

`scalar`

Uniform roughness over the wall, as the measure 'type' names.

Type float, >= 0 m · **Default** not set · **Applies when** give either 'scalar' or 'field'

`thermal model`

How roughness affects wall heat transfer under the compressible wall function.

Type choice · **Default** `owen thomson` · **Applies when** kind is 'wall function' and either temperature is set or heat flux is set

VALUE	WHAT IT DOES
`owen thomson` (default)	Adds a thermal sublayer resistance between the surface and the roughness crest, so the Stanton number rises less than the skin friction.
`reynolds analogy`	Omits that resistance, so the Stanton number rises in proportion to the skin friction.

Guidance: Choosing a wall model › Rough compressible walls (p. 234)

fsi

Fluid-structure interaction model that couples this wall's motion to a structural solve.

KEY	TYPE	DEFAULT	DESCRIPTION
`support radius`	float, > 0	`1`	RBF support radius, also the mode-shape deformation scale used when normalising mode shapes for morphing
`fixed zones`	list of int	not set	Zones held in place while the moving zones deform.
`basis`	choice	`c2`	*modal_model, rbf_multiscale / generic_fsi only* · RBF function type
`base point fraction`	float	required	*modal_model, rbf_multiscale / generic_fsi only* · Fraction of points to use as base points
`file format`	choice	required	*modal_model only* · Structural file format
`nastran casename`	str	required	*modal_model only* · Nastran case name (without extension)
`scale`	list of float	not set	*modal_model only* · Scaling factors [x, y, z]
`mode mapping max distance`	float	required	*modal_model only* · Max distance for mode mapping
`mode mapping power`	float, > 0	`1`	*modal_model only* · Inverse-distance power for mapping mode shapes to the surface
`mode mapping pure idw`	bool	`True`	*modal_model only* · Use pure inverse-distance weighting for the mode mapping
`mode mapping n nearest`	int, >= 1	`4`	*modal_model only* · Number of nearest structural points used per surface point
`recalculate rbfs alpha fraction`	float, >= 0.0	`0`	*modal_model only* · Deformation fraction of the RBF support radius at which the base-point set is rebuilt; 0 never rebuilds it
`newmark alpha`	float	`0.25`	*modal_model only* · Newmark integrator alpha parameter
`newmark delta`	float	`0.5`	*modal_model only* · Newmark integrator delta parameter
`update frequency`	int, >= 1	`1`	*modal_model only* · Cycles between structural updates: the modes are marched and the mesh deformed once every this many solver cycles.

KEY	TYPE	DEFAULT	DESCRIPTION
`fluid force scaling`	float	required	*modal_model only* · Scaling factor for fluid forces
`mode list`	list of int	required	*modal_model only* · List of modes to include
`modal damping`	list of float	required	*modal_model only* · Damping factors for each mode
`harmonic initial state` (p. 116)	settings block		*modal_model only* · Starts every mode on its harmonic response to a point force, instead of at rest
`type`	choice	not set	The structural coupling: `modal_model` uses the built-in Nastran modal model, `generic_fsi` couples to an external structural solver.
`transform type`	choice	not set	Mesh deformation transform used for the coupling
`power`	float or list of float	required	*modal_model, idw / generic_fsi only* · Power parameters for IDW
`stencil size`	int, > = 1	not set	*idw only* · Maximum stencil size (number of nearest neighbors)
`n nearest`	int, > = 1	`100`	*modal_model, idw / generic_fsi only* · Number of nearest points for IDW
`deformation distance`	float, > 0	`1`	*modal_model, idw / generic_fsi only* · Maximum deformation distance
`blending stiffness`	float, 0 to 1	`0.5`	*modal_model, idw / generic_fsi only* · Blending stiffness parameter
`zone`	list of zone ids	required	*generic_fsi only* · Zone IDs coupled to the external solver
`tol`	float	`0.01`	*generic_fsi only* · RBF solve tolerance
`max error`	float	`0.1`	*generic_fsi only* · Maximum allowed RBF interpolation error
`user variables`	settings block	not set	*generic_fsi only* · Arbitrary variables passed to the external coupling solver (genuinely open-ended — not enumerable).

fsi

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "fsi": {
                        "support radius": 1.0,
                        "fixed zones": [...],
```

```
                    "basis": "c2",
                    "base point fraction": <float>,
                    "file format": <choice>,
                    "nastran casename": <str>,
                    "scale": [...],
                    "mode mapping max distance": <float>,
                    "mode mapping power": 1.0,
                    "mode mapping pure idw": True,
                    "mode mapping n nearest": 4,
                    "recalculate rbfs alpha fraction": 0.0,
                    "newmark alpha": 0.25,
                    "newmark delta": 0.5,
                    "update frequency": 1,
                    "fluid force scaling": <float>,
                    "mode list": [...],
                    "modal damping": [...],
                    "harmonic initial state": {...},
                    "type": <choice>,
                    "transform type": <choice>,
                    "power": <float or list of float>,
                    "stencil size": <int>,
                    "n nearest": 100,
                    "deformation distance": 1.0,
                    "blending stiffness": 0.5,
                    "zone": [...],
                    "tol": 0.01,
                    "max error": 0.1,
                    "user variables": {...},
                },
                ...
            },
        },
        ...
    },
    ...
}
```

`support radius`

RBF support radius, also the mode-shape deformation scale used when normalising mode shapes for morphing

Type float, > 0 · **Default** 1

Guidance: Fluid–structure interaction › Interpolation method (p. 320)

basis

RBF function type

Type choice · **Default** `c2`

VALUE	WHAT IT DOES	WHEN TO USE IT
`tps`	A thin-plate spline in the distance scaled by `support_radius`; it is never zero, so the system is dense.	Small surfaces where a global, smooth deformation is wanted.
`thin_plate_spline` *rbf_multiscale only*	Not implemented under this name; the run stops with an unknown-basis error.	Never; the thin-plate spline is `tps`.
`multiquadric` *rbf_multiscale only*	Not implemented; the run stops with an unknown-basis error.	Never; use `c2`.
`inverse_multiquadric` *rbf_multiscale only*	Not implemented; the run stops with an unknown-basis error.	Never; use `c2`.
`gaussian`	A Gaussian whose width is `support_radius`; it is never zero, so every pair of points interacts and the system is dense.	Small surfaces only, as the dense system grows with the square of the number of points.
`linear` *rbf_multiscale only*	Not implemented; the run stops with an unknown-basis error.	Never; use `c2`.
`cubic` *rbf_multiscale only*	Not implemented; the run stops with an unknown-basis error.	Never; use `c2`.
`quintic` *rbf_multiscale only*	Not implemented; the run stops with an unknown-basis error.	Never; use `c2`.
`c0`	A Wendland C0 function that is zero beyond `support_radius`; the deformation is continuous but its slope is not.	Rarely; `c2` gives a smoother deformation with the same sparsity.
`c2` (default)	A Wendland C2 function that is zero beyond `support_radius`, so only nearby points interact and the system is sparse.	The default, and the appropriate choice for most cases.
`c4`	Selects a Wendland C4 function, but the matrix assembly and the volume deformation evaluate `c2`, so the interpolation is inconsistent.	Not at present; use `c2`.
`c6` *rbf_multiscale only*	Not implemented; the run stops with an unknown-basis error.	Never; use `c2`.

Guidance: Fluid–structure interaction › Interpolation method (p. 320)

base point fraction

Fraction of points to use as base points

Type float · **Required** yes

Guidance: Fluid–structure interaction › Interpolation method (p. 320)

`file format`

Structural file format

Type choice · **Required** yes

VALUE	WHAT IT DOES	WHEN TO USE IT
`nastran`	Reads the structural grid from `<nastran_casename>.bdf` and the mode shapes and frequencies from `<nastran_casename>.op2`.	Always; it is the only format supported.

Guidance: Fluid–structure interaction › Modal model (p. 318)

`update frequency`

Cycles between structural updates: the modes are marched and the mesh deformed once every this many solver cycles. With several modal models the smallest value applies to all of them.

Type int, >= 1 · **Default** 1

`type`

The structural coupling: `modal_model` uses the built-in Nastran modal model, `generic_fsi` couples to an external structural solver.

Type choice · **Default** not set · **Applies when** zones is not []

VALUE	WHAT IT DOES
`modal_model`	Selects the built-in Nastran modal-model coupling.
`generic_fsi`	Selects the generic external-solver coupling.

Guidance: Fluid–structure interaction › Modal model (p. 318)

`transform type`

Mesh deformation transform used for the coupling

Type choice · **Default** not set

VALUE	WHAT IT DOES	WHEN TO USE IT
`rbf_multiscale`	Deforms the volume mesh using the multiscale RBF scheme.	
`idw`	Deforms the volume mesh using inverse distance weighting.	

VALUE	WHAT IT DOES	WHEN TO USE IT
`rbf`	Deforms the volume mesh with a single-level RBF, adding surface points one at a time, worst-interpolated first, until each is within `tol`.	Where an error-controlled choice of points is preferred to the fixed `base_point_fraction` of `rbf_multiscale`.

Guidance: Fluid–structure interaction › RBF multiscale transform (p. 318)

`power`

Power parameters for IDW

Type float or list of float · **Required** yes

Guidance: Fluid–structure interaction › Interpolation method (p. 320)

harmonic initial state

Starts every mode on its harmonic response to a point force, instead of at rest

KEY	TYPE	DEFAULT	DESCRIPTION
`location`	vector (3 floats)	required	Point the force acts at; the force there is written to the modal history
`force`	vector (3 floats)	required	Force vector (N); its magnitude sets each mode's initial displacement and velocity
`frequency`	float	required	Frequency of the harmonic force (Hz)

harmonic initial state

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "fsi": {
                        "harmonic initial state": {
                            "location": <[x, y, z]>,
                            "force": <[x, y, z]>,
                            "frequency": <float>,
                        },
                        ...
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

farfield

Farfield boundary condition.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	always `farfield`	Boundary condition applied on these zones; it determines which further settings the block takes.
`zones`	list of zone ids	required	Mesh zone ids this boundary condition applies to.
`kind`	choice	`riemann`	How the farfield state is imposed.
`condition`	str	required	Reference condition that sets the farfield state.
`sponge layer damping` (p. 119)	settings block		Sponge layer that damps the solution towards a reference condition near the farfield boundaries.
`target` (p. 120)	settings block		A lift or drag coefficient the farfield is trimmed to reach, by turning its angle of attack.

farfield

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "type": "farfield",
                    "zones": [...],
                    "kind": "riemann",
                    "condition": <str>,
                    "sponge layer damping": {...},
                    "target": {...},
                },
            },
            ...
        },
    },
    ...
}
```

type

Boundary condition applied on these zones; it determines which further settings the block takes.

Type choice · **Default** always `farfield` · **Applies when** solver.solver settings.type is 'compressible'

Guidance: Choosing boundary conditions (p. 285)

`zones`

Mesh zone ids this boundary condition applies to.

Type list of zone ids · **Required** yes

Guidance: Choosing boundary conditions › Zones (p. 286)

`kind`

How the farfield state is imposed.

Type choice · **Default** `riemann`

VALUE	WHAT IT DOES
`riemann` (default)	Uses Riemann invariants for a non-reflecting inflow with a pressure outflow.
`pressure`	Uses a pressure boundary condition for both inflow (downstream pressure) and outflow (user-specified pressure).
`supersonic`	Uses upstream conditions for both inflow and outflow.
`preconditioned`	Imposes the reference condition as a supersonic inflow on every face, whatever the local flow direction.

Guidance: Choosing boundary conditions › Farfield (p. 286)

sponge layer damping

Sponge layer that damps the solution towards a reference condition near the farfield boundaries.

KEY	TYPE	DEFAULT	DESCRIPTION
`distance`	float, in m	required	Depth of the sponge layer, measured into the domain from the farfield boundaries.
`damping factor`	float, in 1/s	required	Peak damping rate, applied at the farfield boundary and ramped to zero at 'distance' along a cosine profile.
`condition`	str	required	Reference condition the solution is damped towards inside the layer.

sponge layer damping

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "sponge layer damping": {
                        "distance": <float>,
                        "damping factor": <float>,
                        "condition": <str>,
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

target

A lift or drag coefficient the farfield is trimmed to reach, by turning its angle of attack.

KEY	TYPE	DEFAULT	DESCRIPTION
`alpha` (p. 121)	settings block		Turn the angle of attack, starting from the condition's own velocity, until the coefficient is met.
`axes` (p. 125)	settings block		Body axes the angle of attack and the forces are measured in.

target

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "target": {
                        "alpha": {...},
                        "axes": {...},
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

alpha

Turn the angle of attack, starting from the condition's own velocity, until the coefficient is met.

KEY	TYPE	DEFAULT	DESCRIPTION
`cl`	float	not set	Lift coefficient to trim to.
`cd`	float	not set	Drag coefficient to trim to.
`hold`	int, > 0	not set	Cycles to run on once the target is met, after which the run stops.
`tolerance`	float, > 0	not set	How close the coefficient must come to its target to count as met.
`of`	str	not set	Force report the coefficient is read from.
`advanced` (p. 123)	settings block		Tuning of how the angle is stepped and how the coefficient is measured.

alpha

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "target": {
                        "alpha": {
                            "cl": <float>,
                            "cd": <float>,
                            "hold": <int>,
                            "tolerance": <float>,
                            "of": <str>,
                            "advanced": {...},
                        },
                        ...
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

`cl`

Lift coefficient to trim to.

Type float · **Default** not set · **Applies when** give either 'cl' or 'cd'

`hold`

Cycles to run on once the target is met, after which the run stops. Unset, the run ends in the ordinary way.

Type int, > 0 · **Default** not set

`tolerance`

How close the coefficient must come to its target to count as met. Unset, it is 0.001 for CL and 0.0001 for CD.

Type float, > 0 · **Default** not set

`of`

Force report the coefficient is read from. Needed only when there is more than one force report.

Type str · **Default** not set

advanced

Tuning of how the angle is stepped and how the coefficient is measured. Unset, every setting takes its default.

KEY	TYPE	DEFAULT	DESCRIPTION
`max step`	float, > 0 and <= 6	not set	Largest change of the angle of attack in one step, in degrees, up to 6: a larger jump can make the flow solve diverge.
`relaxation`	float, > 0 and <= 2	not set	Fraction of the computed step that is taken, up to 2.
`average over`	int, >= 1	not set	Number of report rows averaged into each measurement of the coefficient.
`max wait`	int, > 0	not set	Cycles to wait after a step for the coefficient to settle, after which the next step is taken regardless.
`slope`	float	not set	First guess of the change of the coefficient per degree of angle of attack.

advanced

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "target": {
                        "alpha": {
                            "advanced": {
                                "max step": <float>,
                                "relaxation": <float>,
                                "average over": <int>,
                                "max wait": <int>,
                                "slope": <float>,
                            },
                            ...
                        },
                        ...
                    },
                    ...
                },
            },
            ...
        },
    },
```

```
    ...
}
```

`max step`

Largest change of the angle of attack in one step, in degrees, up to 6: a larger jump can make the flow solve diverge. Unset, it is 4.

Type float, > 0 and <= 6 · **Default** not set

`relaxation`

Fraction of the computed step that is taken, up to 2. Unset, it is 1.

Type float, > 0 and <= 2 · **Default** not set

`average over`

Number of report rows averaged into each measurement of the coefficient. Unset, it is 10.

Type int, >= 1 · **Default** not set

`max wait`

Cycles to wait after a step for the coefficient to settle, after which the next step is taken regardless. Unset, it is 1500.

Type int, > 0 · **Default** not set

`slope`

First guess of the change of the coefficient per degree of angle of attack. Unset, it is 0.11 for CL, the thin-aerofoil slope of 2π per radian; for CD a first step of half the max step is taken instead.

Type float · **Default** not set

axes

Body axes the angle of attack and the forces are measured in. Alpha turns from forward towards up; drag lies along the flow and lift at right angles to it, towards up.

KEY	TYPE	DEFAULT	DESCRIPTION
`forward`	choice	`+x`	Body axis pointing forward, into the oncoming flow.
`up`	choice	`+z`	Body axis pointing up.

axes

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "target": {
                        "axes": {
                            "forward": "+x",
                            "up": "+z",
                        },
                        ...
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

`forward`

Body axis pointing forward, into the oncoming flow.

Type choice · **Default** `+x`

VALUE	WHAT IT DOES
`+x` (default)	
`-x`	

VALUE	WHAT IT DOES
+y	
-y	
+z	
-z	

up

Body axis pointing up. Lift is positive towards it.

Type choice · **Default** +z

VALUE	WHAT IT DOES
+x	
-x	
+y	
-y	
+z (default)	
-z	

symmetry

Symmetry boundary condition.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	always `symmetry`	Boundary condition applied on these zones; it determines which further settings the block takes.
`zones`	list of zone ids	required	Mesh zone ids this boundary condition applies to.

symmetry

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "type": "symmetry",
                    "zones": [...],
                },
            },
            ...
        },
    },
    ...
}
```

`type`

Boundary condition applied on these zones; it determines which further settings the block takes.

Type choice · **Default** always `symmetry`

Guidance: Choosing boundary conditions (p. 285)

`zones`

Mesh zone ids this boundary condition applies to.

Type list of zone ids · **Required** yes

Guidance: Choosing boundary conditions › Zones (p. 286)

inflow

Pressure-based inflow boundary condition.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	always `inflow`	Boundary condition applied on these zones; it determines which further settings the block takes.
`zones`	list of zone ids	required	Mesh zone ids this boundary condition applies to.
`reference`	str	not set	Reference condition the inflow state and its ratios are taken from.
`total temperature ratio`	float	`1`	Inlet total temperature as a ratio of the reference static temperature.
`kind`	choice	not set	How the inflow state is set.
`static pressure ratio`	float	not set	*pressure only* · Inlet static pressure as a ratio of the reference static pressure.
`total pressure ratio`	float	`1`	*pressure only* · Inlet total pressure as a ratio of the reference static pressure.
`direction`	unit vector (3 floats)	not set	*pressure / massflow only* · Direction the flow enters along; unset, it enters normal to the boundary.
`mach`	float, > 0	not set	*pressure only* · Inlet Mach number; the inlet static pressure then follows from the total pressure ratio and this Mach number rather than from the interior flow.
`mass flow rate`	float, > 0 kg/s	not set	*massflow only* · Mass flow rate through the inlet zones, as a magnitude: the inflow or outflow type sets the direction.
`mass flow ratio`	float, > 0	not set	*massflow only* · Inlet mass flow divided by the reference mass flux, rho_in U_in A_in / (rho_ref U_ref), with A_in in mesh units; a magnitude, since the inflow or outflow type sets the direction.

inflow

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "type": "inflow",
                    "zones": [...],
                    "reference": <str>,
```

```
                    "total temperature ratio": 1.0,
                    "kind": <choice>,
                    "static pressure ratio": <float>,
                    "total pressure ratio": 1.0,
                    "direction": <[x, y, z]>,
                    "mach": <float>,
                    "mass flow rate": <float>,
                    "mass flow ratio": <float>,
                },
            },
            ...
        },
    },
    ...
}
```

`type`

Boundary condition applied on these zones; it determines which further settings the block takes.

Type choice · **Default** always `inflow`

Guidance: Choosing boundary conditions › Inflow (p. 286)

`zones`

Mesh zone ids this boundary condition applies to.

Type list of zone ids · **Required** yes

Guidance: Choosing boundary conditions › Zones (p. 286)

`total temperature ratio`

Inlet total temperature as a ratio of the reference static temperature.

Type float · **Default** `1` · **Applies when** reference is set

`kind`

How the inflow state is set.

Type choice · **Default** not set

VALUE	WHAT IT DOES
`pressure`	Velocity and density are derived from a total pressure ratio and a total temperature ratio relative to 'reference'.
`massflow`	The inlet velocity is set from a mass flow rate or a mass flow ratio.

VALUE	WHAT IT DOES
`default`	Incompressible solver only: the inflow state is taken directly from the reference condition.

static pressure ratio

Inlet static pressure as a ratio of the reference static pressure.

Type float · **Default** not set · **Applies when** reference is set

mass flow rate

Mass flow rate through the inlet zones, as a magnitude: the inflow or outflow type sets the direction. The mesh must be in metres.

Type float, > 0 kg/s · **Default** not set · **Applies when** 'mass flow rate' is required when solver.solver settings.type is 'compressible' and mass flow ratio is unset; give either 'mass flow rate' or 'mass flow ratio', not both

mass flow ratio

Inlet mass flow divided by the reference mass flux, rho_in U_in A_in / (rho_ref U_ref), with A_in in mesh units; a magnitude, since the inflow or outflow type sets the direction.

Type float, > 0 · **Default** not set · **Applies when** reference is set

outflow

Pressure-based outflow boundary condition.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	always `outflow`	Boundary condition applied on these zones; it determines which further settings the block takes.
`zones`	list of zone ids	required	Mesh zone ids this boundary condition applies to.
`reference`	str	not set	Reference condition the outflow ratios are taken relative to.
`kind`	choice	not set	How the outflow state is set.
`static pressure ratio`	float	`1`	*pressure / radial pressure gradient only* · Exit static pressure as a ratio of the reference static pressure.
`mass flow rate`	float, > 0 kg/s	not set	*massflow only* · Mass flow rate through the outlet zones, as a magnitude: the inflow or outflow type sets the direction.
`mass flow ratio`	float, > 0	not set	*massflow only* · Outlet mass flow divided by the reference mass flux, rho_out U_out A_out / (rho_ref U_ref), with A_out in mesh units; a magnitude, since the inflow or outflow type sets the direction.
`reference radius`	float, in m	required	*radial pressure gradient only* · Radius at which the static pressure ratio is defined

outflow

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "type": "outflow",
                    "zones": [...],
                    "reference": <str>,
                    "kind": <choice>,
                    "static pressure ratio": 1.0,
                    "mass flow rate": <float>,
                    "mass flow ratio": <float>,
                    "reference radius": <float>,
                },
            },
            ...
```

```
        },
    },
    ...
}
```

`type`

Boundary condition applied on these zones; it determines which further settings the block takes.

Type choice · **Default** always `outflow`

Guidance: Choosing boundary conditions › Outflow (p. 287)

`zones`

Mesh zone ids this boundary condition applies to.

Type list of zone ids · **Required** yes

Guidance: Choosing boundary conditions › Zones (p. 286)

`kind`

How the outflow state is set.

Type choice · **Default** not set

VALUE	WHAT IT DOES
`pressure`	The exit pressure is set from a static pressure ratio relative to 'reference'.
`massflow`	Scales the local velocity so the integrated mass flow across all zones matches the requested value.
`radial pressure gradient`	As 'pressure', but the static pressure ratio holds at 'reference radius' rather than uniformly.

Guidance: Choosing boundary conditions › Outflow (p. 287)

`mass flow rate`

Mass flow rate through the outlet zones, as a magnitude: the inflow or outflow type sets the direction. The mesh must be in metres.

Type float, > 0 kg/s · **Default** not set · **Applies when** 'mass flow rate' is required when solver.solver settings.type is 'compressible' and mass flow ratio is unset; give either 'mass flow rate' or 'mass flow ratio', not both

mass flow ratio

Outlet mass flow divided by the reference mass flux, rho_out U_out A_out / (rho_ref U_ref), with A_out in mesh units; a magnitude, since the inflow or outflow type sets the direction.

Type float, > 0 · **Default** not set · **Applies when** reference is set

interface

Overset interface: this mesh overlaps another and takes its fringe values from it.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	always `interface`	Boundary condition applied on these zones; it determines which further settings the block takes.
`zones`	list of zone ids	required	Mesh zone ids this boundary condition applies to.
`kind`	choice	not set	How this interface joins its two sides.
`connection` (p. 139)	settings block		Model(s) whose mesh provides donor data at this interface.
`interpolation method`	choice	`least-squares`	*overset only* · How the donor value is built once the containing donor cell is known.
`additional active layers`	int, >= 0	`0`	*overset only* · Number of additional active cell layers near the blanking boundary
`blank enclosed cells`	bool	`True`	*overset only* · Also blank the background cells the hole encloses but no overset cell covers: the ones inside this mesh's own solid bodies
`create overset halos`	bool	`True`	*overset only* · Search for overset donors at the halo cell centre rather than at the boundary face centre, on the background-to-overset direction.
`moving mesh halo layers`	int, >= 0	`0`	*overset only* · Halo fringe layers around the hole when the overset mesh moves (0 = treat every blanked cell as a halo; N > 0 is sufficient when the hole is re-cut every real time step)
`area`	choice	`conservative`	*sliding only* · Which area vector each side's segments use.
`flux`	choice	`segment`	*sliding only* · Where the interface's numerical flux is evaluated.
`segment capacity factor`	int, >= 1	`8`	*sliding only* · Segment storage reserved per interface face.
`sliver tolerance`	float, >= 0.0	`1e-10`	*sliding only* · Overlaps smaller than this fraction of the smaller parent face are discarded.
`closure tolerance`	float, >= 0.0	`0.01`	*sliding only* · Largest fraction of a face's area that may be left uncovered by its segments before setup aborts.
`transform` (p. 141)	choice of settings blocks		*periodic only* · Transformation that maps this boundary onto the one it connects to.

interface

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "type": "interface",
                    "zones": [...],
                    "kind": <choice>,
                    "connection": {...},
                    "interpolation method": "least-squares",
                    "additional active layers": 0,
                    "blank enclosed cells": True,
                    "create overset halos": True,
                    "moving mesh halo layers": 0,
                    "area": "conservative",
                    "flux": "segment",
                    "segment capacity factor": 8,
                    "sliver tolerance": 1e-10,
                    "closure tolerance": 0.01,
                    "transform": {...},
                },
            },
            ...
        },
    },
    ...
}
```

`type`

Boundary condition applied on these zones; it determines which further settings the block takes.

Type choice · **Default** always `interface`

Guidance: Choosing boundary conditions (p. 285)

`zones`

Mesh zone ids this boundary condition applies to.

Type list of zone ids · **Required** yes

Guidance: Choosing boundary conditions › Zones (p. 286)

`kind`

How this interface joins its two sides.

Type choice · **Default** not set

VALUE	WHAT IT DOES
`overset`	This mesh overlaps another and takes the values in its fringe from the mesh beneath it.
`sliding`	Two meshes meet on a shared surface of revolution with non-matching faces and exchange a conservative flux across it.
`periodic`	Two boundaries of the same mesh are treated as one surface, related by a rotation or a translation.

`interpolation method`

How the donor value is built once the containing donor cell is known.

Type choice · **Default** `least-squares`

VALUE	WHAT IT DOES	WHEN TO USE IT
`nearest cell`	Takes the containing donor cell's value directly: a piecewise-constant, first-order transfer.	
`inverse distance`	A distance-weighted average over the containing donor cell and its neighbours.	
`green-gauss`	A Green-Gauss linear reconstruction about the donor cell centre, evaluated at the receptor point.	
`least-squares` (default)	A least-squares linear reconstruction about the donor cell centre, evaluated at the receptor point.	On a moving interface, where only it and `green-gauss` reproduce a field with a gradient at the receptor point.

Guidance: Overset meshes › interpolation_method (p. 329)

`additional active layers`

Number of additional active cell layers near the blanking boundary

Type int, >= 0 · **Default** `0`

Guidance: Overset meshes › additional_active_layers (p. 329)

`blank enclosed cells`

Also blank the background cells the hole encloses but no overset cell covers: the ones inside this mesh's own solid bodies

Type bool · **Default** `True`

Guidance: Overset meshes › blank_enclosed_cells (p. 330)

create overset halos

Search for overset donors at the halo cell centre rather than at the boundary face centre, on the background-to-overset direction. The halo centre is the interior cell centre mirrored across the face, and that is where the solver uses the value: the interface flux extrapolates the halo from it and the boundary gradient averages it with the interior cell. Sampling at the face centre instead puts the value half a cell from where it is used; on the eulervortex static overset case that leaves a 416 Pa pressure overshoot at the interface, against 4 Pa with halo centres – what the conformal mesh gives. A uniform free stream cannot show the difference. The solver applies one value to the whole model, so every overset interface on a model gives the same one.

Type bool · **Default** `True`

moving mesh halo layers

Halo fringe layers around the hole when the overset mesh moves (0 = treat every blanked cell as a halo; N > 0 is sufficient when the hole is re-cut every real time step)

Type int, >= 0 · **Default** `0`

Guidance: Overset meshes › moving_mesh_halo_layers (p. 330)

area

Which area vector each side's segments use. The two options are identical on a planar interface and for matching meshes.

Type choice · **Default** `conservative`

VALUE	WHAT IT DOES
`conservative` (default)	Both sides use the mean of the two parent area maps, so the interface conserves to round-off; the per-face closure error is first order in cell width.
`free stream`	Each side uses its own parent area map, so closure and free-stream preservation are exact; the conservation deficit is second order.

Guidance: Choosing boundary conditions › Sliding (p. 289)

flux

Where the interface's numerical flux is evaluated.

Type choice · **Default** `segment`

VALUE	WHAT IT DOES
`segment` (default)	One flux per intersection polygon, applied with a shared area vector to the cell on each side, so the two sides cancel exactly and the interface conserves.

VALUE	WHAT IT DOES
`parent`	One flux per parent face, against the area-weighted average of that face's donors; quieter as cells pass one another, but not conservative.

Guidance: Choosing boundary conditions › Sliding (p. 289)

`segment capacity factor`

Segment storage reserved per interface face. The supermesh is rebuilt every real time step and its segment count varies with the rotation angle, so storage is reserved once at this multiple of the local interface face count. Setup sweeps a full revolution and aborts with the factor actually needed if this is too small.

Type int, >= 1 · **Default** `8`

`sliver tolerance`

Overlaps smaller than this fraction of the smaller parent face are discarded. Slivers carry no flux worth having but do cost a segment slot each.

Type float, >= 0.0 · **Default** `1e-10`

`closure tolerance`

Largest fraction of a face's area that may be left uncovered by its segments before setup aborts. A real gap between the two sides – mismatched radii, a partial annulus, a wrong axis – shows up here, and would otherwise be a silent leak.

Type float, >= 0.0 · **Default** `0.01`

connection

Model(s) whose mesh provides donor data at this interface. Without it, the solver maps against every preceding model.

KEY	TYPE	DEFAULT	DESCRIPTION
`model`	list of str	required	*overset / sliding only* · Model(s) whose mesh provides donor data at this interface.
`boundary`	str	required	*sliding / periodic only* · Boundary on that model that forms the other side of this interface.

connection

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "connection": {
                        "model": [...],
                        "boundary": <str>,
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

`model`

Model(s) whose mesh provides donor data at this interface.

Type list of str · **Required** yes

Guidance: Overset meshes › connection (p. 330)

`boundary`

Boundary on that model that forms the other side of this interface.

Type str · **Required** yes

Guidance: Choosing boundary conditions > Sliding (p. 289)

transform

Transformation that maps this boundary onto the one it connects to.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	not set	How this boundary maps onto its partner.
`theta`	float, in rad	required	*rotated only* · Angle of the rotation that maps this boundary onto its partner.
`axis`	unit vector (3 floats)	required	*rotated only* · Direction of the rotation axis.
`origin`	vector (3 floats), in m	required	*rotated only* · A point on the rotation axis.
`vector`	vector (3 floats), in m	required	*linear only* · Translation that maps this boundary onto its partner.

transform

```
parameters = {
    "model": {
        "<mesh>": {
            "boundary conditions": {
                "<name>": {
                    "transform": {
                        "type": <choice>,
                        "theta": <float>,
                        "axis": <[x, y, z]>,
                        "origin": <[x, y, z]>,
                        "vector": <[x, y, z]>,
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

type

How this boundary maps onto its partner.

Type choice · **Default** not set

VALUE	WHAT IT DOES
`rotated`	A rotation about an axis maps this boundary onto its partner.
`linear`	A translation maps this boundary onto its partner.

fluid zones

Fluid zone models (e.g. porous, actuator disc).

TYPE	WHAT IT IS
momentum (p. 144)	Momentum source fluid zone.
rotating (p. 147)	A rotating frame on mesh cell zones (MRF); the mesh stays still.
disc (p. 149)	Actuator disc fluid zone.
porous (p. 159)	Porous media fluid zone.
canopy (p. 161)	Canopy model fluid zone for atmospheric boundary layer simulations with vegetation.
abl (p. 163)	Atmospheric boundary layer fluid zone, applied to the whole mesh.
idw transform (p. 164)	Inverse Distance Weighting transform fluid zone.
rbf transform (p. 166)	Radial Basis Function transform fluid zone. Full RBF when base_point_fraction=1.0 (default), Multiscale RBF when base_point_fraction < 1.0.

momentum

Momentum source fluid zone.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	always `momentum`	Fluid zone type
`cell zones`	list of int	not set	Mesh cell zone numbers whose cells form the source region, for example “cell zones”: [4]; give this, ‘definition’, or neither for the whole mesh
`definition`	str	not set	Closed-surface VTP file selecting the source region cells; without it or ‘cell zones’ the source applies to the whole mesh
`kind`	choice	`fixed`	Source kind
`frequency`	int, >= 1	`1`	Update frequency
`func`	Python function, or a Python function	not set	Function of the cell centres giving the source, in the units of ‘source’, in every cell of the source, called as f(*, locations); it returns a list of one source vector of three numbers per location
`source`	vector (3 floats)	not set	Momentum source vector

momentum

```
parameters = {
    "model": {
        "<mesh>": {
            "fluid zones": {
                "<name>": {
                    "type": "momentum",
                    "cell zones": [...],
                    "definition": <str>,
                    "kind": "fixed",
                    "frequency": 1,
                    "func": <function>,
                    "source": <[x, y, z]>,
                },
            },
            ...
        },
    },
    ...
}
```

`type`

Fluid zone type

Type choice · **Default** always `momentum`

Guidance: Choosing a fluid-zone model › Momentum Source (p. 282)

`cell zones`

Mesh cell zone numbers whose cells form the source region, for example “cell zones”: [4]; give this, 'definition', or neither for the whole mesh

Type list of int · **Default** not set · **Applies when** give either 'cell zones' or 'definition', not both

Guidance: Choosing a fluid-zone model › Selecting cells (p. 274)

`definition`

Closed-surface VTP file selecting the source region cells; without it or 'cell zones' the source applies to the whole mesh

Type str · **Default** not set

Guidance: Choosing a fluid-zone model › Selecting cells (p. 274)

`kind`

Source kind

Type choice · **Default** `fixed`

VALUE	WHAT IT DOES	WHEN TO USE IT
`fixed` (default)	Labels the source as constant; the solver evaluates `func` once at start-up whatever the kind.	The default; every kind gives the same result.
`ramp`	Labels the source as ramped, but the solver does not read `kind`, so it behaves as `fixed`.	No effect; leave the default.
`func`	Labels the source as function-driven, but the solver does not read `kind`, so it behaves as `fixed`.	No effect; leave the default.

Guidance: Choosing a fluid-zone model › Momentum Source (p. 282)

`func`

Function of the cell centres giving the source, in the units of 'source', in every cell of the source, called as f(*, locations); it returns a list of one source vector of three numbers per location

Type Python function, or a Python function · **Default** not set

Guidance: Python functions in the deck > Momentum source function (p. 220)

rotating

A rotating frame on mesh cell zones (MRF); the mesh stays still.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	always `rotating`	Fluid zone type
`axis`	unit vector (3 floats)	required	Axis of rotation.
`origin`	vector (3 floats), in m	required	A point on the axis of rotation.
`omega`	float, in rad/s	required	Angular velocity, positive right-handed about 'axis'.
`cell zones`	list of int	required	Mesh cell zone numbers whose cells are solved in the rotating frame, for example "cell zones": [17]

rotating

```
parameters = {
    "model": {
        "<mesh>": {
            "fluid zones": {
                "<name>": {
                    "type": "rotating",
                    "axis": <[x, y, z]>,
                    "origin": <[x, y, z]>,
                    "omega": <float>,
                    "cell zones": [...],
                },
            },
            ...
        },
    },
    ...
}
```

type

Fluid zone type

Type choice · **Default** always `rotating` · **Applies when** the model has no 'transforms › motion' and no overset or sliding interface. A model has one frame: rotating fluid zones on its cell zones, or 'transforms › motion' for the whole mesh. A mesh with an overset or sliding interface turns as a whole, so it takes 'transforms › motion' only. Remove 'motion' from 'transforms', or remove the rotating fluid zones and turn the whole mesh with "transforms": {"motion": {"type": "rotating", "axis": […], "origin": […], "omega": …, "moving mesh": False}}.

Guidance: Choosing a fluid-zone model › Rotating zones (p. 277)

`cell zones`

Mesh cell zone numbers whose cells are solved in the rotating frame, for example “cell zones”: [17]
Type list of int · **Required** yes

Guidance: Choosing a fluid-zone model › Selecting cells (p. 274)

disc

Actuator disc fluid zone.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	always `disc`	
`thrust coefficient`	float	not set	Thrust coefficient
`disc area`	float, > 0	not set	Disc area
`tip speed ratio`	float	not set	Tip speed ratio
`reference density`	float, > 0	not set	Reference density
`verbose`	bool	`False`	Print turbine model and controller details at load
`u ref`	float, > 0	not set	Reference velocity (m/s) override for the BET force calculation — used instead of the sampled freestream, e.g. for hover cases where the freestream is near zero
`controller` (p. 151)	settings block		Turbine controller
`geometry` (p. 153)	settings block		Turbine geometry
`discretisation` (p. 154)	settings block		Turbine discretisation
`model` (p. 155)	settings block		Turbine model definition
`name`	str	required	Zone name
`reference plane`	bool	`False`	Use reference plane
`reference point`	vector (3 floats)	not set	Reference point
`definition`	str	not set	Closed-surface VTP file that selects the disc's cells
`update frequency`	int	`1`	Update frequency
`rotation direction`	str	required	Rotation direction

disc

```
parameters = {
    "model": {
        "<mesh>": {
            "fluid zones": {
                "<name>": {
                    "type": "disc",
                    "thrust coefficient": <float>,
                    "disc area": <float>,
```

```
                    "tip speed ratio": <float>,
                    "reference density": <float>,
                    "verbose": False,
                    "u ref": <float>,
                    "controller": {...},
                    "geometry": {...},
                    "discretisation": {...},
                    "model": {...},
                    "name": <str>,
                    "reference plane": False,
                    "reference point": <[x, y, z]>,
                    "definition": <str>,
                    "update frequency": 1,
                    "rotation direction": <str>,
                },
            },
            ...
        },
    },
    ...
}
```

type

Type choice · **Default** always `disc`

Guidance: Choosing a fluid-zone model › Actuator discs (p. 275)

definition

Closed-surface VTP file that selects the disc's cells

Type str · **Default** not set · **Applies when** 'definition' is required when the fluid zone is an actuator disc. A disc selects its cells with a closed-surface VTP file., and only applies then

Guidance: Choosing a fluid-zone model › Selecting cells (p. 274)

controller

Turbine controller

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	required	Controller type
`pitch`	float	not set	Pitch angle
`omega`	float	not set	Fixed rotational speed (rad/s), used with type='fixed'
`tip speed ratio`	float	not set	Target tip speed ratio, used with type='tsr curve'
`tip speed ratio curve`	list	not set	Tip speed ratio vs. time/velocity curve, as [x, y] pairs

controller

```
parameters = {
    "model": {
        "<mesh>": {
            "fluid zones": {
                "<name>": {
                    "controller": {
                        "type": <choice>,
                        "pitch": <float>,
                        "omega": <float>,
                        "tip speed ratio": <float>,
                        "tip speed ratio curve": [...],
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

type

Controller type

Type choice · **Required** yes

VALUE	WHAT IT DOES	WHEN TO USE IT
`tsr curve`	A target tip-speed ratio driving a torque controller from a lookup curve; the user must specify a tip speed ratio curve.	
`fixed`	For a fixed controller the user must specify an angular velocity and blade pitch angle.	
`schedule`	Sets pitch and rotational speed by interpolating `pitch_schedule` and `speed_schedule` tables against the reference wind speed, and stops the rotor outside their range.	Not available from a validated deck, as the schema does not accept `pitch_schedule` or `speed_schedule`.

Guidance: Choosing a fluid-zone model > Actuator discs (p. 275)

geometry

Turbine geometry

KEY	TYPE	DEFAULT	DESCRIPTION
`centre`	vector (3 floats)	required	Centre coordinates
`inner radius`	float	not set	Inner radius
`outer radius`	float	not set	Outer radius
`normal`	vector (3 floats)	required	Normal vector
`up`	vector (3 floats)	required	Up vector

geometry

```
parameters = {
    "model": {
        "<mesh>": {
            "fluid zones": {
                "<name>": {
                    "geometry": {
                        "centre": <[x, y, z]>,
                        "inner radius": <float>,
                        "outer radius": <float>,
                        "normal": <[x, y, z]>,
                        "up": <[x, y, z]>,
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

`normal`

Normal vector

Type vector (3 floats) · **Required** yes

Guidance: Choosing a fluid-zone model › Actuator discs (p. 275)

discretisation

Turbine discretisation

KEY	TYPE	DEFAULT	DESCRIPTION
type	str	required	Discretisation type
number of elements	int	not set	Number of elements

discretisation

```
parameters = {
    "model": {
        "<mesh>": {
            "fluid zones": {
                "<name>": {
                    "discretisation": {
                        "type": <str>,
                        "number of elements": <int>,
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

model

Turbine model definition

KEY	TYPE	DEFAULT	DESCRIPTION
`kind`	choice	not set	Model kind (propellor or turbine)
`type`	choice	not set	Model type (simple actuator disc or blade-element BET)
`aerofoils`	settings block	not set	Aerofoil polar data keyed by aerofoil name (genuinely open-ended — aerofoil names and curve shapes are user-defined).
`aerofoil positions`	list	not set	Radial position to aerofoil-name mapping, as [position, name] pairs
`blade chord`	list	not set	Blade chord distribution
`blade twist`	list	not set	Blade twist distribution
`number of sections`	int	not set	Number of sections
`number of blades`	int	not set	Number of blades
`power`	float	not set	Fixed power (W)
`power model`	choice	not set	Power model kind
`power curve`	list	not set	Power vs. tip speed ratio curve
`power coefficient`	float, > 0	not set	Fixed power coefficient for the simple propellor model — torque/power are derived from it (zutil.zWind.ForceModels.zPropellorSimpleFixed)
`thrust coefficient`	float	not set	Fixed thrust coefficient
`thrust coefficient curve`	list	not set	Thrust coefficient vs. tip speed ratio curve
`tip loss correction`	choice	not set	Tip loss correction method
`tip loss correction radius`	float	not set	Tip loss correction radius

model

```
parameters = {
    "model": {
        "<mesh>": {
            "fluid zones": {
                "<name>": {
                    "model": {
                        "kind": <choice>,
```

```
                        "type": <choice>,
                        "aerofoils": {...},
                        "aerofoil positions": [...],
                        "blade chord": [...],
                        "blade twist": [...],
                        "number of sections": <int>,
                        "number of blades": <int>,
                        "power": <float>,
                        "power model": <choice>,
                        "power curve": [...],
                        "power coefficient": <float>,
                        "thrust coefficient": <float>,
                        "thrust coefficient curve": [...],
                        "tip loss correction": <choice>,
                        "tip loss correction radius": <float>,
                    },
                    ...
                },
            },
            ...
        },
    },
    ...
}
```

kind

Model kind (propellor or turbine)

Type choice · **Default** not set

VALUE	WHAT IT DOES	WHEN TO USE IT
`propellor`	Treats the disc as a propeller that adds momentum to the flow; with a `simple` fixed model, thrust and power come from `thrust_coefficient` and `power_coefficient` scaled by rotational speed and diameter.	For propellers and other powered rotors.
`turbine`	Treats the disc as a wind turbine that extracts momentum from the flow; with a `simple` fixed model, thrust comes from `thrust_coefficient` on the swept area and power from `power`.	For wind and tidal turbines.

Guidance: Choosing a fluid-zone model › Actuator discs (p. 275)

type

Model type (simple actuator disc or blade-element BET)

Type choice · **Default** not set

VALUE	WHAT IT DOES
`simple`	A simple turbine model, which averages the performance of the turbine over the entire disc, with fixed or curve-driven thrust/power coefficients.
`bet`	A blade element model, which adds the blade/aerofoil description.

Guidance: Choosing a fluid-zone model › Actuator discs (p. 275)

`power model`

Power model kind

Type choice · **Default** not set

VALUE	WHAT IT DOES	WHEN TO USE IT
`curve`	Interpolates power from `power_curve` and thrust coefficient from `thrust_coefficient_curve` against the reference wind speed, and gives zero outside the tabulated range.	For a `simple` turbine model across a range of wind speeds; `kind` is not read.
`fixed`	Uses a single operating point: `thrust_coefficient` with `power` for a turbine, or with `power_coefficient` for a propeller.	For a `simple` model at one operating condition; `kind` must also be set.

Guidance: Choosing a fluid-zone model › Actuator discs (p. 275)

`tip loss correction`

Tip loss correction method

Type choice · **Default** not set

VALUE	WHAT IT DOES	WHEN TO USE IT
`elliptic`	Scales sectional lift and drag by $\sqrt{(1 - r^2)}$, where r is the radius as a fraction of `outer_radius`.	A `bet` model that needs a simple tip loss across the whole span.
`acos-fit`	Scales sectional lift and drag by an arccos fit of the Prandtl factor, which is close to one inboard and falls to zero at the tip.	A `bet` model that needs the loss confined to the outer span.
`acos shift-fit`	Intended as a shifted arccos fit, but the blade element model matches only `acos_shift-fit`, so this value applies no correction.	No effect at present; use `acos-fit`.
`f-fit`	Scales sectional lift and drag by an empirical fit that departs from one only close to the tip.	A `bet` model that needs a correction limited to the last few per cent of span.
`none`	Applies no tip-loss correction.	Equivalent to leaving the key unset.

VALUE	WHAT IT DOES	WHEN TO USE IT
`rstar`	Leaves sections inboard of `tip_loss_correction_radius` uncorrected and scales those outboard by an elliptic factor that falls to zero at the tip.	A `bet` model where the loss should start at a chosen fraction of `outer_radius`.

Guidance: Choosing a fluid-zone model › Actuator discs (p. 275)

porous

Porous media fluid zone.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	always `porous`	Fluid zone type
`cell zones`	list of int	not set	Mesh cell zone numbers whose cells form the porous region, for example “cell zones”: [4]; give this, 'definition', or neither for the whole mesh
`definition`	str	not set	Closed-surface VTP file selecting the porous region cells; without it or 'cell zones' the porous region is the whole mesh
`alpha`	float, >= 0	`1000000000000`	Resistance coefficient (Darcy term)
`c2`	float, >= 0	`0`	Inertial coefficient (Forchheimer term)

porous

```
parameters = {
    "model": {
        "<mesh>": {
            "fluid zones": {
                "<name>": {
                    "type": "porous",
                    "cell zones": [...],
                    "definition": <str>,
                    "alpha": 1000000000000.0,
                    "c2": 0.0,
                },
            },
            ...
        },
    },
    ...
}
```

type

Fluid zone type

Type choice · **Default** always `porous`

Guidance: Choosing a fluid-zone model › Porous Media (p. 281)

`cell zones`

Mesh cell zone numbers whose cells form the porous region, for example “cell zones”: [4]; give this, 'definition', or neither for the whole mesh

Type list of int · **Default** not set · **Applies when** give either 'cell zones' or 'definition', not both

Guidance: Choosing a fluid-zone model › Selecting cells (p. 274)

`definition`

Closed-surface VTP file selecting the porous region cells; without it or 'cell zones' the porous region is the whole mesh

Type str · **Default** not set

Guidance: Choosing a fluid-zone model › Selecting cells (p. 274)

`alpha`

Resistance coefficient (Darcy term)

Type float, >= 0 · **Default** `1000000000000`

Guidance: Choosing a fluid-zone model › Porous Media (p. 281)

`c2`

Inertial coefficient (Forchheimer term)

Type float, >= 0 · **Default** `0`

Guidance: Choosing a fluid-zone model › Porous Media (p. 281)

canopy

Canopy model fluid zone for atmospheric boundary layer simulations with vegetation.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	always `canopy`	
`cell zones`	list of int	not set	Mesh cell zone numbers whose cells form the canopy, for example “cell zones”: [4]; give this, 'definition' or 'field', or neither for the whole mesh
`definition`	str	not set	Closed-surface VTP file selecting the canopy cells; without it, 'cell zones' or 'field' the canopy covers the whole mesh
`field`	str	not set	Field file for canopy definition (alternative to VTP)
`function`	Python function, or a Python function	not set	Function of the cell centres and their wall distances giving the leaf area density (1/m) in every canopy cell, called as f(*, locations, wall_ distances); it returns a list of one leaf area density per location
`cd`	float, >= 0	`0.25`	Drag coefficient for canopy momentum sink
`beta p`	float, >= 0	`0.17`	Canopy turbulence production parameter
`beta d`	float, >= 0	`3.37`	Canopy turbulence dissipation parameter
`ceps 4`	float, >= 0	`0.9`	Canopy C_epsilon_4 turbulence parameter
`ceps 5`	float, >= 0	`0.9`	Canopy C_epsilon_5 turbulence parameter

canopy

```
parameters = {
    "model": {
        "<mesh>": {
            "fluid zones": {
                "<name>": {
                    "type": "canopy",
                    "cell zones": [...],
                    "definition": <str>,
                    "field": <str>,
                    "function": <function>,
                    "cd": 0.25,
                    "beta p": 0.17,
                    "beta d": 3.37,
                    "ceps 4": 0.9,
```

```
                    "ceps 5": 0.9,
                },
            },
            ...
        },
    },
    ...
}
```

type

Type choice · **Default** always `canopy`

Guidance: Choosing a fluid-zone model › Canopy (p. 283)

cell zones

Mesh cell zone numbers whose cells form the canopy, for example “cell zones”: [4]; give this, 'definition' or 'field', or neither for the whole mesh

Type list of int · **Default** not set · **Applies when** give only one of 'cell zones', 'definition' or 'field'

Guidance: Choosing a fluid-zone model › Selecting cells (p. 274)

definition

Closed-surface VTP file selecting the canopy cells; without it, 'cell zones' or 'field' the canopy covers the whole mesh

Type str · **Default** not set

Guidance: Choosing a fluid-zone model › Selecting cells (p. 274)

field

Field file for canopy definition (alternative to VTP)

Type str · **Default** not set

Guidance: Choosing a fluid-zone model › Canopy (p. 283)

function

Function of the cell centres and their wall distances giving the leaf area density (1/m) in every canopy cell, called as f(*, locations, wall_distances); it returns a list of one leaf area density per location

Type Python function, or a Python function · **Default** not set

Guidance: Python functions in the deck › Leaf area density function (p. 220)

abl

Atmospheric boundary layer fluid zone, applied to the whole mesh.

KEY	TYPE	DEFAULT	DESCRIPTION
`type`	choice	always `abl`	Fluid zone type
`roughness length`	float, > 0	not set	Surface roughness length
`friction velocity`	float, > 0	not set	Friction velocity
`reference height`	float, > 0	not set	Reference height
`wind direction`	unit vector (3 floats)	not set	Wind direction vector

abl

```
parameters = {
    "model": {
        "<mesh>": {
            "fluid zones": {
                "<name>": {
                    "type": "abl",
                    "roughness length": <float>,
                    "friction velocity": <float>,
                    "reference height": <float>,
                    "wind direction": <[x, y, z]>,
                },
            },
            ...
        },
    },
    ...
}
```

type

Fluid zone type

Type choice · **Default** always `abl`

Guidance: Choosing a fluid-zone model › Atmospheric Boundary Layer (ABL) (p. 283)

idw transform

Inverse Distance Weighting transform fluid zone.

KEY	TYPE	DEFAULT	DESCRIPTION
`support radius`	float, > 0	`1`	Support radius for interpolation
`fixed zones`	list of int	not set	Zones held in place while the moving zones deform.
`power`	list of float	`[3, 5, 10]`	IDW power parameters
`stencil size`	int, >= 1	not set	Maximum stencil size (number of nearest neighbors)
`n nearest`	int, >= 1	`100`	Number of nearest points (legacy alias for stencil_size)
`deformation distance`	float, > 0	`1`	Maximum deformation distance (IDW specific)
`blending stiffness`	float, 0 to 1	not set	Blending stiffness parameter
`zones`	list of zone ids	required	Boundary zones the morph moves.
`func`	Python function, or a Python function	required	Function of the node position giving the deformed position of every surface node, called as f(*, location); it returns the deformed position as three numbers
`type`	choice	always `idw transform`	

idw transform

```
parameters = {
    "model": {
        "<mesh>": {
            "fluid zones": {
                "<name>": {
                    "support radius": 1.0,
                    "fixed zones": [...],
                    "power": [3.0, 5.0, 10.0],
                    "stencil size": <int>,
                    "n nearest": 100,
                    "deformation distance": 1.0,
                    "blending stiffness": <float>,
                    "zones": [...],
```

```
                    "func": <function>,
                    "type": "idw transform",
                },
            },
            ...
        },
    },
    ...
}
```

`support radius`

Support radius for interpolation

Type float, > 0 · **Default** `1`

Guidance: Fluid–structure interaction > Interpolation method (p. 320)

`power`

IDW power parameters

Type list of float · **Default** `[3, 5, 10]`

Guidance: Fluid–structure interaction > Interpolation method (p. 320)

`func`

Function of the node position giving the deformed position of every surface node, called as f(*, location); it returns the deformed position as three numbers

Type Python function, or a Python function · **Required** yes

Guidance: Python functions in the deck > Mesh deformation function (p. 221)

`type`

Type choice · **Default** always `idw transform`

Guidance: Fluid–structure interaction > Mesh deformation (p. 320)

rbf transform

Radial Basis Function transform fluid zone.

Full RBF when base_point_fraction=1.0 (default), Multiscale RBF when base_point_fraction < 1.0.

KEY	TYPE	DEFAULT	DESCRIPTION
`support radius`	float, > 0	`1`	Support radius for interpolation
`fixed zones`	list of int	not set	Zones held in place while the moving zones deform.
`basis`	choice	`c2`	RBF function type
`base point fraction`	float, > 0 and < = 1	`0.2`	Fraction of points to use as base points. 1.0 = full RBF, < 1.0 = multiscale RBF.
`zones`	list of zone ids	required	Boundary zones the morph moves.
`func`	Python function, or a Python function	required	Function of the node position giving the deformed position of every surface node, called as f(*, location); it returns the deformed position as three numbers
`type`	choice	always `rbf transform`	

rbf transform

```
parameters = {
    "model": {
        "<mesh>": {
            "fluid zones": {
                "<name>": {
                    "support radius": 1.0,
                    "fixed zones": [...],
                    "basis": "c2",
                    "base point fraction": 0.2,
                    "zones": [...],
                    "func": <function>,
                    "type": "rbf transform",
                },
            },
            ...
        },
    },
    ...
}
```

support_radius

Support radius for interpolation

Type float, > 0 · **Default** 1

Guidance: Fluid–structure interaction > Interpolation method (p. 320)

basis

RBF function type

Type choice · **Default** c2

VALUE	WHAT IT DOES	WHEN TO USE IT
tps	A thin-plate spline in the distance scaled by support_radius; it is never zero, so the system is dense.	Small surfaces where a global, smooth deformation is wanted.
thin_plate_spline	Not implemented under this name; the run stops with an unknown-basis error.	Never; the thin-plate spline is tps.
multiquadric	Not implemented; the run stops with an unknown-basis error.	Never; use c2.
inverse_multiquadric	Not implemented; the run stops with an unknown-basis error.	Never; use c2.
gaussian	A Gaussian whose width is support_radius; it is never zero, so every pair of points interacts and the system is dense.	Small surfaces only, as the dense system grows with the square of the number of points.
linear	Not implemented; the run stops with an unknown-basis error.	Never; use c2.
cubic	Not implemented; the run stops with an unknown-basis error.	Never; use c2.
quintic	Not implemented; the run stops with an unknown-basis error.	Never; use c2.
c0	A Wendland C0 function that is zero beyond support_radius; the deformation is continuous but its slope is not.	Rarely; c2 gives a smoother deformation with the same sparsity.
c2 (default)	A Wendland C2 function that is zero beyond support_radius, so only nearby points interact and the system is sparse.	The default, and the appropriate choice for most cases.
c4	Selects a Wendland C4 function, but the matrix assembly and the volume deformation evaluate c2, so the interpolation is inconsistent.	Not at present; use c2.
c6	Not implemented; the run stops with an unknown-basis error.	Never; use c2.

Guidance: Fluid–structure interaction › Interpolation method (p. 320)

base point fraction

Fraction of points to use as base points. 1.0 = full RBF, < 1.0 = multiscale RBF.

Type float, > 0 and < = 1 · **Default** `0.2`

Guidance: Fluid–structure interaction › Interpolation method (p. 320)

func

Function of the node position giving the deformed position of every surface node, called as f(*, location); it returns the deformed position as three numbers

Type Python function, or a Python function · **Required** yes

Guidance: Python functions in the deck › Mesh deformation function (p. 221)

type

Type choice · **Default** always `rbf transform`

Guidance: Fluid–structure interaction › Mesh deformation (p. 320)

transforms

Placement and motion of the mesh. Coordinates go through 'scale', then 'transform', then 'motion'.

KEY	TYPE	DEFAULT	DESCRIPTION
`scale`	list of float	`[1, 1, 1]`	Factors that multiply the mesh file's x, y and z coordinates, for example to convert inches to metres.
`transform`	list or value or Python function, or a Python function	not set	Static placement of the scaled mesh, applied once when the mesh is read: a 4x4 affine matrix whose translation part is in metres, or a function of the scaled node position, called as f(*, location); it returns the placed position as three numbers.
`motion` (p. 171)	choice of settings blocks		Rigid motion of the placed mesh, or the frame its flow is solved in.

transforms

```
parameters = {
    "model": {
        "<mesh>": {
            "transforms": {
                "scale": [1.0, 1.0, 1.0],
                "transform": [...],
                "motion": {...},
            },
            ...
        },
    },
    ...
}
```

scale

Factors that multiply the mesh file's x, y and z coordinates, for example to convert inches to metres. Coordinates go through 'scale', then 'transform', then 'motion'.

Type list of float · **Default** `[1, 1, 1]`

`transform`

Static placement of the scaled mesh, applied once when the mesh is read: a 4x4 affine matrix whose translation part is in metres, or a function of the scaled node position, called as f(*, location); it returns the placed position as three numbers. Coordinates go through 'scale', then 'transform', then 'motion'.
Type list or value or Python function, or a Python function · **Default** not set

motion

Rigid motion of the placed mesh, or the frame its flow is solved in. Its axis, origin and velocity are in placed coordinates. Coordinates go through 'scale', then 'transform', then 'motion'.

KEY	TYPE	DEFAULT	DESCRIPTION
`moving mesh`	bool	`False`	True moves the nodes through an unsteady run.
`ramp time`	float, > 0 s	not set	Physical time over which a moving mesh rises linearly from rest to its full rate; unset starts at full rate.
`axis`	unit vector (3 floats)	required	*rotating only* · Axis of rotation.
`origin`	vector (3 floats), in m	required	*rotating only* · A point on the axis of rotation.
`omega`	float, in rad/s	required	*rotating only* · Angular velocity, positive right-handed about 'axis'.
`type`	choice	required	
`rotate nodes`	bool	`False`	*rotating only* · Rotate the mesh nodes every time step, instead of carrying the rotation as an angle that the overset mapper and sliding supermesh apply.
`velocity`	vector (3 floats), in m/s	required	*translating only* · Full translation velocity.

motion

```
parameters = {
    "model": {
        "<mesh>": {
            "transforms": {
                "motion": {
                    "moving mesh": False,
                    "ramp time": <float>,
                    "axis": <[x, y, z]>,
                    "origin": <[x, y, z]>,
                    "omega": <float>,
                    "type": <choice>,
                    "rotate nodes": False,
                    "velocity": <[x, y, z]>,
                },
                ...
            },
            ...
        },
    },
```

```
    ...
}
```

moving mesh

True moves the nodes through an unsteady run. False keeps the mesh still and solves the flow in the moving frame (MRF).

Type bool · **Default** `False` · **Applies when** the model has an overset or sliding interface and the run is unsteady and compressible. For a steady rotor, or an incompressible run, set 'moving mesh': False to solve the flow in the moving frame.

ramp time

Physical time over which a moving mesh rises linearly from rest to its full rate; unset starts at full rate.

Type float, > 0 s · **Default** not set · **Applies when** 'moving mesh' is True. A frame runs at its full rate from the first step: remove 'ramp time'.

type

Type choice · **Required** yes

VALUE	WHAT IT DOES
`rotating`	
`translating`	

Guidance: Placing and moving a mesh (p. 365)

rotate nodes

Rotate the mesh nodes every time step, instead of carrying the rotation as an angle that the overset mapper and sliding supermesh apply.

Type bool · **Default** `False` · **Applies when** 'moving mesh' is True. A frame never moves the nodes: remove 'rotate nodes'.

Output variables

Every variable zCFD can write, grouped by family. Name volume variables in `volume variables` and surface variables in `surface variables`, both under output settings › solution. Each row says whether the variable is written to volume output, surface output or both.

Flow · Primitive solution variables.

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
`pressure`	Static pressure (Pa)	volume and surface	`p`
`density`	Density (kg/m^3)	volume and surface	`rho`
`velocity`	Velocity vector (m/s)	volume and surface	`V`, `v`
`temperature`	Temperature (K)	volume and surface	`T`, `t`
`mach`	Mach number	volume and surface	`m`
`gauge_pressure`	Pressure relative to the reference value (Pa)	volume and surface	
`V_avg`	Running average of the velocity vector (m/s)	volume and surface	
`V_rms`	RMS of the velocity vector (m/s)	volume and surface	
`p_avg`	Running average of the static pressure (Pa)	volume and surface	
`p_rms`	RMS of the static pressure (Pa)	volume and surface	
`T_avg`	Running average of the temperature (K)	volume and surface	
`T_rms`	RMS of the temperature (K)	volume and surface	

Derived · Quantities computed from the solution.

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
`potentialtemperature`	Potential temperature (K)	volume and surface	
`cp`	Pressure coefficient	volume and surface	
`totalcp`	Total pressure coefficient	volume and surface	
`ek`	Kinetic energy contained in the cell, 0.5 rho IVI^2 times the cell volume (non-dimensional)	volume and surface	

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
enstrophy	Enstrophy - squared magnitude of vorticity	volume and surface	
velocitynd	Non-dimensional velocity	volume and surface	
vorticity	Vorticity vector (1/s)	volume	
vorticitynd	Non-dimensional vorticity	volume	
Qcriterion	Q criterion - vortex core identification	volume	
Qcriterionnd	Non-dimensional Q criterion	volume	
helicity	Helicity	volume	
helicitynd	Non-dimensional helicity	volume	
viscosity	Dynamic viscosity (Pa.s)	volume and surface	mu
nu	Kinematic viscosity (m^2/s)	volume and surface	

Gradients · Gradient vectors and tensors.

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
pressuregrad	Pressure gradient vector	volume and surface	
densitygrad	Density gradient vector	volume and surface	
velocitygrad	Velocity gradient tensor	volume and surface	
temperaturegrad	Temperature gradient vector	volume and surface	
turbulencekineticenergygrad	Turbulence kinetic energy gradient	volume	
turbulenceeddyfrequencygrad	Turbulence eddy frequency gradient	volume	

Turbulence · Turbulence model quantities.

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
eddy	Eddy viscosity (Pa.s)	volume and surface	
reynoldstress	Reynolds stress tensor	volume	
ti	Turbulence intensity	volume	
turbulencekineticenergy	Turbulence kinetic energy (J/kg)	volume and surface	

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
`turbulenceeddyfrequency`	Turbulence eddy frequency (1/s)	volume and surface	
`nuTilde`	Spalart-Allmaras working variable	volume and surface	
`menterf1`	Menter SST blending function	volume	
`lesregion`	DES/DDES/IDDES region flag	volume and surface	
`viscouslengthscale`	Minimum distance between a cell and its neighbours	volume	
`turbulenceintermittency`	Transition model intermittency	volume	
`transitionreynoldsnumber`	Transition momentum-thickness Reynolds number	volume	

Wall · Wall and boundary-layer quantities.

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
`walldist`	Distance to the nearest wall (m)	volume and surface	
`walldistancezone`	Wall-distance zone id	volume and surface	
`yplus`	Non-dimensional wall distance	surface	
`ut`	Friction velocity at the wall	surface	
`cf`	Skin friction coefficient	surface	
`roughness`	Wall roughness height	surface	
`heatflux`	Wall heat flux (W/m^2), positive into the fluid	surface	
`twall`	Wall temperature (K)	surface	
`velocitynearnd`	Non-dimensional velocity at the near-wall cell	volume and surface	

Forces & moments · Per-face force and moment contributions.

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
`pressureforce`	Pressure component of force on a face	surface	
`pressuremoment`	Pressure component of moment on a face	surface	
`pressuremoment<axis>`	Pressure moment about the `<axis>` axis. Replace `<axis>` with x, y or z.	surface	
`frictionforce`	Skin friction component of force on a face	surface	
`frictionmoment`	Skin friction component of moment on a face	surface	

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
`frictionmoment<axis>`	Skin friction moment about the `<axis>` axis. Replace `<axis>` with x, y or z.	surface	

Mesh & geometry · Cell and zone geometry.

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
`zone`	Boundary/mesh zone identifier	volume and surface	
`centre`	Cell or face centre location	volume and surface	
`cell_centre`	Cell centre location	volume and surface	
`cellvolume`	Cell volume	volume	
`cell_volupdate`	Cell volume update rate (moving mesh)	volume	
`cellcolour`	Partition/colouring index used by the mesh partitioner	volume	
`cell_velocity`	Velocity of the mesh at each cell centre from mesh motion (m/s)	volume	
`cellzone`	Cell zone number of each cell, as stored in the mesh file	volume	
`GlobalCellId`	Global cell index, as named in the invalid-cell log lines	volume	

Solver · Residuals, limiters and numerical diagnostics.

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
`timestep`	Local timestep	volume and surface	
`turbulencetimestep`	Local turbulence timestep	volume	
`densitylimiter`	Density limiter value	volume and surface	
`klimiter`	Turbulence kinetic energy limiter value	volume and surface	
`omegalimiter`	Turbulence eddy frequency limiter value	volume and surface	
`sagradientlimiter`	Spalart-Allmaras gradient limiter value	volume	
`celllimiter`	Cell-based reconstruction limiter value for density (1 = unlimited); written by the 'barth jespersen', 'michalak gooch' and 'venkatakrishnan' limiters	volume	

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
`celllimiterpressure`	Cell-based reconstruction limiter value for pressure (1 = unlimited); written by the 'barth jespersen', 'michalak gooch' and 'venkatakrishnan' limiters	volume	
`roelowdissipation`	Roe low-dissipation blending factor	volume	
`FailCell`	1 in a cell whose solution is invalid, 0 elsewhere	volume	
`badcells`	Largest non-orthogonality among the cell's first-order faces under mesh quality remediation	volume	
`sponge`	Sponge layer damping coefficient	volume	
`MomentumSource`	Momentum source term (fluid zone sources)	volume	
`timestepratio`	Local time step ratio	volume and surface	
`artificial_viscosity`	Artificial viscosity applied	volume	
`cellorder`	Spatial order used in each cell	volume	
`lambdavariable`	Spectral radius of the flux Jacobian	volume	
`lescfl`	CFL on the LES length scale used by the DES CFL shield: the larger of the grid relative transport CFL and the eddy turnover CFL	volume and surface	
`lesturnover`	Eddy turnover contribution to lescfl on its own, so the binding mechanism is visible	volume and surface	
`fdshield`	DES/DDES/IDDES shielding function f_d: 1 in resolved LES, 0 in the shielded RANS region	volume and surface	
`var_<n>`	Solution value of solved equation `<n>`. Replace `<n>` with the number of a solved equation, counting from 1. A case solves 5 equations with euler, viscous, les, incomp viscous or incomp les, 7 with rans or incomp rans, 6 with the `sa-neg` turbulence model and 9 with the `sst-transition` turbulence model.	volume and surface	
`resvar_<n>`	Residual of solved equation `<n>`. Replace `<n>` with the number of a solved equation, counting from 1. A case solves 5 equations with euler, viscous, les, incomp viscous or incomp les, 7 with rans or incomp rans, 6 with the `sa-neg` turbulence model and 9 with the `sst-transition` turbulence model.	volume and surface	
`resshare_<n>`	Share of the total squared residual held by each cell, equation `<n>`. Sums to one over the mesh; threshold at 1/Neff (reported when output settings › report › residual statistics is on) to isolate the cells carrying the residual. Replace `<n>` with the number of a solved equation, counting from 1. A case solves 5 equations with euler, viscous, les, incomp viscous or incomp les, 7 with rans or incomp rans, 6 with the `sa-neg` turbulence model and 9 with the `sst-transition` turbulence model.	volume and surface	

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
`vargrad_<n>`	Gradient of solved equation `<n>`. Replace `<n>` with the number of a solved equation, counting from 1. A case solves 5 equations with euler, viscous, les, incomp viscous or incomp les, 7 with rans or incomp rans, 6 with the `sa-neg` turbulence model and 9 with the `sst-transition` turbulence model.	volume and surface	

Overset · Overset data.

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
`overset`	Overset blanking/donor status	volume and surface	
`parent`	Parent cell/zone id (overset)	volume	

FSI & modal · Modal deformation and mesh motion.

NAME	WHAT IT IS	WRITTEN TO	ALSO ACCEPTED AS
`modaldeformation`	Modal deformation of the surface	surface	
`meshvelocity`	Mesh velocity from mesh motion (m/s)	volume and surface	
`modeshapedisplacement_<n>`	Displacement of mode shape `<n>`. Replace `<n>` with a number from 1 to 14.	surface	
`modeshaperotation_<n>`	Rotation of mode shape `<n>`. Replace `<n>` with a number from 1 to 10.	surface	

Command line invocation and flags

To run zCFD on the command line:

```
run_zcfd -c <case> [-n <ranks>] [-t <ranks_per_node>] [-o <threads>] [-d cpu|gpu]
```

FLAG	DEFAULT	MEANING
`-c`, `--case`	required	the control file, `<case>.py`; the `.py` extension may be left out
`-n`, `--ntask`	1	the number of MPI ranks, and so the number of mesh partitions
`-t`, `--tpn`	1	the number of MPI ranks on each node
`-o`, `--otpt`	set by zCFD	the number of OpenMP threads for each rank, exported as `OMP_NUM_THREADS`
`-d`, `--device`	detected for each rank	`cpu` or `gpu`
`-v`, `--version`		print the zCFD version and exit
`-h`, `--help`		print the usage and exit

`zcfd` is a shorter name for the same command and takes the same flags.

The control file names each mesh with the `mesh` (p. 97) key of its entry under `model`. See Running zCFD (p. 212) for worked examples of these flags, and Running zCFD § Multiple Meshes and Overset Cases (p. 213) for a case with more than one mesh, which is run with the same command.

Licence check

```
run_zcfd --check-license
```

checks out the zCFD licence, reports each licensed product with its seats, expiry and source, and exits. It takes no case. See Installation and licensing (p. 199).

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 <case>
```

The script checks the control file against the schema. It does not read the mesh.

Batch Queue Submission

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

Example Slurm submission script "run_zcfd.sub":

```
#!/bin/bash
#SBATCH -J zcfd
#SBATCH --output zcfd.out
#SBATCH --nodes 2
#SBATCH --ntasks 4
#SBATCH --exclusive
#SBATCH --time=10:00:00
#SBATCH --gres=gpu:2
#SBATCH --cpus-per-task=16
source /INSTALL_LOCATION/zCFD-version/bin/activate
run_zcfd -n 4 -t 2 -d gpu -c <case>
```

This system has 2 nodes, each with 2 GPUs. One task is allocated per GPU, giving a total of 4 tasks. The value given to `-n`, in this case 4, on the last line should match Slurm 'ntasks' on line 5, and `-t` gives the 2 tasks on each node. The case is run in 'exclusive' mode which means that zCFD has uncontested use of the devices. Each node has a 64-core CPU, giving 16 CPU cores for each of the 4 tasks. The 'gres' line tells Slurm that there are 2 GPUs available on each node.

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

```
sbatch run_zcfd.sub
```

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

> **NOTE**
>
> Users unfamiliar with the batch submission parameters in a specific system should consult the system administrator.

Environment variables

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

VARIABLE	DEFAULT	DESCRIPTION
ZCFD_GPUDIRECT	1 on GPU ranks	Set to 1 to pass GPU buffers directly to a GPU-aware MPI: Open MPI, Platform MPI or Cray MPI. When the `nv_peer_mem` kernel module is not loaded, zCFD sets it to 0 on a multi-node run, and on a 16-GPU node running more than eight ranks.
ZCFD_PRE_LAUNCH	unset	The path to a script that is sourced just before the solver starts, so that it can override the environment zCFD has set up.
CUDA_VISIBLE_DEVICES	unset	Restricts the GPUs zCFD can use. Typically used to exclude an incompatible GPU or the display GPU.
ZCFD_DISABLE_PINNED	unset	When set, GPU transfer buffers are allocated in ordinary host memory instead of pinned memory. Set it under WSL on Windows, where CUDA limits the amount of pinned memory.
ZCFD_NCCL	unset	Set to 1 to use the NVIDIA NCCL library for parallel exchanges, in builds that include NCCL.
PMPI_CUDA	set by zCFD under Platform MPI	Enables GPU-aware MPI transfers with Platform MPI (custom builds only), when `ZCFD_GPUDIRECT` is 1.
MPICH_GPU_SUPPORT_ENABLED	unset	Set to 1 to enable GPU-aware MPI transfers with Cray MPI (custom builds only), when `ZCFD_GPUDIRECT` is 1.
ZCFD_DISPATCH_SYNC	unset	When set, every kernel launch is synchronous.
ZCFD_DEV_OVERSUBSCRIBE_GPU	unset	**Experimental.** When set, several MPI ranks may share one GPU. When unset, each MPI rank is given its own GPU. Intended for debugging and testing with limited GPU resources.

Parallel invocation

> **WARNING**
>
> zCFD configures its own parallel launch: `run_zcfd` selects the MPI transport, the thread affinity and the GPU for each rank. The settings on this page are for advanced users and are given for reference. For help configuring zCFD on a cluster, contact `zcfd@zenotech.com`.

See Running zCFD § Running in parallel (p. 215) for the Hybrid MPI/OpenMP, Full MPI and GPU parallel execution concepts. This page covers running across multiple machines and the lower-level flags and environment variables that control MPI transport selection and thread affinity.

Running across multiple machines with a cluster scheduler

There are many cluster scheduling systems but commonly used ones are Slurm, PBS Pro or GridEngine. Please refer to either the cluster scheduling software or your local HPC systems' documentation for how to submit jobs to the appropriate nodes on your system as configuration varies between machines.

Intel MPI has tight integration to most commonly used scheduling systems and so will typically autodetect the nodes to run on.

Running across multiple machines without a cluster scheduler

If your cluster does not have a scheduling system then **I_MPI_HYDRA_HOST_FILE** can be set in the environment to point at a file containing the list of hostnames that zCFD will launch on. For example:

```
export I_MPI_HYDRA_HOST_FILE=~/hosts.file
```

> **NOTE**
>
> For this to run you need to be able to ssh without a password to each of the machines listed.

An example contents of a hosts file is shown below:

```
compute01:2
compute02:2
compute03:1
```

This file specifies three hosts, compute01, compute02, compute03 where 2 ranks will be launched on each of compute01 and compute02 and 1 rank will be launched on compute03.

Alternatively, the hosts can be listed without rank counts, in which case processes are launched round robin through the list.

```
compute01
compute02
compute03
```

Libfabric provider selection

zCFD autodetects the provider to use for MPI communication. It tries the following providers in order, and uses the first that `fi_info` lists and that a one-rank MPI test accepts:

```
efa
mlx
verbs
tcp
udp
sockets
```

When **FI_PROVIDER** is set, autodetection is skipped.

Using cluster provided libfabric

By default zCFD will use the version of libfabric that ships with Intel MPI. For some advanced use cases you may wish to make use of preinstalled libfabric. This may be if you have a custom network driver or a specific configuration on your cluster. To enable this, set the **I_MPI_OFI_LIBRARY_INTERNAL** environment variable to 0 before activating the zCFD environment. The cluster's `fi_info` must be on the path.

```
export I_MPI_OFI_LIBRARY_INTERNAL=0
```

Pre launch script

On some systems the defaults setup by zcfd may not be optimal and so a custom script to modify the environment can be injected by setting the environment variable **ZCFD_PRE_LAUNCH** to the path to a script. This script will be sourced from within a bash environment just before the solver is launched.

An example use case for this is to bind a GPU to a specific network interface on a multi-homed network system.

OpenMP affinity

By default zCFD binds its OpenMP threads to cores, setting **OMP_PROC_BIND** to `true` and **OMP_PLACES** to `cores`. When a node runs more MPI ranks than it has NUMA nodes, **OMP_PLACES** is set to `sockets` instead.

Other MPI implementations

Where Intel MPI does not work on a platform, zCFD can be built against a specific implementation of MPI. This is a custom deployment for your facility; contact Zenotech if you need it.

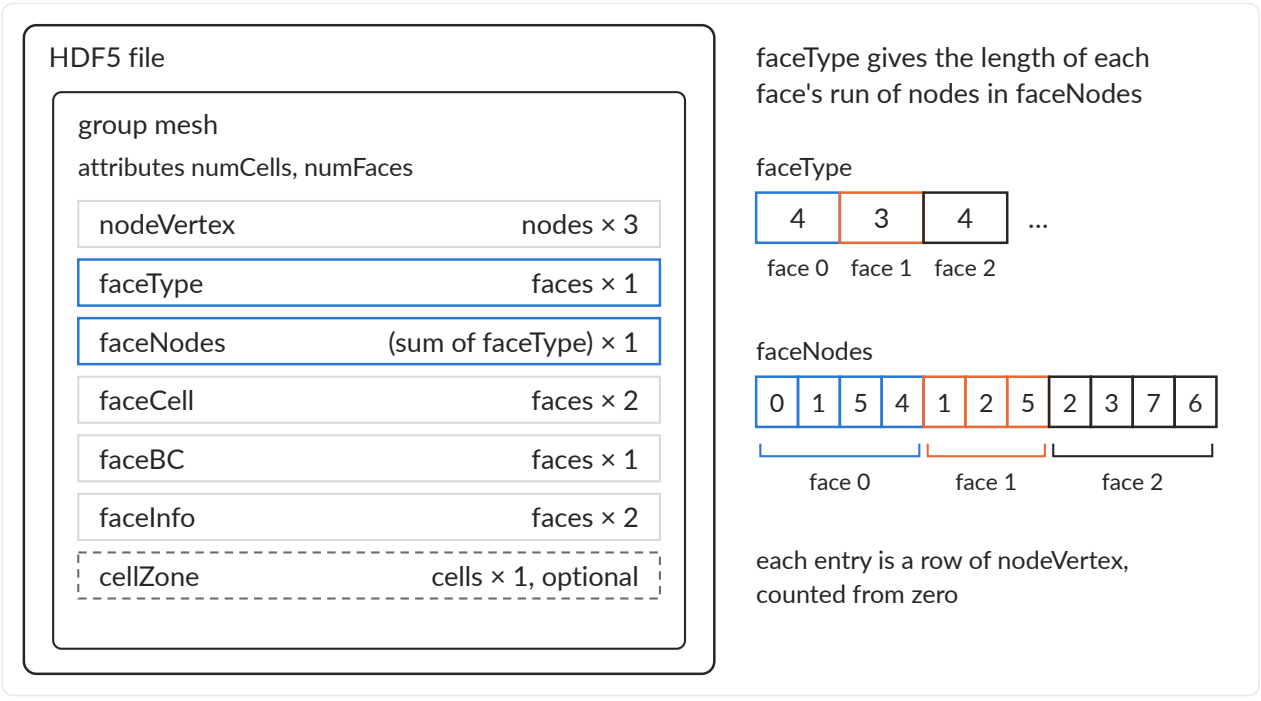
Running at large scale

As zCFD is partly written in python, the install consists of a large quantity of small files, which on a large scale cluster can have a negative impact on the cluster filesystem during startup. To mitigate this, the code can be staged into either local storage or memory on the compute nodes using MPI to broadcast the data.

Contact us if you want to run in this mode.

Mesh file

A zCFD mesh is an HDF5 file holding one group, `mesh`. The group carries two attributes, `numCells` and `numFaces`, and the datasets below. Every index counts from zero. Preparing a mesh (p. 201) describes how to convert a mesh into this format.



The group mesh in an HDF5 file, with its two attributes and seven datasets; faceType gives the number of nodes of each face, and so splits faceNodes into one run of node indices per face

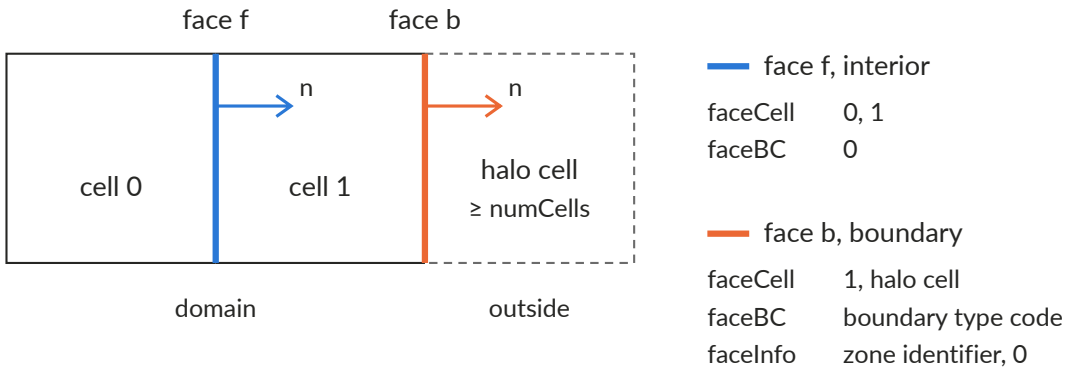
Datasets

DATASET	SHAPE	CONTENTS
`nodeVertex`	nodes × 3, float	the coordinates of each node, in mesh units
`faceType`	faces × 1, integer	the number of nodes of each face
`faceNodes`	(sum of `faceType`) × 1	the node indices of each face in turn, in face order
`faceCell`	faces × 2, integer	the left and right cell of each face
`faceBC`	faces × 1, integer	the boundary type code of each face, `0` for an interior face
`faceInfo`	faces × 2, integer	the zone identifier of each face in the first column; the converters write `0` in the second
`cellZone`	cells × 1, integer	optional; the cell zone number of each cell, which a fluid zone's `cell zones` selects

Cells and faces

The left cell of every face is a real cell, numbered below `numCells`. The right cell of an interior face is also a real cell. The right cell of a boundary face is a halo cell: halo cells are numbered from `numCells` upwards, one for each boundary face.

The node order of a face defines its normal by the right-hand rule. That normal must point out of the left cell and into the right cell, so on a boundary face it points out of the domain. A face wound the other way gives its cells wrong volumes.



An interior face between cells 0 and 1, with its normal pointing from cell 0 into cell 1, and a boundary face between cell 1 and a halo cell numbered from numCells, with its normal pointing out of the domain

A boundary condition in the control file names the zones it applies to by the zone identifiers in `faceInfo`.

Boundary type codes

`faceBC` holds `0` for an interior face and, for a boundary face, the code of its boundary type:

CODE	BOUNDARY TYPE
`0`	interior face
`3`	wall
`4`	inflow
`5`	outflow
`7`	symmetry
`9`	farfield
`18`	periodic
`20`	overset
`22`	sliding

The `type` of the boundary condition that names a zone decides how the solver treats that zone; the code in the mesh does not.

PART 2

Guide

How to install zCFD, set up and run a case, choose its models and schemes, and diagnose problems.

What is zCFD?

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 contains a high performance core, executed via a flexible Python based wrapper. zCFD will run on unstructured polyhedral meshes from a variety of sources, and will export data in VTK HDF format (.vtkhdf) for efficient post-processing with tools such as ParaView. zCFD offers an easier route to coupling multi-disciplinary capabilities and integrating third-party data sources.

Features include both compressible and incompressible finite-volume 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 reliability. 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 uses MPI to manage execution over multiple 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 also includes zm3, or zMesh-Cubed, the Zenotech Cartesian cut-cell mesh generator. This allows users to rapidly generate meshes for CFD from STL files, with minimal user cleanup effort required.

How the deck works

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 using the rich array of python libraries available.

The guide uses four names for this file: the control file, the control dictionary, the input deck and the deck. They are interchangeable.

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

```
parameters = {
    "config_version": 2,
    "solver": {....},
    "model": {....},
}
```

`config_version` is a required integer identifying the schema the rest of the dictionary is validated against. `solver` holds the settings shared across the whole simulation; `model` holds one entry per mesh.

zCFD validates the `parameters` dictionary at run time against a pydantic schema for each key. Values that should be integers and floats are coerced to the correct type where possible; values that should be strings, booleans, lists or dictionaries raise an error if given the wrong type. Which keys are valid depends on the equation set and solver type chosen. A script is provided to check the validity of a control file before submitting a job — see invocation and flags (p. 179).

Required, optional and default keys

Each key in the reference is one of three kinds:

- **Required.** The deck must set it, or validation fails. The reference marks it *required*.
- **Optional, with a default.** Left out, the key takes the default the reference shows, such as `default 1.4`.
- **Optional, with no default.** Left out, the feature the key controls is off. The reference shows *not set by default*.

Some keys apply only under other settings: `gravity`, for example, applies only to the incompressible solver. The reference states the condition after *only applies when*. Setting such a key when its condition does not hold fails validation, and a key that is required only under a condition fails validation when that condition holds and the key is missing.

The reference documents the keys available in a zCFD control dictionary and the valid values for each.

```
parameters = {
    "config_version": 2,
    "solver": {
        "solver_settings": {...},
        "equations": {...},
        "fluid_properties": {...},
        "reference_conditions": {"IC_1": {...}},
        "initialisation": {...},
        "time_settings": {...},
        "convergence_control": {...},
        "numerical_scheme": {...},
        "output_settings": {
            "report": {...},
            "solution": {...},
        },
    },
    "model": {
        "<mesh_name>": {
            "mesh": "<path.h5>",
            "boundary_conditions": {"BC_1": {...}},
            "fluid_zones": {"FZ_1": {...}},
            "transforms": {...},
        },
    },
}
```

- `solver`: `solver_settings` (p. 11), `equations` (p. 18), `fluid_properties` (p. 37), `reference_conditions` (p. 39), `initialisation` (p. 47), `time_settings` (p. 51), `convergence_control` (p. 56), `numerical_scheme` (p. 74), `output_settings` (p. 84) (`report` (p. 84), `solution` (p. 84))
- `model`: `mesh` (p. 97), `boundary_conditions` (p. 98), `fluid_zones` (p. 143), `transforms` (p. 169)

Settings for one mesh

Four solver blocks may also be set inside a mesh's entry: `initialisation`, `convergence_control`, `numerical_scheme` and `output_settings`. A mesh's block lists only what differs on that mesh; everything it leaves out comes from the solver's block of the same name.

```
"model": {
    "rotor": {
        "mesh": "rotor.h5",
        "boundary_conditions": {"BC_1": {...}},
        "numerical_scheme": {"order": "first"},
    },
},
```

Here the rotor mesh runs at first order and keeps every other numerical scheme setting from the solver's block. A mesh that leaves a block out uses the solver's block as it stands.

- Settings merge one by one, at every depth.
- A list, such as the variables to write, replaces the solver's list whole.

- Named forces, monitors and mass flow reports are added to the solver's.
- A sub-block that chooses a different kind, through its `name`, `type` or `kind`, replaces the solver's sub-block whole. A time scheme with a different `name` keeps none of the solver's scheme settings.
- The incompressible coupling scheme governs the whole run, so a mesh cannot set it.

A mistake in a setting the mesh inherits is reported once, at the solver's block. A mistake in a setting the mesh writes is reported at the mesh, even when the solver's block repeats it. When the deck of a running case is edited, a change that a mesh inherits is logged once, against the solver.

Converting an older control file

A control file written for an earlier release of zCFD does not carry `"config_version": 2` in its `parameters` dictionary. Such a file is rejected at start-up, and by `validate_input`. The message names the command that converts it, `update_input`.

The converter reads the old file and writes a new one in the current format. It does not change the original, and it does not run on the solver's start-up path.

Running the converter

```
update_input wing.py --mesh wing.h5
```

The `--mesh` option is optional, but the draft is more complete with it. The mesh supplies three things:

- the name of the model, taken from the file name, so that the model is labelled `wing`;
- the path to the mesh, set as `"mesh": "wing.h5"`. An absolute path is written as given. A relative path is written relative to the converted file when the mesh lies in or below its directory, and as an absolute path otherwise;
- the zone lists for any boundary condition that the old file gives only as a type code, for which the converter reads the zone numbers of that type from the mesh.

Without a mesh, the model name, the mesh path and each such boundary condition are left for you to decide.

What the converter writes

Two files appear beside the original:

- `wing_v2.py`, the converted control file;
- `wing_v2.conversion.md`, a report of every decision the converter made.

The `parameters` dictionary is rewritten, together with the `def` line of each function it uses. Everything else in the file is kept as written: imports, function bodies, constants and comments. A name used inside the dictionary, such as a time step held in a variable, stays a name in the converted file.

zCFD calls every function in the deck with keyword arguments only. The converter therefore gives each function a `def` line that names those keywords after a `*`. A mesh deformation function becomes:

```
python file=wing.py def bend(x, y, z):     return (x, y, z + 0.01 * x * x)
```

```
python file=wing_v2.py def bend(*, location):     x, y, z = location     return (x, y, z + 0.01 * x * x)
```

The rest of the body is unchanged. The report gives each function's old and new `def` line once, with every key that uses it, and what the body returns where the converter can tell. A body that still returns an older `{"v1": x, "v2": y, "v3": z}` dict needs no change, since that form is still accepted.

Where the arguments cannot be matched to the keywords with certainty, or other code in the file calls the function, the function is left as written and becomes a DECIDE item that gives the `def` line to write and the value to return, such as `rewrite bend as def bend(*, location), returning the deformed position as three numbers`. Rewrite the `def` line and the body, then replace the placeholder with the function's name. Python functions in the deck (p. 218) lists what each function receives and returns.

A translation function has no counterpart, since a mesh's motion is not a function. The converter writes a translating `motion` in the model's `transforms`. Where the older file moved the mesh, it sets `"moving mesh": True` and leaves a DECIDE item asking for the `velocity` and the `ramp time`: if the body starts from rest with a linear rise, give the full velocity and the seconds it takes to reach it. Any other profile cannot be expressed. Where the older file did not move the mesh, it sets `"moving mesh": False`, and the DECIDE item asks for the fixed `velocity`. See Placing and moving a mesh (p. 365).

Reading the summary

When it finishes, the converter prints a summary. The figures below are illustrative.

```
$ update_input wing.py --mesh wing.h5
Converting wing.py (config_version 0) to config_version 2

  moved       41 keys
  re-spelled   3 values
  dropped      2 keys   (units: always SI · material: one fluid_properties block)
  DECIDE       2 keys

  1. time marching > scheme > name = "runge kutta", with an unsteady block
     Choose time settings > kind: "dual time stepping" (pseudo-time loop
     each step) or "global timestepping" (explicit march).
  2. BC_3 > ref = 3, with no zone list
     ...

Wrote wing_v2.py (draft: 2 decisions open; replace each DECIDE string)
Wrote wing_v2.conversion.md (every key and where it went)
```

Each key in the original file receives exactly one of four dispositions.

- **Moved.** The key keeps its name and its value, and appears at its new place in the dictionary.
- **Re-spelled.** The key or its value is written differently in the current format, with the same meaning.
- **Dropped.** The current format has no use for the key. The report gives a one-sentence reason for each. The solver always works in SI units, for example, so `units` is dropped.
- **DECIDE.** The converter cannot choose on the engineer's behalf. The draft holds a placeholder string, and the report explains the choice.

The report lists every key under its disposition. It is the reference for anything the summary shows only as a count.

Clearing the DECIDE items

A DECIDE item appears in the draft as a string that names the choice:

```
python file=wing_v2.py "time_settings": {     "type": "unsteady",     "kind": "DECIDE:
'dual time stepping' or 'global timestepping' (see conversion report, item 1)",     ... }
```

The draft will not run as intended until each one is replaced. Most such strings fail validation, but one that stands in for free text, such as the mesh file name, does not. Validate the draft, and search it for the remaining strings:

```
validate_input wing_v2.py
grep -n DECIDE wing_v2.py
```

Each validation error names one open item and the key that holds it. Take the item to the report, read the explanation, replace the string with the value chosen, and validate again. Repeat until the file validates and no DECIDE string remains. An item for a function can leave no string in the file, so work through every numbered item in the report, not only those that the search finds.

For the item above, the two options are as follows.

- `"dual time stepping"` advances physical time in steps, and iterates in pseudo-time within each step until it converges. It suits an unsteady flow that must be time-accurate.
- `"global timestepping"` marches physical time directly, with one update per step and no inner iteration. It suits a flow in which the cost of inner iterations is not justified.

Having chosen, enter the value in place of the placeholder:

```
python file=wing_v2.py "time_settings": {     "type": "unsteady",     "kind": "dual time
stepping",     ... }
```

See Choosing a time marching scheme (p. 237) for the choice between the schemes.

When the original file has errors

The converter first checks the original file against the rules of the release that it was written for. If the file fails, the converter stops and lists the errors. A converted file would carry the faults across.

A file that the original names but that cannot be read from where the converter runs, such as the results of the run to restart from or a field file, does not stop the conversion. The report lists it among its notes. Make sure it is in place before the run.

Correct the original and run the converter again. To convert the file regardless, add `--ignore-legacy-errors`:

```
update_input wing.py --mesh wing.h5 --ignore-legacy-errors
```

The errors then remain in the draft, and may appear again as DECIDE items or as validation errors.

Converting an overset group

An older overset case is run from a file such as `override.py`, which pairs each mesh with its own control file:

```
python file=override.py override = {     "mesh_case_pair": [         ("background.h5",
"background.py"),         ("rotor.h5", "rotor.py"),     ] }
```

Convert the group, not the files it lists:

```
update_input override.py
```

The result is a single control file, `override_v2.py`, with its report beside it. Each pair becomes one model, named after its control file and set to run on its mesh, so the example gives the models `background` and `rotor`. The files the group lists are read but not rewritten. The group names its own meshes, so `--mesh` is not used.

The converter then sorts the settings.

- A setting that every model gives the same value is written once, in `solver`.
- Initialisation, convergence control, the numerical scheme and the output settings may differ from model to model. A block that every model shares is written once, in `solver`. Otherwise every model carries its own block in full, and `solver` holds the settings of the most common block that every model's block also sets. The coupling scheme of an incompressible run is the exception: it holds for the whole run, and is treated as below.
- Any other setting holds for the whole run. Where the models give it different values, it becomes a DECIDE item, and the report lists the value each model gave.
- A sponge layer, or extra overset layers, set in a file that every model shares is applied only to the models that have a farfield or an overset boundary respectively.

Values are written out as numbers in the new file, because each old file's own variables are not carried across. A function is copied into the new file when it needs nothing from its old file except imported modules. Where the old `def` line cannot take the keywords zCFD calls it with, the copy has a keyword-only `def` line in its place. Any other function becomes a DECIDE item: copy it into the new file and write its name in place of the placeholder.

Before running the converted file

Validation shows that the file is well formed. It does not show that the simulation is unchanged. Compare the physics of the converted file with the original before submitting a job.

- Read the dropped keys in the report. Each reason should agree with the intended simulation.
- Check every decision made at a DECIDE item against the intent of the original case.
- Confirm the reference conditions, the boundary conditions and the time settings against the original file.

> **NOTE**
>
> If a file converted without `--ignore-legacy-errors` has no DECIDE items and fails validation, the fault lies in the converter, not in the file. The command exits with an error and asks for the file to be reported to Zenotech.

Installation and licensing

System requirements

COMPONENT	REQUIREMENT
Operating system	64-bit x86 Linux with glibc 2.28 or newer, such as RHEL, Rocky Linux or AlmaLinux 8 or newer, or Ubuntu 20.04 or newer
Windows	Windows 10 or 11, through WSL2
NVIDIA GPU	Compute capability 8.0, 8.6, 8.9, 9.0, 10.0 or 12.x: Ampere, Ada Lovelace, Hopper and Blackwell (see Compute Capability)
NVIDIA driver	580 or newer
CUDA	The CUDA 13.4 runtime is included with zCFD; no CUDA toolkit is needed
MPI	Intel MPI 2021.13 is included with zCFD
Python	Python 3.12 is included with zCFD

On a GPU outside this list, zCFD prints a warning, disables the GPU and runs on the CPU.

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

Linux / Unix

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

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

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

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

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

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

The zCFD environment's Python accepts further packages. With the environment active, `pip install <package>` installs into the zCFD installation, where the solver and any Python functions in a control file

can import it. Packages installed with `pip install --user`, or into a separate virtual environment, are not visible to the solver.

Windows

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

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

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

```
\\wsl$\
```

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

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

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

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

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

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

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

Licence

zCFD requires a licence. To obtain one, contact `zcfd@zenotech.com`. When zCFD starts, it looks for a licence in the following places:

1. Any licence file, such as `zcfd.lic`, in the directory from which the solver is started.
2. Any licence file in the `lic` directory of the zCFD installation, `/<path to zCFD install>/lic/`. This location is searched only when the zCFD environment has been activated.
3. The `RLM_LICENSE` or `zenotech_LICENSE` environment variable.

A licence file is any file whose name ends in `.lic`.

The environment variable can name a licence file:

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

or a licence server, where `port` and `host` are the port number and hostname given in the licence file:

```
export RLM_LICENSE=port@host
```

A system that cannot reach the internet requires a local licence server. The RLM licence server `rlm` and the Zenotech licence server `zenotech` are installed in the `lic` directory of the zCFD installation. See http://www.reprisesoftware.com/admin/software-licensing.php for how to run them.

Checking the licence

To check the licence before submitting a job, run:

```
run_zcfd --check-license
```

zCFD checks out its licence for the installed version, reports it and exits. It needs no case or mesh. A served licence with a seat limit reports:

```
zCFD licence check passed
  zcfd-core       2026.10   4 seats         expires 31-mar-2027   5053@licserver
  zcfd-feat       2026.10   64 seats        expires 31-mar-2027   5053@licserver
  MPI ranks free now: 33
```

`zcfd-feat` needs one seat for each MPI rank, so a job of more ranks than are free waits or fails with error -22. The last column shows the licence server or licence file in use. If the checkout fails, zCFD reports the error code listed below and exits with a non-zero status.

Licence error codes

When a licence cannot be checked out, zCFD reports `Error checking out license` followed by the RLM status value, the product, the version and the number of licences requested, and then stops. The product is `zcfd-core` for the solver itself and ends in `-feat` for a solver feature. A feature is checked out once for each MPI rank.

The table lists the status values most often encountered. For any other value, see the RLM documentation at https://reprisesoftware.com/docs/isv/appendix/appendix-b-rlm-status-values.html or contact us.

ERROR CODE	DESCRIPTION	POSSIBLE REMEDY
-1	No authorisation for product	The licence does not include the product or feature named in the message. Contact us to add it
-3	Licence has expired	Contact us to renew your licence
-4	Wrong host for authorisation	The licence is locked to a different machine. Run on the licensed machine, or contact us to move the licence

ERROR CODE	DESCRIPTION	POSSIBLE REMEDY
-5	Bad key in authorisation	The licence file has been altered or damaged. Restore the file as issued, or contact us for a replacement
-6	Requested version not supported	Your licence is for an older version of zCFD. Contact us to upgrade
-8	Checkout request for too many licences	The licence permits fewer MPI ranks than requested. Run on fewer ranks
-17	Error communicating with server	Check the connection to the licence server, the firewall settings and any endpoint security software
-18	Licence server does not support this	The server holds no licence for this product or version. Check that `RLM_LICENSE` names the correct server
-22	All licences in use	Stop other zCFD runs to free a licence, or wait for them to finish
-37	Licence start date not reached	Your licence is not yet valid. Contact us if you believe this to be in error
-102	Can't read licence data	No readable licence was found. Check that a licence file is in one of the places listed above, that it is readable, or that `RLM_LICENSE` is set
-103	Network error	Check the connection to the licence server. This may require opening a port in the firewall or connecting to the internet
-104	Error writing to network	Check the connection to the licence server, the firewall settings and any endpoint security software
-105	Error reading from network	Check the connection to the licence server, the firewall settings and any endpoint security software
-106	Unexpected response	Check that `RLM_LICENSE` names the correct licence server
-107	HELLO message for wrong server	Check that `RLM_LICENSE` names the correct licence server
-108	Error in private key	The licence file on the server is faulty. Contact us to resolve it
-109	Error signing authorisation	The licence file on the server is faulty. Contact us to resolve it
-110	Internal error	The licence server encountered an error. Contact us if it recurs
-111	Connection refused at server	Check the firewall settings and that `RLM_LICENSE` names the correct licence server
-112	No server to connect to	Check that the licence server is running and that `RLM_LICENSE` names it correctly
-113	Bad communications handshake	Check that the licence server is running and that `RLM_LICENSE` names it correctly
-114	Can't get ethernet address	Check that the user running zCFD has permission to query the network card

During a run, zCFD confirms with the licence server every 30 seconds that the licence is still held. If the server stops responding, zCFD logs `Connection to license server failed retrying` and continues. If the connection is not restored within 10 minutes, zCFD stops with `License heart beat failed`.

Preparing a mesh

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 file system type, such as NFS, GPFS or Lustre. The layout of the file is described in Mesh file (p. 185).

Converting a mesh

The converter for other mesh formats is a utility called *convert_mesh*. This is included in the path set by activating the zCFD command line environment.

To run the utility, use

```
(zCFD): convert_mesh <filename> <new_filename>.h5
```

The utility detects the format from the file extension. The `-f` flag names the format explicitly, as one of `fluent`, `fluentcff`, `su2`, `cgns`, `ugrid`, `foam`, `cba` or `plot3d`:

```
(zCFD): convert_mesh -f <format> <filename> <new_filename>.h5
```

Any arguments given after the two file names are passed to the UGRID, CBA and Plot3D converters, and ignored for the other formats.

Unstructured

An unstructured mesh already stores its cells, faces and boundary zones, so it converts with no options beyond the file names.

Fluent

The converter supports ANSYS Fluent files (`.msh`, `.cas`, `.msh.gz`, `.cas.gz`).

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

This produces the zCFD HDF5 file and a zone helper file, `<new_filename>_zone.py`, which maps the zone names in the Fluent file to zone identifiers. See Zone identifiers.

Fluent CFF

The converter also supports Fluent CFF format files (`.msh.h5`, `.cas.h5`).

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

This produces the zCFD HDF5 file and a zone helper file, `<new_filename>_zone.py`.

SU2

SU2 is an open-source CFD solver. Conversion of 2D meshes is not supported, as they are not handled in zCFD.

```
(zCFD): convert_mesh <filename>.su2 <new_filename>.h5
```

This produces the zCFD HDF5 file and a zone file, `<new_filename>_zones.py`.

CGNS

The converter reads a CGNS file in any of three layouts:

- an unstructured zone of `HEXA_8`, `TETRA_4`, `PYRA_5`, `PENTA_6` or `MIXED` element sections, with `QUAD_4` and `TRI_3` boundary sections;
- an unstructured zone stored face by face, with both an `NGON_n` and an `NFACE_n` section;
- a structured base of one or more blocks, joined by their 1-to-1 connectivity.

Only the first zone of an unstructured file is read. In a structured file, only boundary conditions given as a point range are read; any other is skipped with a warning.

```
(zCFD): convert_mesh <filename>.cgns <new_filename>.h5
```

This produces the zCFD HDF5 file and a zone helper file, `<new_filename>_zone.py`.

UGRID

UGRID is a NASA format for meshes. The file name must follow the UGRID naming convention. The suffix gives the format and the *endianness* of the data (`ascii8`, `b8`, `lb8` or `r8`), which depend on the language, C or FORTRAN, used to write the file. The mapbc file is optional. Given as a third argument, it sets the boundary type code of each zone; without it, every boundary face is coded as a wall.

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

This produces the zCFD HDF5 file.

OpenFOAM

The converter reads an OpenFOAM `polyMesh`, in ASCII or binary, and optionally compressed with `gzip`. Give it the case directory, which holds `constant/polyMesh`, or the `polyMesh` directory itself:

```
(zCFD): convert_mesh <case_directory> <new_filename>.h5
```

This produces the zCFD HDF5 file, and prints the zone identifier given to each boundary patch.

Structured

A structured multiblock file stores each block as an array of grid points, indexed by *i*, *j* and *k*. It holds no faces, and a Plot3D file holds no connectivity or boundary information either. The converter therefore builds the faces itself, merges the nodes that neighbouring blocks share, and pairs the faces where blocks meet. It must also be told which block faces form each zone, which is what the zone file and the options below are for.

A structured CGNS file needs none of these options, since CGNS records its block connectivity and boundary patches; see CGNS.

CBA

The converter supports structured multiblock meshes in the University of Bristol UOBSMB / *flowcode* CBA format (`.blk`). A CBA file is plain text: a header line giving the block count, a symmetry flag and a format version, then each block in turn — a dimension line (`ni nj nk`), the block's grid points (with *i* varying fastest, then *j*, then *k*), and six footer lines, one per face, always in the order `imin, imax, jmin, jmax, kmin, kmax`. Each footer line is `type neighbour_block orientation`: for an inter-block face (type `2`) the other two numbers give the 1-based partner block and the 1-based partner face (`1` = `imin` … `6` = `kmax`); for any other type they are ignored. A 2D block (`nk` = 1) is extruded by one cell along *y*, by the `--span` distance.

Every boundary face carries a type code. Without a zone file, each type maps to a default zone:

TYPE	DEFAULT ZONE NAME	bc	ZONE IDENTIFIER	MEANING
-2	symmetry	symmetry	7	symmetry plane
-1	aero_wall	wall	4	aerodynamic (solid) wall
0	wall	wall	3	wall
1	farfield	farfield	2	farfield
2	n/a	n/a	n/a	inter-block interface, not a boundary
3	periodic_downstream	periodic	5	periodic (downstream side)
4	periodic_upstream	periodic	6	periodic (upstream side)

To convert, run through *convert_mesh* using the `.blk` extension, or explicitly with `-f cba`:

```
(zCFD): convert_mesh <filename>.blk <new_filename>.h5
```

or run the converter directly:

```
(zCFD): cba_convert <filename>.blk <new_filename>.h5 [options]
```

Any options given after the two filenames on a *convert_mesh* command line are passed straight through to `cba_convert`. Its options are:

OPTION	MEANING
`input`	the CBA (`.blk`) mesh to read
`output`	the zCFD *.h5* mesh to write
`--zones FILE`	a zone definition YAML file, overriding the default zone assignment above
`--write-zones FILE`	also write the resolved zone assignment to FILE as YAML, and continue converting
`--no-weld`	keep each block's nodes separate instead of welding coincident nodes at block boundaries
`--span VALUE`	extrusion distance for a 2D mesh (default `1.0`)
`--quiet`	suppress the summary report

This produces the zCFD HDF5 file. A zone file is optional for CBA meshes: see Zone files.

Plot3D

The converter also reads ASCII (formatted), whole-format Plot3D structured multiblock grids (`.p3d`, `.fmt`). The file is a plain sequence of numbers: an optional block-count line, then one dimension line per block (`ni nj nk`, or `ni nj` for a 2D block), then for each block in turn all of its *x* coordinates, then all of its *y* coordinates, then (for a 3D block) all of its *z* coordinates — again with *i* varying fastest, then *j*, then *k*. A single block with no leading count line is also accepted. An extra *IBLANK* integer following each point's coordinates is tolerated and ignored, and FORTRAN-style repeat counts (for example `12*0.0` for twelve repetitions of `0.0`) are expanded. A 2D grid (no *k* dimension) is extruded by one cell along *z*, by the `--span` distance.

Binary (unformatted) Plot3D files, and the plane-by-plane streamed layout some codes write instead of the whole-block layout, are not supported; both are rejected rather than silently misread.

Unlike a CBA file, a Plot3D grid carries no connectivity or boundary information: block-to-block interfaces are discovered from coincident nodes after welding, and every boundary face must be assigned to a zone by hand — see Zone files. A zone file is therefore mandatory.

To convert, run through *convert_mesh* using the `.p3d` or `.fmt` extension, or explicitly with `-f plot3d`:

```
(zCFD): convert_mesh <filename>.p3d <new_filename>.h5 --zones <zonefile>.yml
```

or run the converter directly:

```
(zCFD): plot3d_convert <filename>.p3d <new_filename>.h5 --zones <zonefile>.yml [options]
```

Any options given after the two filenames on a *convert_mesh* command line are passed straight through to *plot3d_convert*. Its options are:

OPTION	MEANING
`input`	the Plot3D (`.p3d`/`.fmt`) grid to read
`output`	the zCFD *.h5* mesh to write
`--zones FILE`	the zone definition YAML file (required, unless only `--write-zones` is given)
`--write-zones FILE`	write a zone template covering every discovered boundary face to FILE
`--no-weld`	keep each block's nodes separate instead of welding coincident nodes at block boundaries
`--span VALUE`	extrusion distance for a 2D grid (default `1.0`)
`--quiet`	suppress the summary report

This produces the zCFD HDF5 file.

Zone files

A zone file is optional for a CBA mesh. Without `--zones`, the default mapping in CBA is used. A Plot3D grid carries no boundary information of its own, so its zone file is mandatory.

A zone file is YAML, with a single top-level `zones` map keyed by an integer zone identifier; each identifier must be unique. Here is a complete zone file for a 4-block ONERA M6 wing grid. Block 1 is upstream of the wing, blocks 2 and 3 run along it, and block 4 is downstream of the tip.

```yaml
zones:
  3:
    name: wing
    bc: wall
    patches:
      - { block: 2, face: jmin }
      - { block: 3, face: jmin }
  7:
    name: symmetry
    bc: symmetry
    patches:
      - { block: 1, face: kmin }
      - { block: 2, face: kmin }
      - { block: 3, face: kmin }
      - { block: 4, face: kmin }
  9:
    name: farfield
    bc: farfield
    patches:
      - { block: 1, face: imin }
      - { block: 1, face: jmax }
      - { block: 2, face: jmax }
      - { block: 3, face: jmax }
      - { block: 4, face: imax }
      - { block: 4, face: jmax }
```

Each zone takes:

- `name` — a free-text label, used only in the summary report. If omitted, it defaults to `zone_` followed by the identifier.
- `bc` — required; one of `wall`, `inflow`, `outflow`, `symmetry`, `farfield`, `periodic`. It sets the boundary type code written for the zone's faces. The boundary condition the solver applies is the one whose `zones` lists the zone identifier.
- `patches` — an optional list of explicit block faces belonging to this zone. `block` is the 1-based block number; `face` is one of `imin`, `imax`, `jmin`, `jmax`, `kmin`, `kmax`.
- `cba_type` — CBA only, optional; a raw CBA footer type code (see CBA). Every boundary patch still carrying that type code, and not already claimed by an explicit patch in this or any other zone, is added to this zone.

A patch normally covers the whole block face. To assign only part of a face, add a 1-based, inclusive node-index range on one or both of the face's two in-plane axes:

FACE	IN-PLANE AXES
`imin`, `imax`	`j`, `k`
`jmin`, `jmax`	`i`, `k`
`kmin`, `kmax`	`i`, `j`

```
zones:
  3:
    name: inlet_strip
    bc: inflow
    patches:
      - { block: 1, face: imax, j: [1, 5], k: [1, 3] }
```

If a patch is listed explicitly, it always takes that zone, even where a `cba_type` catch-all on another zone would otherwise have matched it: explicit patches take precedence over `cba_type` membership.

To find the boundary faces available on a mesh without working them out by hand, write a template and edit it:

```
(zCFD): cba_convert mesh.blk mesh.h5 --write-zones zones.yml
(zCFD): plot3d_convert grid.p3d grid.h5 --write-zones zones.yml
```

For a CBA mesh this writes the *currently resolved* zone assignment, which is the defaults above or the result of `--zones` if given. It still converts the mesh. For a Plot3D grid, if `--zones` is not also given, it instead writes a single template zone (identifier `3`, `bc: wall`) covering every discovered boundary face, split into as many patches as the block structure needs. It stops without writing an *.h5* file. Either way, edit the `name`, `bc` and grouping of the written zones as needed, then rerun with `--zones zones.yml`.

Conversion checks

The converter runs the same checks on every structured mesh, regardless of source format:

- **Node welding.** Coincident nodes at block boundaries are merged into a single global node, within a tolerance derived from the mesh itself (the smaller of one billionth of the mesh's bounding-box diagonal, and a thousandth of the shortest genuine edge in the mesh). `--no-weld` keeps every block's nodes separate; interfaces are still paired even when nodes are not welded.
- **Interface pairing.** For a CBA mesh, every inter-block face declared in the file's footers is checked: the partner face must exist, must point back to the same face (reciprocity), and its nodes must actually coincide with the partner's. For a Plot3D grid, which declares no connectivity, interfaces are discovered instead, purely from which welded nodes turn out to be shared between two blocks.
- **Handedness.** Every block's node ordering is classified as right- or left-handed from its signed volume, and face winding is adjusted per block so that outward normals and cell volumes come out correctly either way. Mixed handedness across blocks in the same mesh is not an error.
- **Six faces per cell, one or two cells per face.** Every cell ends up bounded by exactly six faces, and every face by exactly one cell (a boundary face) or two (an interior or interface face) — never zero, and never more than two.
- **Positive volumes.** Every cell's volume, computed from its bounding faces, must be strictly positive.
- **Periodic pairing.** Boundary faces with `bc: periodic` are grouped by zone identifier and checked pairwise: each periodic zone must map onto exactly one other periodic zone under a pure rotation about the *x*, *y* or *z* axis, and that mapping must be reciprocal. The axis and angle found for each pair are reported.

If any of these fail, conversion stops before a mesh is written, with a message naming the blocks and faces involved:

FAILURE	WHAT IT MEANS
unreciprocated interface	CBA only. One face's footer names another face as its interface partner, but that face's own footer either is not an interface or names a different partner. Fix the footer on the side that is wrong.
non-coincident interface	A declared CBA interface's nodes do not coincide with its partner's at all — the two blocks are offset from one another where they are meant to meet. Check the grid geometry at that face.
partial-face abutment	The same check as above, but only part of a declared CBA interface abuts its partner — a step or a sliver mismatch rather than a full offset. The faces reported are the ones left unpaired.
undeclared coincidence	Two block faces (CBA) are geometrically coincident, but neither declares the other as its interface partner. Either move the blocks apart, or add the missing interface footer.
degenerate block face	A block face has folded onto itself: the block is collapsed at that face, or wrapped with too few cells to keep the face distinct from itself, so some of its quads coincide with each other.
uncovered boundary quads	Some boundary quads were never placed in a zone. The message gives the 1-based node-index range on the face's two in-plane axes — copy it into a `patches` entry in the zone file to cover the gap. The same check, reported as quads assigned to *more than one* zone, catches overlapping patches.
duplicate zone identifier	Two zones in the zone file use the same identifier. Identifiers must be unique; give one of them a different identifier.

Node welding that merges genuinely distinct nodes, rather than the two sides of a declared self-interface, is also rejected — the message names the block and the two node indices involved.

Conversion report

Unless `--quiet` is given, both converters print a summary once conversion succeeds:

- the block count, and the CBA symmetry flag (always `0` for a Plot3D grid)
- the cell count
- the node count, raw and welded, and how many nodes were merged
- the face count, split into interior, interface and boundary faces
- every interface pair, with its face-quad count and whether it was found transposed and/or axis-reversed relative to its partner
- the resolved zones — identifier, `name`, `bc` and the number of boundary faces each ended up with
- how many blocks came out right-handed versus left-handed
- the periodic pairs found (axis and angle in degrees), or `none`
- the minimum cell volume
- the weld tolerance used, and the shortest genuine edge it was derived from
- the time taken in each phase of the conversion (welding, face-building, zone resolution, assembly, checks)

This is the quickest way to confirm a mesh converted the way you expect — in particular, that the interface and zone face counts match what you know about the mesh, and that the minimum cell volume is sensible.

Zone identifiers

Every boundary face of a converted mesh belongs to a zone with an integer zone identifier, and a boundary condition lists the identifiers of the zones it applies to in `zones` (p. 98). Fluid zones select cells by identifier in the same way. The source mesh names its zones, so the converters record which name became which identifier:

CONVERTER	FILE WRITTEN BESIDE THE MESH	CONTENTS
Fluent, Fluent CFF, CGNS	`<new_filename>_zone.py`	the zone helper described below
SU2	`<new_filename>_zones.py`	a single dict, `zones`, from zone name to identifier
UGRID	none	the identifiers are the boundary face tags of the UGRID file
OpenFOAM	none	the patches are numbered from `1`, in the order of the `boundary` file
CBA, Plot3D	none	the identifiers are the keys of the zone file, or the CBA defaults above

The zone helper defines:

NAME	CONTENTS
`zone_name_to_id`	a dict from each boundary zone name to its zone identifier

NAME	CONTENTS
`to_id(names)`	the zone identifiers of a list of boundary zone names
`fluid_zone`	a dict from each fluid zone name to its zone identifier

The Fluent and Fluent CFF helpers also hold one list of zone names for each Fluent boundary type: `wall`, `symmetry`, `farfield`, `velocity_inlet`, `pressure_inlet`, `pressure_outlet` and `interior`. The CGNS helper adds `from_id(id)`, which returns the name of a zone identifier.

The directory of the control file is on the Python import path while the file is read, so a helper beside it can be imported by name. This keeps the deck in terms of the names given in the mesh generator:

```
from wing_zone import to_id

{
    "boundary_conditions": {
        "BC_1": {"type": "wall", "zones": to_id(["wing_upper", "wing_lower"])},
        "BC_2": {"type": "symmetry", "zones": to_id(["symmetry"])},
    },
}
```

A helper whose name is not a valid Python module name, such as one beginning with a digit, is loaded with `zutil.get_zone_info` instead. It takes the file name, with or without `.py`, relative to the control file, and returns the helper as a module:

```
import zutil

zone = zutil.get_zone_info("2d_aerofoil_zone")

{"boundary_conditions": {"BC_1": {"type": "wall", "zones": zone.to_id(["aerofoil"])}}}
```

How a number in `zones` is matched

Each boundary zone of the mesh takes the boundary condition whose `zones` contains its zone number. A zone that no boundary condition names by number falls back to its boundary type code, listed in the zCFD mesh file (p. 186): it takes the boundary condition whose `zones` contains that code. A zone that matches neither way stops the run, with a message naming the zone and its code:

```
Unable to determine boundary condition for zone 12 (ref: 3)
```

> **WARNING**
>
> The fallback applies to every zone left unnamed. In a mesh with a zone numbered `3` and several unnamed wall zones, `"zones": [3]` also applies to every wall zone, since `3` is the code for a wall. List every zone by number to rule this out.

Initial placement

The model's `transforms` (p. 169) places the whole mesh once, when it is read, before the solver starts.

It takes two placement keys, which may be given together:

- `scale` (p. 169) multiplies the *x*, *y* and *z* coordinates of every node by the three factors given.
- `transform` (p. 170) is a 4 × 4 affine matrix applied to every node, or a function that returns each node's placed position.

Where both are given, `scale` is applied first and `transform` second. The matrix therefore acts in metres: where `scale` converts millimetres to metres, a translation in the matrix is given in metres. The order, and the `motion` that follows it, are described in Placing and moving a mesh (p. 365).

A plain unit conversion needs only `scale`:

```
# Scale from mm to metres
parameters["model"]["wing"]["transforms"] = {"scale": [0.001, 0.001, 0.001]}
```

```
# Scale from inches to metres
parameters["model"]["wing"]["transforms"] = {"scale": [0.0254, 0.0254, 0.0254]}
```

A plain anisotropic mesh stretch also needs only `scale`:

```
{
    "model": {
        "canopy_dacosta": {
            "transforms": {"scale": [50.0, 1.0, 1.0]},
            "mesh": "canopy_dacosta.h5",
            ...
        }
    }
}
```

> **NOTE**
>
> `scale` applies to the mesh only — not interpolated surface outputs etc.

Build a `transform` matrix with the `zutil.transform` helpers. Each returns a 4 × 4 numpy array. Given an existing matrix as its optional last argument, `A2B`, a helper applies its own transform after that matrix, so a chain of calls applies the transforms in the order written:

HELPER	TRANSFORM
`rotate(axis, angle_deg, A2B)`	a rotation by `angle_deg` degrees about `axis`, through the origin, by the right-hand rule
`translate(point_b, point_c, A2B)`	a translation by `point_c` minus `point_b`
`scale(scale_factors, A2B)`	a scaling of *x*, *y* and *z* by the three factors

```
import zutil.transform

# Two 90-degree rotations about x, then 10 degrees about y, composed in order
mesh_transform = zutil.transform.rotate([1, 0, 0], 90.0)
mesh_transform = zutil.transform.rotate([1, 0, 0], 90.0, mesh_transform)
mesh_transform = zutil.transform.rotate([0, 1, 0], 10.0, mesh_transform)

{"model": {"naca0012": {"transforms": {"transform": mesh_transform}}}}
```

The matrix is an affine transform (Wikipedia): the upper left 3 × 3 block describes a rotation, reflection, scaling or shear, or any combination of them, and the fourth column a translation. Here a rotation, a scaling and a translation are applied in turn:

```
import zutil.transform

A2B = zutil.transform.rotate([0.0, 1.0, 0.0], 10.0)
B2C = zutil.transform.scale([0.1, 0.1, 0.1], A2B)
C2D = zutil.transform.translate([0.0, 0.0, 0.0], [-1.0, 0.0, 0.0], B2C)

parameters["model"]["wing"]["transforms"] = {"transform": C2D}
```

Rotations follow the conventions of pytransform3d and are active: the mesh is rotated, not the axes. A matrix built directly with pytransform3d can be given as `transform` in the same way.

`scale` and `transform` place the mesh once, when it loads. A mesh that moves during the run, or is solved in a moving frame, is given a `motion` in the same `transforms`; see Placing and moving a mesh (p. 365).

Running zCFD

zCFD is primarily run from the command line; you can run it on your local machine or submit it to a batch scheduler for parallel execution. The following sections give information on how to execute zCFD.

Command Line Execution

zCFD includes a command-line environment, which must be activated before a run, whether zCFD is started directly or through a job scheduler such as Slurm.

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

This sets up the environment for that version of zCFD. While the environment is active, the command prompt carries a zCFD prefix:

```
(zCFD) >
```

To deactivate the environment and restore the previous one, use:

```
zdeactivate
```

The command prompt returns to its normal appearance.

To run zCFD on the command line:

```
run_zcfd -c <case> [-n <ranks>] [-t <ranks_per_node>] [-o <threads>] [-d cpu|gpu]
```

Each flag and its default is listed in invocation and flags (p. 179).

The mesh is not given on the command line. The control file names it with the `mesh` (p. 97) key of each entry under `model`:

```
parameters = {
    "config_version": 2,
    "solver": {...},
    "model": {
        "wing": {"mesh": "wing.h5", ...},
    },
}
```

See Command line invocation and flags (p. 179) for input validation and batch submission.

To run zCFD on a desktop computer with a single 12-core CPU and no GPU:

```
run_zcfd -n 1 -d cpu -c <case>
```

zCFD starts one OpenMP thread for each CPU the operating system reports, unless a smaller number is requested. To run the same case with 6 threads:

```
run_zcfd -n 1 -d cpu -o 6 -c <case>
```

or

```
export OMP_NUM_THREADS=6; run_zcfd -n 1 -d cpu -c <case>
```

To run zCFD on a desktop computer with two GPUs, one rank on each:

```
run_zcfd -n 2 -d gpu -c <case>
```

Multiple Meshes and Overset Cases

A case with more than one mesh (see overset (p. 328)) is run in the same way. Each mesh is a named entry under `model`, with its own `mesh` key, and the command line is unchanged.

An example control file for a background domain with a single rotating turbine:

```
parameters = {
    "config_version": 2,
    "solver": {...},
    "model": {
        "background": {"mesh": "background.h5", ...},
        "turbine": {"mesh": "turbine.h5", ...},
    },
}
```

```
run_zcfd -n 2 -d gpu -c <case>
```

What a run writes

A run of the control file `<case>.py` writes these files into the directory it is started from:

FILE OR DIRECTORY	CONTENTS
`<case>.log`	the screen output of the run. Earlier logs are kept as `<case>.log.1` to `<case>.log.10`, the most recent first
`<case>_status.yaml`	the status file, described below

FILE OR DIRECTORY	CONTENTS
`<case>_OUTPUT/<model>/`	the output of each entry under `model`: its report file `<model>_report.csv`, its checkpoint `<model>_results.h5` and its solution files
`<case>_OUTPUT/LOGGING/`	one log for each MPI rank, `<case>.<rank>.log`

The solution files are described in The output directory (p. 341).

The status file

`<case>_status.yaml` is a YAML file written before the solve starts. It records how the run was started:

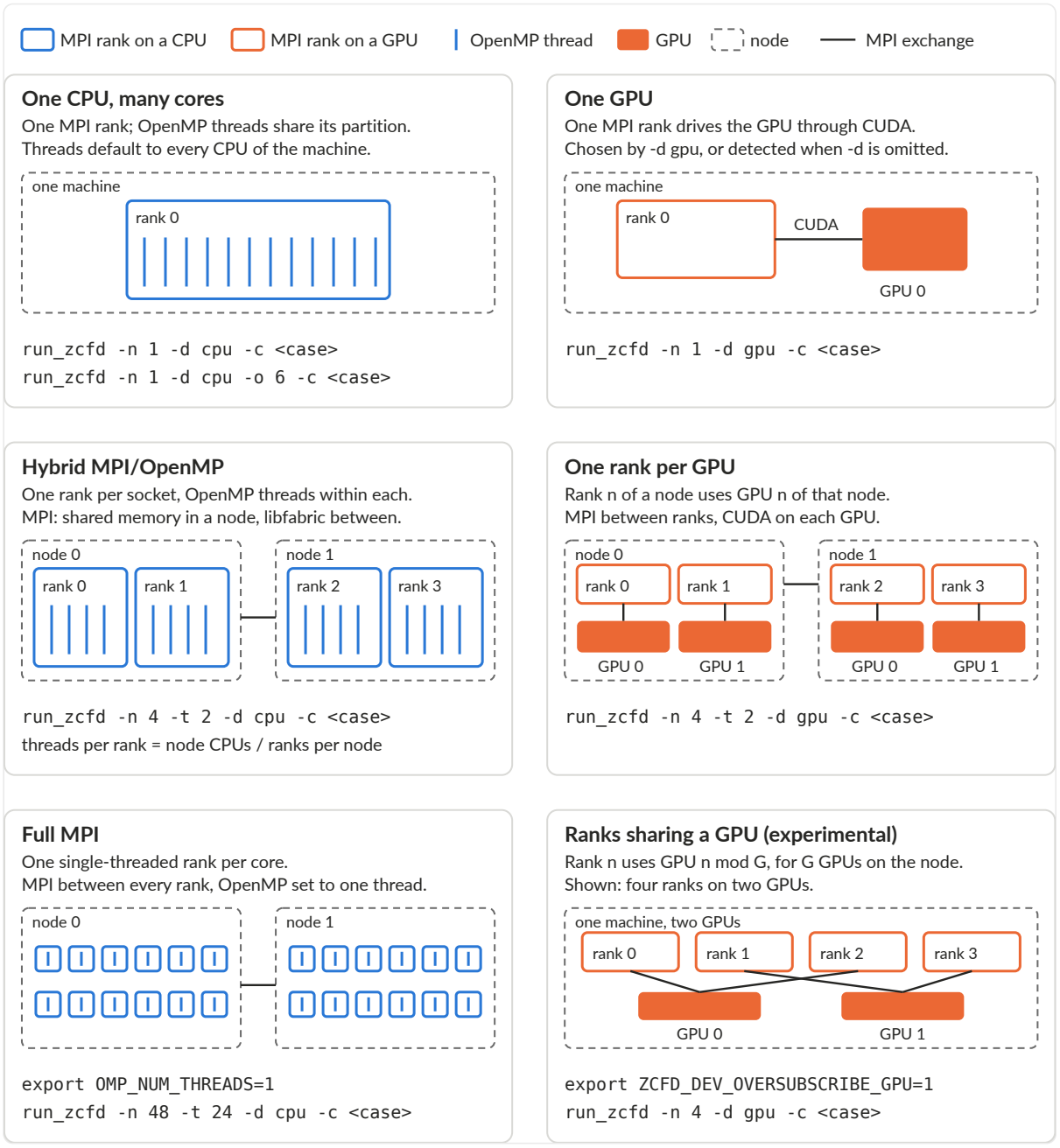
KEY	CONTENTS
`case`	the control file
`num processor`	the number of MPI ranks
`mesh`	the file name of each mesh, with its MD5 checksum
`version`	the zCFD version
`date`	the time and date the run started, as `HH:MM:SS DD-MM-YYYY`
`nodes`	the host name of each node the run used
`devices`	the CPU or GPU used by each rank
`model_ordering`	the entries under `model`, in the order the solver advances them

A run with a farfield target (p. 360) also keeps a `target` record for each target, updated as the run proceeds: its phase (`settle`, `trim` or `hold` during the run, `met` or `not met` at the end), the current angle and the current coefficient. Trimming to a target (p. 363) shows the record. A restart of that run reads the record to start the farfield at the recorded angle.

Three things read it:

- **A restart from another case.** When `restart_casename` (p. 49) names another case, the run reads that case's status file to find its models and its meshes. The run stops if the status file is missing, or if a mesh of the same name now has a different checksum.
- **Post-processing.** The `zutil` tools that open a run's output read the partition count and the model names from it.
- **The checkpoint.** A copy is stored in each `<model>_results.h5`, so the record survives the run directory, as described in What a checkpoint records about the run (p. 342).

Running in parallel



The six ways of running zCFD in parallel, with the command that selects each. On CPUs: one rank with OpenMP threads on one machine; hybrid MPI/OpenMP with one rank per socket and threads within each, over two nodes; full MPI with one single-threaded rank per core. On GPUs: one rank driving one GPU through CUDA; one rank per GPU over two nodes, rank n of a node using GPU n; and, with ZCFD_DEV_OVERSUBSCRIBE_GPU set, four ranks sharing two GPUs, rank n using GPU n mod 2. MPI joins ranks through shared memory within a node and libfabric between nodes.

zCFD divides the mesh into partitions, one for each MPI rank. The `-n` flag sets the number of ranks and `-t` the number of ranks on each node. Within a CPU rank, OpenMP threads share the work of its partition. A GPU rank runs its partition on one GPU through CUDA.

The released package uses Intel MPI. Ranks on the same node communicate through shared memory, and ranks on different nodes through libfabric, which can use interconnects such as InfiniBand or the AWS Elastic Fabric Adapter. The licence is checked out for the number of ranks in the run, so it must cover that number.

There are two strategies for running on CPUs:

Hybrid MPI/OpenMP runs one MPI rank per socket, with OpenMP threads on the cores of that socket. At large core counts this keeps the number of ranks, and with it the cost of communication, low.

Full MPI runs one single-threaded MPI rank per core. It is usually the better choice at low core counts.

Hybrid MPI/OpenMP

When `OMP_NUM_THREADS` is not set, zCFD sets the number of threads for each rank to the CPUs of the node divided by the number of ranks on it. Under Slurm, it divides the CPUs the job holds on each node instead. A smaller value of `OMP_NUM_THREADS`, or of `-o`, takes precedence; a larger one is reduced to this number.

For example, on two hosts each with two 12-core CPUs, request one rank per CPU socket. Where each core runs one hardware thread, each rank then runs 12 threads.

```
run_zcfd -n 4 -t 2 -d cpu -c <case>
```

Full MPI

Set `OMP_NUM_THREADS` to 1 and give the number of ranks with `-n`.

For example, on two hosts each with two 12-core CPUs, one rank per core:

```
export OMP_NUM_THREADS=1
run_zcfd -n 48 -t 24 -d cpu -c <case>
```

GPU Parallel Execution

zCFD assigns GPUs to MPI ranks on GPU-enabled nodes. GPUs are numbered in the order of their PCI bus.

Single rank per GPU: this is the default, and suits most workloads. Rank n on a node uses GPU n of that node, so each rank has a GPU to itself. When `-d` is left out, a rank with no GPU of its own runs on the CPU.

For a node with 4 GPUs, one rank on each:

```
run_zcfd -n 4 -d gpu -c <case>
```

Multiple ranks per GPU (experimental): to let several MPI ranks share a GPU, set the `ZCFD_DEV_OVERSUBSCRIBE_GPU` environment variable (see Command line environment variables (p. 181)). Rank n on a node then uses GPU n mod G, where G is the number of GPUs on the node.

> **WARNING**
> GPU oversubscription is an experimental feature. It may reduce performance, and is intended for debugging where GPUs are few.

It is useful for:

- testing and development with few GPUs
- debugging with smaller problem sizes
- workloads that make little use of the GPU

For a node with 4 GPUs, 8 ranks, 2 on each GPU:

```
export ZCFD_DEV_OVERSUBSCRIBE_GPU=1
run_zcfd -n 8 -d gpu -c <case>
```

Example assignments with oversubscription enabled:

- 4 GPUs, 4 ranks: GPU assignment [0,1,2,3]
- 4 GPUs, 8 ranks: GPU assignment [0,1,2,3,0,1,2,3]
- 4 GPUs, 6 ranks: GPU assignment [0,1,2,3,0,1]

See Command line parallel invocation (p. 182) for running across multiple machines, libfabric provider selection and other advanced parallel settings.

Python functions in the deck

Several parameters in the zCFD control dictionary accept a Python function in place of a value. The solver calls the function when it needs the value, so a field, a source or a ramp can follow any expression you can write in Python.

How zCFD calls a function

Every function is called with keyword arguments only, each named for what it holds. Write the parameters after a bare `*`, so that the signature says at once what the function receives:

```
def shock(*, location, wall_distance, pressure, temperature, velocity):
    return {"pressure": 2.0 * pressure if location[0] < 0.5 else pressure}
```

A function must name every keyword it is given, even one it does not use, or collect them with `**kwargs`.

A vector, whether it comes in or goes out, is three numbers in order x, y, z. Return it as a tuple, a list or a numpy array; a `{"v1": x, "v2": y, "v3": z}` dict is also read. Lengths are in metres, times in seconds, and pressures, temperatures and velocities are dimensional, in Pa, K and m/s.

PARAMETER	CALLED WITH	RETURNS
`cfl_ramp` (p. 69)	`cycle`, `current_cfl`	the new CFL, or a dict of `cfl`, `cfl_turbulence`, `cfl_coarse`
`transform` (p. 94)	`vector`	the transformed vector
`initial_conditions` (p. 48)	`location`, `wall_distance`, `pressure`, `temperature`, `velocity`	a dict of any of `pressure`, `temperature`, `velocity`
`func` (p. 145) of a momentum zone	`locations`	one source vector per location
`function` (p. 162) of a canopy zone	`locations`, `wall_distances`	one leaf area density per location
`func` (p. 168) of a mesh deformation zone	`location`	the deformed position

What validation checks

zCFD checks each function when it reads the control file. It binds the signature to the keywords above, calls the function with representative values, and checks the shape of what comes back. The first call puts any location at the origin or on the x axis. If that call raises, zCFD calls the function again at a point off the axes, so a function that is singular only on an axis, such as a swirl about the z axis, is accepted. A function that raises at both points, or returns the wrong shape, is rejected with a message that names it:

```
solver › output settings › report › forces › FR_1 › transform
  The forces transform my_transform(x, y, z) cannot take the keywords zCFD calls it with:
↪  vector. Write it as def my_transform(*, vector).
```

```
solver › initialisation › initial conditions
  The initial conditions function shock returned {'density': 1.2} (dict) when called with
↪  location=(0.0, 0.0, 0.0), wall_distance=1.0, pressure=101325.0, temperature=288.15,
↪  velocity=(1.0, 0.0, 0.0); it must return a dict of any of 'pressure', 'temperature' and
↪  'velocity'; 'density' is not one of them.
```

Because of those calls, a function should give the same answer however many times it is called. One that keeps count of its own calls sees one or two more than the solver makes.

Ramp CFL Function

The named shapes in cfl_ramp (p. 69) cover most cases. Where the CFL needs to respond to something they cannot express, `cfl_ramp` accepts a function instead of the list.

The function is called once per pseudo-time cycle with the cycle number and the CFL in use, and returns the CFL for that cycle. The result is limited to the `cfl` target.

```python
def my_cfl_ramp(*, cycle, current_cfl):
    if cycle < 500:
        return min(1.0 * 1.005 ** (cycle - 1), 10.0)
    elif cycle < 1000:
        return 15.0
    return 30.0
```

```
{"convergence_control": {"cfl": {"cfl": 30.0, "cfl_ramp": my_cfl_ramp}}}
```

Returning a dictionary drives the turbulence and coarse-level CFL targets alongside the mean-flow one. Any key left out keeps the value from the control dictionary:

```python
def my_cfl_ramp(*, cycle, current_cfl):
    cfl = min(1.0 * 1.005 ** (cycle - 1), 30.0)
    return {"cfl": cfl, "cfl_turbulence": 0.5 * cfl}
```

Transform Function

A transform on a force report turns each force and moment vector into a different coordinate system before it is written. It is most often used to report lift and drag, which lie in the wind axes rather than the solver axes.

```
import zutil

alpha = 10.0


def wind_axes(*, vector):
    return zutil.rotate_vector(vector, alpha, 0.0)
```

```
{"report": {"forces": {"FR_1": {"name": "wing", "zones": [3], "transform": wind_axes}}}}
```

Initialisation Function

A function as the initial condition is called once for every cell, with the cell centre, its wall distance and the state the cell holds before the function runs. It returns the quantities it sets; any it leaves out keep their value.

```
def shock(*, location, wall_distance, pressure, temperature, velocity):
    if location[0] < 0.5:
        return {"pressure": 100000.0, "temperature": 348.4, "velocity": (0.0, 0.0, 0.0)}
    return {"pressure": 10000.0, "temperature": 278.7, "velocity": (0.0, 0.0, 0.0)}
```

A function as the initial condition needs a `reference` (p. 48) condition beside it:

```
{"initialisation": {"reference": "IC_1", "initial_conditions": shock}}
```

Momentum source function

A momentum zone's `func` is called once, when the zone is set up, with the centre of every cell in the zone. It returns one source vector for each centre, in the units of the zone's `source` (p. 144).

```
def momentum_drive(*, locations):
    return [(0.0, 0.0, body_force) for _ in locations]
```

Leaf area density function

A canopy zone's `function` is called with the centre of every cell in the canopy and its wall distance, in the same order. It returns the leaf area density of each cell, in 1/m.

```
def lad_function(*, locations, wall_distances):
    lai = 5.0
    h_can = 20.0
    return [np.interp(d, a_z[0] * h_can, a_z[1]) * lai / h_can for d in wall_distances]
```

Mesh deformation function

The `func` of an RBF or IDW mesh deformation zone is called with the position of each surface node it moves, and returns the node's deformed position. The zone then spreads the surface displacement through the volume mesh.

```
def pitch(*, location):
    return zutil.rotate_vector(location, 2.0, 0.0)
```

Choosing an equation set

Which equations should the solver solve?

Choose the equation set from the physics. Each set adds terms to the one before it:

- `euler` solves the inviscid flow equations. There is no viscosity, and so no boundary layer and no turbulence.
- `viscous` solves the Navier–Stokes equations for laminar flow.
- `rans` solves the Reynolds-averaged Navier–Stokes equations, with transport equations that model the effect of turbulence on the mean flow.
- `les` solves the filtered Navier–Stokes equations. The energy-containing eddies are resolved on the mesh, and only the scales below the filter width are modelled.

Use `euler` where viscosity plays no part in the result, `viscous` for laminar flow, `rans` for turbulent flow whose turbulence is modelled, and `les` where the energy-containing eddies must be resolved. To resolve eddies away from the wall while keeping RANS near it, use `rans` with a hybrid model; see Choosing a turbulence model (p. 226).

type	TURBULENCE	MESH NEAR THE WALL	SOLVER
`"euler"`	none; the flow is inviscid	no boundary layer to resolve	compressible only
`"viscous"`	none; the flow is laminar	resolves the boundary layer	compressible or incompressible
`"rans"`	modelled by transport equations	set by the wall treatment	compressible or incompressible
`"les"`	resolved above the filter width	fine enough to carry resolved eddies	compressible or incompressible

Equation sets

The `equations` block sits under `solver` and selects the equation set:

```
parameters = {
  "config_version": 2,
  "solver": {
    "equations": {
      "type": "rans",
      "turbulence": {"model": "sst"},
    },
    # ... numerical_scheme, fluid_properties, reference_conditions, ...
  },
}
```

`type` (p. 20) is case-insensitive on input, so `"RANS"`, `"rans"` and `"Rans"` validate alike; it is stored in lower case.

The choice between the compressible and the incompressible solver is a separate one, made by `{"solver_settings": {"type": ...}}` with `"compressible"` or `"incompressible"`. Both solvers share the equation types `"viscous"`, `"rans"` and `"les"`, and the incompressible solver accepts further keys alongside them (see Incompressible equation sets). The incompressible solver has no Euler equation set.

precondition

Low-speed Mach preconditioning modifies the pseudo-time system to restore good conditioning at low Mach number. It therefore has meaning only where a pseudo-time loop exists, in steady runs and under dual time stepping. Global time stepping has no pseudo-time loop to precondition, so `precondition` (p. 20) is unavailable there. `False` is the default, and never conflicts with a time-marching choice.

Euler

The compressible Euler equations describe inviscid flow. They suit flows in which momentum greatly outweighs viscosity, at very high speed for example. An Euler mesh need not resolve the boundary layer, and so generally has far fewer cells than the corresponding viscous mesh.

Euler takes only the common `type` and `precondition` keys:

```
parameters["solver"]["equations"] = {
    "type": "euler",
    "precondition": True,
}
```

Viscous

The viscous equations describe flow that is viscous but laminar. The Reynolds number of the flow determines whether it is turbulent.

A viscous mesh must resolve the boundary layer, and so is generally larger than an Euler mesh. On the same mesh a viscous run is faster than a RANS run, since it solves fewer equations.

Compressible viscous flow takes only the common `type` and `precondition` keys:

```
parameters["solver"]["equations"] = {
    "type": "viscous",
    "precondition": True,
}
```

RANS

The Reynolds-averaged Navier–Stokes equations, for fully turbulent flow.

Besides the common keys, `"rans"` requires the `turbulence` (p. 18) dictionary, which names the turbulence model and any hybrid RANS–LES model layered on it. The choice of model is set out in Choosing a turbulence model (p. 226).

```
parameters["solver"]["equations"] = {
    "type": "rans",
    "precondition": True,
    "turbulence": {"model": "sst"},
}
```

LES

Large eddy simulation resolves the energy-containing turbulent scales directly, and models only the sub-grid scales below the mesh filter width. A non-hybrid LES run solves no additional transport equation for turbulence: the sub-grid-scale model is a kernel inside the viscous solver. The `"les"` equation set is therefore handled by the same solver path as `"viscous"`, with the sub-grid closure switched on.

The `turbulence` dictionary of `"les"` is optional, and selects the sub-grid model; see LES sub-grid models (p. 227). Leaving it out gives implicit LES, with no sub-grid model.

Hybrid RANS–LES methods carry a transport-equation RANS model beneath their LES branch, and are configured through the RANS equation set, never through `"les"`.

```
parameters["solver"]["equations"] = {
    "type": "les",
    "turbulence": {"les": "wale"},
}
```

Incompressible equation sets

With the incompressible solver, `{"solver_settings": {"type": "incompressible"}}`, the equation types `"viscous"`, `"rans"` and `"les"` accept further keys beside `type` and `precondition`: the non-orthogonal correction, gradient limiting and central blending, the energy equation, and the stratified-atmosphere settings, among them `base_state` (p. 18) and `theta_perturbation` (p. 18).

The pressure-velocity coupling scheme, its relaxation factors and the SIMPLE pressure settings belong to `convergence_control`; see Incompressible convergence control (p. 251).

```
parameters["solver"] = {
    "solver_settings": {"type": "incompressible"},
    "equations": {
        "type": "viscous",
    },
    # ... numerical_scheme, fluid_properties, reference_conditions, ...
}
```

Incompressible RANS

The incompressible RANS equations combine these keys with the `turbulence` dictionary of RANS. The incompressible solver solves the `sst` and `sa-neg` turbulence models only.

```
parameters["solver"] = {
    "solver_settings": {"type": "incompressible"},
    "equations": {
        "type": "rans",
        "precondition": True,
        "turbulence": {"model": "sst"},
    },
    # ... numerical_scheme, fluid_properties, reference_conditions, ...
}
```

Incompressible LES

The incompressible LES equations combine these keys with the LES `turbulence` dictionary of LES:

```
parameters["solver"] = {
    "solver_settings": {"type": "incompressible"},
    "equations": {
        "type": "les",
        "turbulence": {"les": "wale"},
    },
    # ... numerical_scheme, fluid_properties, reference_conditions, ...
}
```

Reference entries

- `type` (p. 20) and `precondition` (p. 20)
- `turbulence` (p. 18)
- `base_state` (p. 18) and `theta_perturbation` (p. 18)

Choosing a turbulence model

Which turbulence model should close the equations?

This page applies when the equation set is `rans` or `les`; the choice of equation set is set out in Choosing an equation set (p. 222).

For `rans`, name the turbulence `model`. To resolve eddies away from the wall while keeping RANS near it, add a `hybrid` on top of that model. For `les`, choose the sub-grid model.

The RANS turbulence models, selected by `model` (p. 24):

model	MODEL	REFERENCE
`sst`	Menter shear stress transport k-ω, in its 2003 form	Menter (1994); Menter et al. (2003)
`spalart-allmaras`	Spalart–Allmaras, solved in the negative form of `sa-neg`	Spalart and Allmaras (1994); Allmaras et al. (2012)
`sa-neg`	negative Spalart–Allmaras, without the f_{t2} term	Allmaras et al. (2012)
`sst-transition`	Langtry–Menter four-equation transitional SST (γ-Re_θ)	Langtry and Menter (2009)
`sas`	scale-adaptive simulation, run as a standalone RANS model	Menter and Egorov (2010)

The incompressible solver solves `sst` and `sa-neg` only.

Hybrid RANS–LES

A `hybrid` (p. 25) of `DES`, `DDES`, `IDDES` or `SAS` layers an LES branch on the chosen model. On `sst`, `DES` follows Strelets (2001), and `DDES` and `IDDES` follow Gritskevich et al. (2012). On `sa-neg`, `DES`, `DDES` and `IDDES` follow Spalart et al. (1997), Spalart et al. (2006) and Shur et al. (2008) respectively. `SAS` follows Menter and Egorov (2010).

A hybrid model carries a transport-equation RANS model beneath its LES branch; SST-IDDES, for example, still solves the SST k and ω equations. It is configured through the `rans` equation set, with `hybrid` set alongside `model`, and never through `les`. Leave `hybrid` out to solve pure RANS.

The turbulence dictionary

Under `rans`, the `turbulence` (p. 18) dictionary holds `model` (p. 24), `hybrid` (p. 25) and the model options below.

```
parameters["solver"]["equations"] = {
    "type": "rans",
    "precondition": True,
    "turbulence": {
        "model": "sa-neg",
        "hybrid": "IDDES",
    },
}
```

```
parameters["solver"]["equations"] = {
    "type": "rans",
    "precondition": True,
    "turbulence": {
        "model": "sst",
        "rotation_correction": False,
        "limit_gradient_k": 0.5,
        "qcr": 2020,
    },
}
```

qcr

`qcr` (p. 25) adds the non-linear quadratic constitutive relation (QCR) to the turbulent stress, for both SST and SA-neg. QCR has been shown to improve the prediction of separated corner flows, particularly on highly loaded wings. Three versions are available: QCR 2000, QCR 2020 and QCR 2021.

production

`production` (p. 26) selects the production term of the SST model: `"vorticity"` (SST-V), `"strain-incompressible"`, `"strain-compressible"` or `"kato-launder"`. The integers 0 to 3 are also accepted, and stand for the same four names in that order. Left out, the model chooses for itself: `kato-launder` for `sst-transition` and `vorticity` otherwise.

Advanced parameters for turbulence models

The constants in the `turbulence` dictionary tune the turbulence models: `betastar`, `cdes_kw`, `cdes_keps`, `cd1`, `cd2`, `cw`, `a1`, `cdes`, `ca1`, `ca2`, `ce1`, `ce2`, `cthetat`, `sigmagamma`, `sigmathetat` and `separation_correction`. They are advanced settings, and should not normally need adjusting.

LES sub-grid models

Under `les`, the `turbulence` (p. 18) dictionary is a much smaller, LES-specific one, unrelated to the RANS dictionary above. Its `les` (p. 29) key selects the sub-grid model: `wale`, `deardorff` or `none`. `deardorff` is implemented by the incompressible solver only. The whole dictionary is optional: leaving it out gives implicit LES, with no sub-grid model.

```
parameters["solver"]["equations"] = {
    "type": "les",
```

```
    "turbulence": {
        "les": "wale",
    },
}
```

Shielding the LES branch by CFL

`les_cfl_limit` (p. 28) is the CFL number at or below which the LES branch of a DES, DDES or IDDES model is left untouched. Left out, as it is by default, the CFL shield is disabled.

The LES branch replaces the modelled RANS stress with resolved eddies the size of the local filter width Δ. Those eddies are resolved only if the time step can follow them, that is if the cell CFL number is of order one. Where the CFL number is larger the model still drops the RANS stress, but the temporal discretisation cannot supply the resolved stress that should replace it. The shielding function f_d gives no protection against this, because it is built from r_d, a purely spatial ratio.

Setting a limit adds a second shield, built from the cell CFL number. The time step has to resolve two things: the transport of an eddy of size Δ across the cell, and that eddy's own turnover. The shield takes whichever of the two is binding,

$$\text{CFL} = \max\left(\frac{|\mathbf{u}-\mathbf{u}_{grid}|\,\Delta t}{\Delta},\; C_{turn}\,|S|\,\Delta t\right), \qquad f_{CFL} = \text{smoothstep}\left(\text{clamp}\left(\frac{\text{CFL}-\text{limit}}{\text{width}}, 0, 1\right)\right)$$

and is combined with f_d so that the more protective of the two prevails.

The transport term uses the velocity relative to the mesh, so it is indifferent to the frame: a mesh and fluid in rigid-body motion together contribute nothing to it. On a rotating mesh, however, the solid-body term $\boldsymbol{\Omega} \times \mathbf{r}$ is a genuine relative speed and can dominate, so a fine rotating overset mesh is often shielded almost everywhere. That is the criterion reporting a real limitation of the mesh and time step, not an artefact; `les_turnover_coefficient` covers the opposite case. At $f_{CFL} = 1$ the hybrid length scale is exactly the RANS length scale, and the LES branch is fully suppressed. Because f_{CFL} is exactly zero at or below the limit, a simulation whose CFL number is everywhere within the limit is unaffected.

The shield acts only in time-accurate runs, with `time_settings` of type `"unsteady"` under either dual or global time stepping, since a steady run has no meaningful real time step. It applies to the `DES`, `DDES` and `IDDES` settings only, not to `SAS` or to `wale`. Before enabling it, use the `lescfl` output variable to find where the CFL number exceeds the limit.

`les_cfl_width` (p. 29) is the width of the blend above `les_cfl_limit`: RANS is fully enforced at a CFL number of `limit + width`. A wider blend returns to RANS more gradually, which is usually preferable, since the shielded cells also revert to full upwind dissipation.

les_turnover_coefficient

`les_turnover_coefficient` (p. 29) is the weight C_{turn} on the eddy-turnover term of the shielded CFL number. It has no effect unless `les_cfl_limit` is set. Setting it to `0.0` recovers the transport-only criterion.

Transport alone is a one-sided test. Where the flow is slow relative to the mesh it reports a CFL number near zero and permits the LES branch however large the time step, even though an energetic eddy in that cell may turn over many times per step. Stagnation regions, recirculation cores and the neighbourhood of a rotation axis, where $\Omega r \to 0$, all fall in that blind spot.

The turnover term closes it. The velocity of an eddy at the filter width is $u'(\Delta) \sim |S|\,\Delta$, so Δ cancels, and the measure is the shear-timescale Courant number $|S|\,\Delta t$, with $|S| = \sqrt{2S_{ij}S_{ij}}$ the resolved strain-rate magnitude.

The measure uses strain, not vorticity. Solid-body rotation carries $|\omega| = 2\Omega$ with no turnover at all, so a vorticity-based measure would shield every vortex core, which is exactly the content a DES run exists to resolve. The resolved strain is also preferred to the modelled $\sqrt{k}$, which in a shielded cell is the full RANS turbulence kinetic energy rather than its sub-grid part, and would lock a cell into RANS once it got there.

Use the `lesturnover` output variable alongside `lescfl` to see which of the two mechanisms is binding in a given cell.

Literature

- Allmaras, S. R., Johnson, F. T. and Spalart, P. R. (2012). Modifications and Clarifications for the Implementation of the Spalart–Allmaras Turbulence Model. Seventh International Conference on Computational Fluid Dynamics, paper ICCFD7-1902.
- Gritskevich, M. S., Garbaruk, A. V., Schütze, J. and Menter, F. R. (2012). Development of DDES and IDDES Formulations for the k-ω Shear Stress Transport Model. *Flow, Turbulence and Combustion*, 88(3), 431–449. https://doi.org/10.1007/s10494-011-9378-4
- Langtry, R. B. and Menter, F. R. (2009). Correlation-Based Transition Modeling for Unstructured Parallelized Computational Fluid Dynamics Codes. *AIAA Journal*, 47(12), 2894–2906. https://doi.org/10.2514/1.42362
- Menter, F. R. (1994). Two-Equation Eddy-Viscosity Turbulence Models for Engineering Applications. *AIAA Journal*, 32(8), 1598–1605. https://doi.org/10.2514/3.12149
- Menter, F. R. and Egorov, Y. (2010). The Scale-Adaptive Simulation Method for Unsteady Turbulent Flow Predictions. Part 1: Theory and Model Description. *Flow, Turbulence and Combustion*, 85(1), 113–138. https://doi.org/10.1007/s10494-010-9264-5
- Menter, F. R., Kuntz, M. and Langtry, R. (2003). Ten Years of Industrial Experience with the SST Turbulence Model. In K. Hanjalić, Y. Nagano and M. Tummers (editors), *Turbulence, Heat and Mass Transfer 4*, Begell House, 625–632.
- Shur, M. L., Spalart, P. R., Strelets, M. and Travin, A. K. (2008). A Hybrid RANS-LES Approach with Delayed-DES and Wall-Modelled LES Capabilities. *International Journal of Heat and Fluid Flow*, 29(6), 1638–1649. https://doi.org/10.1016/j.ijheatfluidflow.2008.07.001
- Spalart, P. R. and Allmaras, S. R. (1994). A One-Equation Turbulence Model for Aerodynamic Flows. *La Recherche Aérospatiale*, 1, 5–21.
- Spalart, P. R., Deck, S., Shur, M. L., Squires, K. D., Strelets, M. and Travin, A. (2006). A New Version of Detached-Eddy Simulation, Resistant to Ambiguous Grid Densities. *Theoretical and Computational Fluid Dynamics*, 20(3), 181–195. https://doi.org/10.1007/s00162-006-0015-0
- Spalart, P. R., Jou, W.-H., Strelets, M. and Allmaras, S. R. (1997). Comments on the Feasibility of LES for Wings, and on a Hybrid RANS/LES Approach. In C. Liu and Z. Liu (editors), *Advances in DNS/LES*, Greyden Press, 137–147.

- Strelets, M. (2001). Detached Eddy Simulation of Massively Separated Flows. AIAA paper 2001-0879. https://doi.org/10.2514/6.2001-879

Reference entries

- `turbulence` (p. 18): `model` (p. 24), `hybrid` (p. 25), `qcr` (p. 25), `production` (p. 26)
- `les` (p. 29)
- `les_cfl_limit` (p. 28), `les_cfl_width` (p. 29) and `les_turnover_coefficient` (p. 29)

Choosing a wall model

Which wall model should a wall boundary use, and what does the choice ask of the mesh?

Use `no slip` when the mesh resolves the boundary layer, with the first cell at y^+ between 1 and 5. Use `wall function` when the first cell lies clear of the viscous sublayer, at y^+ between 30 and 250, where the law of the wall is applied. Use `slip` where the wall should exert no shear. Use the compressible wall function on a heated wall or at speed. Giving the wall a temperature or a heat flux selects it.

`kind` (p. 100)	FIRST CELL y^+	AT THE WALL
`slip`	any	zero momentum normal to the wall, no shear
`no slip`	1 to 5, in the viscous sublayer	the no-slip condition, applied directly
`wall function`	30 to 250, clear of the viscous sublayer	the law of the wall, from the first cell's state

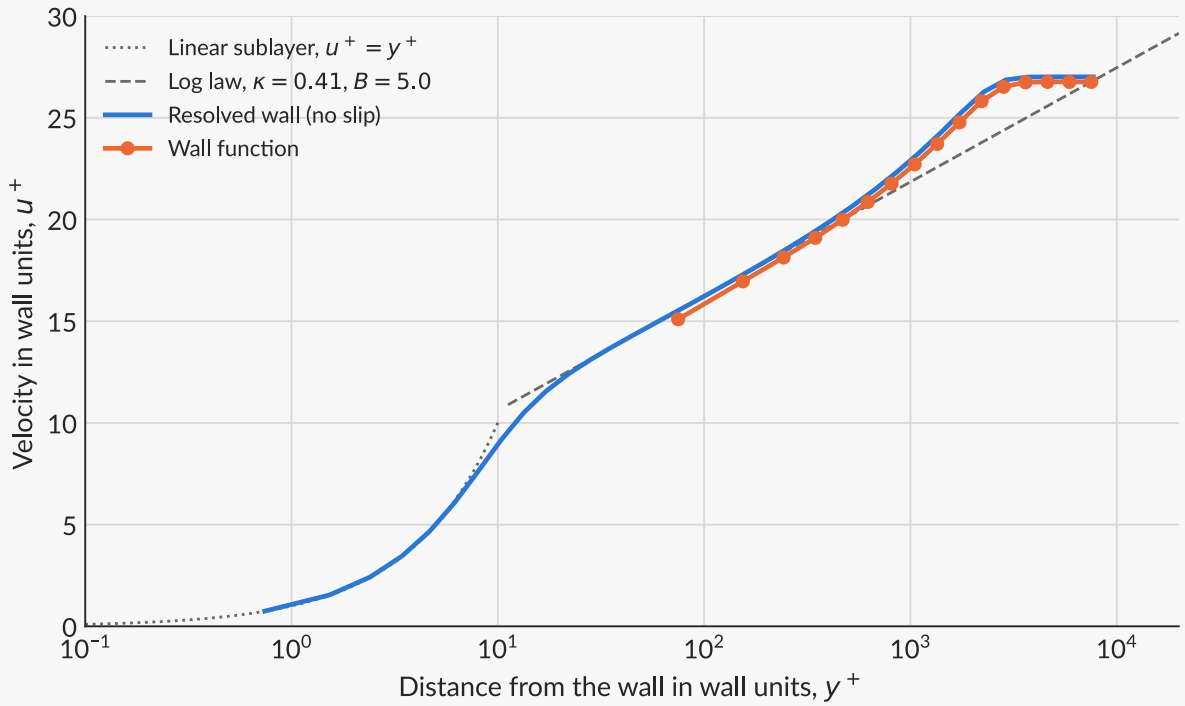
Good practice is to target $y^+ = 1$ on a wall-resolved mesh and $y^+ = 100$ on a wall-modelled mesh.

Slip, no slip or wall function

The `kind` (p. 100) of a wall boundary condition is `slip`, `no slip` or `wall function`. Use `no slip` only where the mesh resolves the viscous sublayer, and a wall function elsewhere. A wall may also carry a `velocity` (p. 99), a `roughness` (p. 99), a `temperature` (p. 99) and an FSI modal model, `fsi` (p. 99).

The best check of a mesh is a histogram of y^+ over the wall from a converged solution, plotted from the `yplus` (p. 173) surface output variable, which is written by default. y^+ scales approximately linearly with the first cell height. If the histogram sits 2.5 times above the target, reduce the first cell height by a factor of 2.5.

Demonstration: what `kind` does FROM THE PLATE CASE



Velocity across the boundary layer of a flat plate in wall units, for a resolved wall and a wall function, against the linear sublayer and the log law

Both runs use the Spalart–Allmaras model on meshes with the same number of cells. The wall-function mesh places its first cell a hundred times further from the wall.

```
setting kind values no slip, wall function case plate
```

Wall function

The law of the wall depends on the turbulence model. Menter SST uses Spalding's law, a single expression that spans the viscous sublayer, the buffer layer and the log layer. SA-neg uses the Allmaras fit, which spans the same range; with the incompressible law of the wall it is applied where the first cell lies below $y^+ = 200$, and above that the log law alone is used. Because both laws hold through the sublayer, a wall-function wall tolerates a first cell anywhere from $y^+ \sim 1$ into the log layer, so a mesh whose first-cell height varies along the wall needs no switch between wall treatments. Even so, size the mesh for a first cell between $y^+ = 30$ and 250.

By default a `'wall function'` wall uses an incompressible law of the wall. The velocity profile is solved with the density and viscosity of the first cell off the wall, and no heat is transferred.

Giving the wall a `temperature` (p. 99) or a `heat flux` (p. 99) switches it to the compressible wall function. This is the Van Driest II / White–Christoph velocity transformation coupled to the Crocco–Busemann thermal law of the wall. It evaluates y^+ and the wall shear stress with the density and viscosity of the wall state, and produces the wall heat flux. With a `heat flux` the wall temperature is recovered; with a `temperature`, the heat flux. Both are available as the `heatflux` and `twall` surface output variables.

The velocity transformation is worth having at speed whether or not the wall exchanges heat, since the boundary layer of an adiabatic wall is still stratified by viscous heating. `compressible wall`

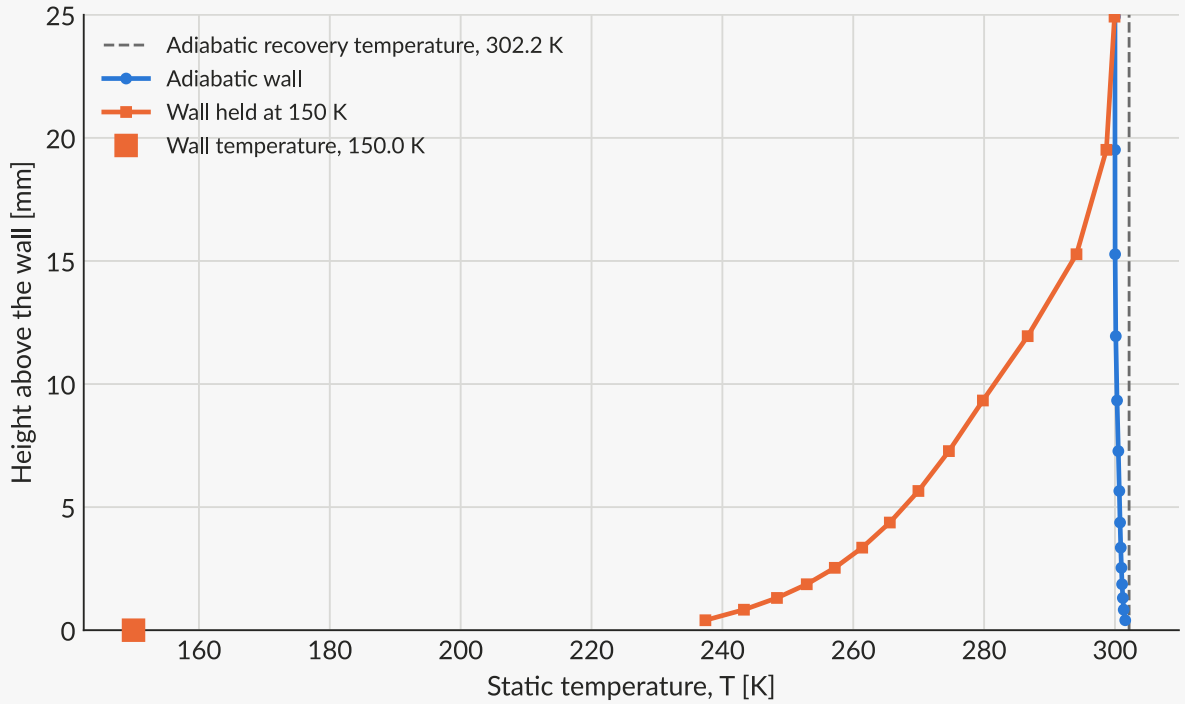
`function` (p. 101) runs it on an adiabatic wall. The wall state is then evaluated at the recovery temperature T_r rather than at a specified one, so the wall heat flux is zero by construction:

```
{
    "boundary_conditions": {
        "BC_2": {
            "zones": [2],
            "type": "wall",
            "kind": "wall function",
            "compressible wall function": True,
        },
    },
}
```

`twall` then reports T_r, and `heatflux` reports zero. The key is redundant alongside a `temperature` or a `heat flux`, which select the same transformation and add the thermal closure to it.

A constant-heat-flux wall is singular where the boundary layer is thin, at a plate leading edge for example. The modelled layer cannot carry an arbitrarily large flux there, and the wall temperature rises until it is limited internally. Check the `twall` output if a prescribed flux is large.

Demonstration: what `temperature` does FROM THE PLATE CASE

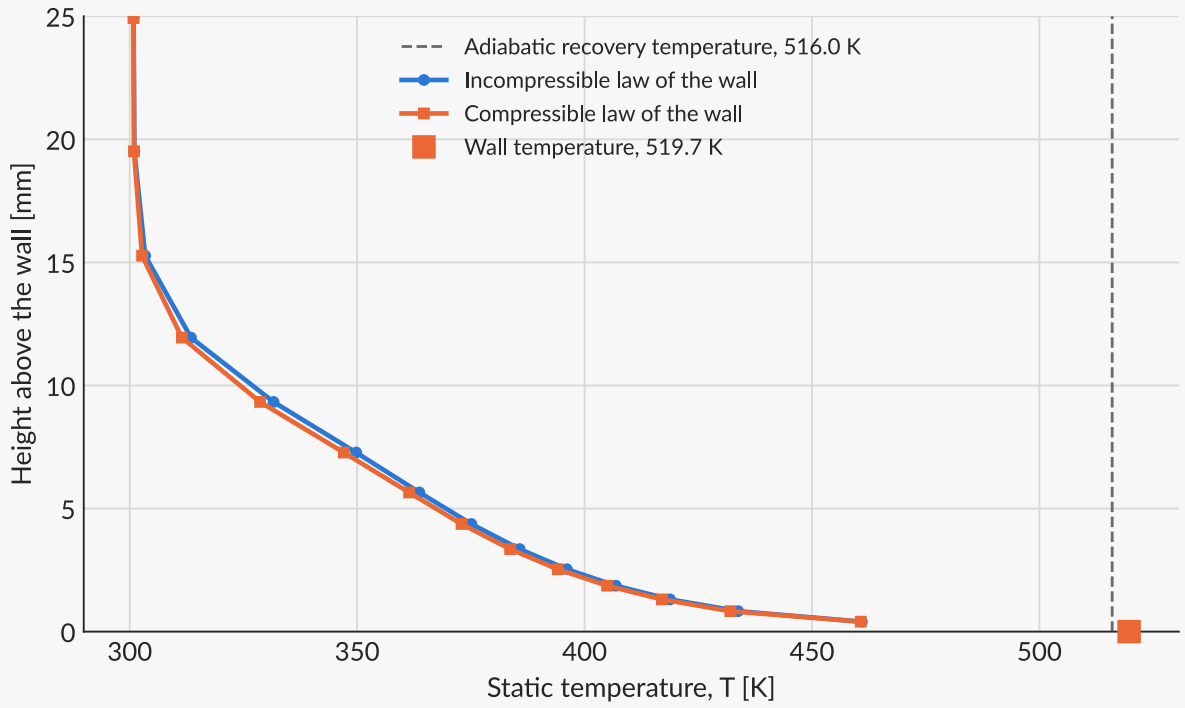


Static temperature across the boundary layer of a flat plate at M = 0.2, for an adiabatic wall and a wall held at 150 K, against the adiabatic recovery temperature

Both runs use Menter SST with a wall function at M = 0.2 on the same mesh. Setting a wall temperature also selects the compressible wall function.

`setting` **`temperature`** `values` **`unset, 150 K`** `case` **`plate`**

Demonstration: what `compressible wall function` does FROM THE PLATE CASE



Static temperature across the boundary layer of a flat plate at M = 2, for the incompressible and the compressible law of the wall, against the adiabatic recovery temperature

Both runs are adiabatic at M = 2, with Menter SST and a wall function on the same mesh.

```
setting compressible wall function values false, true case plate
```

The compressible wall function, however it is selected, is not supported in the incompressible solver, and is rejected at input validation.

Rough compressible walls

Roughness may be combined with the compressible wall function. This is the combination an icing calculation needs. Ice accretion roughens the surface, which raises both the skin friction and the convective heat transfer that in turn drives the accretion, while an ice protection system imposes a wall `heat flux` or `temperature`. Both the roughness and the thermal state accept a `field`, so an ice roughness map and an electrothermal pad layout can be prescribed face by face from VTP files. Each `field` is a path relative to the directory zCFD is run from, and a deck whose file is missing or cannot be read is rejected at validation. A roughness map must carry a point array named `Roughness`, and the solver stops at setup if any of its values is negative or not a number.

The equivalent sand-grain height is given in mesh units, like every other length in the input deck. Nothing rescales the mesh, so a 0.5 mm ice roughness on a mesh in metres is `0.0005`.

On the velocity side, the Van Driest transformed velocity obeys the incompressible rough-wall law, with the roughness Reynolds number k_s^+ formed from the density and viscosity of the wall, not of the first cell. That is the scaling the compressible wall function already uses for y^+ and the wall shear stress; on a heated, anti-iced wall the two differ substantially.

On the thermal side, the Crocco–Busemann closure is an exact Reynolds analogy, so on its own it would

raise the Stanton number in the same ratio as the skin friction. Roughness elements shed momentum by pressure drag but carry heat only by conduction, so in reality the heat transfer rises much less. The default `thermal model` (p. 109), `'owen thomson'`, therefore adds a thermal sublayer resistance

$$\Delta T^{+} = 0.52\,(k_s^{+})^{0.45}\,Pr^{0.8}$$

between the surface and the roughness crest. It is blended in over the same k_s^{+} range as the velocity intercept, so a hydraulically smooth wall is unaffected. Setting `thermal model` to `'reynolds analogy'` removes it.

On a rough wall the `twall` output is the temperature of the surface, not of the roughness crest on which the law of the wall is closed; the two differ by the drop across that resistance. `heatflux` is continuous through it, and so is unambiguous.

An anti-iced leading edge with a measured ice roughness map:

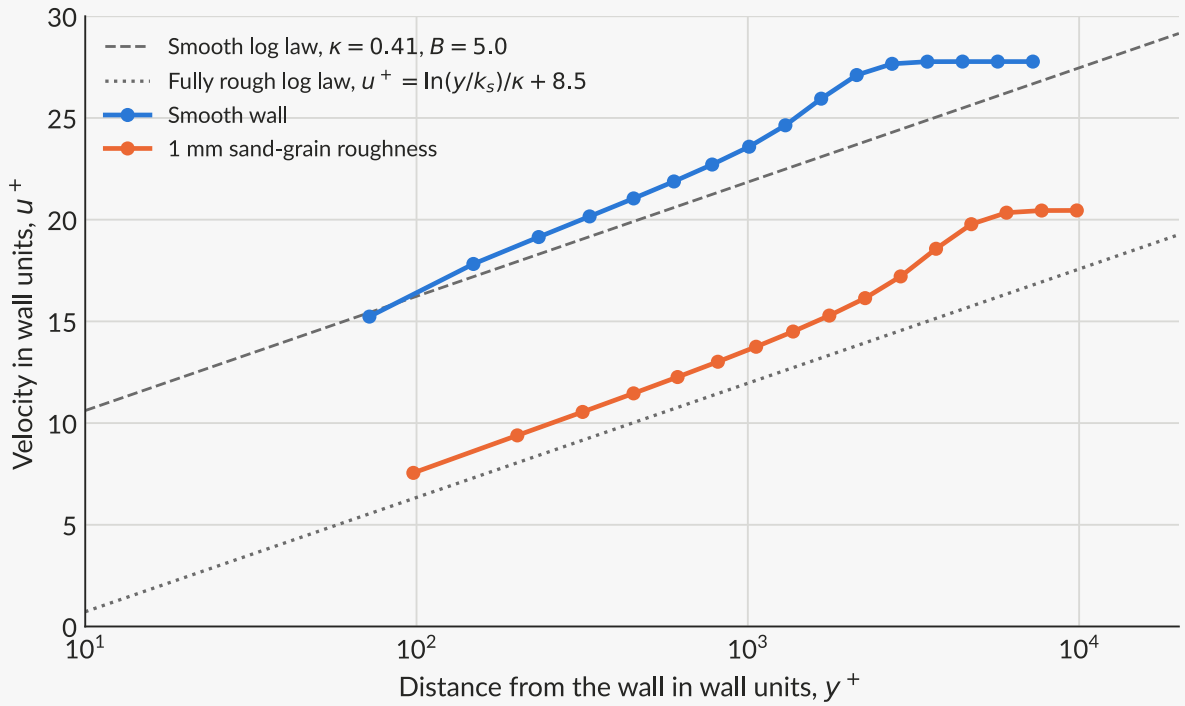
```
{
    "boundary_conditions": {
        "BC_3": {
            "type": "wall",
            "kind": "wall function",
            "heat flux": {"field": "ips_pads.vtp"},
            "roughness": {"type": "height", "field": "ice_roughness.vtp"},
        },
    },
}
```

Roughness

Roughness is applied by the wall-function treatment of both the compressible and the incompressible solver, for the Menter SST and SA-neg turbulence models. On a `no slip` or `slip` wall it has no effect.

Internally the models work in equivalent sand-grain height, so a roughness given as a `length` is converted on input, as `height = length / 0.03`. The `roughness` surface output variable reports that height, not the value written in the deck.

Demonstration: what `roughness` does FROM THE PLATE CASE



Velocity across the boundary layer of a flat plate in wall units, for a smooth wall and a wall with 1 mm sand-grain roughness, against the smooth log law and the fully rough log law

Both runs use Menter SST with a wall function on the same mesh; only the roughness differs.

`setting` **`roughness`** `values` **`unset, height 0.001 m`** `case` **`plate`**

Moving walls

`velocity` (p. 99) applies a surface velocity, for surfaces in relative motion: a car moving over a road surface, for example, or a rotating tyre. A rotating wall velocity takes an `axis`, an `origin` and an `omega`, as a rotating frame does:

```
"BC_3": {"type": "wall", "kind": "wall function", "zones": [14, 20],
         "velocity": {"type": "rotating", "axis": [0.0, -0.9988, 0.0498],
                      "origin": [0.0, -0.6593, 0.2533], "omega": 168.6}},
```

Reference entries

- `wall` (p. 99): `kind` (p. 100), `temperature` (p. 99), `heat flux` (p. 99), `compressible wall function` (p. 101), `roughness` (p. 99), `velocity` (p. 99) and `fsi` (p. 99)

Choosing a time-marching scheme

Should the run be steady or unsteady, and how should its pseudo-time loop be paced?

Run steady when the flow has a steady solution. Run unsteady flow with dual time stepping. An inner pseudo-time loop converges each physical time step, so the time step can be chosen to resolve the flow. An implicit pseudo-time scheme takes a larger CFL number and far fewer cycles than an explicit one, at three to five times the memory. Global time stepping is the last option. It marches every cell at one explicit time step, which the smallest cell limits.

`time_settings` (p. 51)	PHYSICAL TIME	PSEUDO-TIME LOOP	`convergence_control`	TIME ACCURACY FROM
`{"type": "steady"}`	none	the whole run	required	—
`{"type": "unsteady", "kind": "dual time stepping"}`	one time step, each converged by an inner loop	within every physical step	required	the backward difference `order`
`{"type": "unsteady", "kind": "global timestepping"}`	one time step for every cell	none	must not be set	the three-stage TVD Runge–Kutta `scheme`

The pseudo-time schemes, selected by `name` (p. 63) under `convergence_control`:

`name`	KIND	LINEAR SOLVE	USE
`"euler"`	explicit, one stage	none	new or problematic cases; the most stable starting point
`"runge kutta"`	explicit, multi-stage	none	explicit runs, usually with five stages; multigrid
`"implicit euler"`	implicit	the assembled system, by Petsc or AMGX (p. 262)	the default implicit scheme
`"lu-sgs"`	implicit, matrix-free	none; point-implicit sweeps	lower memory and often faster cycles than `"implicit euler"`
`"mf-gmres"`	implicit, matrix-free mean flow	restarted GMRES on the exact analytic Jacobian	between the two, without assembling the mean-flow matrix

Time settings and convergence control

A steady-state solution is marched in pseudo-time towards convergence. A time-dependent simulation uses dual time stepping, with a local pseudo-time loop inside each physical time step, or global time stepping.

Two sibling dictionaries under `solver` control this:

- `time_settings` (p. 51) sets out physical time: whether the simulation is steady, dual time-stepped or globally time-stepped, and, for an unsteady run, the total time, the physical time step and the physical time order. `type` selects steady or unsteady operation, and for an unsteady run `kind` selects dual or global time stepping.
- `convergence_control` (p. 56) sets how each pseudo-time step marches towards convergence: the number of cycles, the pseudo-time scheme, explicit or implicit, and the CFL number and its ramp.

> **WARNING**
>
> `convergence_control` does not apply under global time stepping, which marches physical time with no pseudo-time loop. A deck that sets it alongside `{"time_settings": {"kind": "global timestepping"}}` fails validation. Global time stepping takes its `scheme` inside `time_settings` (see Global time stepping).

```
parameters = {
    "config_version": 2,
    "solver": {
        ...
        "time_settings": {...},
        "convergence_control": {
            "cycles": 5000,
            "scheme": {...},
            "cfl": 30.0,
        },
        ...
    },
    ...
}
```

Running steady state

For a steady-state simulation, set `type` to `"steady"`. It takes no further keys: a steady run has no physical time layout, so `total_time`, `time_step` and `kind` are neither needed nor accepted, and all of its pacing comes from `convergence_control`.

```
"time_settings": {
    "type": "steady",
},
```

> **NOTE**
>
> Setting `total_time` equal to `time_step` under dual time stepping is rejected outright, as a validation error. Use `{"type": "steady"}` for a steady-state run instead.

Dual time stepping

Dual time stepping (DTS) advances the solution in physical time, running an inner pseudo-time loop at each physical time step to converge that step before advancing. The time integration uses the solution at the current physical time step and the contribution of previous steps, through a backward difference formula; `order` (p. 52) sets the order of the formula.

`start` (p. 53) initialises a DTS simulation with a number of steady-state pseudo-time cycles before physical time stepping begins. These run at real-time cycle 0, with no physical time derivative, so they are the same solve as a `"steady"` run of the same length. Physical time stepping begins at real-time cycle 1, where the order of the time integration builds up from first order to `order`. Restarting a DTS case from the results of a steady run, or from a results file written at the end of the start phase, skips real-time cycle 0 and begins time-accurate stepping at once, with the restart field as the time history.

The pseudo-time scheme, CFL number and cycle count for each physical step are set in `convergence_control`, exactly as for a steady run; see Convergence control.

The unsteady laminar cylinder sheds a vortex street at a particular Strouhal number. This deck runs it for 4 seconds of physical time, at a physical time step of 0.002 seconds, small enough to capture the shedding, with 20 pseudo-time cycles of multistage Runge–Kutta per physical step to converge each step:

```
"time_settings": {
    "type": "unsteady",
    "kind": "dual time stepping",
    "total_time": 4.0,
    "time_step": 0.002,
    "order": 2,
    "start": 0,
},
"convergence_control": {
    "cycles": 20,
    "scheme": {"name": "runge kutta", "stage": 5},
    "cfl": {
        "cfl": 2.5,
        "cfl_ramp": [{"type": "exponential", "factor": 1.05, "initial": 0.1}],
    },
},
```

The next uses an implicit pseudo-time scheme, so far fewer cycles are needed per physical step, and starts with 500 steady-state pseudo-time cycles before the unsteady simulation begins:

```
"time_settings": {
    "type": "unsteady",
    "kind": "dual time stepping",
    "total_time": tt,
    "time_step": tt / 400,
    "order": "second",
    "start": 500,
},
"convergence_control": {
    "cycles": 500,
    "scheme": {"name": "implicit euler", "stage": 1},
    "cfl": {
        "cfl": 15,
        "cfl_ramp": [{"type": "exponential", "factor": 1.02, "initial": 1.0}],
    },
```

```
},
```

Global time stepping

Global time stepping (GTS) advances the solution in physical time alone, with the same time step in every cell. It has no pseudo-time loop, so `convergence_control` does not apply. The time step has to meet the CFL limit of the smallest cell, and the solver logs a warning when it does not.

Order of accuracy

Under GTS the `scheme` integrates physical time directly, and sets the order of accuracy of the run. `order` and `start` belong to dual time stepping. To start a GTS run from a converged field, restart it from the results of a steady run.

The one scheme available is the three-stage TVD Runge–Kutta (SSP-RK3), which is third order in physical time. It is written either way:

```
"scheme": {"name": "runge kutta", "stage": 3},
"scheme": {"name": "runge kutta", "stage": "rk third order tvd"},
```

A globally time-stepped unsteady simulation:

```
"time_settings": {
    "type": "unsteady",
    "kind": "global timestepping",
    "total_time": 0.2,
    "time_step": 1.0e-5,
    "scheme": {
        "name": "runge kutta",
        "stage": 3,
    },
},
```

Convergence control

`convergence_control` paces the inner pseudo-time loop towards convergence: how many cycles to run, which scheme drives each pseudo-step, and, on the compressible solver, the CFL number that sizes it. It is required for steady and dual time-stepping runs, and must be absent under global time stepping (see the warning under Time settings and convergence control).

The rest of this section describes the compressible block. `cycles` is common to both solvers; the incompressible block replaces `scheme` with the pressure-velocity coupling scheme, and takes no `cfl` (see Incompressible convergence control).

```
"convergence_control": {
    "cycles": 5000,
```

```
    "scheme": {"name": "implicit euler"},
    "cfl": 30.0,
},
```

Cycles

`cycles` (p. 57) is the total number of pseudo-time steps in a steady-state simulation, or the number in each physical time step of a dual time-stepping simulation.

For most explicit steady-state cases, 5000 cycles is a good starting value. Implicit simulations can use a higher CFL number and so need fewer cycles; 100 to 500 is a good starting point for an implicit steady-state case.

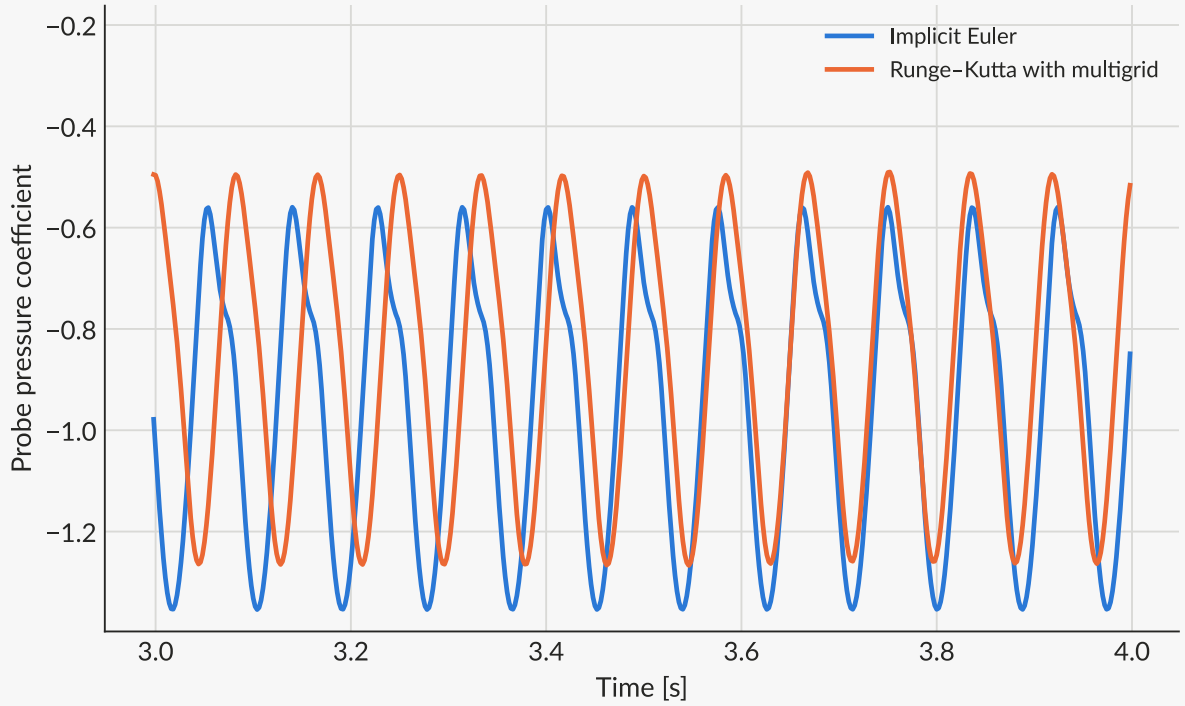
```
"cycles": 5000,
```

Scheme

zCFD has five pseudo-time integration schemes: two explicit, `"euler"` and `"runge kutta"`, and three implicit, `"implicit euler"`, `"lu-sgs"` and `"mf-gmres"`. All five serve steady runs and dual time-stepped unsteady runs. Global time stepping takes only the three-stage TVD Runge–Kutta (see Global time stepping). A case with an overset or sliding interface requires one of the implicit schemes (see Interface (p. 288)).

> **NOTE**
>
> The `"euler"` scheme has a single stage, and is the safest option for new or problematic cases.

Demonstration: what `scheme` does FROM THE CYLINDER CASE



Probe pressure coefficient against time behind a 2D laminar cylinder, with an implicit Euler and a Runge–Kutta inner loop

Both runs use dual time stepping with the same time step and mesh; only the inner-loop scheme, its CFL, its cycles and the multigrid levels change.

```
setting scheme values implicit euler, runge kutta case cylinder
```

name

The solution is integrated in pseudo-time by a scheme that is either explicit, with the variables driving the change evaluated at the previous step, or implicit, with them evaluated at the current step. `"euler"` and `"runge kutta"` are the explicit schemes; `"implicit euler"`, `"lu-sgs"` and `"mf-gmres"` the implicit ones.

An implicit scheme is more stable than an explicit one, and so can take a larger CFL number, and with it a larger pseudo-time step.

> **NOTE**
> The cost of implicit time integration is memory: typically three to five times that of an explicit scheme.

stage

The number of Runge–Kutta stages in each pseudo-time step affects the order of temporal convergence and the largest usable CFL number.

> **NOTE**
> Most cases run with Runge–Kutta integration use five stages. This is a typical industry standard, and has proved reliable for most cases.

A five-stage explicit Runge–Kutta scheme:

```
"convergence_control": {
    "scheme": {"name": "runge kutta", "stage": 5},
    ...
},
```

implicit euler

The default implicit scheme. It solves the full linearised system at each pseudo-time step with the Petsc or AMGX linear solver (p. 262). An implicit scheme with a ramped CFL number, typical of a steady-state RANS case:

```
"convergence_control": {
    "scheme": {"name": "implicit euler"},
    "cfl": {
        "cfl": 100.0,
        "cfl_ramp": [{"type": "exponential", "factor": 1.1, "initial": 0.05}],
    },
    "cycles": 3000,
},
```

lu-sgs

A matrix-free, point-implicit LU-SGS/DP-LUR scheme. It avoids assembling and inverting the full linear system, trading some convergence rate for lower memory use and, often, a faster wall-clock time per cycle than `"implicit euler"`. LU-SGS with a CFL ramp:

```
"convergence_control": {
    "scheme": {"name": "lu-sgs"},
    "cfl": {
        "cfl": 20.0,
        "cfl_ramp": [{"type": "exponential", "factor": 1.1, "initial": 0.05}],
    },
    "cycles": 30000,
},
```

`mf-gmres`

A restarted GMRES solve of the mean-flow system that applies the exact analytic flux Jacobian matrix-free: no mean-flow matrix is assembled. It sits between `"implicit euler"`, which assembles the matrix and solves it with AMG, and `"lu-sgs"`, which relaxes without a Krylov method. Its solver settings sit on the scheme itself, not in `linear_solver_options` (p. 262). `"mf-gmres"` is not available with the SAS turbulence model or the SAS hybrid.

`"mf-gmres"` covers the mean flow only. The turbulence matrix is always assembled, and `turbulence_solver` (p. 64) chooses how it is solved: `"point-implicit"` sweeps it matrix-free, and `"amg"` uses the assembled AMG solve. `"point-implicit"` is the default, except where the backend forces `"amg"`: a CUDA build running on the host only, or HIP.

`"mf-gmres"` as FGMRES with the block-Jacobi preconditioner, for a transonic RANS wing:

```
"convergence_control": {
    "scheme": {
        "name": "mf-gmres",
        "flexible": True,
        "krylov_subspace": 30,
        "linear_tolerance": 0.1,
        "max_iterations": 60,
        "preconditioner": "block jacobi",
        "turbulence_solver": "point-implicit",
    },
    "cfl": 25.0,
    "cycles": 2000,
},
```

Every setting is optional. The solver defaults are:

SETTING	DEFAULT	EFFECT
`krylov_subspace` (p. 64)	10	Restart length m. The solver stores m+1 Krylov vectors, or 2m+1 with `flexible`, so this is the dominant storage cost; size it from the observed iteration count rather than padding it.
`linear_tolerance` (p. 61)	0.1	Relative residual drop required of the linear solve at each implicit step.
`max_iterations` (p. 61)	40	Cap on the total Krylov iterations, across restarts, per implicit step.
`preconditioner` (p. 64)	`"block jacobi"`	`"block jacobi"` inverts the exact analytic 5×5 block diagonal, the same operator on the host and on a GPU. `"lu-sgs"` uses linearised symmetric Gauss–Seidel sweeps. `"lu-sgs nonlinear"` uses the unlinearised sweep, and requires `flexible: True`. `"none"` applies no preconditioner.
`preconditioner_sweeps` (p. 61)	2	Symmetric Gauss–Seidel sweeps per application of the preconditioner (`"lu-sgs"` only).
`preconditioner_over_relaxation` (p. 65)	1.5	Diagonal over-relaxation for the `"lu-sgs"` preconditioner sweeps.
`flexible` (p. 65)	`False`	FGMRES instead of right-preconditioned GMRES. It permits a preconditioner that varies between iterations, at the cost of a second set of m Krylov vectors.
`orthogonalisation` (p. 65)	`"mgs"`	`"mgs"` is modified Gram–Schmidt. `"cgs2"` is classical Gram–Schmidt with one reorthogonalisation pass and batched reductions: O(m) reductions per restart cycle rather than O(m²), at twice the vector work, so it is expected to pay only at high rank counts.
`turbulence_solver` (p. 64)	`"point-implicit"`	How the assembled turbulence matrix is solved (see above).
`turbulence_sweeps` (p. 65)	8	Sweeps for the `"point-implicit"` turbulence solve.

> NOTE
>
> On a GPU the `"lu-sgs"` preconditioner is the weaker DP-LUR Jacobi form rather than symmetric Gauss–Seidel, whereas `"block jacobi"` is the same operator on the host and on the device.

CFL control

The Courant–Friedrichs–Lewy (CFL) number sets the local pseudo-time step with which the solver approaches a converged solution. The larger the CFL number, the faster the solver runs, and the less stable it is. `cfl` (p. 56) takes either a plain number or a dictionary for finer control.

> NOTE
>
> A bare number, such as `"cfl": 30`, is shorthand for `"cfl": {"cfl": 30}`. Any of the other keys can be added alongside it; zCFD gathers them into the equivalent dictionary form.

cfl

Sets the CFL number used in the mass, momentum and energy equations. A typical value for a RANS simulation with explicit time marching:

```
"cfl": 1
```

and with implicit time marching:

```
"cfl": 30
```

> NOTE
>
> After mesh quality, the CFL number is the most important parameter affecting the stability of a simulation. If the simulation does not converge, reduce it.

cfl_turbulence

Sets the working CFL number for the turbulence transport equations, such as those for k and ω. A lower value than `cfl` promotes stability:

```
"cfl": {"cfl": 30, "cfl_turbulence": 20},
```

> NOTE
>
> If the simulation does not converge, and reports high turbulence residuals that increase with successive cycles, set `cfl_turbulence` lower than `cfl`.

cfl_coarse

With multigrid acceleration, `cfl_coarse` is the CFL number on the coarse meshes, while `cfl` still applies to the finest mesh. It applies only to explicit finite-volume solutions, and only when `{"numerical_scheme": {"multigrid": {...}}}` is set. Agglomerated coarse meshes are typically of lower quality than the fine mesh, so a lower CFL number may be needed for numerical stability:

```
"cfl": {"cfl": 2.5, "cfl_coarse": 1.0, "cfl_turbulence": 1.0},
```

cfl_ramp

A ramp allows a relatively large target CFL number to be specified while the simulation starts from a smaller one, for stability while large non-physical transients are cleared from the solution. Starting a simulation at a large CFL number can cause an instability that crashes the solver; a gentler start reduces the effect of those transients.

`cfl_ramp` is always a list. Each entry describes one ramp and names its shape with a `type` key; a single ramp is a list of one. A ramp that starts the CFL number at 0.1 and grows it by a factor of 1.1 until the target of 30 is reached:

```
"cfl": {
    "cfl": 30,
    "cfl_ramp": [{"type": "exponential", "factor": 1.1, "initial": 0.1}],
},
```

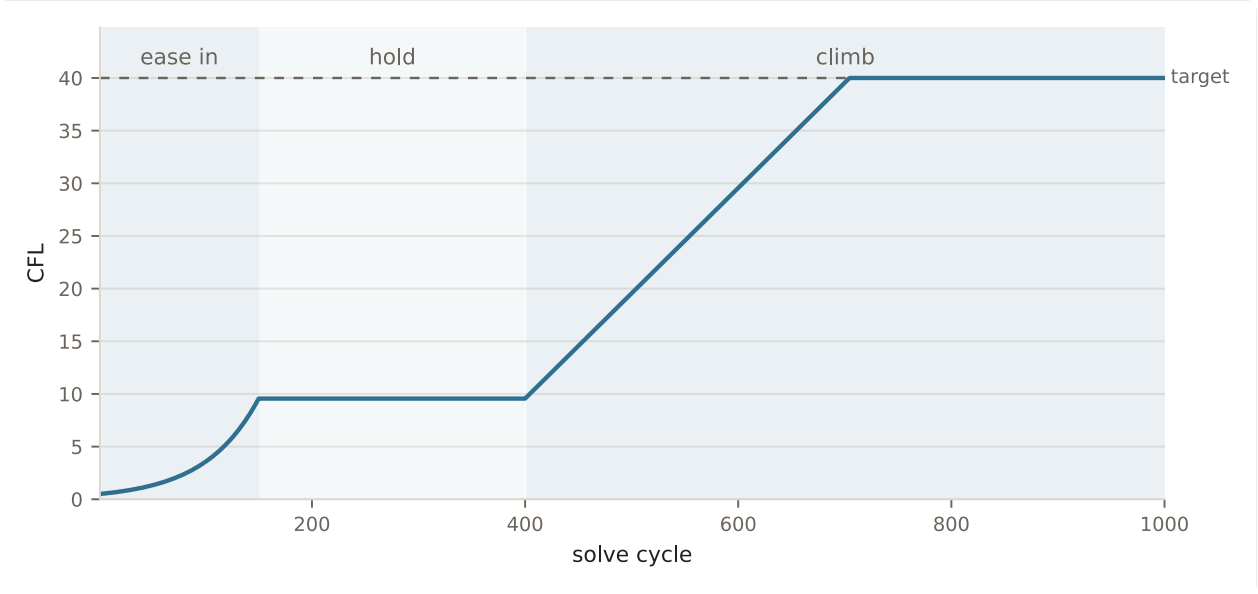
The available shapes are those of the ramp builders in `zutil.control.ramp`: `"linear"`, `"stepped_linear"` and `"exponential"`.

Several entries run one after another. Every entry is called on every cycle, each acting on the output of the one before, so an entry confines itself to its own stretch of the run with `start_cycle` and `end_cycle`. Easing in exponentially for 200 cycles, then climbing steadily to the target:

```
"cfl": {
    "cfl": 30,
    "cfl_ramp": [
        {"type": "exponential", "factor": 1.1, "initial": 0.1, "end_cycle": 200},
        {"type": "linear", "increment": 0.5, "start_cycle": 201},
    ],
},
```

A `"linear"` entry with an `increment` of `0` holds the CFL number steady for a stretch. This is useful when a case needs time at a low CFL number to shed its start-up transients before the CFL number is raised: ease in, hold while the flow settles, then climb to the target. Here the first entry eases in from 0.5, the second holds while the start-up transients wash out, and the third climbs steadily to the target:

```
"cfl": {
    "cfl": 40,
    "cfl_ramp": [
        {"type": "exponential", "factor": 1.02, "initial": 0.5, "end_cycle": 150},
        {"type": "linear", "increment": 0.0, "start_cycle": 151, "end_cycle": 400},
        {"type": "linear", "increment": 0.1, "start_cycle": 401},
    ],
},
```



CFL rising exponentially to cycle 150, holding flat until cycle 400, then climbing linearly to the target of 40

Cycles that no entry covers hold the CFL number where it is, so the middle entry above could be left out and the profile would be the same. Writing the hold makes it visible to the next reader of the control dictionary.

Every shape takes `initial`, the CFL number the ramp starts from. It is returned at the ramp's `start_cycle`, so the ramp sets its own starting point:

```
"cfl": {
    "cfl": 30,
    "cfl_ramp": [
        {
            "type": "exponential",
            "factor": 1.1,
            "initial": 0.1,
            "max_allowed": 30,
        }
    ],
},
```

A Python function can be given instead of the list; see the ramp function (p. 219) reference. The function is called as `f(cycle=..., current_cfl=...)`, so it is written `def f(*, cycle, current_cfl)`, and returns either the new CFL number or a dictionary mapping any of `cfl`, `cfl_turbulence` and `cfl_coarse` to drive those alongside it; keys it does not return keep their static value.

cfl_viscous_factor

A factor on the diffusive, viscous, time step when the CFL condition is applied. Viscous effects are rarely the cause of solver instability, so their contribution to the time step can be reduced by setting `cfl_viscous_factor` below 1.

In a low Reynolds number case the viscous effects dominate, but do not contribute to instability. A value of 1e-6 then effectively discounts viscosity in setting the time step:

```
"cfl": {
    "cfl": 200,
    "cfl_viscous_factor": 1.0e-6,
    "cfl_ramp": [{"type": "exponential", "factor": 1.05, "initial": 0.1}],
},
```

Inner cycle convergence

Choosing `cycles` for an unsteady simulation means guessing how many pseudo-time iterations each real time step needs at the chosen `cfl`. The `inner_convergence` (p. 56) dictionary lets the solver decide instead: each real time step ends once its inner iteration has converged, and `cycles` becomes an upper bound rather than a fixed count.

It applies only to dual time-stepping runs. A steady or globally time-stepped run has no inner iteration to converge, and a deck that sets it for one fails validation. Leave the dictionary out to run the full `cycles` on every real time step. Both solvers honour the same dictionary; what `target_orders` is measured on differs between them, and is set out under Choosing a target.

Each real time step here runs until the mean-flow residual has dropped two orders, but never for more than 50 inner cycles:

```
"time_settings": {
    "type": "unsteady",
    "kind": "dual time stepping",
    "total_time": 1.7,
    "time_step": 0.002,
    "order": "second",
},
"convergence_control": {
    "cycles": 50,
    "cfl": 2.0,
    "inner_convergence": {
        "target_orders": 2.0,
        "min_cycles": 5,
        "check_frequency": 5,
    },
},
```

The criterion is not applied to the first real time step. That step runs the `start` cycles from a uniform initial field while the CFL number is still ramping, so its residual history is not comparable with the rest of the simulation.

> **NOTE**
>
> A real time step that converges early still runs one further inner cycle, on which it writes its report row, checkpoint and visualisation output.

Choosing a target

`target_orders` (p. 59) is measured on `InnerRatio`: the largest, across the selected equations, of each residual divided by its own physical rate of change. Two properties matter. Dividing each equation by its own rate makes the comparison dimensionless, so taking the largest across equations is meaningful where taking the largest raw residual would not be. And taking the largest tracks the equation with the most work still to do.

`InnerDrop`, by contrast, is the smallest drop across the same equations, and is reported as a diagnostic only. Residuals are not comparable between equations, so one can start well below the others because that is its scale, with almost nothing left to resolve. The smallest drop then picks exactly that equation, and stays near zero however well the rest of the field converges. In a representative IDDES case the momentum equations dropped 2.2 orders while density dropped 0.07 and governed, so a criterion on `InnerDrop` could never have been met. `InnerRatio` on the same case fell a consistent 1.8 orders per real time step.

Because `target_orders` is a reduction in a ratio rather than an absolute level, one value carries between cases. To choose it, run once with the criterion off, and follow `InnerRatioDrop` through a real time step. Most of the reduction happens in the first handful of inner cycles at the chosen `cfl`, and the curve then has a long flat tail. On that IDDES case it reached 0.7 orders by cycle 5, 1.0 by cycle 10 and 1.8 by cycle 50, so the first five cycles achieved some four fifths of the reduction in residual, and the remaining forty-five the rest. Set `target_orders` near the knee, with `min_cycles` at the knee's cycle number.

Use `target_ratio` (p. 59) only when an absolute floor is wanted as well. It is combined with `target_orders` by a logical OR, and its value has to be measured case by case.

The equation governing `InnerRatio` can change from one inner cycle to the next, so `InnerRatioDrop` compares the worst ratio now against the worst at the first cycle rather than following a single equation. It reads as how far the bottleneck has improved, and flattens when a slower equation takes over; that is the signal, not an artefact.

> **NOTE**
>
> A solution restarted from a steady state and left to develop, from RANS into DES or a wake that has yet to break down, looks converged on every real time step while the unsteadiness is still small: the residual starts low, barely moves, and sits far below the physical rate of change. Ending those steps early would cut the inner iteration short exactly while the instability that has to grow is being resolved. Use `start_real_time_step` (p. 60) to leave that phase alone, and let the criterion take over once the flow is established.

The incompressible measure

The incompressible solver has no pseudo-time residual to form a ratio from. Its report carries the nonlinear residual of the outer iteration, already normalised by the matrix diagonal and by each variable's own range, so it is free of scale for each equation before anything divides it. `target_orders` is therefore judged on `InnerDrop`, the orders the worst selected residual has fallen since the first inner cycle of the step, and `InnerRatio` and `InnerRatioDrop` are reported as zero. `target_ratio` has nothing to test, and is rejected.

Under the coupled scheme, `"mean flow"` is momentum and pressure; the pressure row is the continuity constraint, and is the equation that most often governs. Under SIMPLE, the pressure entry in the report

is the residual left by the pressure-correction solve, bounded by the linear solver's tolerance rather than by the flow, so it is not judged, and `"mean flow"` is momentum alone.

The default of 1.0 was set on the coupled scheme, where a developed cylinder wake ends the median real time step after 10 of 50 inner cycles, and the cycles given up are worth about a quarter of an order. SIMPLE converges each step less far. On the same flow its momentum residual falls about 0.8 orders in the first five cycles and then holds, so it needs a lower target, 0.5 in the regression case, and meets it on most steps rather than all. Follow `InnerDrop` through a few steps with the criterion off before choosing, exactly as for `InnerRatioDrop` above.

Reading the inner convergence

Whether or not an automatic criterion is set, a dual time-stepping run reports its inner-cycle convergence to the report file in the `InnerCycle`, `InnerDrop`, `InnerRatio` and `InnerRatioDrop` columns, joined by `InnerCells` on the compressible solver only; see the table of columns for which carry meaning on each solver. `InnerDrop` is the number of orders of magnitude the worst mean-flow residual has dropped since the first inner cycle of the current real time step, and is the quantity to judge `cycles` by.

The global residual is a poor guide here, for two reasons. It is an absolute value, never normalised per real time step, so its magnitude says nothing about how far the current inner iteration has come. And it is an RMS over cells, so a handful of cells whose limiter switches between iterations can hold the whole norm at a noise floor while the bulk of the field is still converging cleanly.

To see the convergence of each real time step separately:

```
from zutil.plot import zCFD_Plots
plots = zCFD_Plots("cylinder.py")
plots.plot_inner_convergence()
```

This plots `InnerDrop` against the inner cycle index, with one curve for each real time step. Once the start-up transient has passed, the curves collapse onto each other, and the cycle at which they flatten is where `cycles` should be set. At the end of the run the solver also logs the median drop achieved, and whether `cycles` looks too high or too low.

If the curves flatten well short of the target, see troubleshooting (p. 354). A stall caused by limiter switching is cured by freezing the limiter, not by adding cycles; see also Choosing a limiter (p. 258).

Inner cycle convergence columns

Dual time-stepping simulations report extra columns describing the convergence of the inner, pseudo-time, iteration within each real time step. The residual columns themselves are absolute, and never normalised per real time step, so they cannot say whether a real time step has converged; these columns can. The first four are always present, and which of them carries meaning depends on the solver.

COLUMN	SOLVER	MEANING
`InnerCycle`	both	The inner cycle index within the current real time step, counting from 1. Use it as the x axis when plotting the convergence of a single real time step.

COLUMN	SOLVER	MEANING
`InnerDrop`	both	Orders of magnitude the worst selected residual has dropped since the first inner cycle of the current real time step. It is what `target_orders` tests on the incompressible solver; on the compressible solver it is a diagnostic only (see Choosing a target).
`InnerRatio`	compressible	The largest, across the mean-flow equations, of each residual divided by its own physical rate of change. Zero on real-time cycle 0, where there is no time history to compare against, and always zero on the incompressible solver, which has no pseudo-time residual.
`InnerRatioDrop`	compressible	Orders of magnitude `InnerRatio` has fallen within the current real time step. It is what `target_orders` tests on the compressible solver. Always zero on the incompressible solver.
`InnerCells`	compressible	The effective number of cells carrying the residual. A small value means limiter switching in a few cells rather than a stalled field. Not written by the incompressible solver, which has no slope limiter.

See Reading the inner convergence for how to plot these columns and use them to choose `cycles`.

> **NOTE**
> With `inner_convergence` (p. 56) enabled, the number of inner cycles varies between real time steps, so the `Cycle` column does not advance by a fixed stride per real time step. Group on `RealTimeStep` rather than assuming a stride.

Incompressible convergence control

The incompressible solver marches each pseudo-step with an implicit solve rather than a CFL-limited time step, so its `convergence_control` carries `cycles`, the pressure-velocity coupling `scheme` and two start-up ramps. There is no `cfl` key.

```
"convergence_control": {
    "cycles": 5000,
    "scheme": {"name": "coupled", "implicit_relax": 0.9},
},
```

Coupling scheme

The coupling scheme decides how velocity and pressure are solved together. `{"scheme": {"name": ...}}` selects `"coupled"` or `"simple"`, and the relaxation that drives it, such as `implicit_relax` (p. 61), sits in the same block.

The coupling scheme applies to the whole run, since it selects the solver itself. Unlike the rest of `convergence_control`, it cannot be overridden per mesh.

Under `"simple"` three further keys configure the pressure-correction solve: `pressure_relax` (p. 66), `pressure_regularisation` (p. 66) and `pressure_correction_limit` (p. 61). They are rejected under `"coupled"`, which performs no such solve. The PETSc options for the SIMPLE momentum and pressure-correction solves are separate; see Choosing a linear solver (p. 262).

```
"convergence_control": {
    "cycles": 5000,
    "scheme": {
        "name": "simple",
        "implicit_relax": 0.7,
        "pressure_relax": 0.3,
    },
},
```

Startup ramps

Both ramps, `viscosity_ramp` (p. 56) and `relaxation_ramp` (p. 56), ease an impulsive start, and are active only during the first real time step. Each takes `initial`, the value on the first pseudo-cycle, and `cycles`, the number of pseudo-cycles it takes to reach 1. Here the run starts heavily relaxed and diffusive, and eases to the configured values over 500 cycles:

```
"convergence_control": {
    "cycles": 5000,
    "scheme": {"name": "coupled", "implicit_relax": 0.9},
    "viscosity_ramp": {"initial": 100.0, "cycles": 500.0},
    "relaxation_ramp": {"initial": 0.2, "cycles": 500.0},
},
```

Multigrid

Multigrid is configured on `numerical_scheme` (p. 74), with the spatial discretisation. It applies only to compressible cases on the explicit pseudo-time path: it needs the pseudo-time loop to exist, in a steady or dual time-stepping run but never under global time stepping, and it needs the pseudo-time `scheme` to be explicit, `"euler"` or `"runge kutta"`, or unset. Setting it under the incompressible solver, under global time stepping, or alongside an implicit scheme is rejected.

To accelerate an explicit finite-volume solution, coarse meshes are generated from the original, fine, mesh by agglomerating cells into larger ones. `levels` (p. 81) sets how many times the operation is applied, each application giving a coarser mesh with fewer cells. Solving on the hierarchy of coarser meshes as well as on the fine mesh moves information through the domain faster, and reaches a converged solution more efficiently. The multigrid scheme is designed so that, in theory, the acceleration does not change the result that solving on the finest mesh alone would give; in practice, multigrid does eventually stall convergence on most meshes.

`levels` is a maximum. Coarsening stops after the first level on which every partition holds fewer than max(32, 5760/N) cells, for N partitions, or 32 cells in serial. A run on few partitions, as on GPUs, can therefore build fewer levels than a CPU run of the same case. The log prints one coarsening ratio for each level built.

Use multigrid for a set number of `cycles` (p. 81), and then revert to the finest mesh alone. This takes the more efficient scheme at the start, and avoids the stall that multigrid causes at the end of a solution.

Three levels of multigrid are enough in most cases to cut the number of cycles by an order of magnitude.

If this causes instability, reducing it to two may cure it:

```
"numerical_scheme": {
    ...
    "multigrid": {"levels": 3},
    ...
},
```

In one case, after 20000 explicit cycles the multigrid stall prevented convergence to the correct drag. Turning multigrid off after that point recovers the correct value on the finest mesh:

```
"numerical_scheme": {
    ...
    "multigrid": {"levels": 3, "cycles": 20000},
    ...
},
```

NOTE

This is geometric multigrid, on agglomerated coarse meshes. An implicit scheme solved with AMGX (see Choosing a linear solver (p. 262)) uses algebraic multigrid instead. It coarsens the assembled matrix, so it needs no coarse meshes and no `multigrid` setting.

Reference entries

- `time_settings` (p. 51): `type` (p. 51), `kind` (p. 52), `order` (p. 52), `start` (p. 53) and `scheme` (p. 51)
- `convergence_control` (p. 56): `cycles` (p. 57), `scheme` (p. 56), `cfl` (p. 56) and `inner_convergence` (p. 56)
- `multigrid` (p. 74)

Choosing a spatial scheme

Which spatial order, reconstruction and inviscid flux scheme should the compressible solver use?

Keep the defaults for general work: second order, `muscl` reconstruction from Green–Gauss gradients, and the `roe` flux. Use first order only to start a difficult case, through `first_order_cycles` (p. 74). Use `hllc` or `rusanov` for strong shocks, and a `low-diss` variant for scale-resolving simulations.

These settings are all in `numerical_scheme`:

```
{
  "numerical_scheme": {
    "order": "second",
    "reconstruction": "muscl",
    "gradient": "green-gauss",
    "inviscid_flux_scheme": "roe"
  }
}
```

A finite-volume solver stores one value of each variable per cell. The flux through a face needs a state on each side. The spatial order sets the accuracy of those states. The reconstruction sets how they are built from cell values and gradients. The inviscid flux scheme turns the pair into a flux. A slope limiter bounds the reconstruction near discontinuities; see Choosing a limiter (p. 258).

Spatial order

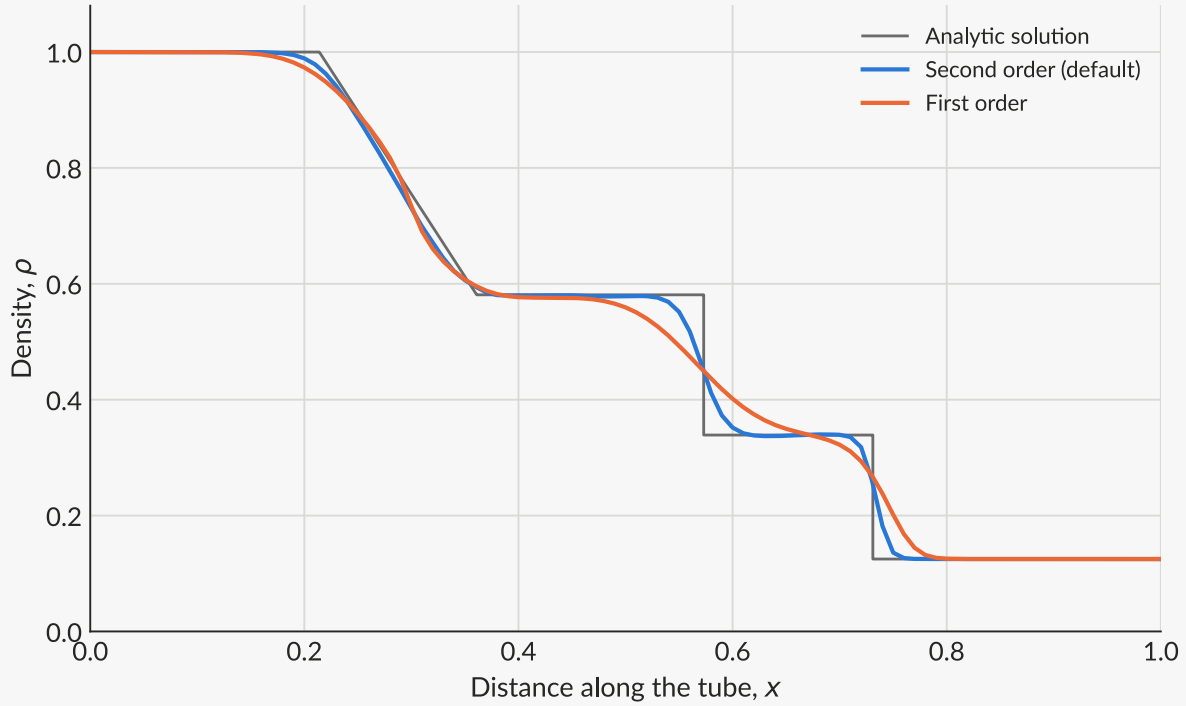
`order` (p. 75) sets the order of accuracy of the face states.

order	FACE STATES
`first`	the cell values; stable but diffusive
`second` (the default)	reconstructed from the cell values and gradients, for every variable
`euler_second`	density, velocity and pressure reconstructed at second order; turbulence variables first order

At first order a discontinuity spreads over several cells, and the error falls only linearly with mesh spacing. Second order halves the error on the shock tube below. Use it for production work. The incompressible solver also accepts `central`.

`first_order_cycles` (p. 74) runs that number of cycles at first order, then switches to the chosen order. It helps a case that diverges from a uniform initial state.

Demonstration: what `order` does FROM THE SHOCKTUBE CASE



Density along a shock tube at first and second order, against the analytic solution

Both runs use the Roe scheme and explicit three-stage Runge–Kutta on the same mesh; only the spatial order changes.

```
setting order values first, second case shocktube
```

Reconstruction

At second order each side of a face takes its cell value plus an increment. For the left cell L of face f, with right neighbour R:

$$u_{L,f} = u_L + \phi_L \left[(1-\kappa)\, \nabla u_L \cdot (\mathbf{x}_f - \mathbf{x}_L) + \tfrac{1}{2}\kappa\, (u_R - u_L) \right]$$

and the right side is built in the same way from cell R. The first term extrapolates the cell gradient to the face. The second is the jump between the two cells. The blend weight κ is 1/3. The limiter value ϕ lies between 0 and 1. $\phi = 1$ gives the full reconstruction. $\phi = 0$ gives first order on that side of the face. The reconstructed variables are density, velocity and pressure, and the turbulence variables when they are solved.

`reconstruction` (p. 78) selects how the backward difference for a one-dimensional limiter is formed. Without limiting, all options give the same face states.

`reconstruction`	BACKWARD DIFFERENCE FOR THE LIMITER
`muscl` (the default)	the cell gradient along twice the cell-to-face vector
`umuscl`	the cell gradient along the full vector between the two cell centres
`lpumuscl`	as `muscl`

The two forms differ only where a face does not lie midway between the cell centres, and only with the one-dimensional limiters. The cell-based limiters compute their own bound, so the reconstruction does not affect them.

`permit_backward_difference_during_reconstruction` (p. 74) switches the one-dimensional limiters to a backward difference taken from the cell gradient alone, on faces where the two cell-to-face vectors lie within 45° of each other. It is off by default.

Gradients

The reconstruction and the cell-based limiters read the cell gradients. `gradient` (p. 78) selects how they are computed.

gradient	METHOD
`green-gauss` (the default)	the sum of face values times face areas, divided by the cell volume
`least-squares`	the least-squares fit to the differences between a cell and its neighbours

The face values in the Green–Gauss sum are weighted by inverse distance from the two cell centres. `linear_gradients` (p. 79) replaces that weighting with a plain average. It has no effect with `least-squares`.

Inviscid flux scheme

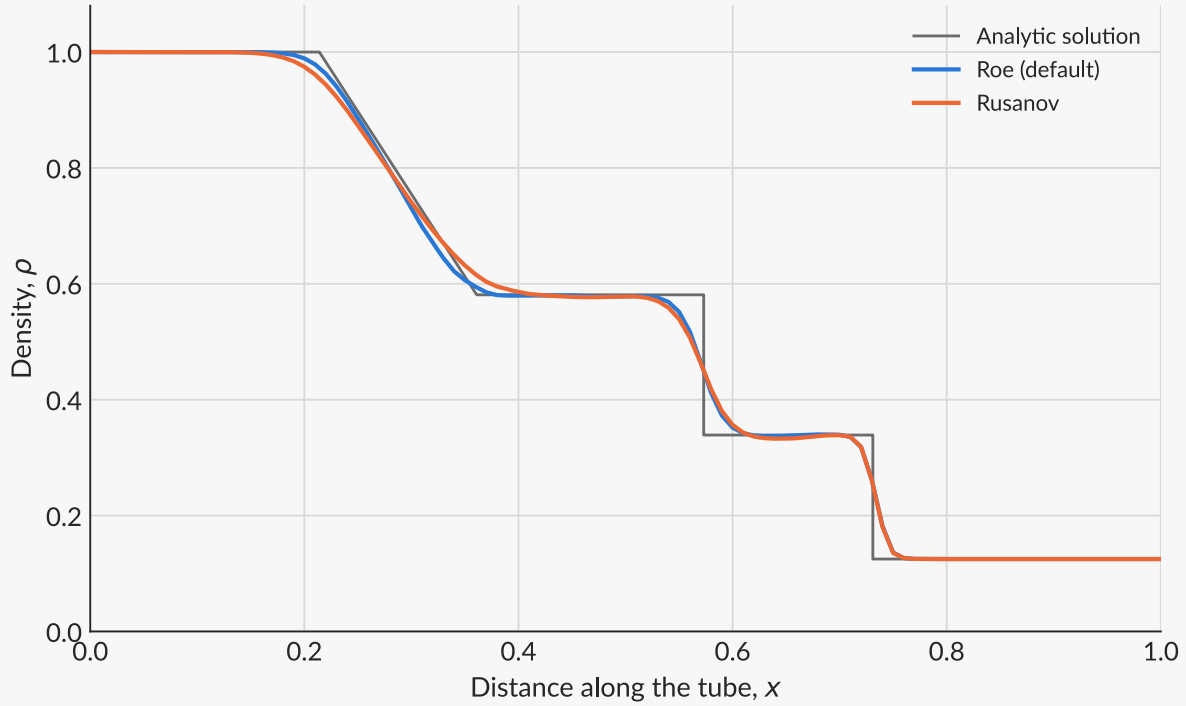
`inviscid_flux_scheme` (p. 78) selects the approximate Riemann solver that turns the two face states into a flux.

inviscid_flux_scheme	SCHEME
`roe` (the default)	Roe's linearised solver; each wave has its own dissipation
`hllc`	the HLLC solver, which resolves the contact and shear waves
`rusanov`	local Lax–Friedrichs: one dissipation scaled by the fastest wave speed; the most diffusive
`roe low-diss`	`roe` with its dissipation scaled down by a sensor
`hllc low-diss`	`hllc` with its dissipation scaled down by a sensor

`entropy_fix` (p. 74) floors the Roe wave speeds at that fraction of the largest. This adds dissipation where a wave speed passes through zero. It is zero by default. The other schemes ignore it.

The `low-diss` variants reduce the upwind dissipation where the flow is resolved. This stops resolved turbulence in a DES or LES being damped. `roe_low_dissipation_sensor` (p. 79) selects the sensor: `FD`, `NTS` or `DES`. The default, `NONE`, keeps the full dissipation. A `low-diss` scheme therefore needs a sensor to differ from its base scheme. `roe_dissipation_sensor_minimum` (p. 74) is the floor on the scaled dissipation. `roe_low_dissipation_sensor_parameter` (p. 79) is the field the sensor reads.

Demonstration: what `inviscid flux scheme` does FROM THE SHOCKTUBE CASE



Density along a shock tube for the Roe and Rusanov flux schemes, against the analytic solution

Both runs are second order with the van Albada limiter and explicit three-stage Runge–Kutta on the same mesh; only the flux scheme changes.

```
setting inviscid flux scheme values roe, rusanov case shocktube
```

Reference entries

- `order` (p. 75) and `first_order_cycles` (p. 74)
- `reconstruction` (p. 78) and `permit_backward_difference_during_reconstruction` (p. 74)
- `gradient` (p. 78) and `linear_gradients` (p. 79)
- `inviscid_flux_scheme` (p. 78) and `entropy_fix` (p. 74)
- `roe_low_dissipation_sensor` (p. 79)
- `limiter` (p. 76), on Choosing a limiter (p. 258)

Choosing a limiter

Which slope limiter should bound the second-order reconstruction of the compressible solver?

What a limiter does

At second order the solver builds each face state from the cell value plus an increment taken from the cell gradient; see Reconstruction (p. 255). In smooth flow this increment is accurate. Across a shock or a contact the gradient is large. The extrapolated face value can then overshoot the values in the neighbouring cells. The overshoot creates a new maximum or minimum. This grows into oscillations on either side of the discontinuity and can drive density or pressure negative.

A slope limiter prevents this. It multiplies the increment by a factor between 0 and 1. The factor is chosen so that the face value creates no new maximum or minimum. In smooth regions the factor stays at or near 1, so the reconstruction keeps its second-order accuracy. At a discontinuity it falls towards 0, and the face takes the cell value, as at first order.

The limiter has two costs. It also acts at smooth extrema, such as the peak of a vortex or the suction peak on an aerofoil. It clips them, which lowers the accuracy there. Where its factor switches between iterations, the residual can stall at a noise floor instead of converging.

The limiters

The limiters fall into two families. A one-dimensional limiter works face by face. It compares the jump across the face with a backward difference taken from the cell gradient. A cell-based limiter bounds every face value of a cell by the range of values over all of its neighbours. This is the stricter guarantee.

limiter	FAMILY	BEHAVIOUR
`van albada` (the default)	one-dimensional	smooth; the most diffusive of the three in the shock-tube comparison below
`van leer`	one-dimensional	smooth; slightly more compressive than van Albada
`symmetric ratio`	one-dimensional	no limiting over a wide range of difference ratios; the least dissipative of the three
`none`	—	no limiting: second order everywhere, with overshoots at discontinuities
`barth jespersen`	cell-based	bounded by the neighbours; limits wherever the bound is active, and can stall the residual
`michalak gooch`	cell-based	bounded by the neighbours; smooth; switches off in smooth flow
`venkatakrishnan`	cell-based	bounded by the neighbours; switches off in smooth flow; its shape has one kink

Which to use

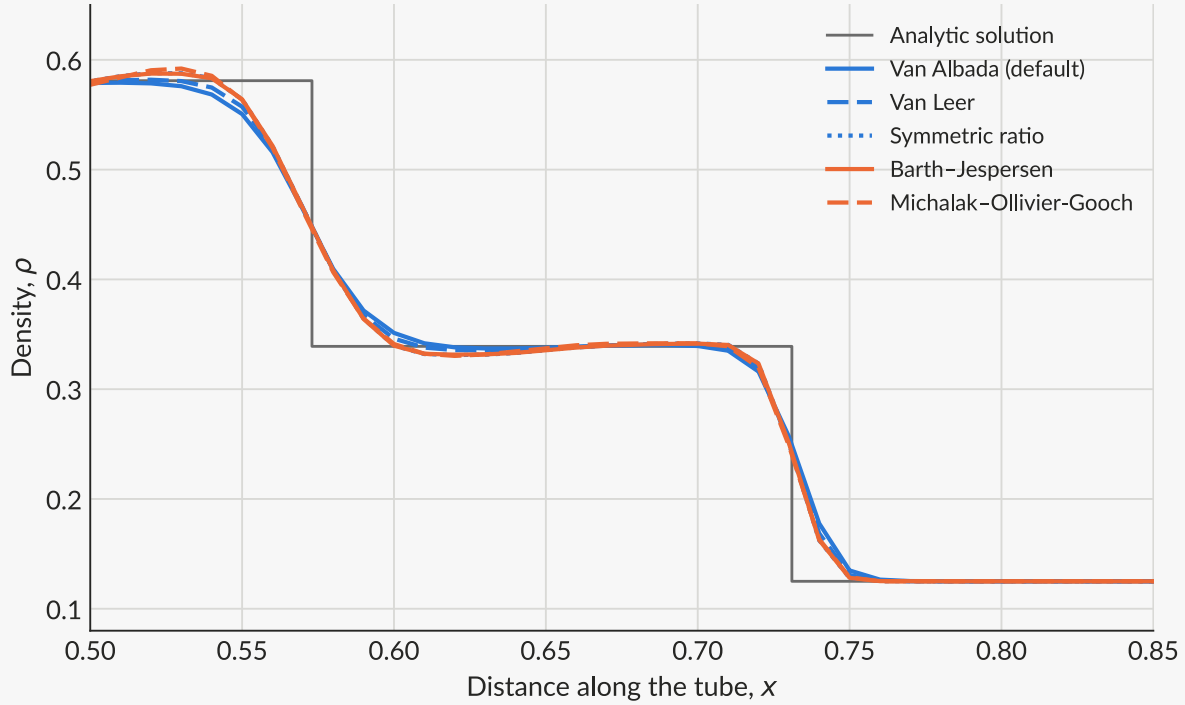
Keep the default, `van albada`, for general work. Where the reconstructed face values must stay within the range of the neighbouring cells, as they must across a strong shock, use `michalak gooch`: it is bound preserving, smooth enough not to stall the residual, and switches itself off in smooth flow so that second-order accuracy survives mesh refinement. Use `none` only for smooth flow with no shocks or strong gradients.

The choice is made with `limiter` (p. 76) in `numerical_scheme`:

```
{ "numerical_scheme": { "order": "second", "limiter": "michalak gooch" } }
```

The limiter acts on the second-order reconstruction only. It has no effect when `order` (p. 75) is `"first"`, nor during the cycles covered by `first_order_cycles` (p. 74); see Choosing a spatial scheme (p. 254). It belongs to the compressible solver; the incompressible solver limits its gradients through `gradient_limiter_type` (p. 21) and `gradient_limiter` (p. 18) in `equations` instead.

Demonstration: what `limiter` does FROM THE SHOCKTUBE CASE



Density across the contact and the shock of a shock tube for five limiters, against the analytic solution

Every run is second-order Roe with implicit dual time stepping on the same mesh; only the limiter changes.

`setting` **`limiter`** `values` **`van albada, van leer, symmetric ratio, barth jespersen, michalak gooch`** `case` **`shocktube`**

One-dimensional limiters

`van albada`, `van leer` and `symmetric ratio` are functions $\psi(a, b)$ of two directional differences: a is the jump across the face and b a backward difference reconstructed from the cell gradient. Each is applied independently to every reconstructed variable and to each side of every face, and ψ scales the whole reconstruction increment on that side.

They are regularised by a threshold

$$\varepsilon = 0.001 \cdot \tfrac{1}{2}(u_L + u_R) + 10^{-8}$$

which is proportional to the magnitude of the variable itself; `symmetric ratio` uses its absolute value. For a variable that changes sign, a velocity component for example, ε becomes small where the variable passes through zero, and the limiter then acts on correspondingly smaller variations.

The symmetric-ratio plateau

Writing $r = b/a$ for the ratio of the two differences, `symmetric ratio` is

$$\psi = \min\left(1, \frac{4\min(a,b)}{a+b}\right)$$

It is zero for $r \le 0$, rises as $4r/(1+r)$, is exactly 1 across the whole plateau $r \in [1/3, 3]$, and then decays as $4/(1+r)$. It satisfies $\psi(1/r) = \psi(r)$. The wide plateau is why it is the least dissipative of the one-dimensional limiters: `van albada` and `van leer` reach $\psi = 1$ only at $r = 1$ exactly.

r	symmetric ratio	van albada	van leer
0.1	0.364	0.198	0.331
1/3	1.000	0.600	0.750
1	1.000	1.000	1.000
3	1.000	0.600	0.750
10	0.364	0.198	0.331

Cell-based limiters

`barth jespersen`, `michalak gooch` and `venkatakrishnan` are multidimensional. They bound the reconstructed face value by the range of values over the cell's whole set of face neighbours, which the one-dimensional limiters do not. They are evaluated per cell, in a pass of their own, and the reconstruction then reads the result.

For cell i, take $u_{\max}$ and $u_{\min}$ over its face neighbours, and write

$$\Delta_f = \nabla u_i \cdot (\mathbf{x}_f - \mathbf{x}_i)$$

for the unlimited extrapolation to face f. Define

$$y_f = \begin{cases} (u_{\max} - u_i)/\Delta_f & \Delta_f > 0 \\ (u_{\min} - u_i)/\Delta_f & \Delta_f < 0 \end{cases}$$

Then $\phi_i = \min_f P(y_f)$, where the shape $P(y)$ distinguishes the three limiters:

limiter	$P(y)$	CONTINUITY
barth jespersen	$\min(1, y)$	C^0: a kink at $y = 1$
michalak gooch	$y - \frac{4}{27}y^3$ for $y < 3/2$, otherwise 1	C^1, and never above $\min(1, y)$
venkatakrishnan	$\min\left(1, (y^2 + 2y)/(y^2 + y + 2)\right)$	C^0: the clip adds a kink at $y = 2$

A kink makes the limiter value flicker between cycles, which can hold the residual at a noise floor. That is the case against barth jespersen, and the reason michalak gooch is preferred.

> **NOTE**
>
> ϕ scales the whole reconstruction increment, the gradient term and the face-jump term together. $\phi = 0$ in a cell therefore gives first order there exactly, and $\phi = 1$ the full $\kappa = 1/3$ reconstruction.

Switching off in smooth flow

michalak gooch and venkatakrishnan blend ϕ towards 1 wherever the flow is smooth. A cell counts as smooth when the range $u_{\max} - u_{\min}$ over its neighbours, normalised by the reference state, is small against the threshold

$$\varepsilon^2 = \left(\frac{Kh}{L_{ref}}\right)^3$$

where K is limiter_coefficient (p. 77), h the cell length scale and L_{ref} the reference_length (p. 40) of the reference condition named by reference in initialisation. K defaults to 5.0. A larger value treats more of the flow as smooth, and so limits less. The cube falls faster than the $O(h^2)$ reconstruction error, so under refinement the limiter switches itself off in smooth regions and formal order is kept. barth jespersen has no such switch and limits wherever the bound binds; the one-dimensional limiters ignore limiter_coefficient.

Freezing the limiter

If a few cells whose limiter switches between iterations hold the residual at a noise floor, freeze the limiter with freeze_limiter_cycle (p. 77). With a cell-based limiter the values are held per cell rather than per face. See Limiter switching (p. 356) for when the freeze applies under each kind of time marching.

Reference entries

- limiter (p. 76), with the formula for each option
- limiter_coefficient (p. 77)
- freeze_limiter_cycle (p. 77)
- order (p. 75), on Choosing a spatial scheme (p. 254)

Choosing a linear solver

Which linear solver settings suit the implicit schemes, RBF mesh motion and the incompressible solver?

Start from the defaults: they are tuned for each equation, and most cases need no change. The library is set by the hardware. On CPUs zCFD uses Petsc, configured from the control dictionary; on NVIDIA GPUs it uses AMGX, configured from json files read at run time. Change a setting only for a measured reason, such as an outer iteration that stalls, and then only for the one equation concerned. The two backends are tuned independently, so a setting that helps one is not evidence for the other.

A linear system is solved in these cases:

WHEN	SYSTEMS SOLVED	DICTIONARY	AMGX FILE
the convergence-control scheme (p. 241) is `"implicit euler"`	mean flow and turbulence	`flow`, `turbulence`	`amgx.json`, `turbamgx.json` or `turbamgx_sa.json`
the scheme is `"lu-sgs"` or `"mf-gmres"`, with `turbulence_solver: "amg"`	turbulence only	`turbulence`	`turbamgx.json` or `turbamgx_sa.json`
RBFs are used for mesh motion	the RBF system	`rbf`	`RBF_amgx.json`
the incompressible solver is selected	every equation	all five, by coupling scheme	one file per equation

The mean-flow solve of `"mf-gmres"` is matrix-free GMRES, configured on the scheme itself (see `mf-gmres` (p. 243)), and takes nothing from these settings. `"lu-sgs"` and `"mf-gmres"` also take the assembled turbulence solve wherever the backend forces it: on a CUDA build running on the host only, and on HIP.

Both libraries offer a wide range of solvers and preconditioners, and the settings can make a large difference to the performance and convergence of the solver. For the settings each backend uses, equation by equation, see Side by side: Petsc and AMGX.

The Petsc settings sit under `linear_solver_options` (p. 11), in `solver_settings`:

```
parameters = {
    "config_version": 2,
    "solver": {
        ...
        "solver_settings": {
            "type": "compressible",
            "linear_solver_options": {
                "flow": {...},
                "turbulence": {...},
                "rbf": {...},
```

```
                "double_precision": False,
            },
        },
        ...
    },
    ...
}
```

Petsc

The options available for the Petsc KSP linear solvers are described in the Petsc KSP manual. zCFD passes them to Petsc from the `linear_solver_options` dictionary.

Each of `flow`, `turbulence` and `rbf` is a plain dictionary of Petsc command-line flags, with both key and value strings. Where a Petsc option takes no value, give an empty string. Most settings in the Petsc documentation can be passed in this way. The compressible defaults are:

```
"linear_solver_options": {
    "flow": {
        "-ksp_min_it": "2",
        "-ksp_type": "fgmres",
        "-ksp_rtol": "1.0e-3",
        "-ksp_converged_reason": "",
    },
    "turbulence": {
        "-ksp_min_it": "2",
        "-ksp_type": "fgmres",
        "-ksp_rtol": "1.0e-5",
        "-ksp_converged_reason": "",
    },
    "rbf": {
        "-ksp_type": "fgmres",
        "-ksp_rtol": "1.0e-5",
    },
    "double_precision": False,
}
```

WARNING

`-ksp_rtol` is relative to the initial guess, not to the right-hand side. zCFD solves every system with `KSPSetInitialGuessNonzero`, keeping the previous outer iteration's solution, so only the first solve of a run starts from zero, with `||r0|| == ||b||`. After that `||r0||` is typically a hundredth of `||b||` or less, and a given `-ksp_rtol` is correspondingly tighter in absolute terms than it looks. Two `-ksp_rtol` values therefore cannot be compared across equations whose initial guesses differ in quality, and a value that looks generous can still exhaust `-ksp_max_it`. `-ksp_monitor_true_residual` prints `||r(i)||/||b||` beside the residual, and shows what a solve is really achieving.

`-ksp_converged_reason` prints one line for each linear solve, giving the number of iterations it took and whether it converged. A solve that reports `DIVERGED_ITS` ran out of iterations rather than diverging. It still returns its best answer, but the tolerance asked for was not met, and a run in which that happens on most cycles is doing work the outer iteration cannot use. `-ksp_monitor` prints a line for each Krylov

iteration. It is useful when investigating a solve, but is not a default, because on a long run it dominates the log; add it to any of these dictionaries when it is needed.

One options namespace per solver

Petsc has a single options database for the whole process. zCFD builds a separate solver for each system it solves, and gives each its own Petsc options prefix, taken from the name of the dictionary it comes from:

DICTIONARY	PREFIX
`flow` (p. 14)	`-flow_`
`turbulence` (p. 14)	`-turb_` (and `-energy_` for the incompressible energy equation)
`momentum` (p. 15)	`-momentum_`
`pressure_correction` (p. 15)	`-pcorr_`
`rbf` (p. 14)	`-rbf_`

So `"-ksp_type": "bcgs"` in the `momentum` dictionary reaches Petsc as `-momentum_ksp_type bcgs`, and applies to that solver alone. The keys in a control file are written without the prefix, and zCFD adds it. The prefix matters only for options given from outside the control file, through the `PETSC_OPTIONS` environment variable for instance, where the prefix must be written out to reach a single solver.

The Hypre library of algebraic multigrid methods is available through Petsc PCHYPRE, and its settings are given in the same dictionaries. Hypre's BoomerAMG package, as a preconditioner, has performed well with the incompressible solver:

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

Further Hypre options are described in the Petsc PCHYPRE documentation.

Incompressible solver

With `{"solver_settings": {"type": "incompressible"}}`, `linear_solver_options` gains two further keys, `momentum` and `pressure_correction`, and takes its own defaults, modelled on OpenFOAM's `simpleFoam`.

Why they look the way they do

Under the SIMPLE scheme every equation is solved inside an outer iteration, so its linear solve need not be accurate. It needs to be cheap, because the outer loop revisits it a few thousand times. That is what `simpleFoam` encodes: across the OpenFOAM tutorials every equation is solved to `relTol 0.1`, one order of magnitude, and only the kind of solver changes with the equation. The motorBike tutorial is the exception, tightening `p` to `0.01`; the zCFD pressure-correction default follows that tighter figure, and every other equation the 0.1. zCFD follows the same mapping:

fvSolution	OPENFOAM SETTING	ZCFD EQUIVALENT
`p`	GAMG + GaussSeidel, `relTol 0.1`	`pressure_correction`: CG + Hypre BoomerAMG
`U`	PBiCGStab + DILU, `relTol 0.1`	`momentum`: BiCGStab + block-Jacobi/ILU
`k`, `omega`, `epsilon`	smoothSolver, `relTol 0.1`	`turbulence`: BiCGStab + block-Jacobi/ILU

The pressure correction is the one equation that needs coarse levels. It is elliptic, and a purely local preconditioner such as ILU cannot carry information across the mesh however strong it is made, so it exhausts its iteration limit without converging. Algebraic multigrid is the cure, and is what OpenFOAM uses for `p`.

The coupled scheme is the exception. There the block-4 velocity-pressure solve is the step itself rather than an inner iteration, so `flow` keeps a tight tolerance. `simpleFoam` has no counterpart to it.

The defaults:

```
"linear_solver_options": {
    # Coupled scheme only: this solve is the step, so it is solved tightly.
    "flow": {
        "-ksp_min_it": "2",
        "-ksp_type": "fgmres",
        "-ksp_rtol": "1.0e-7",
        "-ksp_atol": "1.0e-16",
        "-ksp_max_it": "50",
        "-pc_type": "bjacobi",
        "-sub_pc_type": "ilu",
        "-ksp_converged_reason": "",
    },
    # simpleFoam's k/omega/epsilon/nuTilda entry. Also used for the energy equation.
    "turbulence": {
        "-ksp_min_it": "1",
        "-ksp_type": "bcgs",
        "-ksp_rtol": "0.1",
        "-ksp_atol": "1.0e-16",
        "-ksp_max_it": "50",
        "-pc_type": "bjacobi",
        "-sub_pc_type": "ilu",
        "-ksp_converged_reason": "",
    },
    # SIMPLE only: simpleFoam's U entry.
    "momentum": {
```

```
        "-ksp_min_it": "1",
        "-ksp_type": "bcgs",
        "-ksp_rtol": "0.1",
        "-ksp_atol": "1.0e-12",
        "-ksp_max_it": "50",
        "-pc_type": "bjacobi",
        "-sub_pc_type": "ilu",
        "-ksp_converged_reason": "",
    },
    # SIMPLE only: simpleFoam's p entry.
    "pressure_correction": {
        "-ksp_min_it": "1",
        "-ksp_type": "cg",
        "-ksp_rtol": "0.01",
        "-ksp_atol": "1.0e-12",
        "-ksp_max_it": "100",
        "-pc_type": "hypre",
        "-pc_hypre_type": "boomeramg",
        "-zcfd_constant_nullspace": "1",
        "-zcfd_pc_reuse_cycles": "20",
        "-ksp_converged_reason": "",
    },
    "rbf": {
        "-ksp_type": "fgmres",
        "-ksp_rtol": "1.0e-5",
    },
    "double_precision": True,
}
```

`-zcfd_constant_nullspace` is not a Petsc option: zCFD consumes it and never passes it on. It attaches a constant null space to the pressure-correction operator, which keeps the solve well posed when the domain is enclosed and the pressure has no datum. zCFD detaches it again if the boundary conditions include a fixed-pressure outlet, since that outlet then sets the datum.

`-zcfd_pc_reuse_cycles` is not a Petsc option either; zCFD consumes it too. It sets how often the preconditioner is rebuilt. The matrix is reassembled at every outer iteration, so by default Petsc rebuilds the multigrid hierarchy before every solve, and that is the dominant cost. Profiled with `-log_view` on a 52k-cell plate SIMPLE case, `PCSetUp` took 5.01 s of the 8.50 s spent in `KSPSolve`, 59% of all linear-solver time, for a matrix whose sparsity never changes. Building it once and freezing it is no answer either: on the plate case that takes the pressure solve from 1.26 iterations per cycle over the first fifty cycles to 8.3 over the last fifty, as the start-up transient moves the operator away from it. Rebuilding every N solves cuts the setup cost by a factor of N, while keeping the preconditioner at most N cycles stale; it is the trade OpenFOAM's GAMG makes with `cacheAgglomeration`. Set it to `1` to rebuild for every solve. Whatever the value, it cannot change the answer: the Krylov solver applies the true operator, so a stale preconditioner costs iterations and nothing else.

`pressure_correction` (p. 15) and `momentum` (p. 15) apply only when the incompressible solver uses the SIMPLE coupling scheme, `{"convergence_control": {"scheme": {"name": "simple"}}}`. The coupled scheme builds neither solver, and setting either key under it is rejected.

AMGX

The AMGX settings for the mean-flow, turbulence and RBF linear systems are read from json files, not from the control dictionary: `ZCFD_HOME/amgx.json`, `ZCFD_HOME/turbamgx.json` and `ZCFD_HOME/RBF_amgx.json`. The files are in AMGX's own configuration format, and are unrelated to the `config_version` key of the zCFD control file; the `"config_version": 2` below is the version of AMGX's file format, not zCFD's. The mean-flow settings are:

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

These settings perform well for most cases. If the linear system is hard to converge, reducing `"tolerance"` may help. Otherwise, changing the aggregation selector from `"selector": "SIZE_8"` to `"SIZE_4"` or `"SIZE_2"` builds smaller aggregates, and so more coarse levels, improving convergence at the cost of memory. Increasing `"gmres_n_restart"` may also improve convergence, again at the cost of memory.

AMGX has too many options to describe here, but the AMGX example configurations cover the different solvers.

Incompressible AMGX configurations

The incompressible solver reads its own set of files, one for each equation, so that each system has a configuration suited to it. Each file is looked for first in the working directory and then in `ZCFD_HOME`, so a copy placed beside the control file overrides it for that case.

FILE	SYSTEM
`incomp_amgx.json`	Coupled scheme: the block-4 velocity-pressure system.
`incomp_simple_mom_amgx.json`	SIMPLE: the block-3 momentum predictor. FGMRES + block-Jacobi, `alt_rel_tolerance` 0.1.
`incomp_pcorr_amgx.json`	SIMPLE: the block-1 pressure-correction Poisson. Aggregation AMG, as for `simpleFoam`'s `p`.
`incomp_turb_amgx.json`	Turbulence and energy transport.

`"structure_reuse_levels": -1` in `incomp_amgx.json` and `incomp_pcorr_amgx.json` keeps the multigrid hierarchy from cycle to cycle. The matrix coefficients change at every outer iteration but the sparsity pattern never does, so re-aggregating every cycle would be wasted work. Leave `"coarse_solver": "NOSOLVER"` in `incomp_pcorr_amgx.json` as it is: with no fixed-pressure outlet the

pressure-correction operator is singular, and a direct coarse solve meets a zero pivot on it. That is also why OpenFOAM leaves `directSolveCoarsest` off.

> **NOTE**
>
> The Petsc settings above do not transfer wholesale to AMGX, and were measured separately. Two changes that are clearly right on the Petsc side proved wrong here, in a GPU comparison on the lid-driven cavity and the plate RANS case:
>
> - Moving the momentum predictor from FGMRES to BiCGStab, `simpleFoam`'s `PBiCGStab`, cost 68% more iterations, from 6564 to 10995, and was no better with `MULTICOLOR_DILU` in place of block-Jacobi. OpenFOAM pairs `PBiCGStab` with `DILU`, a much stronger preconditioner than anything cheap available here, and without it BiCGStab has no strong preconditioner to support it. FGMRES is kept.
> - Loosening the turbulence `alt_rel_tolerance` from `1.0e-6` to OpenFOAM's `relTol 0.1` moved `wall_Fx` on the plate case by 53%. The equivalent change is safe on Petsc only because the turbulence solve there already ended on `-ksp_atol`, so the relative tolerance never bounded it; `alt_rel_tolerance` has no such floor, and the loosening is real. It is kept at `1.0e-6`.
>
> The AMGX files therefore diverge from the Petsc options by design: the mapping under Why they look the way they do describes the Petsc path.

On NVIDIA GPUs, with AMGX, the one linear-solver setting that still comes from the zCFD control dictionary is `double_precision` (p. 14), on `linear_solver_options`:

```
"linear_solver_options": {"double_precision": False}
```

`double_precision` sets whether the mean-flow linear system is solved in single or double precision. Single precision reduces memory use and speeds up the solver; more demanding cases may need double precision.

Side by side: Petsc and AMGX

The same equation is solved by a different library on CPU and on GPU, and the two are configured in different places: Petsc through the control dictionary, AMGX through its json files. This section sets them side by side, equation by equation, so that the settings a case is running can be read in one place.

The two backends are tuned independently, and are not expected to match. Where they differ, the difference is usually deliberate and measured; the note under Incompressible AMGX configurations gives the two cases where a Petsc setting was tried on AMGX and rejected.

How the settings correspond

CONCEPT	PETSC	AMGX
Krylov method	`-ksp_type`	`solver`
Preconditioner	`-pc_type`, `-sub_pc_type`	`preconditioner`, and its `smoother` when that is AMG
Relative tolerance	`-ksp_rtol`	`alt_rel_tolerance`, or `tolerance` under `RELATIVE_INI_CORE`
Absolute tolerance	`-ksp_atol`	`tolerance`, under `COMBINED_REL_INI_ABS` only
Iteration limit	`-ksp_max_it`	`max_iters`

CONCEPT	PETSC	AMGX
Minimum iterations	`-ksp_min_it`	no equivalent
GMRES restart	`-ksp_gmres_restart` (unset, so 30)	`gmres_n_restart`
Setup amortisation	`-zcfd_pc_reuse_cycles`	`structure_reuse_levels` (see below)

The tolerance rows depend on the `convergence` mode, which is why the same `tolerance` key means different things in different files:

- `COMBINED_REL_INI_ABS`, used by the incompressible files, stops when `|r| < tolerance` or `|r|/|r0| <= alt_rel_tolerance`. `tolerance` is then the absolute floor and `alt_rel_tolerance` the relative one: the two Petsc tolerances, in the opposite order to the one the names suggest.
- `RELATIVE_INI_CORE`, used by the compressible files, stops when `|r|/|r0| <= tolerance`. Here `tolerance` is the relative tolerance, and there is no absolute floor at all.

> **WARNING**
>
> `structure_reuse_levels` and `-zcfd_pc_reuse_cycles` do similar jobs but are not equivalent. `structure_reuse_levels` keeps the multigrid structure, the aggregation and colouring, and still rebuilds the coarse operators at every setup. `-zcfd_pc_reuse_cycles` skips the preconditioner setup outright for all but every Nth solve. AMGX has no equivalent of the second, and Petsc none of the first.

Incompressible, SIMPLE scheme

Momentum predictor — `momentum` / prefix `momentum_` / `incomp_simple_mom_amgx.json`:

SETTING	PETSC	AMGX
Solver	`bcgs`	`FGMRES`, restart 30
Preconditioner	`bjacobi` + `ilu`	`BLOCK_JACOBI`
Relative tolerance	`0.1`	`0.1`
Absolute tolerance	`1.0e-12`	`1e-10`
Iteration limit	`50`	`50`

Pressure correction — `pressure_correction` / prefix `pcorr_` / `incomp_pcorr_amgx.json`:

SETTING	PETSC	AMGX
Solver	`cg`	`FGMRES`, restart 32
Preconditioner	`hypre` / BoomerAMG	aggregation AMG, `SIZE_8`, V-cycle, `BLOCK_JACOBI` smoother
Sweeps	BoomerAMG defaults	0 pre, 2 post, `coarse_solver` `NOSOLVER`
Relative tolerance	`0.01`	`0.01`
Absolute tolerance	`1.0e-12`	`1e-10`

SETTING	PETSC	AMGX
Iteration limit	`100`	`50`
Setup amortisation	rebuild every 20 solves	`structure_reuse_levels -1`

This is the closest-matched pair: multigrid at `relTol 0.01` on both sides.

Turbulence and energy — `turbulence` / prefixes `turb_` and `energy_` / `incomp_turb_amgx.json`:

SETTING	PETSC	AMGX
Solver	`bcgs`	`FGMRES`, restart 32
Preconditioner	`bjacobi` + `ilu`	aggregation AMG, `BLOCK_JACOBI` smoother, 0 pre / 3 post
Relative tolerance	`0.1`	`1.0e-6`
Absolute tolerance	`1.0e-16`	`1e-10`
Iteration limit	`50`	`100`

The relative tolerances differ by five orders of magnitude, by design; see the note under Incompressible AMGX configurations. The energy equation shares the `turbulence` dictionary and the same json file on both backends; only its Petsc options prefix is its own.

Incompressible, coupled scheme

Block-4 velocity-pressure — `flow` / prefix `flow_` / `incomp_amgx.json`. Turbulence and energy are as above; `momentum` and `pressure_correction` are not built at all under this scheme.

SETTING	PETSC	AMGX
Solver	`fgmres`	`FGMRES`, restart 100
Preconditioner	`bjacobi` + `ilu`	aggregation AMG, `MULTICOLOR_GS`, 1 pre / 3 post
Relative tolerance	`1.0e-7`	`1e-4`
Absolute tolerance	`1.0e-16`	`1e-10`
Iteration limit	`50`	`30`
Setup amortisation	rebuild every 20 solves	`structure_reuse_levels -1`

WARNING

On a large parallel case the coupled flow solve does not reach `-ksp_rtol` at any iteration limit, and `-ksp_max_it` is what ends it. The 7.46M-cell Windsor body on 16 ranks reports `DIVERGED_ITS` on every cycle at a cap of 20, 50 and 400 alike: block-Jacobi/ILU stalls on a block-4 saddle-point system of that size, its convergence factor per iteration worsening from 0.82 over the first twenty iterations to 0.91 averaged over fifty.

That is no reason to lower the cap. The outer iteration needs a better linear solve, not a converged one, and the deeper solve pays for itself: at 500 cycles it takes `rhoV[0]` from `3.78e-06` to `1.11e-06` for 1.73 times the cost per cycle, so at equal wall time it is still about twice as converged. Read `DIVERGED_ITS` on this solver as a statement

about the reach of the preconditioner, not as a failure to act on. A stronger preconditioner for the coupled system, not a larger cap, is what would remove it.

NOTE

The two relative tolerances in this table are not comparable, and the Petsc one is not the tighter of the pair, despite the smaller number. Petsc applies `-ksp_rtol` to the residual of the initial guess, and this solver keeps the previous outer iteration's solution as that guess. From the second solve onwards `||r0||` is therefore a hundredth of `||b||` or less, and `1.0e-7` relative to it is nearer `1.0e-9` relative to `||b||`. AMGX's `alt_rel_tolerance` is likewise a reduction from its own starting residual, but its aggregation multigrid reaches it in about eight iterations, where block-Jacobi/ILU stalls on this saddle-point system.

Measured on the lid-driven cavity at 300 cycles, comparing the outer residual each setting delivers rather than the number in the file, AMGX at `alt_rel_tolerance 1e-4` reaches `rhoV[0] 1.9e-09`, where Petsc at `-ksp_rtol 1.0e-7` reaches `3.4e-08`. Matching the numbers would make matters worse: Petsc at `1.0e-4` stalls the outer iteration at `1.2e-05`, and never reaches `1e-06`. The two backends agree in the limit: solved almost exactly, with `-ksp_rtol 1.0e-12` and 200 iterations, Petsc reaches `3.2e-13`.

Compressible

Mean flow: `flow` / prefix `flow_` / `amgx.json`. Turbulence: `turbulence` / prefix `turb_` / `turbamgx.json`, or `turbamgx_sa.json` when the turbulence model is `sa-neg`.

SETTING	PETSC	AMGX
Solver (both systems)	`fgmres`	`FGMRES`, restart 5
Preconditioner	not set, so Petsc's own default	aggregation AMG, `MULTICOLOR_GS` (`BLOCK_JACOBI` for `sa-neg`)
Relative tolerance, mean flow	`1.0e-3`	`1e-3`
Relative tolerance, turbulence	`1.0e-5`	`1e-5`
Absolute tolerance	not set, so `1e-50`	none, under `RELATIVE_INI_CORE`
Iteration limit	not set, so `10000`	`10`

The tolerances match exactly here. The iteration limits do not: a compressible Petsc solve is bounded only by its tolerance, whereas the AMGX one stops after ten iterations, converged or not.

RBF mesh motion

`rbf` / prefix `rbf_` / `RBF_amgx.json`. This is the least closely matched pair in zCFD: the two backends use a different Krylov method, a different cycle and different kinds of tolerance.

SETTING	PETSC	AMGX
Solver	`fgmres`	`PCG`
Preconditioner	not set	AMG, `D2` interpolation, W-cycle, `BLOCK_JACOBI` smoother
Tolerance	`1.0e-5` relative	`1e-12` absolute
Iteration limit	not set, so `10000`	`100`

Summary

EQUATION	RELATIVE TOLERANCE	KRYLOV METHOD	PRECONDITIONER CLASS
Incompressible pressure correction	same, `0.01`	differs	same, multigrid
Incompressible momentum	same, `0.1`	differs	same, block-Jacobi
Incompressible turbulence, energy	differs	differs	differs
Incompressible coupled flow	differs (not comparable — see above)	same	differs
Compressible mean flow	same, `1.0e-3`	same	differs
Compressible turbulence	same, `1.0e-5`	same	differs
RBF	differs	differs	differs

Reading the settings a case actually used

On the Petsc path the solver logs every option it sets, under the prefix it sets it under, so the log is the authoritative record of what a run used, including anything overridden from the control dictionary. On the AMGX path, set `"print_solve_stats": 1` in the relevant json file to have each solve report its iteration count and residuals.

Reference entries

- `linear_solver_options` (p. 11): `flow` (p. 14), `turbulence` (p. 14), `momentum` (p. 15), `pressure_correction` (p. 15), `rbf` (p. 14) and `double_precision` (p. 14)
- `scheme` (p. 56) and `turbulence_solver` (p. 64)

Choosing a fluid-zone model

Which fluid zone represents the device, obstruction or motion in the flow?

A fluid zone is a region of cells in which the solver applies a source term or a reference frame. Each zone is a named block in the model's `fluid_zones`. Its `type` sets what is applied. The cells are selected by `zones`, a list of the mesh's cell zones, or by `definition`, a closed surface.

Choose the zone type by what the flow must feel. A rotor, propeller or turbine whose blades the mesh does not resolve is an actuator disc. A distributed resistance, such as a radiator, a fence or a packed bed, is a porous zone, and a tree canopy over terrain is a canopy zone. Any other body force is a momentum source, written as a Python function. A part of a still mesh that turns, such as a wheel or an impeller, is a `rotating` zone: its cells are solved in a rotating frame. A frame over the whole mesh, a translating frame, and a mesh that moves are not fluid zones; they are the `motion` in the model's `transforms`, described in Placing and moving a mesh (p. 365).

`type`	REPRESENTS	SELECTS ITS CELLS BY
`rotating` (p. 147)	a rotating frame over part of a still mesh	`cell zones`
`disc` (p. 149)	an actuator disc: a rotor, propeller or turbine	its geometry and `definition`
`porous` (p. 159)	a porous medium, as viscous and inertial resistance	`cell zones`, `definition`, or the whole mesh
`canopy` (p. 161)	a tree canopy in a terrain wind model	`cell zones`, `definition`, `field`, or the whole mesh
`momentum` (p. 144)	an arbitrary body force, from a Python function	`cell zones`, `definition`, or the whole mesh
`rbf transform` (p. 166), `idw transform` (p. 164)	a surface deformation carried into the volume mesh	`zones`, the moving boundary faces
`abl` (p. 163)	validated, but has no effect on the solution	the whole mesh

Fluid zones are a dictionary of named blocks, keyed in the same way as boundary conditions (p. 285):

```
{
    'model': {
        '<mesh_name>': {
            'mesh': '<mesh_name>.h5',
            'fluid_zones': {
                'FZ_1': {'type': '...', ...},
                'FZ_2': {'type': '...', ...},
            },
            'boundary_conditions': {...},
```

```
        }
    }
}
```

Each block's `type` (p. 143) selects the kind of zone, and determines which other keys are valid and required in that block. An unrecognised key anywhere in a fluid zone is rejected at validation, not silently ignored.

Selecting cells

A fluid zone that acts on a volume selects its cells in one of three ways:

- `cell zones`, a list of the mesh's cell zone numbers. Every cell labelled with one of those numbers belongs to the fluid zone. This suits a region that was meshed as a body of its own, such as a wheel, a fan or a radiator core.
- `definition`, the path to a closed-surface VTP file. The cells the surface encloses belong to the fluid zone. This suits a region the mesh does not mark, or one drawn after the mesh was made.
- neither, in which case the fluid zone covers the whole mesh.

A fluid zone takes at most one of these. Which it may take depends on its type:

type	cell zones	definition	NEITHER
`rotating`	required	not accepted	not accepted
`momentum`, `porous`	accepted	accepted	the whole mesh
`canopy`	accepted	accepted	the whole mesh
`disc`	not accepted	required	not accepted
`abl`	not accepted	not accepted	always the whole mesh

A `canopy` zone may instead give `field`, a canopy height map, which selects the cells beneath the canopy height; it takes only one of `cell zones`, `definition` and `field`. A `disc` zone places its forces with its `geometry` and selects the cells that receive them with its `definition`.

```
"fluid zones": {
    "fan": {"type": "momentum", "cell zones": [4], "func": fan_force},
    "screen": {"type": "porous", "definition": "screen.vtp", "alpha": 1.0e-7},
    "drive": {"type": "momentum", "func": pressure_gradient},
},
```

Cell zone numbers belong to the cells of the mesh, and are separate from the boundary zone numbers in a boundary condition's `zones`. The two may share a number without any connection between them. See Finding cell zone numbers for how to read them from a mesh.

The `zones` of an `rbf transform` or `idw transform` zone are boundary zones: the moving surfaces whose deformation is carried into the volume mesh, numbered as for boundary conditions (p. 286).

Fluid zones in the incompressible solver

All the source-term zone types, `momentum`, `porous`, `canopy`, `disc` and `abl`, are available to the incompressible solver, `{"solver_settings": {"type": "incompressible"}}`, as well as to the compressible one. Three points differ.

The incompressible momentum equation is solved per unit mass, with the density divided out, so a `momentum` zone's `func` must return a force per unit mass rather than per unit volume. The porous, canopy and actuator disc models handle this internally, so their inputs (`alpha`, `C2`, `cd`, and the disc's geometry and model dictionaries) are the same for both solvers.

A strong `porous` or `canopy` zone under the segregated SIMPLE scheme, `{"convergence_control": {"scheme": {"name": "simple"}}}`, needs the scheme's `pressure_regularisation` set. Their drag adds to the momentum diagonal, so in a heavily blocked cell every pressure-correction face coefficient collapses towards zero, and that cell's row of the p′ Laplacian becomes nearly empty. The conjugate-gradient solver then reports an indefinite matrix, and the run diverges. A value around `1e-2` restores stability, though a near-solid block still converges more slowly and less far than under the coupled scheme; for strong drag, prefer the `'coupled'` scheme. Weak drag, such as a typical canopy, converges under either scheme with the default of `0`, and the two agree to discretisation level. `momentum`, `disc` and `abl` zones are explicit body forces that never touch the momentum diagonal, and are unaffected either way.

The `canopy` turbulence source is defined only for the `sst` model. Under any other turbulence model the canopy applies its momentum sink alone, and the solver logs a warning.

Actuator discs

A wind turbine or a propeller affects the flow, but it is often impractical to resolve the flow down to the scale of its blades. An actuator disc represents the device instead, as a set of source terms superimposed on the flow.

Turbines and propellers behave almost identically in the actuator disc model; the differences lie only in the coordinate systems and sign conventions usually used to define them. In zCFD an actuator disc is a fluid zone (p. 149) of type `'disc'`.

The `type` (p. 156) key of the disc's `model` block selects a `'simple'` disc, with fixed or curve-driven thrust and power coefficients, or a `'bet'` disc, based on blade-element theory.

The `normal` (p. 153) sets the direction of the disc's force. In zCFD the normal points in the direction of the force on the fluid: for a propeller it points with the flow, and for a turbine it points towards the incoming flow.

A disc zone is built from four dictionaries and a few keys of its own:

KEY	HOLDS
`controller` (p. 149)	how the disc's forces are controlled: `fixed`, with `omega` and `pitch`, or `tsr curve`, with a tip speed ratio curve
`geometry` (p. 149)	`centre`, `normal`, `up`, `inner radius` and `outer radius`

KEY	HOLDS
`discretisation` (p. 149)	the representation of the disc, such as `{"type": "disc", "number of elements": 48}`
`model` (p. 149)	the force model: `simple`, averaged over the whole disc, or `bet`, by blade elements
`name` (p. 149)	the name of the disc
`definition` (p. 150)	a closed-surface VTP file selecting the disc's cells
`reference_plane` (p. 149) and `reference_point` (p. 149)	for a `simple` model, where the reference velocity is taken: on a reference plane when `reference_plane` is true, otherwise at `reference_point`, or from the solver's reference condition if neither is set
`update_frequency` (p. 149)	how often, in cycles, the disc's source is re-sampled
`rotation_direction` (p. 149)	the direction of rotation, such as `clockwise`

A simple propeller with a fixed controller:

```
{
  "fluid_zones": {
    "FZ_1": {
      "type": "disc",
      "name": "R01",
      "definition": "propellor_vtp/R01.vtp",
      "reference plane": true,
      "update frequency": 10,
      "rotation direction": "clockwise",
      "controller": { "type": "fixed", "omega": 314.16, "pitch": 0.0 },
      "geometry": {
        "centre": [0.0, 0.0, 0.0],
        "normal": [0.0, 0.0, -1.0],
        "up": [1.0, 0.0, 0.0],
        "inner radius": 0.039,
        "outer radius": 0.175
      },
      "discretisation": { "type": "disc", "number of elements": 24 },
      "model": {
        "type": "simple",
        "kind": "propellor",
        "power model": "fixed",
        "thrust coefficient": 0.06,
        "power coefficient": 0.048
      }
    }
  }
}
```

A `bet` model in place of the simple one describes the blades: their number, chord and twist along the radius, as fractions of it, and the lift and drag of each aerofoil against angle of attack:

```
{
  "model": {
    "type": "bet",
    "kind": "propellor",
    "number of blades": 2,
    "number of sections": 24,
    "blade chord": [
      [0.2, 0.03],
      [1.0, 0.015]
    ],
    "blade twist": [
      [0.2, 40.0],
      [1.0, 12.0]
    ],
    "aerofoil positions": [
      [0.0, "naca4412"],
      [1.0, "naca4412"]
    ],
    "aerofoils": {
      "naca4412": {
        "cl": [
          [-10.0, -0.6],
          [0.0, 0.4],
          [10.0, 1.4]
        ],
        "cd": [
          [-10.0, 0.02],
          [0.0, 0.01],
          [10.0, 0.02]
        ]
      }
    },
    "tip loss correction": "rstar",
    "tip loss correction radius": 0.8
  }
}
```

NOTE

Keys may be written with spaces, `'inner radius'` and `'tip speed ratio'`, or with underscores, `inner_radius` and `tip_speed_ratio`. Both forms are normalised to the same field before validation.

Rotating zones

A `rotating` zone solves the flow in part of a mesh in a frame that turns about an axis, while the mesh itself stays still. It is the multiple reference frame (MRF) treatment: the cells of the zone carry the Coriolis and centrifugal sources of the rotating frame, and the cells outside it are solved in the stationary frame. It suits a body that turns steadily inside a stationary one, such as a road wheel under a car, a fan in a duct or the impeller of a compressor, where a steady run is to give the time-averaged flow.

KEY	VALUE	MEANING
`cell zones` (p. 148)	a list of cell zone numbers	the cells solved in the rotating frame
`axis` (p. 147)	a vector	the axis of rotation, in placed coordinates
`origin` (p. 147)	a point	a point on the axis, in placed coordinates
`omega` (p. 147)	rad/s	the rate of rotation, positive right-handed about `axis`

A rotating zone is always a frame on a still mesh. It takes no `definition`, no `moving mesh` and no ramp, and it runs at its full rate from the first cycle. A cell zone may belong to only one rotating zone, but a model may carry several rotating zones, each with its own axis and rate.

A rotating zone or a `motion`

Both a rotating zone and the `motion` in the model's `transforms` can solve flow in a rotating frame. They differ in how much of the mesh the frame covers, and in whether the mesh may move:

THE MESH	WRITE
a still mesh, part of which turns	a `rotating` fluid zone on the cell zones that turn
a still mesh that turns as a whole, such as a rotor in its own mesh	`transforms > motion` with `"moving mesh": False`
a still mesh in a translating frame	`transforms > motion` of type `translating`, with `"moving mesh": False`
a mesh that moves through the run	`transforms > motion` with `"moving mesh": True`, and an overset or sliding interface

A model takes one source of frame: a model with a rotating zone carries no `motion`, and a model with a `motion` carries no rotating zone. A mesh with a sliding or overset interface turns as a whole, so it takes a `motion` and no rotating zone; see Placing and moving a mesh (p. 365). A rotating zone avoids that interface: the turning and stationary cells stay in one mesh, joined by ordinary interior faces.

Preparing the mesh

The region that turns must be meshed as one or more cell zones of its own, enclosing the body that turns and nothing that does not. The surface between those cells and the stationary ones carries the change of frame, so it should:

- be a surface of revolution about the axis, such as a cylinder or a disc, so that the flow crossing it is close to uniform around the axis;
- lie clear of the blades or spokes, with a few cells between the body and the surface;
- be made of faces shared by the cells either side, with no boundary zone or interface on it.

A rotating cell zone that cuts through a stationary wall, or a surface that wraps closely around blades, places the change of frame where the flow varies most around the axis, and the solution then depends on where the surface was drawn.

A wall bounding the cells of a rotating zone turns with the frame, so the blades and hub of a body inside the zone need no `velocity` of their own. A wall `velocity` takes precedence over the frame: a stationary wall inside the zone, such as a shroud, carries `"velocity": {"type": "linear", "vector": [0.0, 0.0, 0.0]}`, and a surface that turns but lies outside the zone, such as the tread of a tyre, carries a rotating wall `velocity` (p. 236) with the same axis, origin and rate as the zone.

Finding cell zone numbers

A zCFD mesh labels each cell with a cell zone number, held in its `cellZone` dataset. The Fluent and Fluent CFF converters keep the cell zone numbers of the Fluent file, and the zone helper they write beside the mesh maps each Fluent cell zone name to its number in `fluid_zone`; see Zone numbers (p. 208):

```
from wheel_zone import fluid_zone

parameters["model"]["wheel"]["fluid zones"] = {
    "wheel": {"type": "rotating", "cell zones": [fluid_zone["wheel"]], ...},
}
```

The helper may not list every cell zone in the mesh. For any mesh, the numbers can be read from a solution: add `cellzone` to the volume output variables and run a few cycles:

```
"output settings": {"solution": {"volume variables": ["cellzone", "V", "p"]}},
```

Open the volume output in ParaView and colour it by `cellzone`. A clip or slice through the turning body shows the number of the region around it; a Threshold filter on `cellzone` isolates one region, which confirms that it encloses the turning body and no more.

Example: a road wheel

A wheel turning on a road, with the car's speed as the oncoming flow and the floor moving with it. The wheel's radius is 0.2635 m, so at 160 km/h it turns at 168.6 rad/s. The cells around the wheel are cell zone 17, and the tyre's tread and sidewall are boundary zones 14 and 20:

```
speed = 160.0 / 3.6
wheel_radius = 0.52709 / 2.0
rot_rate = speed / wheel_radius
wheel_axis = [0.0, -0.9988, 0.0498]
wheel_origin = [0.0, -0.6593, 0.2533]

parameters["model"]["wheel"] = {
    "mesh": "wheel.h5",
    "fluid zones": {
        "wheel": {
            "type": "rotating",
            "cell zones": [17],
            "axis": wheel_axis,
            "origin": wheel_origin,
            "omega": rot_rate,
        },
```

```
    },
    "boundary conditions": {
        "floor": {
            "zones": [8],
            "type": "wall",
            "kind": "wall function",
            "velocity": {"type": "linear", "vector": [speed, 0.0, 0.0]},
        },
        "tyre": {
            "zones": [14, 20],
            "type": "wall",
            "kind": "wall function",
            "velocity": {
                "type": "rotating",
                "axis": wheel_axis,
                "origin": wheel_origin,
                "omega": rot_rate,
            },
        },
        "hub": {"zones": [10, 11, 12, 13, 15, 19], "type": "wall", "kind": "wall function"},
        "symmetry": {"zones": [9], "type": "symmetry"},
        "farfield": {"zones": [4, 5, 6, 7], "type": "farfield", "condition": "IC_1", "kind":
        ↪  "riemann"},
    },
}
```

The axis is tilted slightly from the y axis, to follow the camber of the wheel, and the origin lies on the axle. The frame carries the flow through the spokes and the rim; the tyre's walls carry the same rotation, and the floor moves at the oncoming speed, so the contact patch is at rest relative to the road.

Example: a turbocharger compressor

A turbocharger compressor wheel turning at 5,000 rad/s about an axis along $-x$, in still air. The Fluent mesh has two cell zones: cell zone 1, named `fluid`, is the passage around the wheel, and cell zone 0 is the stationary inlet and housing around it. The frame covers cell zone 1, and the inlet and outlet hold `IC_2`, a reference condition at rest:

```
import turbo_zone as tz

impeller_axis = [-1.0, 0.0, 0.0]
impeller_origin = [0.0, 0.005967, 0.37498]
impeller_rate = 5000.0

parameters["model"]["turbo"] = {
    "mesh": "turbo.h5",
    "fluid zones": {
        "impeller": {
            "type": "rotating",
            "cell zones": [tz.fluid_zone["fluid"]],
            "axis": impeller_axis,
            "origin": impeller_origin,
```

```
            "omega": impeller_rate,
        },
    },
    "boundary conditions": {
        "wheel": {
            "zones": tz.to_id(["comp", "comp-fine", "comp-fe", "comp-extra-fine"]),
            "type": "wall",
            "kind": "wall function",
            "velocity": {
                "type": "rotating",
                "axis": impeller_axis,
                "origin": impeller_origin,
                "omega": impeller_rate,
            },
        },
        "housing": {
            "zones": tz.to_id(["body", "body-fine"]),
            "type": "wall",
            "kind": "wall function",
            "velocity": {"type": "linear", "vector": [0.0, 0.0, 0.0]},
        },
        "ports": {
            "zones": tz.to_id(["vel-inlet", "pressure-outlet"]),
            "type": "farfield",
            "condition": "IC_2",
            "kind": "pressure",
        },
    },
}
```

The boundary between the cell zones does not follow the walls exactly. Part of the wheel's surface lies against stationary cells, and part of the housing lies against rotating ones, so each wall states its own `velocity`: the wheel turns with the frame wherever it lies, and the housing stays still where it passes through the rotating passage. Where the zones and walls are uncertain, a `velocity` on every wall that touches a rotating zone removes the doubt. A radial pressure gradient outlet on a diffuser exit takes its axis from the frame of the cells beside it; see Outflow (p. 287).

Porous Media

A porous zone adds the momentum source

$$S_i = -\left(\frac{\mu}{\alpha} V_i + \frac{1}{2}\rho C_2 |V| V_i\right)$$

where S_i is the source term in the i-th direction, μ the dynamic viscosity, α the permeability, ρ the fluid density, C_2 the inertial resistance factor and V_i the velocity component. The first term is Darcy's law; in laminar flow it gives the pressure drop

$$\Delta P = \frac{\mu}{\alpha} V$$

where ΔP is the pressure drop, μ the dynamic viscosity, α the permeability and V the velocity.

`alpha` (p. 160) defaults to 10^{12}, which is effectively no resistance, so a porous zone does nothing until `alpha` is lowered or `C2` (p. 160) is raised.

> **NOTE**
> The model constants are given in mesh units: α is an area and C_2 an inverse length. The values below assume a mesh in metres, with α in m² and C_2 in 1/m.

MATERIAL	PERMEABILITY α (M²)
Fence	1×10^{-7}
Car radiator	1×10^{-9}
Sand	1×10^{-11}

MATERIAL	INERTIAL RESISTANCE FACTOR C_2 (1/M)
Open cell foam	1000
Gravel	500
Sand	100
Fine mesh	10000

Momentum Source

A momentum source (p. 144) applies an arbitrary body force, supplied as a Python function of the cell centres. The momentum source function (p. 220) returns one source vector per cell, in non-dimensional units. This is a force per unit volume in the compressible solver and a force per unit mass in the incompressible solver.

`func` is sampled once, when the zone is set up, and is not re-evaluated during the solve; an actuator disc, by contrast, re-samples its source at every `update_frequency`. `kind` and `frequency` are accepted by the schema but have no effect on a `momentum` zone.

```
def momentum_drive(*, locations):
    return [(0.0, 0.0, body_force) for _ in locations]

{
    'fluid_zones': {
        'FZ_1': {
            'type': 'momentum',
            'definition': 'pipe_periodic_region.vtp',
            'func': momentum_drive,
        }
    }
}
```

Canopy

A canopy zone represents a tree canopy in a terrain wind model.

`cd` (p. 161), `beta_p` (p. 161), `beta_d` (p. 161), `Ceps_4` (p. 161) and `Ceps_5` (p. 161) are defined in the literature, for example Desmond (2014).

```
def lad_function(*, locations, wall_distances):
    lai = 5.0
    h_can = 20.0
    return [np.interp(d, a_z[0] * h_can, a_z[1]) * lai / h_can for d in wall_distances]

{
    'fluid_zones': {
        'canopy_zone': {
            'type': 'canopy',
            'definition': 'canopy_dacosta.vtp',
            'function': lad_function,
            'cd': 0.25,
            'beta_p': 0.17,
            'beta_d': 3.37,
            'Ceps_4': 0.9,
            'Ceps_5': 0.9,
        },
    }
}
```

The leaf area density varies with 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 `field` (p. 162) is used instead of `definition`, the height of the canopy at each cell is found from the point of the supplied VTK file nearest the cell centre, and read from a node-based scalar array named `'Height'`. A file without that array is rejected at validation.

Atmospheric Boundary Layer (ABL)

WARNING

An ABL zone (p. 163) has no physical effect on the solution. Its fields are validated but not read by the solver: the zone reduces to a bare momentum-source kernel over the selected cells, with no source derived from `roughness_length`, `friction_velocity`, `reference_height` or `wind_direction`. An atmospheric boundary layer profile is set on a reference condition instead; see ABL (p. 297).

Moving or deforming a mesh

zCFD has four distinct mechanisms for placing, moving or deforming a mesh, or the frame its flow is solved in, each configured per model, that is per mesh:

MECHANISM	WHAT IT DOES	WHERE IT IS SET
Initial placement	a single static rotation, translation or scaling of the whole mesh, applied once when it loads	the `scale` and `transform` in the model's `transforms` (p. 169)
Rigid motion of a whole mesh	a mesh rotating or translating during the run, such as a rotor or a moving overset component	the `motion` (p. 169) in the model's `transforms`, with `"moving mesh": True`; see Placing and moving a mesh (p. 365)
Moving reference frame	cells of a still mesh solved in a rotating or translating frame	for part of the mesh, a `rotating` (p. 147) fluid zone on its cell zones; see Rotating zones. For the whole mesh, the `motion` (p. 169) in the model's `transforms`, with `"moving mesh": False`
Mesh morphing	a known deformation of a surface, such as a wing or blade deflecting under load, carried into the volume mesh by RBF or IDW interpolation	an `'rbf transform'` or `'idw transform'` entry in `fluid_zones` (p. 143); see Fluid–structure interaction (p. 320)

Reference entries

- Fluid zones (p. 143), with each zone type
- `rotating` (p. 147), `disc` (p. 149), `porous` (p. 159), `canopy` (p. 161), `momentum` (p. 144)
- `transforms` (p. 169), with its `motion`

Choosing boundary conditions

Which boundary condition should each boundary of the mesh take?

Give every solid surface a wall, and choose its wall model separately. Close an external flow with a farfield on the compressible solver, and an internal flow with an inflow and an outflow; the incompressible solver always uses the latter pair. Put a symmetry condition on a plane of mirror symmetry. Where a boundary is not closed at all but joined to another part of the domain, whether an overlapping mesh, an abutting rotating mesh or its own periodic partner, make it an interface.

`type`	USE IT FOR	SELECTED BY `kind`	NAMES ITS FLOW STATE WITH
`wall` (p. 99)	solid surfaces; see Choosing a wall model (p. 231)	`slip`, `no slip`, `wall function`	—
`farfield` (p. 117)	open boundaries of an external flow, compressible solver only	`riemann`, `pressure`, `supersonic`, `preconditioned`	`condition`
`inflow` (p. 128)	inlets	`pressure`, `massflow`, `default` (incompressible)	`reference`
`outflow` (p. 131)	outlets	`pressure`, `massflow`, `radial pressure gradient`	`reference`
`symmetry` (p. 127)	planes of mirror symmetry	—	—
`interface` (p. 134)	overset boundaries, sliding interfaces and periodic pairs	`overset`, `sliding`, `periodic`	—

Declaring a boundary condition

Boundary conditions are a dictionary of named blocks under the mesh's model entry. Each block is keyed by a name of your choosing, conventionally `BC_1`, `BC_2` and so on, and describes one boundary condition:

```
parameters = {
    "config_version": 2,
    "solver": {...},
    "model": {
        "naca0012": {
            "mesh": "naca0012.h5",
            "boundary_conditions": {
                "BC_1": {"zones": [2, 3], "type": "symmetry"},
                "BC_2": {"zones": [4], "type": "wall", "kind": "no slip"},
                "BC_3": {
                    "zones": [1],
                    "type": "farfield",
                    "condition": "IC_1",
```

```
                    "kind": "riemann",
                },
            },
        }
    },
}
```

Every block requires a `type`, which selects the boundary condition applied to its zones, and `zones`, which lists those zones. Most types also take a `kind`, which selects a variant of the type with its own further keys. A wall's `kind`, for example, chooses between `slip`, `no slip` and `wall function`, and the keys available below it change with the choice.

Zones

`zones` (p. 98) is a list of the zone identifiers of the mesh to which the boundary condition applies. The converters record which source zone name became which identifier; see Zone identifiers (p. 208).

Wall

See Choosing a wall model (p. 231).

Farfield

A farfield boundary closes an external flow, and switches between inflow and outflow face by face according to the local flow direction. It names its flow state through `condition`, a reference condition (p. 39) entry. The incompressible solver has no farfield boundary; close an incompressible domain with inflow and outflow boundaries instead.

The `kind` (p. 118) sets how each face is treated:

kind	TREATMENT
`riemann`	Riemann invariants for a non-reflecting inflow; a viscous outflow takes the pressure from outside.
`pressure`	a pressure boundary for both inflow and outflow.
`supersonic`	the upstream state on inflow and outflow alike.
`preconditioned`	the reference state imposed as a supersonic inflow on every face.

A farfield's `target` (p. 117) turns its angle of attack until a lift or drag coefficient reaches a set value; see Trimming to a target (p. 360).

Inflow

An inflow boundary takes one of three kinds, selected by `kind` (p. 129):

kind	THE INFLOW STATE IS SET FROM
`default`	the reference condition alone; this is the incompressible solver's velocity inlet
`pressure`	a total pressure ratio and a total temperature ratio relative to the reference condition
`massflow`	a mass flow rate or a mass flow ratio

The `default` kind is for the incompressible solver. A compressible deck uses `pressure` or `massflow`, and validation rejects `default` there.

All three name their flow state through `reference` (p. 128), a reference condition (p. 39) entry. The key is `reference`, not `condition`, which belongs to the farfield. On the compressible solver, the `pressure` and `massflow` kinds take a `direction` (p. 128) for the flow to enter along; without one, the flow enters normal to the boundary. A `pressure` inflow also takes a `mach` (p. 128), which fixes the inlet Mach number: the inlet static pressure then follows from the total pressure ratio rather than from the flow inside the domain.

> **NOTE**
>
> A mass flow rate is given in kg/s, and can be used only if the mesh is in metres. A mass flow ratio is non-dimensional, with A_{in} in mesh units: $massflowratio = \rho_{in} U_{in} A_{in} / (\rho_{ref} U_{ref})$. Both must be positive: the boundary type sets the direction of the flow, so zero or a negative value is rejected at validation.

Outflow

An outflow boundary is a pressure, mass flow or radial-pressure-gradient outlet, selected by `kind` (p. 132) as for an inflow. All three kinds name their flow state through `reference` (p. 131). A radial-pressure-gradient outlet holds its `static_pressure_ratio` (p. 131) at the `reference_radius` (p. 131); elsewhere the exit pressure follows radial equilibrium about the axis of the frame the outlet's cells are solved in: the model's rotating `motion`, whether a moving mesh or a frame, or the `rotating` fluid zone (p. 277) that holds those cells.

An outflow sets no temperature, so it takes no total temperature ratio; that belongs on a `pressure` or `massflow` inflow.

> **NOTE**
>
> A mass flow rate is given in kg/s, and can be used only if the mesh is in metres. A mass flow ratio is non-dimensional, with A_{out} in mesh units: $massflowratio = \rho_{out} U_{out} A_{out} / (\rho_{ref} U_{ref})$. Both must be positive: the boundary type sets the direction of the flow, so zero or a negative value is rejected at validation.

Demonstration: what `kind` does FROM THE PIPE CASE



Outlet mass flow against cycle in a turbulent pipe, for a mass-flow outlet and a static-pressure outlet, with the imposed mass flow marked

Both runs use the same mesh, Menter SST with wall functions and a total-pressure inlet; only the outlet changes. The pressure ratio is not tuned to the 10 kg/s target: a pressure outlet sets the pressure, and the flow follows.

`setting kind values massflow, pressure case pipe`

Interface

An interface boundary, `{"type": "interface"}`, joins a boundary to another part of the domain rather than closing it: the solution on one side is supplied by the other. The `kind` (p. 135) selects how the two sides are joined:

kind	USE IT WHERE	WHAT CROSSES THE JOIN
`overset`	this mesh overlaps another	fringe values, taken from the mesh beneath
`sliding`	two meshes meet on a shared surface of revolution with non-matching faces	a conservative flux
`periodic`	two boundaries of the same mesh are one surface, related by a rotation or a translation	the flow leaving one boundary enters the other

Every interface names what it connects to in its `connection` (p. 134) block. What the connection holds depends on the `kind`, and is described under each heading below.

NOTE

A case with an overset or sliding interface requires an implicit pseudo-time scheme, `"implicit euler"`, `"lu-sgs"` or `"mf-gmres"`; an explicit scheme alongside either is rejected at validation. See Choosing a time-marching

scheme (p. 241).

Overset

An overset interface goes on the outer boundaries of a mesh that is overset on one or more other meshes. Its connection names the model or models whose meshes provide donor data; without a connection, the solver maps against every preceding model. See Overset meshes (p. 328) for how an overset case is set up and solved.

```
"BC_2": {
    "zones": [10],
    "type": "interface",
    "kind": "overset",
    "connection": {"model": ["background"]},
},
```

Sliding

A sliding interface joins two models whose meshes abut without overlapping on a shared cylindrical, conical or annular surface, typically a rotating region and the stationary region around it. The faces on the two sides need not match. The two face sets are clipped against each other into a common set of intersection polygons, one numerical flux is computed for each polygon, and that flux is applied to the cell on each side. The interface is therefore conservative, not interpolated.

A sliding interface is declared on both models, and each names the other. The connection gives the model and the boundary on the other side; that boundary must itself be a sliding interface whose connection names this one back.

The interface must be a surface of revolution about the rotation axis of the side that rotates. That axis comes from the model's rotating `motion` (p. 169), with `"moving mesh": True` for a turning rotor or `"moving mesh": False` for a frozen rotor; the interface itself takes no `axis` or `origin`. When neither side rotates, the two meshes are joined directly, and the axis is found from the shape of the interface itself.

The interface may be a cylinder, a cone, a flat annulus, a spherical band, a disc or spherical cap that reaches the axis, or a whole sphere centred on the axis. It must cover the full circle about the axis; a periodic sector is not supported. The summary printed at the start of the run names the shape the solver found, which is a quick check that the right zones were listed.

A closed or compound surface, such as the two faces and the side of a drum, can be declared either way:

- as one sliding interface on each side that lists every zone. The solver groups the zones into single surfaces, a cylinder meshed as two zones counting as one, and pairs each surface with the matching one on the other side;
- as one sliding interface per surface, each naming its partner, when the pairing should be stated explicitly.

A zone that matches no surface on the other side stops the run at setup, with the zone, the model and the shape it was found to have.

```
"outer": {"boundary_conditions": {
    "BC_2": {"zones": [10, 63, 62], "type": "interface", "kind": "sliding",
             "connection": {"model": "disc", "boundary": "BC_1"}}}},
"disc": {"boundary_conditions": {
    "BC_1": {"zones": [21, 22, 23], "type": "interface", "kind": "sliding",
             "connection": {"model": "outer", "boundary": "BC_2"}}}},
```

A rotor turning inside the stator around it, each connected to the other:

```
"model": {
    "rotor": {
        "mesh": "rotor.h5",
        "transforms": {
            "motion": {
                "type": "rotating",
                "axis": [0.0, 0.0, 1.0],
                "origin": [0.0, 0.0, 0.0],
                "omega": 100.0,
                "moving mesh": True,
            },
        },
        "boundary_conditions": {
            "BC_3": {
                "zones": [21],
                "type": "interface",
                "kind": "sliding",
                "connection": {"model": "stator", "boundary": "BC_2"},
            },
        },
    },
    "stator": {
        "mesh": "stator.h5",
        "boundary_conditions": {
            "BC_2": {
                "zones": [20],
                "type": "interface",
                "kind": "sliding",
                "connection": {"model": "rotor", "boundary": "BC_3"},
            },
        },
    },
},
```

Two settings trade conservation against other properties of the interface:

SETTING	`conservative` / `segment` (THE DEFAULTS)	`free stream` / `parent`
`area` (p. 137)	both sides use the mean of the two parent area maps: conservative to round-off, closure error first order in cell width	each side uses its own parent map: closure and free-stream preservation exact, conservation deficit second order

SETTING	`conservative` / `segment` (THE DEFAULTS)	`free stream` / `parent`
`flux` (p. 137)	one flux per intersection polygon, with a shared area vector, so the two sides cancel exactly	one flux per parent face, against the area-weighted average of its donors: quieter as cells pass, but not conservative

The two `area` settings are identical on a planar interface and for matching meshes. Under `parent` the flux does not change when the intersection pattern changes, so the noise a rotating interface makes as cells pass one another is lower; but each side fluxes its own faces and nothing cancels. `segment` is the default for that reason.

`segment_capacity_factor` (p. 138), `sliver_tolerance` (p. 138) and `closure_tolerance` (p. 138) are rarely needed. The first sizes the segment storage reserved for each interface face; setup sweeps a full revolution, and stops with the factor actually needed if it is too small. The closure tolerance is the largest fraction of a face's area that may be left uncovered before setup stops, so that a real gap between the two sides, from mismatched radii, a partial annulus or a wrong axis, shows at setup rather than as a silent leak.

The rotor's `motion` turns its mesh every time step and requires an unsteady run. A frozen rotor, steady or unsteady, sets `"moving mesh": False` on the same `motion`, which solves the rotor in a rotating frame over every cell of its mesh. Both are described in Placing and moving a mesh (p. 365).

Periodic

A periodic interface treats two boundaries of the same mesh as one surface: the flow leaving one boundary enters the other. It is declared in pairs. Each boundary's connection names the other boundary of the pair, and that boundary must name this one back.

Each side carries the `transform` (p. 134) that maps it onto the other, so the two transforms must be inverses of each other. A rotation of θ about an axis pairs with a rotation of −θ about the same axis line, or equivalently θ about the reversed axis; a translation by a vector pairs with the translation by its negative. Both sides of a pair use the same transform `type`, `'linear'` or `'rotated'`.

> **WARNING**
>
> The incompressible solver supports only the `'linear'`, translational, periodic transform; `'rotated'` periodic interfaces need the compressible solver. A rotated pair on an incompressible case, `{"solver_settings": {"type": "incompressible"}}`, passes validation but is ignored, with a warning in the log.

A linear periodic pair:

```
"BC_3": {
    "zones": [11],
    "type": "interface",
    "kind": "periodic",
    "connection": {"boundary": "BC_4"},
    "transform": {"type": "linear", "vector": [0.0, 0.0, 1.0]},
},
"BC_4": {
```

```
    "zones": [12],
    "type": "interface",
    "kind": "periodic",
    "connection": {"boundary": "BC_3"},
    "transform": {"type": "linear", "vector": [0.0, 0.0, -1.0]},
},
```

A rotated periodic pair for a 120° sector:

```
"BC_5": {
    "zones": [5],
    "type": "interface",
    "kind": "periodic",
    "connection": {"boundary": "BC_6"},
    "transform": {
        "type": "rotated",
        "theta": math.radians(120.0),
        "axis": [1.0, 0.0, 0.0],
        "origin": [0.0, 0.0, 0.0],
    },
},
"BC_6": {
    "zones": [6],
    "type": "interface",
    "kind": "periodic",
    "connection": {"boundary": "BC_5"},
    "transform": {
        "type": "rotated",
        "theta": math.radians(-120.0),
        "axis": [1.0, 0.0, 0.0],
        "origin": [0.0, 0.0, 0.0],
    },
},
```

Reference entries

- Boundary conditions (p. 98), with `type` and `zones`
- `wall` (p. 99), `farfield` (p. 117), `inflow` (p. 128), `outflow` (p. 131), `symmetry` (p. 127) and `interface` (p. 134)
- Reference conditions (p. 39)

Choosing initialisation and reference state

Which named flow state fills the field at the start, which one non-dimensionalises the results, and when should a run restart instead?

Define each flow state once, as a named entry of `reference_conditions`, and refer to it by name wherever it is used. Name the state that fills the field in `initial_conditions`. Leave `reference` out unless the results must be non-dimensionalised against a different state; it then follows `initial_conditions`. Where a solution already exists, restart from it instead: on the same mesh with `restart`, and on a different mesh with `interpolate_restart` as well.

ROLE	SET BY	WHERE	IF LEFT OUT
fill the flow field at the start	`initial_conditions` (p. 48)	`initialisation`	a validation error, unless `restart` is set
non-dimensionalise the force reporting	`reference` (p. 48)	`initialisation`	the name given in `initial_conditions`
set the flow entering or leaving	`condition` on a farfield, `reference` on an inflow or outflow	the boundary condition	—
continue from an earlier solution	`restart` (p. 48)	`initialisation`, or per mesh	a fresh start

Reference conditions

A reference-condition entry defines a fluid state. The entries sit in the `reference_conditions` (p. 39) dictionary under `solver`, each keyed by a name, conventionally `IC_1`, `IC_2` and so on:

```
parameters["solver"]["reference_conditions"] = {
    "IC_1": {....},
    "IC_2": {....},
}
```

The same dictionary, and the same shape of entry, serves three roles; there is no structural split between the initial condition and a reference condition:

- `initial_conditions` in `initialisation` names the entry that fills the flow field at the start of the simulation. There is no default such as `IC_1`: the entry must be named.
- `reference` in `initialisation` names the entry that non-dimensionalises the force reporting.
- A boundary condition names the entry that sets the flow entering or leaving the domain: a farfield through `condition`, and an inflow or outflow through `reference` (see boundary conditions (p. 285)).

Each entry is validated against a model chosen by `{"equations": {"type": ...}}`. An `euler` entry rejects the viscosity and turbulence keys outright, a `viscous` entry adds viscosity, and a `rans` or `les` entry adds the full turbulence set. The routing is automatic: the deck supplies the keys relevant to its equation type, and the schema picks the model.

One entry can inherit from another by naming the base entry in a `reference` key of its own. The inheritance is resolved before validation: the keys are merged and overridden, and `reference` is then discarded, so it never reaches the solver. This is how a lean, ratio-only entry, used for boundary conditions specified relative to a base flow state, is written without repeating every key. See `total_pressure_ratio` (p. 40) and its siblings for the ratio keys available on every entry.

```
parameters["solver"]["reference_conditions"] = {
    "IC_1": {
        "v": {
            "vector": [1.0, 0.0, 0.0],
            "mach": 0.15,
        },
        "pressure": 101325.0,
        "temperature": 273.15,
        "reynolds_no": 6.0e6,
        "reference_length": 1.0,
        "turbulence_intensity": 5.2e-2,
        "eddy_viscosity_ratio": 1.0,
        "ambient_turbulence_intensity": 5.2e-2,
        "ambient_eddy_viscosity_ratio": 1.0,
    },
}
```

Initial state

```
parameters["solver"]["initialisation"] = {
    "initial_conditions": "IC_1",
}
```

`initial_conditions` (p. 48) names the entry of `reference_conditions` that provides the initial flow field. If a Python callable is given instead of a name, it is called once for every cell to provide the initial flow variables (see the initialisation function (p. 220)), and `reference` must then be set explicitly (see Reference state).

> **NOTE**
>
> There is no implicit fallback. `initial_conditions` must be set, to a name or a callable, unless `restart` is `True`; leaving it out of a fresh run is a validation error.

Any mesh in the `model` block can set its own `initialisation`. It lists only what differs on that mesh; everything it leaves out comes from the solver's `initialisation`, as described in settings for one mesh (p. 190). This is useful in a multi-mesh case, an overset case for example, where only one mesh should restart, or where the meshes start from different named conditions. Here the wing restarts and keeps the solver's initial conditions:

```
parameters["model"]["wing"]["initialisation"] = {"restart": True}
```

Reference state

```
parameters["solver"]["initialisation"] = {
    "reference": "IC_1",
    "initial_conditions": "IC_1",
}
```

`reference` (p. 48) names the reference condition that provides the reference quantities for the non-dimensionalisation of the force reporting. The name must be an entry defined in `reference_conditions`.

Left out, `reference` falls back to the name given in `initial_conditions`. It must be set explicitly when `initial_conditions` is a Python callable, an initialisation function (p. 220), since a callable has no static name to fall back on; a missing `reference` is then a validation error, not a silent default.

If the initial condition has zero velocity, as in a quiescent cavity, set `reference` to a different reference condition that carries the physical velocity and Reynolds number, so that the solver has something non-zero to non-dimensionalise against.

Velocity

Velocity is given in one of two ways: as a bare vector, or as a vector and a Mach number. With a Mach number, the vector gives the direction, and the magnitude is computed from the Mach number and the speed of sound at the given pressure and temperature.

A velocity of 50 mesh units per second in the x direction:

```
..
"v": {"vector": [50.0, 0.0, 0.0]}
..
```

A velocity of Mach 0.2 in the x direction:

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

Viscosity

The dynamic viscosity, also called the shear, absolute or molecular viscosity, is defined at the static temperature of the entry. It is given either as a dimensional quantity or through a Reynolds number and a reference length. Both forms are valid only for the `viscous`, `rans` and `les` equation types; leave both out for `euler`.

```
"viscosity": 1.83e-5,
```

or

```
"reynolds_no": 5.0e6,
"reference_length": 1.0,
```

The Reynolds number is defined as

$$Re = \frac{\rho V L_{ref}}{\mu}$$

with ρ the density, V the velocity, L_{ref} the reference length and μ the viscosity. Exactly one of `reynolds_no` (p. 41) and `viscosity` (p. 41) must be set; setting both, or neither, is a validation error.

Turbulence intensity and eddy viscosity

Turbulence intensity is the ratio of the velocity fluctuations u' to the mean flow velocity. An intensity of 1 per cent is considered low, and one above 10 per cent high. The keys below apply to the `rans` and `les` equation types, and are not valid for `euler` or `viscous`. `turbulence_intensity` (p. 41) defaults to 0.01 and `eddy_viscosity_ratio` (p. 41) to 0.1.

```
"turbulence_intensity": 0.01,
"ambient_turbulence_intensity": 0.01,
```

Both intensities are given as fractions. The ambient keys set the level at which the turbulence is sustained.

The eddy viscosity ratio, μ_t/μ, depends on the type of flow. For external flows it varies from 0.1 to 1, and in a wind tunnel from 1 to 10.

For internal flows it depends more strongly on the Reynolds number, since the largest eddies are limited by the characteristic lengths of the geometry, such as the height of a channel or the diameter of a 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": 0.1,
"ambient_eddy_viscosity_ratio": 0.1,
```

Both ambient keys, `ambient_turbulence_intensity` (p. 41) and `ambient_eddy_viscosity_ratio` (p. 41), default to 1.0e-20: a near-zero numerical floor, not the freestream value. They do not fall back to the

freestream turbulence level; to sustain the ambient level at the freestream value, set them explicitly to the same values:

```
"reference_conditions": {
    "IC_1": {
        ..
        "turbulence_intensity": 5.2e-2,
        "eddy_viscosity_ratio": 1.0,
        "ambient_turbulence_intensity": 5.2e-2,
        "ambient_eddy_viscosity_ratio": 1.0,
    }
},
```

Velocity profile

A reference condition can also carry a `profile` (p. 39): the variation of the flow state near a boundary, read from a file or given as an atmospheric boundary layer.

field

The file named by `field` (p. 43) is in the ParaView/VTK VTP format. It contains a node array with one or more of the array names `'Pressure'`, `'Temperature'`, `'Velocity'`, `'TI'` and `'EddyViscosity'`. zCFD looks up those values in the solution by finding the nearest point in the file. `use_wall_distance` (p. 43) localises the field by wall distance rather than by the z coordinate. The path is relative to the directory zCFD is run from, and a deck whose file is missing or cannot be read is rejected at validation.

```
..
"profile": {
    "field": "inflow_field.vtp",
    "use_wall_distance": True,
},
..
```

> **NOTE**
> The field overrides the conditions given elsewhere in the entry, so the entry need specify only the conditions that differ from the default.

ABL

An atmospheric boundary layer profile is set on a reference-condition entry as `abl` (p. 43) in its `profile`:

```
{ "profile": { "abl": { "roughness_length": 0.1, "friction_velocity": 0.5 } } }
```

`ground_level` (p. 46) is the canonical spelling; `z0` is also accepted, and normalised to `ground_level`. `up` (p. 46) is a vertical direction vector of its own, independent of the gravity vector set on the fluid model.

NOTE

`surface_layer_height`, `monin_obukhov_length`, `tke`, `up` and `ground_level` are accepted only when the solver is compressible, or when `{"solver_settings": {"type": ...}}` is unset. The incompressible inflow boundary reads only `roughness_length` and `friction_velocity` from the profile, so setting any of the other five on an incompressible deck is a validation error rather than a setting that would silently have no effect.

roughness_length

`roughness_length` (p. 45) is a measure of the height at which the mean wind speed falls to zero, through the friction between the air and the surface.

It is usually written z_0. It is the height above the surface at which the logarithmic wind profile, extrapolated downwards, reaches zero speed. The logarithmic profile describes the variation of wind speed with height in the atmospheric boundary layer.

The roughness length depends on the surface and on the objects on it. Forests, urban areas and bodies of water have different roughness lengths: smooth open water has a small one, and a densely forested area a larger one.

The wall boundary of the mesh is taken to lie at the height of the roughness length rather than at ground level, so the first cell should not be smaller than the roughness length.

friction_velocity

The friction velocity, `friction_velocity` (p. 45), usually written $u_\star$ and read 'u-star', is the characteristic velocity of the turbulent motion in the surface layer of the atmosphere, generated by the shear stress between the air and the surface. It is the square root of the shear stress divided by the air density:

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

where $u_\star$ is the friction velocity, τ the shear stress between the air and the surface, and ρ the density of the air.

The shear stress arises from the transfer of momentum between the moving air and the roughness elements of the surface. It is related to the wind speed and the roughness length through the logarithmic wind profile,

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

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

The friction velocity measures the intensity of the turbulence near the surface. It bears on heat and moisture exchange, on the dispersion of pollutants and on the structure of the boundary layer. Its magnitude depends on the surface roughness, the wind speed and the atmospheric stability, among other factors.

surface_layer_height

The surface layer is the lowest part of the atmospheric boundary layer, in which the wind speed, temperature, humidity and turbulence are governed by the surface beneath, and in which the logarithmic profile holds. Its height is usually estimated from empirical relationships or from the atmospheric stability. It can range from a few metres to a few hundred metres, depending on the roughness length, the wind speed and the stability, and it changes through the day with daytime heating and night-time cooling.

In zCFD, `surface_layer_height` (p. 45) caps the profile: above it, the profile holds the value it reaches there. With the SST model, a surface layer height left unset while `friction_velocity` is given is taken as the height at which the logarithmic profile reaches the reference speed, $z_0 \exp(\kappa U_{ref}/u_\star)$; a `friction_velocity` left unset is derived from the surface layer height in the same way.

monin_obukhov_length

The Monin–Obukhov length, `monin_obukhov_length` (p. 45), usually written L, characterises the stability of the air near the surface. It is named after A. S. Monin and A. M. Obukhov, whose work founded the similarity theory of the surface layer.

It is formed from the friction velocity and the surface buoyancy flux:

$$L = -\frac{u_\star^3\,\theta_v}{\kappa\, g\, \overline{w'\theta_v'}}$$

where L is the Monin–Obukhov length, $u_\star$ the friction velocity, θ_v the virtual potential temperature, κ the von Kármán constant, approximately 0.4, g the acceleration due to gravity, and $\overline{w'\theta_v'}$ the vertical flux of virtual potential temperature.

L measures the balance between the mechanical production of turbulence by wind shear and its buoyant production or suppression by vertical temperature gradients; $|L|$ is roughly the height above which buoyancy matters as much as shear. Positive values indicate stable conditions, and negative values unstable ones. A neutral atmosphere, with no buoyancy flux, corresponds to $|L| \to \infty$. $L = 0$ is rejected as an input value; leave the key out for neutral stability.

The Monin–Obukhov length bears on the structure of the turbulence, on heat and moisture exchange and on the dispersion of pollutants. It plays a large part in the behaviour of the atmospheric boundary layer, including the vertical mixing of air and the development of turbulence. It depends on the surface properties, the wind speed, and the gradients of temperature and humidity.

Restart

A restart loads a solution from a previous run and continues the solver from that point. By default the solver looks for a `<casename>_results.h5` file to restart from; `restart_casename` (p. 49) names a different case. `restart_ignore_history` (p. 49) set to `True` begins the simulation from that results file while ignoring its cycle history.

The default restart assumes the same mesh, and stops if the case named by `restart_casename` ran on a different one. To start from that case's solution on another mesh, see Restarting on a different mesh.

The restart settings sit in `initialisation`, as the global default, and a mesh can change any of them in its own `initialisation`. `solution_smooth_cycles` (p. 12) is a system setting rather than a restart one, and sits separately in `solver_settings`:

```
{ "solver_settings": { "solution_smooth_cycles": 0 } }
```

A restart from the previous solution of the case `cylinder_steady_start_init`:

```
parameters["solver"]["initialisation"] = {
    "initial_conditions": "IC_1",
    "restart": True,
    "restart_casename": "cylinder_steady_start_init",
}
```

Restarting on a different mesh

`interpolate_restart` (p. 49) starts a case from another case's solution on a different mesh: a finer or coarser one, one of another cell type, or one of a modified geometry. It is available for the compressible solver.

```
parameters["solver"]["initialisation"] = {
    "initial_conditions": "IC_1",
    "restart": True,
    "interpolate_restart": True,
    "restart_casename": "cylinder_coarse",
}
```

`restart_casename` names the case to start from. That case records the mesh it ran on, and the mesh file must still be present beside it. For an older results file with no such record, `restart_meshname` (p. 49) names the mesh.

KEY	DEFAULT	MEANING
`interpolation` (p. 49)	`linear`	How each new cell's value is formed: `linear` or `injection`
`interpolate_outside_limit` (p. 50)	`0.001`	Largest fraction of the new cells that may lie outside the old mesh
`restart_models` (p. 50)	none	Which old model each new model starts from, where the names differ
`restart_meshname` (p. 49)	none	The old mesh, for a results file that does not record it
`restart_ignore_history` (p. 49)	`False`	Start the time history afresh rather than continue the old case's

How each cell's value is calculated

Each new cell is located by its centre $\mathbf{x}$: the old cell c that contains $\mathbf{x}$ supplies the value. Every conserved variable is treated separately.

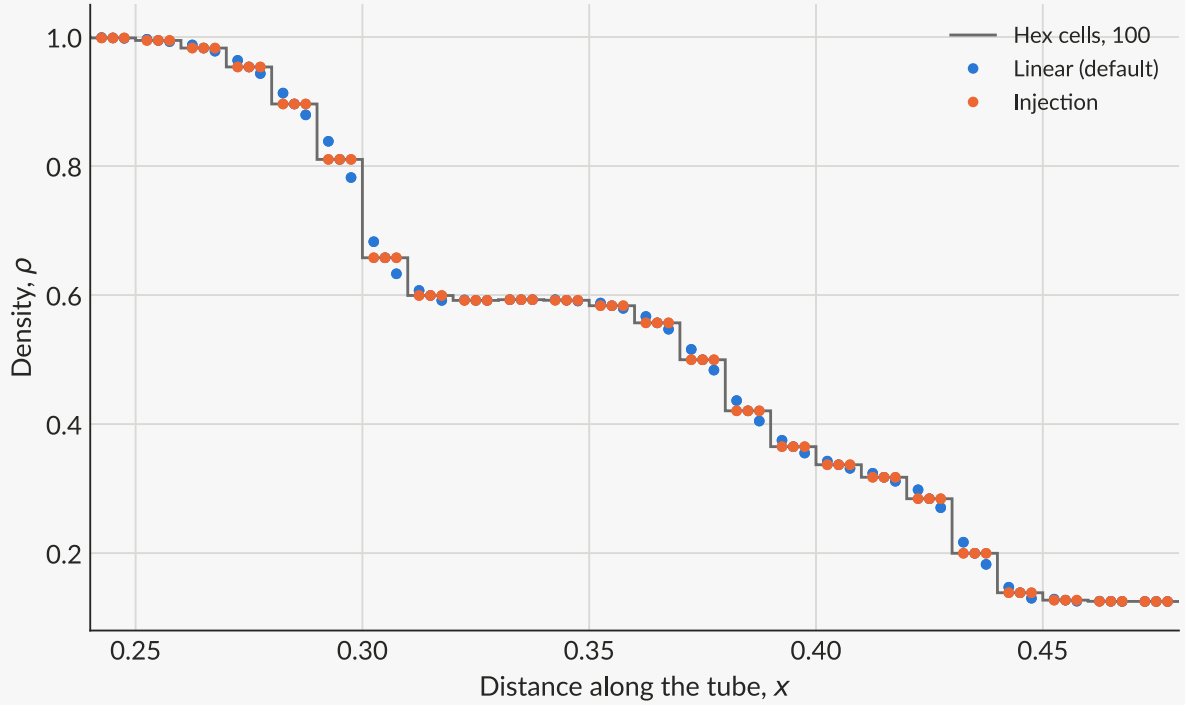
With `interpolation` set to `linear`, the default, the value is reconstructed from the old cell's value ϕ_c and its gradient,

$$\phi(\mathbf{x}) = \phi_c + \nabla\phi_c \cdot (\mathbf{x} - \mathbf{x}_c),$$

where $\mathbf{x}_c$ is the old cell's centre. The gradient is a least-squares fit to the cells that share a face with c, each weighted by the inverse square of its distance from $\mathbf{x}_c$. The result is then held between the smallest and largest values of ϕ in c and those neighbours. A field that varies linearly is reproduced exactly, and no new maximum or minimum is created, so density and energy stay positive wherever the old solution was. Where the fit cannot be made, for a cell with too few neighbours, the old cell's value is copied. In a parallel run the fit uses the neighbours held on the same task as c.

With `interpolation` set to `injection`, each new cell takes the value of the old cell that contains its centre. This is first-order: a new mesh finer than the old one sees the old mesh's steps. It is appropriate when the values must be exactly those the old case held.

Demonstration: what `interpolation` does FROM THE SHOCKTUBE CASE



Density in each cell of a shock tube interpolated from 100 hex cells onto 600 tet cells, with linear and with injection, against the hex cells it came from

```
setting interpolation values linear, injection case shocktube
```

Neither method conserves mass or energy exactly; the totals on the new mesh differ from the old ones by the interpolation error.

Cells outside the old mesh

A new cell whose centre lies in no old cell takes the value of the closest old cell, judged by cell centre. In a parallel run the search looks first among the old cells on the same task, so the cell chosen can be a near neighbour of the closest. A handful of such cells is expected where the two meshes approximate a curved boundary differently. When more than `interpolate_outside_limit` of the new cells lie outside, 0.1% by default, the run stops before the first cycle and reports how many did. That usually means the

two meshes do not cover the same domain: check the mesh, its scale and its transforms before raising the limit.

A changed geometry

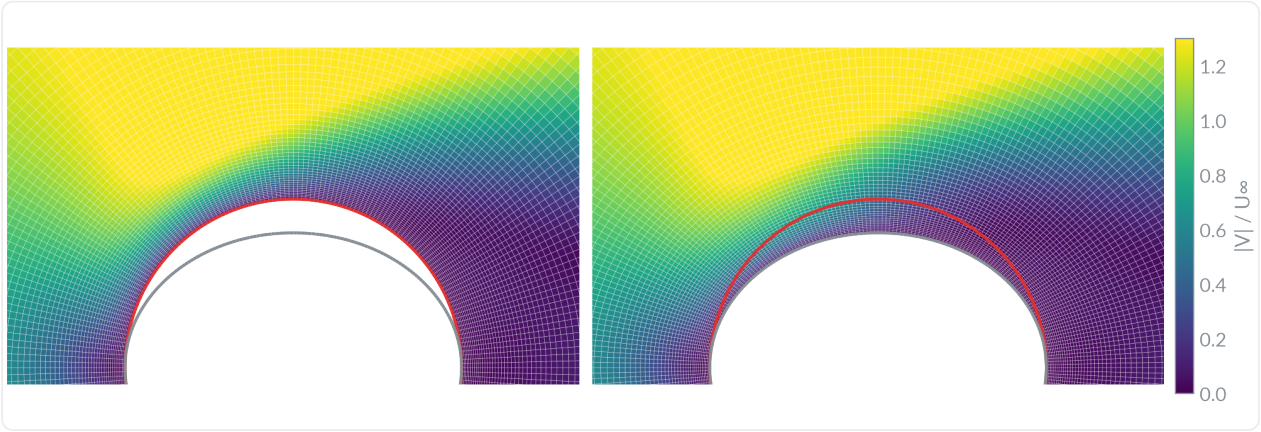
Before the cells are located, every wall face of the new mesh is measured against the old walls: its distance from the nearest old wall face, along that face's normal, and that distance in first-cell heights. A remesh of the same geometry moves the wall by a fraction of a first-cell height, and the cells are located as above. When any wall has moved by more than one first-cell height, the geometry has changed, and the cells near the walls are located by wall distance instead.

Each new cell within a blend length L of a wall, four times the largest wall movement, is sampled at the point that lies at its own distance d from the old wall, beside the point of the old wall nearest its own wall foot. Away from the wall the sample point returns to the cell centre:

$$\mathbf{x}_s = \mathbf{x} + w(d/L)\,(\mathbf{p}_{old} + d\,\mathbf{n}_{old} - \mathbf{x}), \qquad w(r) = (1-r)^4(4r+1),$$

where $\mathbf{p}_{old}$ is the old wall point and $\mathbf{n}_{old}$ the old wall's normal into the flow. Momentum is turned through the same fraction w of the angle between the old and new wall normals, so the flow beside a tilted wall stays tangent to it. The value at $\mathbf{x}_s$ is then formed by `interpolation` as above.

The cells beside the new wall therefore start from the old boundary layer at the same wall distance. Cells in the space the old body filled, which lie outside the old mesh, are sampled in the old flow instead and are not counted as outside. The flow away from the body starts from the old solution unchanged.



Flow speed above a cylinder at Reynolds number 40, and the same flow on an ellipse mesh as the interpolated restart writes it, with the cylinder wall in red and the ellipse wall in grey

The walls are those this case's boundary conditions name, applied to both meshes, so the wall zones must carry the same numbers in the two meshes.

Variables, time and models

The variables both cases solve are mapped. The rest come from the initial conditions, which is why `initial_conditions` is required: a RANS case started from an Euler solution takes its turbulence variables from them. A case cannot start from an incompressible solution, nor an incompressible case from a compressible one.

Physical time continues from the old case. For a dual time stepping case the earlier time levels are

mapped as well, so the time history carries across. `restart_ignore_history` set to `True` starts the time history afresh.

Each mesh of this case starts from the old case's mesh of the same model name. `restart_models` pairs them where the names differ; keys are this case's models and values the old case's:

```
parameters["solver"]["initialisation"]["restart_models"] = {"wing_fine": "wing"}
```

The wall distance and every other mesh quantity are computed on the new mesh as usual.

What the log reports

The start of the log reports where each value came from. A cylinder solution started on an ellipse of the same chord:

```
Interpolated restart from cylinder_re40_OUTPUT/body/body_results.h5
  source             cylinder_re40.h5   32,768 cells
  target             ellipse_ar0p8.h5   34,560 cells
  interpolation      linear
  mapped             rho, rhoU, rhoV, rhoW, rhoE
  from IC_1          (none)
  wall movement      0.1  (24.8 first-cell heights); 10,468 cells within 0.4 of a wall mapped
 ↪  by wall distance
  outside source     none
  mass, energy       +0.008%, +0.008% against the restart case
  time               0.6 s  (read 0.4, search 0.2, map 0.0)
```

A further line, `flow volume`, gives the change in the volume of the flow. Neither method conserves mass or energy exactly, and a changed body or far field changes the volume of the flow as well, so a change in mass that matches the change in volume comes from the domain, not the interpolation. A dual time stepping case also lists its time levels after the mapped variables.

Restarting on a different number of tasks

A restart appends to the VTKHDF output it finds rather than replacing it, so a restarted case continues one time series instead of starting a second. That holds only while the partition count is unchanged. A VTKHDF series stores its sizes as one entry per partition per step, so a run appending at a different task count cannot extend the series, and the earlier output would in practice be lost.

When a restart is about to run on a different number of tasks from the solution it restarts from, the `.vtkhdf` files already in the output directory are therefore moved into a subdirectory named for the partition count that produced them, before the solver opens its output:

```
plate_coarse_OUTPUT/background/P2/background.vtkhdf
plate_coarse_OUTPUT/background/P2/background_wall.vtkhdf
```

The new run then starts its own series at the new count, and the old output stays where it is, and readable. The log reports the move:

```
Restarting from a solution generated on 2 partition(s) with 1 task(s):
moved 4 VTKHDF file(s) aside so the existing output is not written over.
plate_coarse_OUTPUT/background/P2
```

The archive follows these rules:

- Only the case's own output directory is swept. With `restart_casename` naming another case, that case's output is left alone, since this run does not write into it.
- Each file is labelled from its own recorded partition count, so output left by an earlier run of the same case at a third count is labelled correctly, not by the restart source's count.
- A file already written at the new task count is left in place, so a restart on the same number of tasks extends its series as usual.
- Passing through the same count twice, as in 2, then 1, then 2, then 1, archives into `P2` and then `P2.1`; an existing archive is never written over.
- The archive is a subdirectory of the model directory, which post-processing does not look inside: `zutil` finds models one level below `<case>_OUTPUT` and lists output non-recursively. An archive is therefore never mistaken for a mesh, nor picked up as live output.
- Checkpoint (`_results.h5`) and report files are not moved. Only `.vtkhdf` output is affected.

Advanced restart

The SA-neg solver can restart from results generated by the Menter SST solver, and the SST solver from SA-neg results; by default the turbulence fields are reset to zero. An approximate SA-neg field cannot be generated from SST results from a control file, since the turbulence schema has no key for it.

Reference entries

- `reference_conditions` (p. 39), with `v` (p. 39), `reynolds_no` (p. 41), `viscosity` (p. 41) and `profile` (p. 39)
- `initialisation` (p. 47): `initial_conditions` (p. 48), `reference` (p. 48), `restart` (p. 48), `interpolate_restart` (p. 49), `interpolation` (p. 49), `interpolate_outside_limit` (p. 50) and `restart_models` (p. 50)

Aeroacoustics (FWH)

A Ffowcs-Williams Hawkings (FWH) solver, which is used for prediction of farfield noise, is included in the zCFD distribution. The zCFD FWH solver uses the Farassat 1A formulation, and can accept both permeable and impermeable FWH surfaces.

What is an FWH solver for?

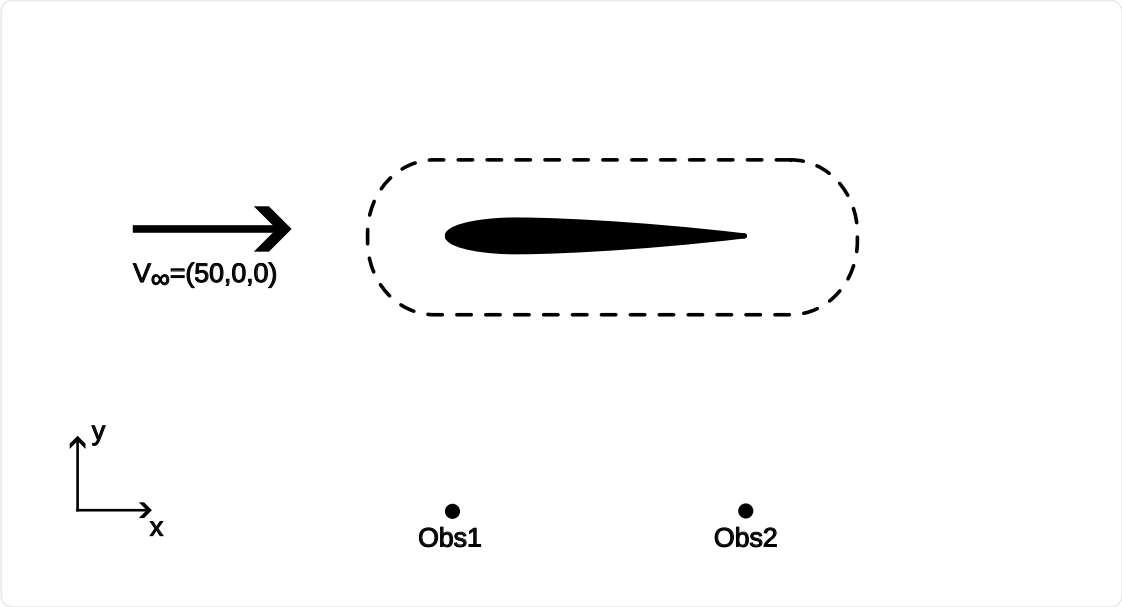
A common problem in CFD is the prediction of the farfield aerodynamic noise which results from a particular airflow. Two issues make prediction of farfield aerodynamic noise challenging. First of all, many of the sources of aerodynamic noise (turbulence, vortex shedding etc.) require high fidelity, time accurate simulations and fine CFD meshes to be properly captured. Secondly, a mesh must be very fine and the simulation must have very low numerical dissipation to propagate aerodynamic noise any distance within a CFD mesh.

The first issue of noise simulation can be made tractable by the use of techniques like FRPM (Fast Random Particle Method), but the second issue of far-field mesh requirements for noise propagation means that e.g. predicting the noise from a helicopter main rotor a kilometre away is not feasible using a conventional CFD simulation.

This issue of noise propagation in the far-field is solved by the use of the Ffowcs-Williams Hawkings (FWH) equations. These equations propagate sound to an observer an arbitrary distance away without the use of a mesh which extends all the way to the observer.

How does an FWH solver work?

To use the FWH equations to propagate noise to a far-field observer, a CFD simulation is run first to record the flow variables ($p(\mathbf{x}, t)$, $\rho(\mathbf{x}, t)$ and $\mathbf{u}(\mathbf{x}, t)$) on an 'FWH surface' in the near-field, noise generating region. Figure 1 shows an FWH surface enclosing the nearfield for a simulation of an aerofoil in a wind tunnel.



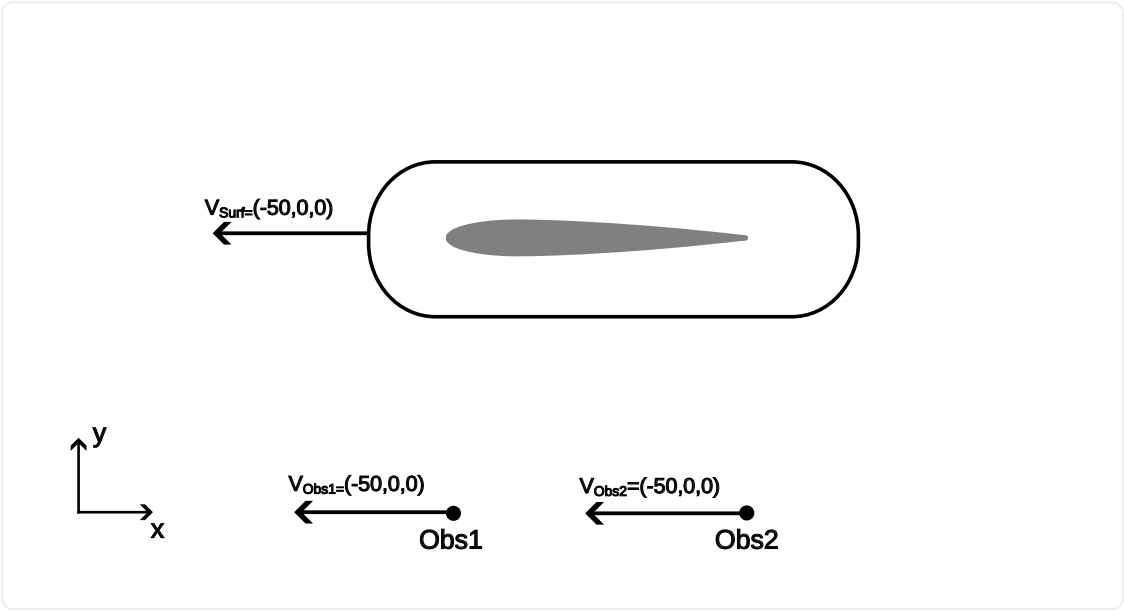
CFD simulation of an aerofoil, including an FWH surface

The FWH surface should enclose the noise generating region, and the CFD mesh should be sufficiently fine in the near-field region to propagate noise to the FWH surface without excessive dissipation. Ideally, the FWH surface should not intersect the nearfield region of any wakes.

In some situations, it is sufficient to collect FWH surface data on the wall instead of using a separate FWH surface inside the fluid domain. FWH surfaces coincident with the wall are called *impermeable* (as opposed to *permeable*) FWH surfaces, and only require p (not p, ρ and $\mathbf{u}$) to be recorded.

Once the flow variables have been recorded on the FWH surface, the noise can be propagated to the observers using an FWH solver. To do so, the motion of the FWH surface and observers must be defined in the FWH simulation. The FWH equations assume a quiescent medium (i.e. still air, with a free-stream velocity of zero), which means that the FWH surface and observer motions are frequently different to the motions used in the CFD simulation used to produce the FWH surface data.

In the example of an aerofoil in a wind tunnel, the CFD simulation has a freestream velocity, which does not satisfy the 'quiescent medium' condition. When using an FWH simulation to calculate the noise at the farfield observer microphone in Figure 1, a co-ordinate transformation is therefore required, in which both the FWH surface and the observer microphone are 'flown' forwards through the quiescent medium, as in Figure 2.



FWH simulation of an aerofoil, with quiescent medium co-ordinate transformation applied

Once the FWH surface data has been collected from a CFD simulation and the FWH surface and observer microphone settings have been correctly defined, the FWH equations are used to propagate the pressure and velocity signals from the FWH surface to the farfield observer locations. The equations used in the zCFD FWH solver are the Farassat 1A equations, with a retarded time formulation. The retarded time is the time τ_* at which a sound wave must be emitted by a point on the FWH surface in order for it to reach an observer at time t, given the prescribed motions of both surface and observer.

> **NOTE**
>
> The Farassat 1A equations used by the zCFD FWH solver are only valid when the FWH surface moves through the quiescent medium subsonically. The retarded time algorithm in the FWH solver will also struggle to converge as the closing speed between an FWH surface and an observer approaches the speed of sound.

Running the zCFD FWH solver

There are two ways of running the zCFD FWH solver - the fwh python module and the run_fwh utility. The *fwh python module* is flexible and intended for advanced users with knowledge of the python programming language. The *run_fwh utility* is a simpler interface which requires no knowledge of the python programming language and is intended for FWH data generated using *zCFD*.

This documentation actually introduces the *fwh python module* first, since doing so illustrates some mechanics of FWH solvers which all users should be aware of. The simpler *run_fwh utility* method is subsequently introduced as an alternative, more automated method to get the same results. For a quick introduction to the zCFD FWH solver which starts with the *run_fwh utility*, see the FWH tutorial (p. 426).

To use the zCFD FWH solver in either format, FWH surface data must first be generated during a zCFD simulation (see `output_settings` (p. 84) for more details). The FWH surface data is stored in a HDF5 format. While no mesh conversion tools are provided, the zCFD FWH file format can easily be interrogated and replicated with tools like `h5ls` and `h5py` should you wish to convert data from other simulation tools to this format.

Using the zCFD FWH python module

The zCFD FWH solver consists of a series of python modules, which are accessible from within the zCFD command-line environment (p. 212). Three python dictionaries, controlling the surface, observer, and solver settings, are required as inputs to the `fwh.solve` function. This function then returns time histories of the pressures at the specified observer points, in the nested dictionaries `pOut` and `tOut`.

```
from solvers import fwh, motion

mySurfaces = { surfaces }
myObservers = { observers }
mySolverSettings = { solverSettings }

pOut,tOut = fwh.solve(mySurfaces, myObservers, solverSettings)
```

For an aerofoil in a wind tunnel (as in Figure 1 and Figure 2), an appropriate python script (including a small amount of post-processing) might look like this:

```
from solvers import fwh, motion
from matplotlib import pyplot as plt
import json

v = [-50,0,0]
surfaceCentreMotion = motion.ConstantVelocityPoint([0,0,0],v)
surfaceMotion = motion.NonrotatingSurface(surfaceCentreMotion)
obs1Motion = motion.ConstantVelocityPoint([1,0,0],v)
obs2Motion = motion.ConstantVelocityPoint([2,0,0],v)

mySurfaces = {
    "fwhSurf1": {
        "motion": surfaceMotion,
        "fileName": "./zcfdSim_OUTPUT/aerofoil/ACOUSTIC_DATA/fwhsurf1_FWHData.h5"
    }
}
myObservers = {"Obs1": obs1Motion, "Ob2": obs2Motion}
mySolverSettings = { "c": 340, "rho0": 1.2, "p0": 1e5}

pOut,tOut = fwh.solve(mySurfaces,myObservers,solverSettings)

fig,ax = plt.subplots()
ax.plot(tOut["fwhSurf1"]["Obs1"],pOut["fwhSurf1"]["Obs1"],label="Obs1")
ax.plot(tOut["fwhSurf1"]["Obs2"],pOut["fwhSurf1"]["Obs2"],lable="Obs2")
ax.set_xlabel("t (s)")
ax.set_ylabel("p (Pa)")
fig.savefig("./FWH_figure.pdf")

dataForJson = {"p": pOut, "t": tOut}
with open("./FWH_data.json","w") as f:
    json.dump(dataForJson,f)
```

When there is more than one surface, the `pOut` and `tOut` dictionaries will contain a `Surface Summation` entry in addition to the entries for each surface. This contains, for each observer, a sum of the

contributions from each surface. This is useful when noise originates from several distinct regions, each covered by separate FWH surfaces (e.g. UAVs with several propellors). The `Surface Summation` entry will likely cover a shorter time period than the individual surface entries, because it only covers the time period when the observer is receiving valid noise signals from all of the FWH surfaces.

Surfaces dictionary

The zCFD FWH solver is capable of using multiple FWH surfaces in the same FWH calculation. The Surfaces dictionary therefore contains a key-value pair for each FWH surface to be used, where the key is the surface's name (to be used in the FWH solver) and the value is an individual surface definition dictionary. The surface name must be a string, and cannot be 'Surface Summation'.

Example usage:

```
from solvers import motion
v = [-50,0,0]
surfaceCentreMotion = motion.ConstantVelocityPoint([0,0,0],v)
surfaceMotion = motion.NonrotatingSurface(surfaceCentreMotion)

mySurfaces = {
    "fwhSurf1": {
        "motion": surfaceMotion,
        "fileName": "./zcfdSim_OUTPUT/aerofoil/ACOUSTIC_DATA/fwhsurf1_FWHData.h5"
    }
}
```

Individual surface definition dictionary

KEYWORD	REQUIRE	DEFAULT	VALID VALUES	DESCRIPTION
'motion'	Yes		A surface motion class	The motion of the surface through the FWH medium
'fileName'	Yes		A readable file path	The FWH surface data, in the FWH HDF5 format outputted by the zCFD solver
'permeable'	No	Automatically deduced from the FWH surface data file	True I False	Whether or not the surface is permeable or not (see above).
'flipSurfaceNormals'	No	False	True I False	Whether or not to invert the FWH surface normals before running the FWH solver. The surface normals should always point outwards towards the farfield. For FWH surface data generated using the fwh interpolate (p. 87) keyword in the zCFD solver, the surface normal direction depends on the .stl file used to define the FWH surface, and can be checked using a visualisation tool (e.g. ParaView).

KEYWORD	REQUIRE	DEFAULT	VALID VALUES	DESCRIPTION
'transformSurfaceVelocity'	No	True	True I False	Relevant for permeable surfaces only, and should normally be kept `True`. Setting this to `False` means that the FWH surface fluid velocity used in the FWH solver does not include the velocity resulting from the motion of the surface (as defined my the 'motion' keyword above). This option may be required when using data from other solvers, in the case where the FWH surface moves through a quiescent medium during the simulation used to create the FWH surface data.

Observers dictionary

The Observers dictionary defines a list of observer microphones at which farfield noise should be calculated. In the observers dictionary, keys are used to set the observer's name in the FWH solver output, and the value sets the motion of the observer. Observer motions must be defined using the point motion classes.

Example usage:

```
from solvers import motion
v = [-50,0,0]
obs1Motion = motion.ConstantVelocityPoint([1,0,0],v)
obs2Motion = motion.ConstantVelocityPoint([2,0,0],v)

myObservers = {"Obs1": obs1Motion, "Ob2": obs2Motion}
```

Solver settings dictionary

The solver settings dictionary sets the properties of the FWH medium, and the numerical settings used in the FWH solver.

Example usage:

```
mySolverSettings = { "c": 340, "rho0": 1.2, "p0": 1e5}
```

KEYWORD	REQUIRE	DEFAULT	VALID VALUES	DESCRIPTION
'c'	Yes		Positive number	speed of sound in the far-field medium, m/s
'rho0'	Yes		Positive number	density of the far-field medium, kg/s

KEYWORD	REQUIRE	DEFAULT	VALID VALUES	DESCRIPTION
'p0'	Yes		Positive number	static pressure of the far-field medium, Pa
'dt'	No	set equal to the smallest timestep between outputs in all the FWH surface data files specified in the surfaces dictionary	Positive number	timestep between observer microphone recording times, s
'breakTolAbs'	No	'dt' set in the solver settings dictionary / 1000	Positive number	the time accuracy (s) required of the prediction of the retarded time τ_* in the retarded time calculation.
'maxIts'	No	50	Positive integer	The maximum number of iterations to use in the algorithm used to predict retarded time
'observerStartTime'	No		Number	The simulation time (s) at which observers start recording data. If this is not set, this will be set automatically to be the earliest time when a surface/observer pair has a valid signal.
'observerEndTime'	No		Number	The simulation time (s) at which observers end recording data. If this is not set, this will be set automatically to be the last time when a surface/observer pair has a valid signal.

Point (observer) motion classes

FWH Observers are points which move through the FWH quiescent medium according to the function $x = f(t)$. In the zCFD FWH solver, the motion of observers is defined using one of the following classes, which are available inside the `solvers.motion` module inside the zCFD command line environment (see above).

MOTION CLASS	INPUTS	DESCRIPTION
`OriginPoint()`		A point which is always on the origin, i.e. $f(t) = [0, 0, 0]$.
`StationaryPoint(X0)`	X0: a 3-d vector $[x0, y0, z0]$	A point which is stationary, i.e. $f(t) = [x0, y0, z0]$.
`ConstantVelocityPoint(X0,dXdt)`	X0: a 3-d vector $[x0, y0, z0]$; dXdt: a 3-d vector $[dxdt, dydt, dzdt]$	A point with constant velocity, i.e. $f(t) = [x0 + t * dxdt, y0 + t * dydt, z0 + t * dzdt]$

Surface motion classes

Just like FWH Observers, FWH Surfaces require their motion to be defined. There are two surface motion classes - NonrotatingSurface and RotatingSurface. Both use a point motion class to define the

motion of their 'centre point'. In this way, complex motions like that of a propellor which is both rotating and translating through the FWH medium can be defined.

MOTION CLASS	INPUTS	DESCRIPTION
`NonrotatingSurface(centrePoint)`	centrePoint: a point motion class	A surface which does not rotate, and translates at the same velocity as the 'centrePoint'.
`RotatingSurface(centrePoint,axis,omega)`	centrePoint: a point motion class; axis: a 3-d vector $[ax_x, ax_y, ax_z]$; omega: a number.	A surface which translates at the same velocity as the `centrePoint`, and also rotates about the centrePoint with rotation axis `axis` and rotation angular velocity (rad / s) `omega`

Using the zCFD run_fwh utility

When using the fwh python module defined above, the FWH surface, observer and solver settings dictionaries used in `fwh.solve()` are defined by hand. This gives fine-grained control, but for most FWH simulations which are based on zCFD CFD simulations, the input dictionaries to `fwh.solve()` can actually be generated automatically by reading the zCFD input parameters file (p. 189). This is what the `run_fwh` utility does. Within the zCFD command-line environment (p. 212), the `run_fwh` utility can be called as below:

```
run_fwh -c <zcfd control file> --reference_condition <reference condition> --observer_locations <observe
```

The `run_fwh` utility reads the zcfd control file and the observers .csv file, and based on these, automatically runs the FWH solver. Assuming the `zcfd control file` input was `zcfdSim.py` and the `observers .csv file` specified the observers `Obs1` and `Obs2`, the `run_fwh` utility would:

1. automatically generate the inputs to the `fwh.solve()` function, setting the observer and surface motions based on the reference condition and using all FWH surfaces that the CFD simulation defined by `zcfdSim.py` outputs
2. Run the `fwh.solve()` function, saving the screen output to `zcfdSim.fwh.log`
3. Save the sum of the contributions of all FWH surfaces at the observers `Obs1` and `Obs2` to the `.csv` files `zcfdSim.fwh.Obs1.csv` and `zcfdSim.fwh.Obs2.csv` respectively.

For the example outlined in the fwh python module example, the `run_fwh` command would look something like this:

```
run_fwh -c zcfdSim.py --reference_condition IC_1 --observer_locations observers.csv
```

zCFD control file

The first input to the `run_fwh` utility is the `zCFD control file`. This is the control file for the zCFD simulation which generated the FWH surface data files which are to be used. For the example outlined in Figure 1 and the fwh python module example, the `zCFD control file` would be called `zcfdSim.py` and would look something like the example below.

```
parameters = {
    "config_version": 2,
    "solver": {
        ...
        "reference_conditions": {
            "IC_1": {
                "temperature": 288.0,
                "pressure": 1e5,
                "v": {"vector": [50.0, 0.0, 0.0]},
                ...
            },
        },
        "output_settings": {
            "solution": {
                ...
                "fwh_interpolate": ["fwhsurf1.stl"],
            },
        },
    },
    "model": {
        "aerofoil": {...},
    },
}
```

The `run_fwh` utility reads the zCFD parameter file and finds the paths to all the FWH surface output files which were generated by the zCFD CFD simulation defined by `zcfdSim.py`. In this case, it would find the file `./zcfdSim_OUTPUT/aerofoil/ACOUSTIC_DATA/fwhsurf1_FWHData.h5`.

Reference condition

In the fwh python module example, the solver settings and the motions of the observers and surfaces through the FWH quiescent medium are defined by hand. However, if all the surfaces and observers are in the same *translational reference frame*, the motions and solver settings can be set directly from an entry in zCFD's `reference_conditions` (p. 39).

Being in the same *translational reference frame* means that all the surfaces and observers translate through the FWH quiescent medium at the same velocity. In practice, this means that all observers are *static* in the 'mesh' frame of reference, i.e. do not 'fly by'. Figure 1 and Figure 2 show that for the example, `Obs1`, `Obs2` and the FWH surface are in the same translational reference frame, hence can be used in the `run_fwh` utility.

In the example case, the reference frame would be set to `IC_1`. This is the freestream conditions in the zcfdSim.py CFD simulation. The pressure and temperature in `IC_1` set the `c`, `p0` and `rho0` in the solverSettings dictionary. The velocity in `IC_1` sets the motion of the surfaces and observers through the FWH quiescent medium. Since `IC_1` defines a freestream velocity of `[50,0,0]`, the surface and observers move through the FWH quiescent medium at a velocity of `[-50,0,0]`, just as in Figure 1 and Figure 2.

In most cases, a zCFD control file already contains a reference condition suitable for `run_fwh`. A control file can hold reference conditions that nothing else uses, so add a suitable one if none exists.

Which *translational reference frame* an FWH surface is in is independent of whether the FWH surface is rotating or not - see below for more details.

Observer .csv file

In the Observers .csv file, the locations of the observers are defined (in the 'mesh fixed' frame of reference). To match the observers shown in Figure 1 and Figure 2, the `observers .csv file` should contain the following:

```
Obs1, 1.0,  0.0,  0.0
Obs2, 2.0,  0.0,  0.0
```

Advanced use

The `run_fwh` utility accounts for overset (p. 328) zCFD simulations and for FWH surfaces that rotate with a model's rotating `motion`, whether the mesh moves or is solved in a rotating frame. Simulations such as a drone with several rotating propellers can therefore be processed with the `run_fwh` utility.

Where the input zCFD control file is an overset (p. 328) 'override' file, FWH surfaces from all control files specified in the override dictionary are used and the reference condition (e.g. `IC_1`) is taken from the first control file specified in the override dictionary. Where FWH surfaces lie on a rotating mesh, they remain in the same translational reference frame as the observers, but also rotate with the `axis`, `origin` and `omega` of the model's `motion`.

A frame, a rotating `motion` with `"moving mesh": False`, covers the whole mesh, so it rotates every FWH surface on that mesh. See Placing and moving a mesh (p. 365).

The `run_fwh` utility does not allow a translating `motion`, whether the mesh moves or not, or translational boundary conditions, since both would introduce ambiguity into what the correct translational reference frame for the FWH surfaces and observers should be. As explained above, 'fly-by' observers are also not possible in the `run_fwh` utility.

In the fwh python module, `fwh.solve()` outputs the contribution from each FWH surface to each observer pressure history, as well as the sum of all contributions. The `run_fwh` utility only records the sum of contributions from all FWH surfaces in the output `.csv` files. This means that in the `run_fwh` utility, every noise source in the zCFD simulation is assumed to be enclosed by a single FWH surface. This means that, for instance, including both `fwh_interpolate` and `fwh_wall_data` FWH surface data output in a zCFD control file is likely to lead to double accounting in the `run_fwh` utility.

Where the `run_fwh` utility is not suitable, it is still possible to use components of the `run_fwh` utility to automate parts of a fwh python module workflow. A simplified summary of what happens within the `run_fwh` executable is detailed below, showing the use of the `fwh_helper_functions` which can be used in the `fwh python module` to automate workflows.

```
from solvers import fwh_helper_functions
rho0, p0, c, freestreamV = fwh_helper_functions.get_zcfd_ref_conditions(
    "zcfdSim.py", "IC_1"
)
surfaces = fwh_helper_functions.get_zcfd_surfaces(
    "zcfdSim.py", freestreamV
```

```
)
observers = fwh_helper_functions.get_csv_observers(
    "observers.csv", freestreamV
)
solverSettings = {"rho": rho0, "p0": p0, "c": c}
pOut, tOut = fwh.solve(surfaces, observers, solverSettings)
fwh_helper_functions.write_fwh_data_to_zcfd_csv("zcfdSim.py", tOut, pOut)
```

FWH file utilities

As well as the FWH solver itself, two FWH file utilities are provided - `writeFWHSurfaceFrequenciesFile` and `writeBlankedOffFWHSurfaceFile`. Users may find these utilities useful for investigating the sources of noise within their CFD simulation.

writeFWHSurfaceFrequenciesFile

This python function takes an FWH surface file as input, and creates a file where the FWH surface data can be viewed in the frequency, instead of time, domain.

This can be useful for visually indicating where particular frequencies originate from in an FWH observer pressure signal. To generate the FWH surface frequencies file, an FFT (using a Hann window) of the FWH data is calculated at each surface face. The data stored in the surface frequencies file at a particular surface face and chosen frequency is then the magnitude of that FFT's amplitude, at the chosen frequency. The data can be viewed by opening the `.xdmf` file corresponding to the output file in ParaView - the 'time' chosen in ParaView corresponds to the frequency being viewed. The data at frequency = 0 corresponds to the time average of the FWH surface data.

Example usage:

```
from solvers import fileManipulation
fileManipulation.writeFWHSurfaceFrequenciesFile("./myFWHSurface.h5","./myFWHSurface_
↪  Frequencies.h5")
```

writeBlankedOffFWHSurfaceFile

This python function writes a copy of an FWH surface file where specified faces are ignored by the FWH solver.

Sometimes it is useful to 'blank off' certain areas from an FWH surface output file before the FWH solver is run - for example, if CFD setup constraints mean the noise from that region of the FWH surface is non-physical and should therefore be ignored when calculating FWH noise data. This function takes an input FWH surface file, copies it to an output FWH surface file, then sets to zero the face area of any face in the output file where the python function shouldBlankOff(x,y,z) returns 'True' for the face's (x,y,z) co-ordinates. When the face area of a face has been artificially set to 0, that face will not contribute to the observer location pressures calculated by the FWH solver. The region which has been set to have 0 face area can be inspected visually by opening the `.xdmf` corresponding to the output file in ParaView

Example usage:

```
from solvers import fileManipulation

def isPositiveX(x: float, y: float, z: float):
    if x > 0:
        return True
    else:
        return False

fileManipulation.writeBlankedOffFWHSurfaceFile("./myFWHSurface.h5","./myFWHSurface_
↪  IgnorePositiveX.h5", isPositiveX)
```

Fluid–structure interaction

zCFD supports two fluid-structure interaction (FSI) coupling schemes: a built-in **modal model** driven by mode shapes imported from a Nastran modal analysis, and a **generic** loosely-coupled scheme for driving an external FEA solver through a user-supplied `genericFSI.py` script. Both schemes use the same underlying mesh-deformation machinery (RBF, RBF multiscale, or IDW - see Mesh deformation) to propagate the motion of the coupled surface into the volume mesh.

Where FSI settings live

An FSI coupling is attached to the wall boundary condition whose surface is the moving, coupled structural boundary, via that boundary condition's `'fsi'` (p. 99) key:

```
{
    "model": {
        "<case_name>": {
            "boundary_conditions": {
                "BC_FSI": {
                    "zones": [1, 2, 3],
                    "type": "wall",
                    "kind": "slip",
                    "fsi": {
                        "type": "modal_model",  # or "generic_fsi"
                        ...
                    },
                },
                ...
            },
        }
    },
}
```

The `'fsi'` block is discriminated by its `'type'` key: `'modal_model'` selects the built-in Nastran modal-model coupling, `'generic_fsi'` selects the generic external-solver coupling. For the modal model, the moving zones are **not** repeated inside the `'fsi'` block - they are inherited from the owning boundary condition's own `'zones'` list. Generic FSI is the exception: it carries its own `'zone'` key (singular, unlike every other zone list in zCFD, which is why it stands out below).

Restarting an FSI simulation

FSI restart behaviour is controlled by `'restart'` (p. 48) in the `'initialisation'` block - the same flag that governs restarting the flow solution. The FSI coupling resolves this flag through the same model-block-then-solver-block precedence used everywhere else in zCFD: if the model that owns the FSI-coupled boundary condition sets its own `'initialisation'` block, the `'restart'` value there takes

priority; only when the model block does not set `'initialisation'` does the coupling fall back to the top-level solver block:

```
{
  "solver": { "initialisation": { "restart": true } },
  "model": { "<case_name>": { "initialisation": { "restart": true } } }
}
```

This determines whether the modal model resumes its structural state (modal displacements and velocities, and, for the RBF transform, the previously built base-point set) from a previous run, so a per-model `'initialisation'` block, when present, is always the one that decides FSI restart - not the solver-level one.

Modal model

The modal model reduces the structural response to a small set of Nastran mode shapes, integrates the modal equations of motion (via a Newmark-beta scheme) forced by the aerodynamic pressure load, and maps the resulting modal deformation back onto the fluid surface and, via RBF or IDW, into the volume mesh.

> **NOTE**
> Unlike generic FSI, the modal model does not accept a `'user_variables'` dictionary - modal-model behaviour is controlled entirely by its own keywords.

By default the modes are marched and the mesh deformed after every solver cycle. `'update_frequency'` (p. 114) spaces the structural updates out to once every that many cycles; when several walls carry a modal model, the smallest value applies to all of them.

RBF multiscale transform

Set `'transform_type': 'rbf_multiscale'` to deform the volume mesh using the multiscale RBF scheme (see Interpolation method for how RBF and RBF multiscale work).

IDW transform

Set `'transform_type': 'idw'` to deform the volume mesh using inverse distance weighting (see Interpolation method for the IDW weighting formulae).

> **NOTE**
> `'support_radius'` and `'stencil_size'` are alternative ways of bounding the IDW stencil - setting both explicitly is rejected as ambiguous.

Harmonic initial state

An optional `'harmonic_initial_state'` (p. 116) block starts every mode on its steady harmonic response to a point force, instead of at rest. The magnitude of `force` and the `frequency` set each mode's initial displacement and velocity at the start of a run that is not a restart; no force is applied while the run marches. The force at `location` is written to the modal history.

Modal model example

RBF multiscale coupling of a panel to a Nastran modal basis:

```
{
    "boundary_conditions": {
        "BC_FSI": {
            "zones": [1, 2, 3],
            "type": "wall",
            "kind": "slip",
            "fsi": {
                "type": "modal_model",
                "file_format": "nastran",
                "nastran_casename": "panel_nm_0.04m",
                "transform_type": "rbf_multiscale",
                "base_point_fraction": 1.0,
                "support_radius": 0.02,
                "scale": [1.0, 1.0, 1.0],
                "mode_mapping_max_distance": 0.01,
                "fluid_force_scaling": 0.08 / 0.000166667,
                "mode_list": [0, 4, 10, 13],
                "modal_damping": [0.0, 0.0, 0.0, 0.0],
            },
        },
    },
}
```

The same panel coupled with an IDW transform instead, with a set of surrounding zones held fixed:

```
{
    "boundary_conditions": {
        "BC_FSI": {
            "zones": [1, 2, 3],
            "type": "wall",
            "kind": "slip",
            "fsi": {
                "type": "modal_model",
                "file_format": "nastran",
                "nastran_casename": "panel_nm_0.04m",
                "transform_type": "idw",
                "power": [3.0, 8.0, 10.0],
                "deformation_distance": 0.05,
                "blending_stiffness": 0.0,
                "n_nearest": 350,
                "fixed_zones": [4, 5, 6, 7, 8, 9],
                "scale": [1.0, 1.0, 1.0],
                "mode_mapping_max_distance": 0.01,
                "fluid_force_scaling": 0.08 / 0.000166667,
                "mode_list": [0, 4, 10, 13],
                "modal_damping": [0.0, 0.0, 0.0, 0.0],
            },
        },
    },
```

```
}
```

Generic FSI

Generic FSI is a loose (explicit) coupling to an external FEA program. By default, `genericFSI.py` performs no changes - it is a hook to be implemented to exchange pressures, displacements and node coordinates with your own external solver.

Generic FSI example

```
{
    "boundary_conditions": {
        "BC_FSI": {
            "zones": [1, 2, 3],
            "type": "wall",
            "kind": "slip",
            "fsi": {
                "type": "generic_fsi",
                "zone": [1, 2, 3],
                "fixed_zones": [4, 5, 6],
                "transform_type": "idw",
                "n_nearest": 100,
                "deformation_distance": 1.0,
                "blending_stiffness": 0.5,
                "power": [3.0, 5.0, 10.0],
                "user_variables": {"restart_file": "structure_state.dat"},
            },
        },
    },
}
```

Mesh deformation

A known deformation of a surface in the mesh is propagated out into the volume using either radial basis function or inverse distance weighted interpolation. Wing deflection and turbine-blade deflection under a static load are two uses of this functionality. Set up as an entry in the model's `fluid_zones` (p. 143) with `'type': 'rbf transform'` (p. 166) or `'type': 'idw transform'` (p. 164).

Two zone types apply a geometric transform to a subset of the mesh rather than a physical source term: the RBF and IDW mesh deformation schemes, typically used to propagate a moving boundary's displacement into the surrounding volume mesh. Both require `zones` and a `func` (p. 221) that returns the deformed position of each surface node; the solver takes the displacement as the difference from the original position.

Interpolation method

Selects the interpolation method used to propagate the surface deformation into the volume mesh. Two `type` values are available:

'rbf transform' with `base_point_fraction` (p. 168) set to `1.0` uses the 'greedy' method of Allan et al, where the deformation is started with a subset of points and additional points are added until the surface deformation converges.

'rbf transform' with `base_point_fraction` set below `1.0` instead uses the multiscale RBF method, where a subset of surface points with a fixed support radius are solved implicitly and the remaining points are solved explicitly with a support radius determined as part of the solution process. The base set of points are assigned the `support_radius` (p. 167) and hence can have a wider sphere of influence than the remaining points which are used in the multiscale process.

> **NOTE**
> Increasing the number of points in the base set and/or increasing `support_radius` generally makes the surface deformation propagate further into the volume mesh, reducing the likelihood of generating poor quality cells.

'idw transform' uses a simple inverse distance weighted interpolation. The **power** (p. 165) parameter provides a list of three exponents used in the IDW scheme. The first (p1) is applied to the interpolation weight of points close to the surface, the second (p2) to points which are the deformation distance away from the surface and the third (p3) is used to blend between the two extremes.

$$wt_1 = (deformation_distance/distance)^{p1}$$

$$wt_2 = (deformation_distance/distance)^{p2}$$

$$blend = ((deformation_distance - distance)/deformation_distance)^{p3}$$

Then the weight is:

$$wt = (wt1 * (1.0 - blend) + blend * wt2)$$

Examples

RBF Example

```
import zutil

alpha = 10.0

def my_transform(*, location):
    return zutil.rotate_vector(location, alpha, 0.0)


{
    "fluid_zones": {
        "TR_1": {
            "type": "rbf transform",
            "zones": [4],
```

```
            "func": my_transform,
            "support_radius": 200.0,
            "basis": "c2",
        }
    }
}
```

RBF Multiscale Example

```
import zutil

alpha = 10.0

def my_transform(*, location):
    return zutil.rotate_vector(location, alpha, 0.0)


{
    "fluid_zones": {
        "TR_1": {
            "type": "rbf transform",
            "zones": [4],
            "func": my_transform,
            "support_radius": 200.0,
            "basis": "c2",
            "base_point_fraction": 0.1,
        }
    }
}
```

IDW Example

```
import zutil

alpha = 10.0

def my_transform(*, location):
    return zutil.rotate_vector(location, alpha, 0.0)


{
    "fluid_zones": {
        "TR_1": {
            "type": "idw transform",
            "zones": [4],
            "func": my_transform,
            "fixed_zones": [1],
            "n_nearest": 500,
            "power": [4.0, 8.0, 10.0],
            "deformation_distance": 100.0,
            "blending_stiffness": 0.5,
```

```
        }
    }
}
```

Monitoring a run

zMon is a run monitor that opens in a web browser. It plots the residuals, forces, moments and monitor points of a running case as the solver writes them, alongside the solver log, and can ask the solver to stop cleanly.

Starting zMon

zMon is installed with zCFD. Activate the zCFD environment, change to the case directory and name the control file:

```
source <ZCFD_INSTALL_PATH>/bin/activate
cd ~/zcfd_tutorial_1/
zmon naca0012.py
```

zMon prints the address it is serving, such as `zMon is at http://localhost:40213`, and opens that address in the default browser. Where no browser can be opened, it prints the address for you to open by hand. Run `zmon` without a file to start with the file browser instead. zMon keeps running until you press *Ctrl + C* in its terminal; closing the browser tab leaves it running.

To use zMon outside a zCFD installation, install it from PyPI:

```
pip install zmon
```

The command accepts these options:

OPTION	EFFECT
`--port`	The port to listen on. Without it, zMon uses the port in the `ZMON_PORT` environment variable, or else an unused port chosen by the operating system.
`--host`	The address to bind, by default `127.0.0.1`. Any other address, such as `0.0.0.0`, exposes the file browser to anyone who can reach it over the network.
`--no-browser`	Print the address without opening a browser.
`--exit-when-idle`	Shut down once no browser is connected. zMon waits 30 seconds for the first browser, and stops shortly after the last one disconnects.
`--version`	Print the installed version and exit.

Opening a run

zMon opens a run from either of two files:

- **The control file.** zMon reads the deck to find every file the solver writes for it: each model's report file, the run log and the real time step. This is the usual way to open a run, and the only one from which the run can be stopped.
- **A report file.** Opening a `<model>_report.csv` directly shows that model's plots and nothing else. There is no deck to read, so the time step is unknown and the plots are drawn against cycles and time steps rather than simulated seconds.

A control file can be opened before the solver has started. zMon predicts the report file of each entry under the deck's `model` (p. 8) key, and the plots fill in once the solver begins writing them.

The file browser opens when zMon is started without a file, and from the *Open File* button on the toolbar. It lists one directory at a time, with a filter box, pinned folders and recently opened files and folders. Two kinds of file carry a badge: `report` marks a report file, and `control` marks a control file with a mesh of the same name beside it. Several files can be selected and opened together.

Each opened run appears as a tab across the top of the page, so several runs can be watched at once. Closing a tab stops zMon watching that run; it does not affect the solver.

Reading the dashboard

Every panel reads the model's report file, which the solver appends to every `frequency` (p. 90) cycles. zMon checks the file each second and adds new rows to the plots as they arrive, so the dashboard updates at the reporting interval. The columns themselves are described in The report file (p. 342).

Drag and Lift. Two summary cards show the latest value of a force coefficient with a short history beneath it. Each card has a menu listing every `forces` (p. 90) block in the report, split into total, pressure, friction and transformed components. The X, Y and Z buttons choose the direction, which is X for drag and Z for lift until changed.

Residuals. The plot has one line per equation, drawn from the report's residual columns on a logarithmic axis. The momentum components `rhoV[0]`, `rhoV[1]` and `rhoV[2]` are labelled `rhoU`, `rhoV` and `rhoW`. When the report carries `residual_statistics` (p. 91) columns, a row of buttons above the plot switches between the residual itself and the `Rinf`, `Rrms`, `Rtop` and `Neff` families; hovering over a button describes the measure. `Neff` is a count of cells and is drawn on a linear axis. The measures are explained in Residual statistics columns (p. 344).

Monitor points. This panel lists every other named column in the report. One or more can be plotted together. It holds the `monitor` (p. 90) columns and the `mass_flow` (p. 90) columns, named as described in Monitor points (p. 345) and Mass flow (p. 348).

Force and moment panels. Six plots, one for each force and moment direction, each with a menu of the matching columns. Choosing a column in any one panel moves the other five to the same forces block and component. The column names follow Output names (p. 346). Zooming along the horizontal axis of one force or moment plot applies the same range to the others; a double-click resets the view.

Any panel can be closed with the cross at its corner and restored from the panel menu on the toolbar. The browser remembers which panels were open.

Cycles, time steps and spectra

A strip of buttons at the edge of each plot chooses the horizontal axis:

- **Cycle** plots every report row against the solver cycle.
- **Time step** plots against the real time step of an unsteady run. When zMon has read the deck and `type` (p. 51) is unsteady, the axis is the simulated time in seconds, from `time_step` (p. 51).

In either mode a second pair of buttons chooses between every inner cycle, drawn as a sawtooth, and one converged value per real time step: the last inner cycle for forces and monitor points, and the smallest residual for the residuals. Once the report shows an unsteady run, zMon switches the panels to the time-step axis, and all but the residuals to converged values.

In time-step mode, the monitor-point, force and moment panels and the two summary cards offer a frequency view: the amplitude spectrum of the converged values. A plot's spectrum covers the range visible in its time view, so zooming in on the time view first excludes the start-up transient. The frequency is in hertz when the time step is known, and in cycles per time step otherwise.

Overset runs

An overset run opened from its control file appears as one tab with a sub-tab for each model, named by its key under `model` (p. 8), and an *Overview* sub-tab. Each model's sub-tab is the full dashboard for that model's report file. A dot beside each model's name shows whether its report has been written in the last 30 seconds (green), not recently (amber), or not at all (red).

The Overview shows two things across all the models, each model in a colour of its own:

- **Combined residuals.** The density residual of every model on one plot, or the first residual column where there is no density equation. The buttons above the plot show or hide each model.
- **Forces and moments summary.** A table of each model's latest transformed total force and moment, with their sum across all the models, and thrust and torque where a model reports them, on a rotating mesh or in a rotating frame. It reads the transformed columns only, so it is populated only for forces blocks that set a `transform` (p. 94).

The layout of an overset deck is described in Overset meshes (p. 328).

Log and cycle time

Below the plots, the *Log Output* panel shows the last 500 lines of the run log, `<case>.log` in the case directory, with the solver's colours preserved. It refreshes every two seconds. In an overset run every sub-tab shows the same log, because the run writes one.

The *Cycle Time* card shows the latest `AvgCycleTime` value from the report, the mean wall-clock time of a solver cycle over the run so far, with its history beneath it. Its meaning, and why it is not comparable between runs, is covered in Cycle time column (p. 344).

Stopping a run

The *Stop* button on the toolbar asks the solver to stop cleanly. After confirmation, zMon writes an empty file named `<case>.stop` into the case directory, where `<case>` is the control file's name without

`.py`. One file stops the whole run, including every model of an overset run. Creating the file by hand has the same effect:

```
touch naca0012.stop
```

The solver looks for the file each time it writes a report row. A steady run writes its final output and stops at the end of that cycle. A dual time-stepping run completes the current real time step first. The solver deletes the file when it acts on it, and removes any left over from an earlier run when it starts.

> **NOTE**
> The stop request reaches the solver only when the run was opened from its control file. A report file opened on its own lies inside the output directory, and a stop file written beside it is never read.

What zMon needs to read a run

To open a control file, zMon executes it in a separate Python interpreter and reads the run's layout with the same `zutil` routines that zCFD's own post-processing tools use. Three things follow:

- **A zCFD environment.** The interpreter running zMon must be able to import `zutil`. It can do so inside an activated zCFD environment, but not in an installation made with `pip install zmon` alone. There, opening a control file fails with a message to run zMon in a zCFD environment. A report file opens in either.
- **A single deck for the run.** The control file must declare `"config_version": 2`, with every mesh as an entry under `model` (p. 8). A driver file that lists mesh and control-file pairs in an `override` dictionary is refused, with a message describing the single-deck form.
- **The solver's own layout.** zMon expects the files where the solver writes them: `<case>.log` beside the control file, and each model's report at `<case>_OUTPUT/<model>/<model>_report.csv`. A run whose output has been moved or renamed cannot be resolved from its control file, though its report files can still be opened directly.

A control file that takes longer than 20 seconds to execute is abandoned with a timeout message.

Overset meshes

zCFD is capable of automatically oversetting multiple meshes into a single simulation. An overset case is defined entirely within one control file: every mesh in the simulation is declared as its own named entry directly under the control file's top-level `model` key, each carrying its own `mesh` path, and each mesh that receives donor data from another mesh must declare an overset interface on its hole-cutting surface (see Overset interface below).

Setting up an overset case

One model entry is the background mesh, typically the largest of the meshes that the others overlap with. Each further entry is a mesh that oversets onto another entry (the background mesh, or another overset mesh), declared through an overset interface whose `connection` names the donor model(s) (see below).

Example `model` structure (one background mesh, a refinement region overset onto it, and two further meshes overset onto the refinement region):

```
"model": {
    "background": {"mesh": "background.h5", ...},
    "refinement_region": {"mesh": "refinement_region.h5", ...},
    "cylinder_1": {"mesh": "cylinder.h5", ...},
    "cylinder_2": {"mesh": "cylinder.h5", ...},
}
```

Run the case with a plain:

```
(zCFD)> run_zcfd -c overset_cylinders.py
```

All meshes are read from the `mesh` key of each entry under `model`; no separate mesh/case file pairing and no command-line mesh argument is needed.

Overset interface

Overset mapping options are set directly on the boundary condition that marks a mesh's overset (hole-cutting) surface, using `'type': 'interface'` with `'kind': 'overset'` inside that model's `boundary_conditions` block — see the `interface` (p. 134) boundary condition reference and Interface (p. 288).

Example usage, adapted from a real multi-mesh overset testcase (one background mesh, a refinement region overset onto it, and two further meshes overset onto the refinement region):

```
parameters = {
    ...
    "model": {
        "background": {
            "mesh": "background.h5",
            "boundary_conditions": {
                "BC_1": {"zones": [9, 11], "type": "symmetry"},
                "BC_2": {"zones": [10], "type": "farfield", "condition": "IC_1", "kind":
                ↪  "riemann"},
            },
        },
        "refinement_region": {
            "mesh": "refinement_region.h5",
            "boundary_conditions": {
                "BC_1": {"zones": [9, 11], "type": "symmetry"},
                "BC_2": {
                    "zones": [10],
                    "type": "interface",
                    "kind": "overset",
                    "connection": {"model": ["background"]},
                },
            },
        },
        "cylinder_1": {
            "mesh": "cylinder.h5",
            "boundary_conditions": {
                "BC_1": {"zones": [11, 12], "type": "symmetry"},
                "BC_2": {
                    "zones": [13],
                    "type": "interface",
                    "kind": "overset",
                    "connection": {"model": ["refinement_region"]},
                },
                "BC_3": {"zones": [10], "type": "wall", "kind": "no slip"},
            },
        },
    },
    ...
}
```

interpolation_method

interpolation_method (p. 136) selects how the donor value is built once the containing cell is known: `nearest cell` takes that cell's value directly, `inverse distance` is a distance-weighted average over the cell and its neighbours, `green-gauss` is a Green-Gauss linear reconstruction about the donor cell centre, and `least-squares` (the default) is a least-squares linear reconstruction about the same point. Each method is described under Interpolation below.

additional_active_layers

Adds additional active cells which may have been made inactive by an overlapping mesh. For example, where a rotating mesh is used, the resolution of the rotating boundary may result in rotated overset mesh boundary points, entering cells made inactive. This will result in no mapping between background

and overset meshes at these points. To resolve this issue, it is possible to add additional active cells (for safety) to ensure a mapping between overset and background meshes. Whilst not necessary, this parameter may be useful if issues with mapping between meshes occur.

moving_mesh_halo_layers

Relevant only when the overset mesh is itself moving, that is when its model's `motion` has `"moving mesh": True`; see Moving a whole mesh. Controls how many halo fringe layers are kept around the re-cut hole each time the mesh moves; a value of 0 treats every blanked cell as a halo, which is safe but conservative, while N > 0 is sufficient when the hole is re-cut every real time step and avoids remapping cells that don't need it.

blank_enclosed_cells

Coverage blanks a background cell by testing whether each of its nodes lies inside an overset cell, which cannot see the cells inside the overset mesh's own solid bodies — a blade, a hub, a fuselage — since the overset mesh has no cells inside its walls. Left active, those cells solve as fluid inside a solid with no donor to correct them, and appear as islands in the middle of the hole in the `overset` output variable.

With **blank_enclosed_cells** (p. 136) on (the default), a flood fill from the active cells outside the overset mesh's bounding box identifies what the hole walls in, blanks it, and keeps it out of the halo set. It only runs for an overset mesh that has a wall boundary: a mesh with no wall encloses nothing, and one carrying an inward-facing overset interface has deliberately left its interior to the background. The count blanked is logged at each hole cut.

connection

The **connection** (p. 134)'s `model` names the model(s) whose mesh supplies donor data at this overset interface, and is used to derive the blanking priority order: models are topologically sorted so that a model always appears after the model(s) it connects to, with root meshes (no overset interfaces) placed first. Naming the connection explicitly is preferred to omitting it, as it avoids building mappers for mesh pairs that do not overlap. The `model` accepts either a single model name or a list, as shown in the example above.

> **NOTE**
>
> There is no control-file option for detailed per-mesh-pair mapper diagnostics. The underlying C++ verbose-mapping flag exists, but nothing in the current control-file schema or launcher wires a value to it, so it cannot be enabled from a control deck.

Moving a whole mesh

An overset mesh that moves during the run carries a `motion` (p. 169) with `"moving mesh": True` in its model's `transforms`. The motion is rigid and moves the whole mesh, and it needs an unsteady compressible run. Its keys, the order in which `scale`, `transform` and `motion` apply, a ramp from rest and worked examples are described in Placing and moving a mesh (p. 365). A steady rotor is the same block with `"moving mesh": False`, which solves the mesh in a rotating frame over every cell and does not move it.

At each real time step the moving mesh is placed where the motion puts it at the end of the step, and

every overset mapping or sliding intersection that involves it is recomputed.

By default a rotating mesh keeps its coordinates and carries the rotation as an angle, which the overset mapper and the sliding intersection apply to what they transfer, and the mesh is solved in its own mesh-attached frame (see Moving meshes). `rotate nodes` (p. 172) set to `True` rotates the mesh nodes every step instead, and solves the mesh in the inertial frame. Both treatments are exact for the interior, since a pure rotation changes no volume and no area. A translating mesh always moves its nodes.

A moving mesh whose motion has a zero `omega` or a zero `velocity` still runs, with a warning from validation: the mesh never moves, but its interfaces are re-coupled every step, which slows the run.

Viewing an overset run

Each model writes its own solution under `<case>_OUTPUT/<model>/`, holding every cell of its mesh, including the cells another mesh covers. Add `overset` to the `volume variables` (p. 85) to tell them apart: the `overset` (p. 173) variable is 1 in an active cell and 0 in a cell covered by another mesh.

To view the domain as one solution in ParaView:

1. Load the volume solution of each model, `<case>_OUTPUT/<model>/<model>.vtkhdf`.
2. Select every model's volume and apply *Filters > Group Datasets*, then *Filters > Merge Blocks*. ParaView reads each .vtkhdf file as one block per partition, which *Append Datasets* cannot take.
3. Apply *Filters > Threshold* on `overset`, with the lower threshold raised from 0 to 1, to keep only the active cells.

To make the first step easier, `generate_paraview_helpers` writes ParaView scripts for the run. Run `paraview_view_volumes.py` once with `pvpython` to save `paraview_view_volumes.pvsm`, which loads every model's volume with *File > Load State...*; see Opening a run in ParaView (p. 337).

Tutorial 5 (p. 442) works through these steps on two cylinders.

How overset works in the solver

Each mesh in an overset case keeps its own solver, its own partitioning and its own boundary conditions. Nothing is merged into a single mesh. The coupling is a set of directed mesh pairs, and everything below happens per pair.

Mesh ordering

The connection on each overset interface defines a directed graph over the models, which is required to be acyclic. The models are topologically sorted so that a model always appears after the model or models it connects to, with root meshes (those with no overset interface) first. That order is used for hole cutting, for the data exchange and for advancing the solvers, so donor data is always one step ahead of the mesh that consumes it.

Each edge of the graph becomes a mapping pair holding two mappers: a *forward* mapper carrying data from the overset mesh into the background mesh, and a *reverse* mapper carrying data from the background mesh into the overset mesh.

Hole cutting

For each pair, every node of the background mesh is tested against the cells of the overset mesh. A background cell is blanked when **all** of its nodes are found inside the overset mesh; a cell with even one uncovered node stays active. Blanked cells take no part in the solution.

This minimal blanking is recorded before anything else modifies it, and is what the `overset` output variable reports. `additional_active_layers` then runs N sweeps over the interior faces, and on each sweep any blanked cell sharing a face with an active cell is promoted back to active. The hole therefore shrinks by N cell layers, which is how an overlap band — a region where both meshes hold a valid solution — is created.

When one mesh is the background of several overset meshes, all of its active flags are reset once, then every pair's hole is cut, and only then is any mapping computed. Cutting and mapping pair by pair would let a later pair's reset erase an earlier pair's hole.

Fringe cells

The cells that receive interpolated data are drawn from the blanked set. By default the fringe is the two layers of blanked cells adjacent to the remaining active region.

When the overset mesh moves, a cell blanked now may be uncovered by the next hole cut, so every blanked cell is treated as a fringe cell. That is safe but means interpolating the whole hole volume. Setting `moving_mesh_halo_layers` to N replaces this with an N-layer fringe, which is sufficient when the hole is re-cut every real time step.

Donor search

Donors are found with a bounding volume hierarchy built over the donor mesh — on the GPU when a device is present, on the host otherwise. The tree is built once per mesh and reused while the mesh coordinates stay where they are.

For a rotating mesh without `rotate_nodes` the mesh coordinates and the tree are **not** moved. The rotation is carried as an angle and applied to the *query points* instead, mapping them into the donor mesh's own frame before the search. This is why such a case costs little more per remap than a static one. A translating mesh, and a rotating mesh with `rotate_nodes`, move their nodes, so the tree is refitted to the moved nodes or rebuilt at every remap.

A donor only counts if it is itself active. A fringe point that lands in a blanked region of the donor mesh finds nothing, and that face falls back to the unmapped face treatment rather than to interpolated data. This is the usual reason to increase `additional_active_layers`.

Interpolation

`interpolation_method` selects the interpolant once the donor cell is known:

- **nearest cell** — the value of the containing cell is taken directly, with unit weight. This is a piecewise-constant transfer, first-order accurate.
- **inverse distance** — the containing cell and its neighbours are combined with weights $(R - d)/(R + 4d)$, where d is the distance from the query point to each donor cell centre and R is 1.2 times the largest of those distances.

- **green-gauss** — a Green-Gauss linear reconstruction about the donor cell centre, evaluated at the query point.
- **least-squares** — a least-squares linear reconstruction about the donor cell centre, evaluated at the query point.

`nearest cell` and `inverse distance` combine cell values, so they return the donor field at a point offset from the query point; `green-gauss` and `least-squares` reconstruct a gradient and evaluate it at the query point itself.

What is exchanged, and when

The exchange runs **once per pseudo-time cycle**, before any solver advances, and moves both the conserved cell data and the cell gradients. Walking the models in topological order, each model first sends the data it owns as a donor, then receives the data it needs as a dependent, and is then synchronised to the device and has its parallel halos updated before the next model is handled.

The exchange is not repeated between the stages of a multi-stage scheme. An explicit Runge-Kutta march would therefore advance the interior several times against interface data that is frozen for the whole cycle, and for that reason an overset case requires one of the implicit pseudo-time schemes, `implicit euler`, `lu-sgs` or `mf-gmres` (see Choosing a time-marching scheme (p. 241)). A control file that asks for an explicit scheme alongside an overset interface is rejected at validation.

Moving meshes

An overset mesh whose model has a rotating `motion` with `"moving mesh": True` (see Moving a whole mesh) is solved in its own mesh-attached frame, unless `rotate nodes` is set; only the mapped data crosses between frames. At the end of each real time step the rotation angle is set to the angle the motion has reached, and the momentum components of the current solution and of both stored time levels are counter-rotated by the angle turned, so that the time history the dual-time scheme differences is expressed consistently in the current frame.

At the end of a real time step, if either mesh of a pair reports that it has moved, the per-mesh record of which faces are mapped is cleared and **all** pairs are re-cut and re-mapped — not only the stale ones, because cutting one donor's holes begins with a full reset of its active flags. Clearing the record matters on a moving mesh: a face mapped at a previous position but now outside donor coverage would otherwise stay marked as found and its fringe cell would be left unfilled.

Blanked cells in the rest of the solver

Blanking is not confined to the mapper:

- the residual is zeroed in blanked cells, so they cannot drive the solution or contribute to the reported residual norms;
- the implicit update is masked by the same flag, so a blanked row is left untouched;
- multigrid agglomeration propagates the flag to the coarse levels, and a coarse cell is marked blanked if **any** of its children is blanked. A coarse cell with a mixture of active and blanked children logs `Parents are mixed active and inactive`. A large count means the coarsening and the hole boundary are poorly aligned.

Practical consequences

- Meshes must genuinely overlap. A hole cut exactly to the overset mesh's own footprint leaves no band in which both meshes hold an independently computed solution, and the fringe then draws its donors from cells that are themselves interpolated. Use `additional_active_layers` to open a real overlap band.
- Keep cell sizes similar across the interface. The transfer is at best a distance-weighted average of donor cell values, so a coarse donor cannot represent a gradient the fine receiver resolves.
- Use an implicit pseudo-time scheme. Multigrid is not available with the implicit schemes, so an overset case runs without it.
- A body inside a hole is not seen by the mesh whose cells are blanked, so a wall must always live in the mesh that resolves it.

Post-processing and visualisation

zCFD by default will output data in VTK HDF format (.vtkhdf). This format can be read directly by many post-processing tools including ParaView by Kitware.

> **NOTE**
> The .vtkhdf output format requires ParaView 6.1.1 or later.

zCFD can also write EnSight output; see EnSight Output.

VTK HDF Output

By default, zCFD will output data in VTK HDF format (.vtkhdf). To force VTK HDF output explicitly, set `format` (p. 86) in the `solution` block of `output_settings`:

```
{ "output_settings": { "solution": { "format": "vtk" } } }
```

EnSight Output

To output in EnSight format, set `format` (p. 86) to `ensight`:

```
{ "output_settings": { "solution": { "format": "ensight" } } }
```

ParaView Catalyst Scripts

ParaView provides a lightweight interface that allows parallel processing of simulation data during program execution called ParaView Catalyst. This is driven by scripts (Python files) that can be called when data is output. The solver reads the list of scripts from `scripts` (p. 86) in the `solution` block. The output `format` (p. 86) must be `vtk`, and the scripts should be in the same directory as the input files:

```
{
  "output_settings": {
    "solution": {
      "format": "vtk",
      "scripts": ["paraview_catalyst1.py", "paraview_catalyst2.py"]
    }
  }
}
```

Output variables

The variables written to the solution files are named in `surface_variables` (p. 85) and `volume_variables` (p. 85) in the `solution` block. A monitor point samples the names in its own `variables` list from the volume set. The output variables (p. 173) page lists every name, with its meaning and units.

```
{
  "output_settings": {
    "solution": {
      "surface_variables": ["p", "cp", "yplus"],
      "volume_variables": ["V", "p", "T", "mach"]
    }
  }
}
```

Which names are valid depends on the equation type, set by `type` (p. 20) in `equations`. The tiers are cumulative: each equation type offers every name of the types above it in the table, and adds its own. The incompressible solver offers the same names as the compressible solver for the same equation type.

EQUATION TYPE	NAMES OFFERED
`euler`	The base names, such as `pressure`, `velocity` and `cp`, and the Euler tier, such as `vorticity` and the pressure forces on a surface
`viscous`	Adds the viscous tier, such as `viscosity`, `nu`, `yplus`, `cf` and the friction forces
`rans` or `les`	Adds the LES and RANS tiers, such as `reynoldstress`, `ti` and the turbulence model variables

Two further sets are offered on top of the equation type. The transition variables, `turbulenceintermittency` and `transitionreynoldsnumber`, are offered when the turbulence `model` (p. 24) is `sst-transition`, the model that solves for them. The modal variables, such as `modaldeformation` and `meshvelocity`, are offered when a wall boundary condition sets `fsi` (p. 99).

Surface and volume sets differ. Some names, such as `yplus`, `ut` and `cf`, exist only on a surface; others, such as `vorticity` and `ti`, only in the volume.

In an unsteady run, `compute_average_and_rms` (p. 84) keeps a running average and RMS of velocity, pressure and temperature in each cell, updated once per real time step. Request them as `V_avg`, `V_rms`, `p_avg`, `p_rms`, `T_avg` and `T_rms`. Averaging starts at the real time step set by `average_start_time_cycle` (p. 84); before that the statistics are reset every step. The RMS is of the value itself, not of its fluctuation, so the standard deviation is $\sqrt{(\text{rms}^2 - \text{avg}^2)}$. A restart reads the statistics back from the checkpoint and continues them. In a compressible run, requesting one of them without `compute_average_and_rms` stops the run.

`var_i`, `resvar_i`, `resshare_i` and `vargrad_i` are the solution value, residual, share of the total squared residual and gradient of solved equation `i`. They are offered for `i` from 1 up to the number of equations solved:

EQUATION TYPE	TURBULENCE MODEL	EQUATIONS SOLVED
`euler`, `viscous`, `les`	not used	5
`rans`	`sa-neg`	6
`rans`	`sst-transition`	9
`rans`	any other model	7

A name outside the valid set fails validation, and the deck is rejected before the solver starts. The message names the variable, the equation type and the context; see Output variable rejected (p. 357) for the names most often mistaken.

ParaView Client

To download the free ParaView software for visualising VTK HDF format data on the local device, visit the ParaView download page. ParaView 6.1.1 or later is required for .vtkhdf file support.

Opening a run in ParaView

`generate_paraview_helpers` writes ParaView scripts that load a run's output, so a case with several models or boundary types opens in one step. Run it in the zCFD environment once the solver has written output, naming the control file:

```
(zCFD)> generate_paraview_helpers naca0012.py
```

It writes these scripts beside the control file:

SCRIPT	WHAT IT LOADS
`paraview_view_volumes.py`	The volume solution of each model
`paraview_view_surfaces_<boundary>.py`	One boundary type, such as `wall`, for every model
`paraview_view_all_surfaces.py`	Every boundary type, grouped by type
`paraview_view_all.py`	The volumes and every boundary type

Each model of an overset (p. 328) case is loaded as its own source, named after its key under `model`, and given its own colour. A script loads only the files the run has written.

Open a script in the ParaView GUI with *File › Run Script…*, or run it with `pvpython`:

```
(zCFD)> pvpython paraview_view_all.py
```

When run with `pvpython`, a script also saves the pipeline as a ParaView state file of the same name, such as `paraview_view_all.pvsm`, which opens with *File › Load State…*. When connected to a ParaView server, it does not save the state.

OPTION	EFFECT
`-o, --output-dir`	Write the scripts to this directory, creating it if needed
`--python`	Write scripts that load the output without saving a state file

The scripts find the output through `_base_dir`, set near the top of each script to the directory they were written to. To open them where the run is mounted at a different path, change `_base_dir` to match.

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 not 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 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 official documentation.

Alternatively, use the version bundled with zCFD. The 'pvserver' included in the zCFD distribution includes plugins for reading the zCFD HDF5 mesh format directly.

To start a ParaView Server on your computer within the zCFD environment run:

```
> pvserver
```

ParaViewConnect

ParaViewConnect is a helper utility for connecting to remote ParaView servers. It reduces the complexity of setting up the required ssh connections and works well with zCFD. More information and installation instructions can be found on the GitHub page.

ParaView Post-Processing Best Practices for Parallel zCFD Data

When post-processing solution data generated in parallel by zCFD using ParaView, it is important to understand the differences between various ParaView filters and their impact on data interpolation. This section provides guidance on recommended pipelines for accurate results.

CleantoGrid vs MergeBlocks

Two commonly used ParaView filters for processing parallel datasets are **CleantoGrid** and **MergeBlocks**. Each has distinct characteristics:

CleantoGrid Filter:

- **Purpose**: Merges points that are exactly coincident and removes degenerate cells

- **Interpolation**: Performs minimal interpolation, preserving original data values at cell centres and vertices
- **Use case**: Recommended when data accuracy is critical and you want to preserve the original computed values
- **Characteristics**: May result in slightly fragmented visualisations at block boundaries but maintains data fidelity

MergeBlocks Filter:

- **Purpose**: Combines multiple blocks into a single unstructured grid
- **Interpolation**: May perform interpolation across block boundaries to create smooth transitions
- **Use case**: Better for smooth visualisations and streamline generation across block boundaries
- **Characteristics**: Creates visually smoother results but may alter original computed values through interpolation

Recommended Pipelines

For **quantitative analysis** and **data extraction** (probes, line plots, force calculations):

```
# Recommended pipeline for quantitative analysis
import paraview.simple as pvs

# Load data
reader = pvs.OpenDataFile('solution.vtkhdf')

# Use CleantoGrid to preserve data accuracy
clean_data = pvs.CleantoGrid(Input=reader)

# Convert to point data if needed
point_data = pvs.CellDatatoPointData(Input=clean_data)

# Proceed with analysis (probes, calculators, etc.)
```

For **visualisation purposes** (streamlines, contours, volume rendering):

```
# Pipeline for smooth visualisations
import paraview.simple as pvs

# Load data
reader = pvs.OpenDataFile('solution.vtkhdf')

# Use MergeBlocks for smoother visualisations
merged_data = pvs.MergeBlocks(Input=reader)

# Convert to point data for smooth contouring
point_data = pvs.CellDatatoPointData(Input=merged_data)

# Apply visualisation filters (streamlines, contours, etc.)
```

Important Considerations

1. **Data Accuracy**: Always use CleantoGrid when extracting quantitative data for analysis or validation
2. **Visualisation Quality**: MergeBlocks may provide better visual results for presentations and general visualisation
3. **Performance**: CleantoGrid is typically faster as it performs less processing
4. **Block Boundaries**: Be aware that different filters may show artefacts differently at parallel decomposition boundaries
5. **Regression Testing**: When comparing results across different runs, ensure consistent filter usage

Loading data in a script

When processing zCFD parallel data, load the output one way for analysis and another for visualisation:

```
import paraview.simple as pvs


def load_for_analysis(filename):
    reader = pvs.OpenDataFile(filename)
    reader.UpdatePipeline()
    return pvs.CleantoGrid(Input=reader)


def load_for_visualisation(filename):
    reader = pvs.OpenDataFile(filename)
    reader.UpdatePipeline()
    merged = pvs.MergeBlocks(Input=reader)
    return pvs.CellDatatoPointData(Input=merged)
```

This keeps quantitative analysis faithful to the computed values and gives smooth visualisations.

Batch post-processing with Python scripts

Post-process a run with a Python script stored beside its control file. A script gives the same result on every run, sits under version control with the case, and runs unattended on a cluster login node or as a batch job. Every zCFD validation case is post-processed this way.

Run the script with the Python in the zCFD environment, which provides `zutil`, ParaView's `paraview.simple` and matplotlib:

```
(zCFD)> python naca0012_post.py
```

`get_zcfd_result` reads the control file and finds everything the run wrote, so a script need not build output paths by hand:

ATTRIBUTE	WHAT IT HOLDS
`report.data`	The report file as a pandas `DataFrame`, one column per heading
`force_prefix()`	The prefix of the force columns, for building names such as `wall_Ftx`

ATTRIBUTE	WHAT IT HOLDS
`volume_file_path`	The volume solution, `<case>.vtkhdf`
`wall_boundary_path`	The wall surface solution; other boundary types have matching attributes, such as `farfield_boundary_path`
`model_names`, `model_reports`	For an overset case, the model names and each model's report

The following script plots the density residual and a chordwise pressure coefficient for an aerofoil:

```
import matplotlib.pyplot as plt
import paraview.simple as pvs

from zutil.fileutils import get_zcfd_result
from zutil.post import clean_vtk, cp_profile

result = get_zcfd_result("naca0012.py")

report = result.report.data
fig, ax = plt.subplots()
ax.semilogy(report["Cycle"], report["rho"])
ax.set_xlabel("Cycle")
ax.set_ylabel("Density residual")
fig.savefig("naca0012_residuals.png", dpi=200)

wall = clean_vtk(pvs.VTKHDFReader(FileName=result.wall_boundary_path))
profile = cp_profile(wall, [0.0, 1.0, 0.0], [0.0, 0.5, 0.0])
fig, ax = plt.subplots()
ax.plot(profile[0]["chord"], profile[0]["cp"])
ax.invert_yaxis()
ax.set_xlabel("x/c")
ax.set_ylabel("Cp")
fig.savefig("naca0012_cp.png", dpi=200)
```

`zutil.post` also provides `cf_profile` for skin friction, `calc_force_wall` and `calc_moment_wall` for loads integrated from the surface, and `get_monitor_data` for monitor points. `zutil.plot` provides the axis, legend and logo helpers used for zCFD's own figures.

The output directory

The **output** directory is created in the working directory from which the solver is run, and is named after the control file:

```
<case_name>_OUTPUT
```

It holds one sub-directory for each entry under `model` (p. 96), named by that entry's key, and a `LOGGING` sub-directory with one log for each MPI rank.

For the 'vtk' format the volume data of a model is written to its sub-directory with the file name:

```
<case_name>_OUTPUT/<model>/<model>.vtkhdf
```

For the 'vtk' format the surface data is written to the same sub-directory, one file for each boundary condition type:

```
<case_name>_OUTPUT/<model>/<model>_<boundary>.vtkhdf
```

The FWH interpolated surface data (`fwh_interpolate` (p. 87)) and FWH wall data (`fwh_wall_data` (p. 87)) are written to the *ACOUSTIC_DATA* sub-directory of the model's sub-directory, `<case_name>_OUTPUT/<model>/ACOUSTIC_DATA`.

What a checkpoint records about the run

A `<model>_results.h5` checkpoint carries the run's own configuration alongside the flow solution, so the file explains where it came from without depending on the run directory still being intact:

LOCATION IN THE FILE	CONTENTS
`version` (root attribute)	Layout version of the checkpoint, currently `2`. This is the file layout, not the deck's `'config version'`.
`/provenance/status`	The `<case>_status.yaml` file verbatim — zCFD version, date, host names, devices, partition count and the mesh MD5 checksums. Written once.
`/provenance/control/<filename>`	The source of every Python file the control-file read executed, one dataset each, with the full path and mtime as attributes. The set is the whole include chain, not only the file named on the command line, so a deck built from a base plus overrides is recorded in full.

The control deck can be edited while a run is in progress — the solver polls it and applies any change the schema marks updatable — so it is re-read on each checkpoint and rewritten whenever it has actually changed. Reading it back is a plain HDF5 string read:

```
import h5py

with h5py.File("case_OUTPUT/background/background_results.h5") as f:
    print(f.attrs["version"])
    print(f["provenance/status"][()].decode())
    for name in f["provenance/control"]:
        print("---", f[f"provenance/control/{name}"].attrs["path"])
        print(f[f"provenance/control/{name}"][()].decode())
```

Recording these is best effort: a checkpoint that could not be annotated is still a valid, restartable checkpoint, and the run reports a warning rather than stopping.

The report file

zCFD has the capability to output certain parameters to a <casename>_report.csv file as the solver progresses. These can be used to monitor the solver's convergence. By default, the solver outputs

volume integrated residuals for each solution variable to the <casename>_report.csv file, but adding to the `report` (p. 84) block means other quantities of interest can also be monitored as the simulation progresses. zMon plots these columns while the solver runs; see Monitoring a run (p. 324).

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

> **NOTE**
>
> When `inner_convergence` (p. 56) is set, its **'check_frequency'** cycles are also reporting cycles, so rows may appear more often than **'frequency'** alone would suggest.

'scale_residuals_by_volume' scales the reported volume-integrated residuals by cell volume. It applies to the `resvar_i` and `resshare_i` output variables as well as to the report, so the field and the norm of that field always describe the same quantity.

'monitor', **'forces'** and **'mass_flow'** are dictionaries keyed by whatever name is chosen for each block; the keys are not otherwise validated, but naming them consecutively with a prefix (e.g. `MR_1`, `FR_1`, `MF_1`) keeps a large control file readable:

REPORTING ON	SUGGESTED KEY PREFIX
'monitor'	MR_?
'forces'	FR_?
'mass_flow'	MF_?

Example usage:

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

```
# Report every 10 cycles
"report": {
    "frequency": 10,
},
```

A simulation with two monitor points and one forces block

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

```
"report": {
    "frequency": 10,
    "monitor": {
        "MR_1": {
```

```
            "point": [0.0, 0.0, 0.0],
            "variables": ["p", "V"],
        },
        "MR_2": {
            "point": [1.0, 0.0, 0.0],
            "variables": ["p", "V"],
        },
    },
    "forces": {
        "FR_1": {
            "zones": [4],
        },
    },
},
```

Cycle time column

Every row carries an **'AvgCycleTime'** column: the mean wall-clock time, in seconds, of a solver cycle over every cycle of the run so far. It is the same measurement the console report block and the end-of-run summary give, recorded per row so the cost of a run can be read from its history. The last row is the run's mean cycle time.

The cycle timed is the solve - mesh mapping, the solver advance and source term updates - and excludes writing the report and the solution files, which happen after the cycle is timed.

> **NOTE**
> Wall-clock time is not a like-for-like column between runs: it depends on the machine, the partition count and anything else running on the node. It is a within-run diagnostic. In an overset case every mesh's report file carries the same value, because one cycle advances all of them.

Residual statistics columns

Setting **'residual_statistics'** (p. 91) adds four columns per equation, describing the *true* residual $R(Q)$ rather than the solution update. They answer a question the RMS cannot: an RMS says "not converged" without saying that the field is converged everywhere except a handful of cells, which is the case that takes longest to diagnose.

The columns are named prefix-first (`Rinf_rho`, not `rho_Rinf`) and are written to <casename>_report.csv only - they are deliberately kept off the console residual line, which four extra columns per equation would otherwise swamp.

COLUMN	MEANING
`Rinf_<eq>`	$\|R\|_\infty$, the largest residual anywhere in the mesh for that equation.
`Rrms_<eq>`	RMS of the true residual. Note this is *not* the equation's own residual column, which is the RMS of the solution update $\Delta Q/\Delta\tau$; the two coincide only when the linear solve is exact.

COLUMN	MEANING
`Neff_<eq>`	Participation ratio $(\sum R^2)^2 / \sum R^4$: the number of cells the residual is effectively spread over. Equal to the cell count when every cell contributes equally, and to 1 when a single cell carries everything. A small value against a large mesh means a localised stall, not a stalled field.
`Rtop_<eq>`	Fraction of $\sum R^2$ held by the single worst cell.

To find *where* those cells are, add the `resshare_i` (p. 173) volume output variables. `resshare_i` is each cell's share of the same $\sum R^2$, summing to one over the mesh, so thresholding it at `1/Neff_<eq>` isolates the cells carrying more than an equal share - the cells `Neff` is counting.

> **NOTE**
>
> This is a different measure from 'InnerCells' (p. 250), and neither supersedes the other. `InnerCells` is a single scalar over the solution *update*, used to steer dual-time inner cycles. These are per-equation figures over the true residual, for diagnosing where a steady-state stall is sitting.

On by default. Set **'residual_statistics'** to `False` to save the extra pass over the residual field and the two MPI reductions per report.

Monitor points

Monitor points report the specified variable at fixed mesh locations.

'point' should be the x,y,z location of the point of interest in the mesh as it appears in the solver (i.e. after any mesh scaling due to parameters scaling (p. 169) parameters).

'variables' controls which variables are reported at this point. In the <casename>_report.csv the column name will be a combination of the **'name'** (or, if omitted, the block's own dictionary key in lower case, with spaces written as underscores) and the variable name.

'name' overrides the column heading used for variables from this monitor point. If omitted, the dictionary key given to the block is used instead, in lower case with spaces written as underscores: a block keyed `MR_1` reports its pressure as `mr_1_p`, and one keyed `Inlet Probe` as `inlet_probe_p`.

Example usage:

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

```
"report": {
    "frequency": 10,
    "monitor": {
        "MR_1": {
            "name": "monitor_1",
            "point": [180.0, 100.0, 50.0],
            "variables": ["V", "p", "T", "ti"],
```

```
        },
    },
},
```

Forces

Force reporting outputs the x,y,z total forces and moments on a surface or set of surfaces. These are presented as force coefficients (using the non-dimensionalisation convention used for aerofoil lift, drag and moment coefficients), as opposed to dimensional values. The force and moment coefficients resulting exclusively from viscous forces and exclusively from pressure forces are also given.

Example usage:

```
"report": {
    "frequency": 10,
    # Report force coefficients in the grid axes, using a user defined transform
    "forces": {
        "FR_1": {
            "name": "force_1",
            # Required: Zones to be included
            "zones": [1, 2, 3],
            # Transformation function
            "transform": my_transform,
            "reference_area": 0.112032,
            "reference_length": 1.0,
            "reference_point": [0.0, 0.0, 0.0],
            # Calculate forces using a specified reference pressure rather than absolute
            "reference_pressure": 0.0,
        },
    },
},
```

Output names

The forces will be written into the <casename>_report.csv prefixed with the supplied **'name'** (or, if not given, the block's own dictionary key in lower case, with spaces written as underscores, so a block keyed `Main Wing` writes `main_wing_Fx`). The forces will be written with the following names:

ITEM	REPORTING OUTPUT NAME
Total forces and moments	**'name'**_F[x,y,z] and **'name'**_M[x,y,z]
Forces and moments due only to pressure forces	**'name'**_Fp[x,y,z] and **'name'**_Mp[x,y,z]
Forces and moments due only to viscous (friction) forces	**'name'**_Ff[x,y,z] and **'name'**_Mf[x,y,z]
Transformed total forces and moments (if transform function is set)	**'name'**_Ft[x,y,z] and **'name'**_Mt[x,y,z]
Transformed forces and moments due only to pressure forces (if transform function is set)	**'name'**_Ftp[x,y,z] and **'name'**_Mtp[x,y,z]
Transformed forces and moments due only to viscous forces (if transform function is set)	**'name'**_Ftf[x,y,z] and **'name'**_Mtf[x,y,z]

ITEM	REPORTING OUTPUT NAME

NOTE

On a model whose rotating `motion` (p. 169) moves the mesh without `rotate nodes`, `'inertial_frame'` (default `True`) transforms the totals above **in place** before they are written — there is no separate set of "static frame" columns. Set `'inertial_frame': False` to report the forces and moments in the rotating (blade) frame instead, still under the same `_F`/`_M` column names. With `rotate_nodes` the mesh nodes themselves turn, the forces are already in the inertial frame and `'inertial_frame'` has no effect, as it has none on a frame, whose mesh does not move.

A rotating `motion`, moving mesh or frame, also adds the thrust, torque and power about its own axis, as `'name'`_*model*_Thrust, `'name'`_*model*_Tq and `'name'`_*model*_Tqp, where *model* is the model's name. A forces block named `wall` on a model named `rotor` writes `wall_rotor_Thrust`, `wall_rotor_Tq` and `wall_rotor_Tqp`. There is no keyword to configure these.

Total force

Total force, $\mathbf{F}$, is $\mathbf{F}_p + \mathbf{F}_v$, and likewise for moments.

Non-dimensionalisation

Forces are non-dimensionalised into force coefficients as $C_x = \frac{\mathbf{F}_x}{q_\infty S}$, where q_∞ is the reference dynamic pressure and S is the reference area.

Moments are non-dimensionalised into moment coefficients as $C_{Mx} = \frac{\mathbf{M}_x}{q_\infty S c}$, where c is the reference length.

Dimensional pressure force

$$\mathbf{F_p} = \oint -(p - p_{ref})\,\hat{\mathbf{n}}\,\mathrm{d}A$$

Where p is surface pressure and $\hat{\mathbf{n}}$ is the surface normal (pointing towards the wetted area).

Dimensional viscous force

$$\mathbf{F}_v = \oint \tau_w\,\mathrm{d}A$$

Here τ_w is the wall shear stress vector. It lies in the surface and points along the velocity of the fluid relative to the wall.

Dimensional pressure moment

$$\mathbf{M}_p = \oint (\mathbf{x} - \mathbf{x}_{ref}) \times \left(-(p - p_{ref})\,\hat{\mathbf{n}}\right)\,\mathrm{d}A$$

Where $\mathbf{x}$ is the surface position and $\mathbf{x}_{ref}$ is the `reference_point`.

Dimensional viscous moment

$$\mathbf{M}_v = \oint (\mathbf{x} - \mathbf{x}_{ref}) \times \tau_w\,\mathrm{d}A$$

NOTE

The reference pressure p_{ref} is a static pressure, set in pascals by `reference_pressure` on the force block. Its default of 0 gives forces computed from the absolute pressure.

Mass flow

'zones' The mass flows through all the zones are summed to give a single output value for the mass flow monitor. Flow out of the domain is considered positive massflow, flow into the domain is considered positive massflow and the massflows are dimensional (typically kg per second).

The ‹casename›_report.csv column is always named after the block's dictionary key in lower case, with spaces written as underscores — e.g. a block keyed `MF_1` reports as `mf_1_massflow`, and one keyed `Total Inlet` as `total_inlet_massflow`.

Two monitor, force or mass flow blocks whose keys differ only in case or spacing, such as `Total Inlet` and `total_inlet`, would write the same columns, so zCFD refuses the control file and names both keys.

> **NOTE**
>
> Unlike `'monitor'` and `'forces'`, a mass flow block has no `'name'` field to override the reported column name — only `'zones'` is a valid key here. Where a friendlier column name than the dictionary key is needed, the key itself can be chosen accordingly (e.g. `"pipe_outflow"` instead of `"MF_1"`).

Example usage:

```
"report": {
    "frequency": 10,
    # Report massflow
    "mass_flow": {
        "MF_1": {
            # Zones to be included
            "zones": [1, 2, 3],
        },
    },
},
```

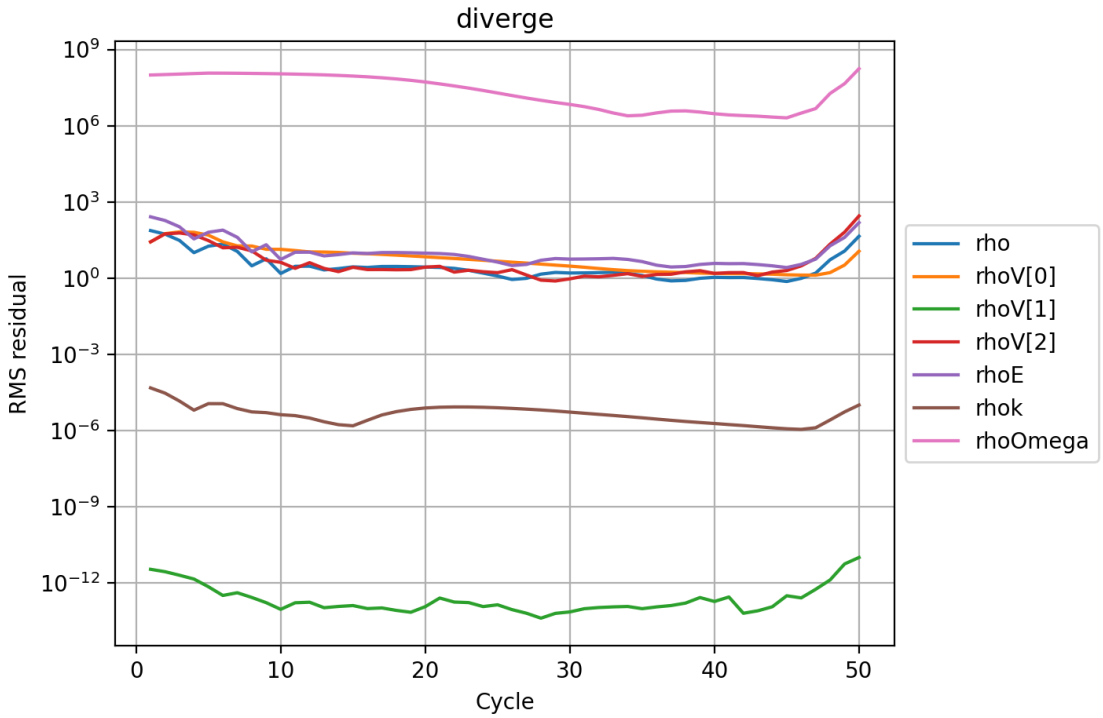
Troubleshooting

zCFD validates every deck before the solver starts, so a key that is misspelt, out of range or inconsistent with its neighbours is reported before the first cycle. Most of the symptoms below are ones validation cannot catch. Find the symptom in the table, confirm it with the check its section gives, then work through the causes in the order listed.

SYMPTOM	WHAT THE RUN SHOWS	SECTION
Residuals rise until the run stops	One or more residual columns of the report file grow from cycle to cycle	Diverging residuals
The run stops on a non-finite value	`Report check at cycle ...` in the log	NaN or `inf` in the report
A compressible run fails and the cells that cause it are unknown	`has an invalid solution` lines in the log, and a `<model>_fail` output	Locating the failing cells
Residuals fall, then plateau	A residual history that flattens well short of the target	Convergence stall
The inner-cycle residual flattens at the same level in every real time step	A floor in each real time step, while the integrated forces continue to settle	Limiter switching
Divergence persists at a low CFL number and first order, or a stall is local	A residual carried by a handful of cells	Poor mesh quality
The solver stops on a licence message, at start-up or during the run	`The zCFD licence could not be initialised.`, `No zCFD licence could be checked out for zcfd-core.` or `zCFD lost contact with the licence server for more than 10 minutes.`	Licence and launch failures
Validation rejects an output variable	`Output variable error: Variables [...] are not valid for equation type ...`	Output variable rejected

Diverging residuals

The residuals in the `<casename>_report.csv` file should fall gradually as the solver converges, per real time step if the flow is unsteady. A run is diverging when one or more of them turns and rises instead. The divergence may follow a period of otherwise stable convergence, because a particular pattern in the flow can appear during the solution and destabilise the scheme. A diverging run that is left to continue ends on a non-finite value.



The typical form of a divergent set of residuals. Rather than decreasing, the values all start increasing significantly from cycle 45 onwards. In this case the NACA0012 case was deliberately set with an unstable explicit CFL number of 6.0.

Confirm it. Plot the residual columns of the report file against the cycle, or watch them in zMon (p. 324). For an implicit (p. 63) run using the AMGX library, the log also reports each solve of the implicit linear system: the number of iterations and the residual after each one. Typically 4 to 5 iterations are enough to achieve the desired reduction in residual, which is 3 to 4 orders of magnitude. If the number of iterations climbs above 6, the linear system may be poorly conditioned as a result of numerical instability in the solution.

```
Cycle 5003 (real time cycle: 0 time: 0)
CFL 20.0 (20.0) - MG 20.0 (coarse mesh: 0)
        iter      Mem Usage (GB)       residual           rate
        ------------------------------------------------------------
            Ini             36.9899   3.037596e-06
            0              36.9899   4.565379e-07         0.1503
            1              36.9899   1.138337e-07         0.2493
            2              36.9899   3.056741e-08         0.2685
            3              36.9899   1.061114e-08         0.3471
            4              36.9899   3.308933e-09         0.3118
            5              36.9899   1.293730e-09         0.3910
        ------------------------------------------------------------
        Total Iterations: 6
        Avg Convergence Rate:   0.2743
        Final Residual:         1.293730e-09
        Total Reduction in Residual:    4.259057e-04
        Maximum Memory Usage:   36.990 GB
```

```
        ------------------------------------------------------------
Total Time: 0.869665
    setup: 0.109454 s
    solve: 0.760211 s
    solve(per iteration): 0.126702 s
        iter      Mem Usage (GB)       residual           rate
        ------------------------------------------------------------
            Ini             36.9899   1.713350e-04
            0             36.9899   1.060694e-07    0.0006
            1             36.9899   1.898567e-10    0.0018
        ------------------------------------------------------------
        Total Iterations: 2
        Avg Convergence Rate:   0.0011
        Final Residual:     1.898567e-10
        Total Reduction in Residual:    1.108102e-06
        Maximum Memory Usage:   36.990 GB
        ------------------------------------------------------------
```

Example output from the AMGX implicit linear solver. In this case the turbulence equations are solved after the primary flow variables. The primary flow variables are solved in 5 cycles of the AMGX solver, achieving an order of magnitude 3 reduction in the residuals.

Work through the causes in this order. If the run still diverges once the first three have been tried, the cause is most likely the boundary conditions, preconditioning or the mesh quality.

CAUSE	SETTING	FIRST CHANGE
CFL number above the stability limit of the mesh	`cfl` (p. 56)	Below 1.0, say 0.5, for an explicit run; 5.0 or less for an implicit run
Second-order spatial accuracy	`order` (p. 75)	`first`
A less dissipative inviscid flux scheme	`inviscid_flux_scheme` (p. 78)	`rusanov`
Wall function iterations that do not converge	wall `kind` (p. 100)	`slip` or `no slip`
An inconsistent set of boundary conditions	`boundary_conditions` (p. 98)	Simplify them, as described below
Preconditioning unsuited to the mesh	`precondition` (p. 20)	`False`
Poor quality cells	Poor mesh quality	Confirm it, then remediate or improve the mesh

Numerical instability

The solver advances the flow solution towards convergence at the required level of spatial and temporal accuracy, subject to a user-defined limit called the CFL (p. 56) (Courant–Friedrichs–Lewy) number. The CFL number limits the update to the solution in each cycle to maintain the stability of the underlying numerical scheme.

There are formal criteria for the CFL number of a given scheme, but these typically apply only to idealised meshes: orthogonal or structured, with a very gradual variation in the volume ratio from cell to cell. For a complex mesh, or where cell quality is poor even in only a few cells, the limit for stability may be significantly lower. In explicit mode the theoretical CFL limit is approximately 1.0 for most schemes, though in many cases a higher value of 1.5 to 2.0 will work. In implicit mode the theoretical CFL limit is 1000 or more, and values between 10 and 100 are common.

Reducing the spatial `order` (p. 75) to `first` and the `inviscid_flux_scheme` (p. 78) to `rusanov` gives the most stable, though least accurate, discretisation.

Wall functions

zCFD scales a turbulent wall function on a solid boundary automatically. An internal iterative solution of a set of equations matches the flow speed and the wall shear stress in the cell adjacent to the wall to the ideal profile of a turbulent boundary layer. The number of internal iterations should be low, fewer than 20, if the solution is physically valid. zCFD reports the time per cycle in the `AvgCycleTime` column of the report file, and a value that starts climbing may indicate that the wall function iterations are not converging. To test whether the wall functions are the cause, change the wall `kind` (p. 100) from `wall function` to `slip` or `no slip`, whichever is the closer approximation to the desired behaviour.

Inconsistent boundary conditions

zCFD checks that each boundary condition in `boundary_conditions` (p. 98) is correctly specified, but it cannot determine whether a combination of boundary conditions is correct. A closed fluid domain with every boundary a solid wall except for a single mass flow condition, for example, will appear to converge at first, until the pressure rises or falls to non-physical levels. The effects of mismatched boundary conditions can also be more subtle, especially where inflow and outflow conditions over-specify or under-specify the solution.

To diagnose inconsistent boundary conditions, simplify them: replace wall functions with slip walls, and inflow and outflow conditions with farfield (p. 118) conditions of kind `riemann`. It can also help to repeat the simulation and stop the solver shortly before the crash, to inspect the solution for unexpected features. The temperature field often points to the areas of the flow that are causing problems, particularly where poor quality cells are present.

Preconditioning

The timestep of zCFD in explicit mode is limited by a local numerical wave speed in the solver scheme. In regions of low-speed flow, such as near stagnation points, this may be overly conservative. Preconditioning modifies the scheme to increase the local timestep, subject to a local stability condition. The stability limits of the preconditioned scheme may be inappropriate for a given mesh, however, and the flow field then diverges. To test this, set `precondition` (p. 20) in `equations` to `False`.

Parallel log files

For a parallel run, the main log file may not contain all of the information about a crash that the solution on one partition has caused. The log file for each partition is stored in the `<case>_OUTPUT/LOGGING` directory as `<case>.<rank>.log`.

NaN or `inf` in the report

The run stops with one of these messages, the first from a compressible run and the second from an incompressible one:

```
Report check at cycle <cycle>: the solution is invalid in <count> cells. It had already
 ↪  diverged, so these may not be the origin.
Report check at cycle <cycle>: the report is not finite.
```

A compressible run also names the invalid cells and writes them out; see Locating the failing cells.

Confirm it. The row holding the non-finite value is written to `<casename>_report.csv` before the run stops, so the last row of the file shows which column failed, and the rows above it show how the run arrived there. For a parallel run, read the log file of each partition as well as the main log.

WHAT THE ROWS BEFORE IT SHOW	CAUSE AND REMEDY
Residuals that rose over the preceding rows	A divergence: see Diverging residuals
A residual carried by a handful of cells, or divergence at a low CFL number	See Poor mesh quality

> **NOTE**
> A column that reads `NaN` in every row written after a restart, while the run continues, is not a failure. The restart was seeded from a run with a different turbulence model, and the column belongs to that model's closure, which this run does not solve.

Locating the failing cells

When the solution of a compressible run becomes invalid, the run names the cells responsible, writes them out for inspection and stops. A cell is invalid when any of its conserved variables is NaN or `inf`, or when its density or pressure is not positive.

The log of the partition that owns an invalid cell carries one `ERROR` line for it. The line gives the global cell ID, the rank and the coordinates of the cell centre, then each conserved variable and the pressure, in SI units with the non-dimensional value alongside:

```
Cell <id> on rank <rank> at (<x>, <y>, <z>) has an invalid solution: <name>=<value> <unit>
 ↪  [nd <value>], ..., p=<value> Pa [nd <value>]
```

Each partition lists at most 20 cells and counts the remainder in a single line, `<count> more invalid cells on rank <rank>`. The main log then names the check that found the cells, the output it wrote and the last checkpoint written before the failure:

```
Report check at cycle <cycle>: the solution is invalid in <count> cells. It had already
 ↪  diverged, so these may not be the origin.
Wrote <case>_OUTPUT/<model>/<model>_fail.vtkhdf (volume + surface). Last good checkpoint:
 ↪  <path> at cycle <cycle>. Rerun from it with safe mode on.
```

If no checkpoint has been written, the line reads `Last good checkpoint: none. Rerun with safe mode on.` When safe mode is already on, the line ends at the checkpoint.

CHECK	WHEN IT RUNS
`Start-up check`	Before the first cycle, on the solution the run starts from. A restart also logs the file and the turbulence model that wrote it
`Safe-mode update check`	After each explicit update on every multigrid level, when safe mode is on. On a coarse level the log names the level, and the cells listed are those of the fine mesh
`Checkpoint check`	Each time a checkpoint is due
`Report check`	When a row of the report file holds a non-finite value. The summary adds that the cells may not be where the divergence began

The failure output. The `<model>_fail` output is written to `<case>_OUTPUT/<model>/`: a volume file, `<model>_fail.vtkhdf`, and one surface file per boundary with the same prefix. It does not replace the checkpoint. Its variables are fixed and take the place of the lists in the deck:

- volume: `FailCell`, `GlobalCellId`, `var_1` to `var_N` and `resvar_1` to `resvar_N` for the N conserved variables, `pressure`, `temperature`, `mach`, `velocity`, `eddy`, `timestep`, `celllimiter`, `cellvolume`, `bad cells maximum non orthogonality angle`, `walldistance` and `cellzone`;
- surface: `FailCell`, `pressure`, `temperature`, `cp`, `yplus` and `zone`.

`FailCell` is 1 in an invalid cell and 0 elsewhere. On a surface, each face takes the value of the cell it belongs to.

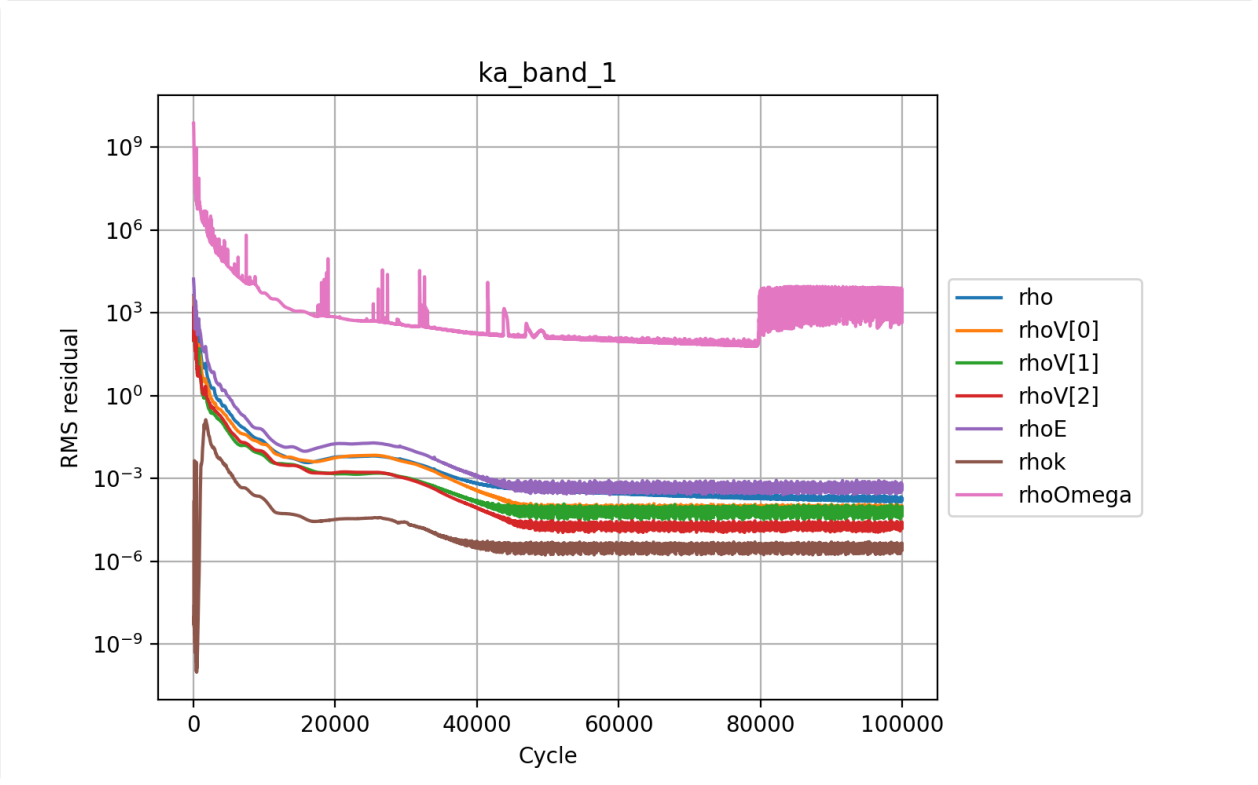
Inspect it in ParaView. Open the volume file and apply a Threshold filter on `FailCell` with a lower bound of 1 to isolate the invalid cells. To find a cell named in the log, use Find Data on `GlobalCellId` with its global cell ID. Colour the surface files by `FailCell` to see which boundaries the invalid cells touch. The cell volume, the non-orthogonality angle and the wall distance in the same file help to judge whether the mesh is at fault, as described in Poor mesh quality.

Find the origin. With `safe` (p. 11) off, the update itself is not checked, and a non-finite value usually reaches the report first. By then the solution has already diverged over several updates, so the marked cells show where the divergence ended, which may not be where it began. Rerun from the last good checkpoint, setting `restart` (p. 48) to `True` in the `initialisation` block and `safe` to `True` in the `solver settings` block. Safe mode then checks every cell after each explicit update, and stops the run at the first update that leaves a cell invalid.

Safe mode synchronises every stage of the update, which slows the run considerably, so use it to diagnose a failing case and leave it off in production. It does not check implicit updates or the coarse multigrid levels. The incompressible solver does not name invalid cells.

Convergence stall

More common than divergence is convergence stall, in which the reported residuals fall at first and then plateau.



The convergence of the steady-state explicit solver has achieved an average residual reduction of 5 orders of magnitude and then stalled. The noise in the solution from 40k cycles onwards suggests that either the flow is not fundamentally steady-state or that multigrid is adding an error to the solution. The user must determine whether the level of convergence is sufficient for their purposes and if not whether to run an unsteady simulation or degrade the spatial accuracy.

Confirm it. The residual history in the report file flattens and stays level. The residual statistics columns (p. 344) show whether the stall belongs to the whole field or to a handful of cells: a small `Neff` against a large mesh means a localised stall, which is a mesh quality question. The flow residuals may be stalled while the integrated properties of interest, such as lift, drag or mass flow, are converged well enough to be useful. That judgement rests with the engineer.

CAUSE	SETTING	FIRST CHANGE
An error introduced by multigrid	`cycles` in `multigrid` (p. 81)	Turn multigrid off after a set number of cycles
Physical unsteadiness in a steady-state simulation	`type` (p. 51) in `time_settings`	Run unsteady from the stalled solution, or degrade the accuracy
A physical time step too large for dual time stepping	`time_step` (p. 51)	Reduce it
The limiter switching within the inner cycles	Limiter switching	Freeze the limiter

Multigrid

For explicit simulations, multigrid (p. 74) accelerates convergence by using a set of coarse meshes to propagate the longer-wavelength components of the solution through the domain faster than the CFL condition would allow on the finest mesh. In theory, and on idealised meshes, multigrid does not change the steady-state solution that is reached, or the pseudo-steady-state solution of an unsteady flow. On real meshes around complex geometry, however, multigrid introduces an error, and this can

limit convergence. Setting `cycles` (p. 81) in the `multigrid` block turns multigrid off after that number of cycles, which removes the error.

Flow unsteadiness

A very common cause of convergence stall, particularly in steady-state simulations, is physical unsteadiness in the underlying flow. The solver is then attempting to find a steady-state solution to an unsteady problem, and the solution keeps changing from cycle to cycle. There are two approaches, depending on the flow features of interest.

If the unsteadiness matters, use the unconverged steady-state flow as the starting point for an unsteady simulation. If the stall is within the pseudo-steady cycles of an unsteady dual time-stepping solution, the physical `time_step` (p. 51) may be too large, and should be reduced.

If the unsteadiness is not of interest, the accuracy of the solution can be deliberately degraded to give a more 'average' steady-state flow. This might mean the `rusanov` inviscid flux scheme, first-order spatial accuracy, a coarser mesh, or a combination of these. The implicit mode of the solver can also filter the solution in time, since a higher CFL number allows larger time steps. This approach can cause solver instability and inaccuracy in the final solution, because the 'averaging' process is non-physical.

Limiter switching

Within the inner, pseudo-time cycles of an unsteady dual time-stepping simulation on the compressible solver, a common cause of apparent stall is the MUSCL limiter switching state from one iteration to the next in a small number of cells. Those cells keep producing a solution change that never decays. The reported residual is an RMS over all cells, so a handful of them is enough to hold the whole norm up at a floor, while the bulk of the field may be converging perfectly well behind it.

Confirm it. The residual drops cleanly for the first several inner cycles and then flattens at the same level in every real time step, while the integrated forces continue to settle. Judging this from the global residual is unreliable. Use the inner cycle convergence columns (p. 250) instead, which measure the drop within each real time step rather than an absolute level.

Adding inner cycles does not help, because the floor is not a problem of convergence rate.

CAUSE	SETTING	FIRST CHANGE
The limiter changing state in a few cells	`freeze_limiter_cycle` (p. 77) in `numerical_scheme`	Set it, to snapshot the limiter and hold it fixed thereafter
A solution that genuinely needs the limiter to keep adapting	`time_step` (p. 51)	Reduce it, so less has to change in each real time step
The outer-iteration floor of the incompressible coupled scheme	linear solver (p. 262), `time_step` (p. 51)	Tighten the linear solver, or reduce the time step
The under-relaxation floor of the incompressible SIMPLE scheme	`target_orders` (p. 59)	Set the target from a run with the criterion off

The incompressible solver has no slope limiter and ignores `freeze_limiter_cycle`. An inner residual that flattens there is the outer iteration's own floor. Under the coupled scheme, tighten the linear solver

or reduce the physical time step. Under SIMPLE, the floor is set by the under-relaxation, `implicit_relax` (p. 61) and `pressure_relax` (p. 66) in the `scheme` block, and it is reached within a handful of cycles. A `target_orders` above that floor can never be met, so set the target from a run with the criterion off rather than adding cycles.

Licence and launch failures

The solver stops at start-up with `The zCFD licence could not be initialised.` or `No zCFD licence could be checked out for zcfd-core.` A run that has already started may instead log `Connection to license server failed retrying`, and later stop with `zCFD lost contact with the licence server for more than 10 minutes.`

Confirm it. The lines printed before the stop name the failure. A failed start-up prints `Error initializing license system` or `Error initializing license heart beat`, followed by the licensing system's own description of the error. A failed checkout prints `Error checking out license` with the RLM status value, then the product, version and count requested, and the same description. Look the status value up in the licence error codes (p. 199).

CAUSE	WHERE TO LOOK
No licence file in any of the locations searched	Licence (p. 198): a `.lic` file in the directory the solver is started from, the `lic` directory of the installation, or `RLM_LICENSE`
A licence that has expired, or has not yet started	Status value -3 or -37
More licences requested than are free	Status value -8 or -22: reduce the number of ranks, or stop other running processes
No connection to the licence server	Status value -17, -103 to -107 or -111 to -113: check the server, the firewall and `RLM_LICENSE`
A connection lost during the run	The heartbeat messages: see below

Once a run has started, a lost connection to the licence server does not stop it at once. The solver logs the retry warning and continues for up to 10 minutes. If the connection is not restored within that time, the run stops with `zCFD lost contact with the licence server for more than 10 minutes.`

Output variable rejected

Validation rejects the deck with a message of this form:

```
Output variable error: Variables ['kinematicviscosity'] are not valid for equation type
↪  'viscous' in volume context. Valid variables include: [...]...
```

Confirm it. The message names the rejected variables, the equation type and whether the list is the surface or the volume one. Where a close spelling is valid, it adds `Did you mean`. The list it closes with shows only the first few valid names, so look the name up on the output variables (p. 173) page instead. Which names are valid, and why, is set out under Output variables (p. 336).

Several names are rejected because the name an engineer reads in an output file, or would naturally write, is not the one a deck asks for:

REJECTED NAME	REQUEST INSTEAD	WHY
`kinematicviscosity`	`nu` (p. 173)	`nu` is written to the output file as `kinematicviscosity`
`turbulenceintensity`	`ti` (p. 173)	`ti` is written as `turbulenceintensity`
`walldistance`	`walldist` (p. 173)	`walldist` is written as `walldistance`
`velocitygradient`	`velocitygrad` (p. 173)	Only the shorter spelling is valid
`facecentre`	`centre` (p. 173)	`centre` is the valid spelling in both contexts
`frictionvelocity`	`ut` (p. 173)	Only the short name is valid

A valid name can also be rejected because the case does not offer it:

CAUSE	SETTING
The name belongs to a higher tier than the equation type solved	`type` (p. 20) in `equations`
A transition variable without the transition model	`model` (p. 24) in `turbulence`, set to `sst-transition`
A modal variable without FSI	`fsi` (p. 99) on a wall boundary condition
A surface-only name in the volume list, or the reverse	`surface_variables` (p. 85), `volume_variables` (p. 85)
A `var_i`, `resvar_i`, `resshare_i` or `vargrad_i` index above the number of equations solved	`type` (p. 20) and the turbulence model

A running average or RMS variable, such as `p_avg` or `V_rms`, passes validation whether or not averaging is on. If `compute_average_and_rms` (p. 84) is not set, the solver raises this error when it first writes the variable:

```
Averaged output variable requested but averaging storage is unallocated - set output_
↪  settings.compute_average_and_rms
```

Poor mesh quality

A run that still diverges once the CFL number, the spatial order and the inviscid flux scheme have been reduced, or a stall carried by a handful of cells, points to poor quality cells. The formal CFL criteria apply only to idealised meshes, and a few poor cells are enough to lower the limit for stability of the whole run.

Confirm it. A small `Neff` in the residual statistics columns (p. 344), against a large mesh, means that the residual is carried by a few cells. Add the `resshare_i` (p. 173) volume output variables and threshold them at `1/Neff` to find those cells. The temperature field often points to the same areas.

CAUSE	SETTING	FIRST CHANGE
Poor cells lowering the limit for stability	`cfl` (p. 56)	Reduce it
Poor cells that cannot be removed from the mesh	Mesh quality remediation	Set a threshold angle

CAUSE	SETTING	FIRST CHANGE
A mesh that can be improved	Preparing a mesh (p. 201)	Improve the mesh, so the whole domain stays second order

Mesh quality remediation

```
{
  "solver_settings": {
    "type": "compressible",
    "mesh_quality_remediation_angle_threshold": 75
  }
}
```

A strongly non-orthogonal face can stop a second-order compressible solve from converging. The non-orthogonality of a face is the angle between the line joining the two cell centres on either side of it and the face normal. It is 0 degrees on an ideal mesh and grows as the cells skew.

Setting `mesh_quality_remediation_angle_threshold` computes the convective flux at first order on every interior face whose non-orthogonality exceeds the threshold, in degrees. Every other face keeps the order the numerical scheme sets. The key is unset by default, and no face is changed. When convergence problems are traced to mesh quality, 75 degrees is a reasonable first threshold. The first-order faces add numerical dissipation where they sit, so an improved mesh remains preferable to a remediated one for a final solution.

The setting applies to the compressible solver only. An incompressible deck that sets it fails validation.

At start-up the log reports the spread of face angles across the mesh and the number of faces changed:

```
[wing] mesh quality remediation: interior faces whose non-orthogonality exceeds 75 degrees
↪  use a first-order convective flux
Mesh quality remediation: face non-orthogonality histogram
Angle (deg) |  0 -> 20 | 20 -> 40 | 40 -> 60 | 60 -> 80 | 80 ->100 | 100->120 | 120->140 |
↪  140->160 | 160->180 |
Num faces   |    812345|     40211|      2310|       188|         4|         0|         0|
↪  0|         0|
Mesh quality remediation: face non-orthogonality ranges from 0 to 81.6 degrees
Mesh quality remediation: 57 interior faces exceed the 75 degree threshold and use a
↪  first-order convective flux
```

Add `badcells` to `volume_variables` in the solution output to see where those faces lie. Each cell next to a first-order face holds the largest non-orthogonality among its first-order faces; every other cell holds 0.

Trimming to a target

A farfield can turn its own angle of attack until a force coefficient reaches a value you set. The run trims, holds the trimmed angle, and reports the angle it found:

```
"BC_3": {
    "zones": [1],
    "type": "farfield",
    "condition": "IC_1",
    "kind": "riemann",
    "target": {"alpha": {"cl": 1.0778, "hold": 500}},
}
```

This asks for the angle of attack that gives a lift coefficient of 1.0778, and for 500 cycles at that angle once it is found.

Setting up a trim

1. **Report the force on the body.** The coefficient is read from a force report on the walls of the body, so the case needs one under `output_settings` › `report` › `forces`, written as coefficients (the default) with `reference_area` set to the body's reference area (it is 1 unless set). With more than one force report, name the one to trim with `of`.

   ```
   "report": {"frequency": 10, "forces": {"FR_1": {"name": "wall", "zones": [4]}}},
   ```

2. **Start the farfield near the answer.** The trim starts from the angle of the farfield's own condition. Any angle works, but a start within a few degrees of the answer saves a step or two. The speed, the Mach number and the sideslip of the condition are kept.
3. **Add the target** to the farfield, as above: the coefficient (`cl` or `cd`) and, usually, a `hold`.
4. **Give the run enough cycles.** `cycles` stays the budget. Allow for the trim, a few hundred cycles for each step, plus the hold. The aerofoil above trims from 0° to 10° in about 2300 cycles on a 449 × 129 mesh.
5. **Read the result.** The log ends with the trimmed angle and the coefficient it reached, the report file carries the angle and coefficient on every report row, and the status file records whether the target was met. To reuse the angle in another run, paste the vector printed on the end line into the condition, or restart from this run.

The run must be steady and compressible. Any other run keeps the condition's own angle and says so in a warning.

What a run does

The run starts at the angle of the farfield's own `condition`. Before each step it waits for the coefficient to settle: three successive means, each over a few report rows, must agree within `tolerance`. Two are not enough, because after a step the coefficient can sit on a short plateau before it falls to its real value. It then steps the angle towards the target, by no more than 4° at a time, and waits again. The first step for `cl` follows from the thin-aerofoil lift slope; later steps follow from the slope fitted through the settled values seen so far. The target is met when the settled value lies within `tolerance` of it.

Well above zero lift, the coefficient creeps towards its settled value for many hundreds of cycles. When the last few means are closing in steadily, the run works out the value they are heading for and steps on that, rather than waiting for the creep to finish. A step taken this way shows the predicted value in its log line, as `cl 0.7531 (heading for 1.0412)`. A prediction only ever starts a step; the target is met, and a stall is confirmed, only on a settled value.

If a higher angle gives less lift rather than more, the aerofoil has stalled. The run does not step beyond the angle of greatest measured lift, and a `cl` target above that lift is reported as not met, naming the peak it measured.

Drag is the same at plus and minus the same angle, so a `cd` target is searched for on one side only: on the side of the starting angle, or at positive angles when the run starts at 0°.

Once the target is met the angle is frozen and the run carries on under its ordinary convergence control (p. 56). If the coefficient drifts outside `tolerance` while the angle is held, the run trims again.

KEY	MEANING
`cl`, `cd`	the target lift or drag coefficient; give exactly one
`hold`	the number of cycles to run once the target is met, after which the run stops; left out, the run ends in the ordinary way
`tolerance`	the largest absolute difference from the target that counts as met; 0.001 for `cl` and 0.0001 for `cd` if left out
`of`	the name of the force report (p. 90) the coefficient comes from
`axes`	the forward and up axes the angle is measured in; see Axes

`of` may be left out when the deck has a single force report. With more than one it is required, and it must name one of them.

Hold and cycles

`cycles` (p. 57) remains the budget for the whole run, and the ordinary stops still apply, so a `hold` can only end a run sooner. A run whose budget runs out during the hold stops at the budget and says so in a warning, for example `held for only 230 of 500 cycles`. Allow enough cycles for the trim and the hold together.

When the target is not met

A run that reaches the end of its budget without meeting the target still finishes normally: it writes its usual output and exits without an error. Its last line states the shortfall plainly, and the status file

records the target as not met, so a post-processing script can check it without reading the log.

Tuning

The defaults suit most runs, and a deck need not mention any of this. A run that steps too eagerly, or waits too long, can be tuned with an `advanced` block inside `alpha`:

```
"target": {
    "alpha": {
        "cl": 1.0778,
        "hold": 500,
        "advanced": {"max_step": 1.0, "average_over": 20},
    }
}
```

KEY	DEFAULT	MEANING	WHEN TO CHANGE IT
`max_step`	4	the largest change of angle in one step, in degrees, up to 6	lower it near stall or for a sensitive configuration; above 6° a single jump can make the flow solve diverge
`relaxation`	1	the fraction of the computed step that is taken, greater than 0 and at most 2	lower it if the angle overshoots and swings about the target
`average_over`	10	the number of report rows averaged into each measurement	raise it when the coefficient oscillates from row to row
`max_wait`	1500	the cycles to wait after a step before stepping anyway, settled or not	raise it for a flow that settles slowly; lower it to trade accuracy for a shorter trim
`slope`	see below	a first guess of the change of the coefficient per degree	set it when the lift slope is known to differ from that of a thin aerofoil, or for `cd`

Left out, `slope` is 0.11 per degree for `cl`, the thin-aerofoil slope of 2π per radian. For `cd` there is no such guess, so the first step is a probe of half `max_step`.

Axes

The angle of attack is measured in the plane of a forward and an up axis, from forward towards up. Lift is perpendicular to the freestream, towards up, and drag lies along it.

The axes default to forward +x and up +z, the ESDU convention that `zutil.vector_from_angle` also uses. With angle of attack α and sideslip β, the velocity is

$$\mathbf{v} = |\mathbf{v}|\,(\cos\alpha\cos\beta,\ \sin\beta,\ \sin\alpha\cos\beta),$$

so α turns the flow from +x towards +z and β turns it out of that plane towards +y. A mesh built with another up axis states its own:

```
"target": {"alpha": {"cl": 0.5}, "axes": {"forward": "+x", "up": "+y"}}
```

Each axis is one of `+x`, `-x`, `+y`, `-y`, `+z` and `-z`, and the two must be at right angles. Turning the angle keeps the speed and the sideslip of the condition unchanged.

What the run reports

Each line of the log about the target begins with `target`, followed by the boundary condition's key as the deck spells it. A line appears only when something changes: once at the start, stating the axes, then at each step, when the target is met, and at the end. This is the NACA 0012 at Mach 0.15 trimmed to its lift coefficient at 10°:

```
target BC_3: trimming alpha (x–z plane, forward +x, up +z) for cl 1.0778, then hold 500 cycles
target BC_3: cycle  200  alpha  0.000° → 4.000°   cl 0.0000   target 1.0778
target BC_3: cycle  600  alpha  4.000° → 8.000°   cl 0.4369 (heading for 0.4378)   target
↪ 1.0778
target BC_3: cycle 1100  alpha  8.000° → 9.790°   cl 0.8245 (heading for 0.8808)   target
↪ 1.0778
target BC_3: cycle 1500  alpha  9.790° → 9.934°   cl 1.0397 (heading for 1.0622)   target
↪ 1.0778
target BC_3: cycle 1900  alpha  9.934° → 9.980°   cl 1.0709 (heading for 1.0732)   target
↪ 1.0778
target BC_3: cycle 2100  alpha  9.980° → 9.998°   cl 1.0764   target 1.0778
target BC_3: cycle 2300  alpha  9.998° → 9.984°   cl 1.0790   target 1.0778
target BC_3: cycle 2500  cl 1.0782 met at alpha 9.984° — holding 500 cycles
target BC_3: alpha trimmed to 9.984° for cl 1.0778 (cl 1.0780 at end, cycle 3000); as the
↪ condition's vector: zutil.vector_from_angle(9.984, 0.000)
```

The end line also gives the trimmed angle in a form ready to paste into a condition, `zutil.vector_from_angle(9.984, 0.000)`. With axes other than the default it gives the trimmed vector itself. A run that does not meet its target ends instead with a warning such as:

```
target BC_3: cl target NOT met: cl 0.6830 at alpha 6.000° (target 1.0778, 36.6 % short) after
↪ 3000 cycles
```

The coefficient in each of these lines, and in the end line in particular, is the settled value: the mean of the last `average_over` report rows, not a single row.

The report file gains two columns for each target, written on every report cycle: the angle, `<bc>_alpha`, in degrees, and the coefficient, such as `<bc>_cl`.

The status file (p. 214) holds a `target` record for each target, keyed as the log lines are:

```
target:
  BC_3:
    phase: met
    alpha: 9.803
    cl: 1.0783
    met_cycle: 4300
```

The phase is `settle`, `trim` or `hold` while the run proceeds, and `met` or `not met` at the end. `cl` is the

settled value, and `met_cycle` is the cycle at which the target was met; it is empty until then.

Restarting

A restarted run with a target starts the farfield at the angle recorded in the previous run's status file, not at the angle of its `condition`. A run whose target was met goes straight to the hold; one still trimming carries on. The line after the opening line says so, for example `target BC_3: resuming at alpha 9.803° (met in the previous run)`. After a restart the flow settles once more before the angle moves.

A restarted run without a target uses its `condition` as written. To keep a trimmed angle in a later run without a target, paste the end line's `zutil.vector_from_angle` call into the condition's `vector`.

What can be trimmed

A target turns the angle of attack, `alpha`, for a lift or drag coefficient, on a steady compressible run. The farfield's `condition` must give its velocity as a `vector`.

On any other run, unsteady (dual or global time stepping (p. 237)) or incompressible, the target is ignored: validation warns, naming the boundary condition, the log repeats the warning at the start, and the run proceeds at the condition's own angle.

Placing and moving a mesh

Every mesh is placed and moved by one block: the `transforms` (p. 169) of its model entry. It holds three settings. `scale` converts the units of the mesh file, `transform` puts the mesh where it belongs in the case, and `motion` describes how it moves, or the frame its flow is solved in, during the run. Each is optional.

The order of operations

The coordinates of every node pass through the three settings in a fixed order:

```
x_file ──scale──▶ x_m ──transform──▶ x_placed ──motion(t)──▶ x(t)
```

1. `scale` multiplies the coordinates read from the mesh file, so that they are in metres.
2. `transform` is applied once, when the mesh is read, to the scaled coordinates. A translation in a `transform` matrix is therefore in metres, whatever the units of the file.
3. `motion` acts on the placed mesh. Its `axis`, `origin` and `velocity` are given in placed coordinates: the position of the mesh in the case, after `scale` and `transform`.

A rotor whose mesh was built about the origin in inches, and placed 2 m along x, therefore turns about an `origin` of `[2.0, 0.0, 0.0]`, not about the origin of its mesh file.

Keys

KEY	VALUE	EFFECT
`scale` (p. 169)	three positive numbers, default `[1, 1, 1]`	Multiplies the x, y and z coordinates of the mesh file, for example to convert inches to metres.
`transform` (p. 170)	a 4 × 4 matrix, or a function of `location` that returns three numbers	Places the scaled mesh, once, when it is read.
`motion` (p. 169)	a settings block	Rigid motion of the placed mesh, or the frame its flow is solved in.
`motion` › `type` (p. 172)	`rotating` or `translating`	The kind of motion.
`motion` › `axis` (p. 171)	a vector	Rotating only. The axis of rotation, in placed coordinates.
`motion` › `origin` (p. 171)	a point	Rotating only. A point on the axis, in placed coordinates.
`motion` › `omega` (p. 171)	rad/s	Rotating only. The rate, positive right-handed about `axis`.

KEY	VALUE	EFFECT
`motion` > `velocity` (p. 171)	m/s, three numbers	Translating only. The full velocity.
`motion` > `moving mesh` (p. 172)	`True` or `False`, default `False`	`True` moves the nodes through an unsteady run. `False` keeps the mesh still and solves its flow in the moving frame, the multiple reference frame (MRF) treatment.
`motion` > `ramp time` (p. 172)	seconds	A moving mesh rises linearly from rest to its full rate over this time.
`motion` > `rotate nodes` (p. 172)	`True` or `False`, default `False`	Rotates the nodes of a rotating moving mesh every time step, instead of carrying the rotation as an angle.

Which settings go together

Validation applies these rules and names the setting and its remedy when a deck breaks one.

- A mesh may carry one `motion`.
- `"moving mesh": True` requires an unsteady compressible run and an overset or sliding interface on that mesh. A steady rotor, or any motion in the incompressible solver, is written with `"moving mesh": False`.
- `ramp time` is allowed only with `"moving mesh": True`. A frame runs at its full rate from the first cycle.
- `rotate nodes` is allowed only on a rotating motion with `"moving mesh": True`. A frame never moves the nodes.
- `transform` is either a matrix or a function. A function is called once during validation with a representative location, and must return three numbers.

A motion, moving mesh or frame, always covers the whole mesh. A region that moves on its own, such as a fan inside a duct, is either a mesh of its own, joined to the rest by a sliding (p. 289) or overset (p. 328) interface and carrying its own `motion`, or, when it turns steadily, a set of cell zones solved in a rotating frame by a `rotating` fluid zone (p. 277). A model with a rotating fluid zone carries no `motion` and no overset or sliding interface. A wall's `velocity` (p. 286) turns or slides that surface without moving the mesh.

A moving mesh and a frame differ in what the solver integrates. A moving mesh is advanced to its new position every real time step, and every overset mapping or sliding intersection that involves it is recomputed. A frame leaves the mesh where it is and adds the Coriolis and centrifugal sources of a rotating frame, or the velocity of a translating one, to every cell. The incompressible solver applies the frame velocity at every cell and face but adds no Coriolis or centrifugal source.

Examples

A rotor

An overset cylinder, placed at y = 1.2 m and z = 1.2 m, turning about its own axis at 104.72 rad/s. The `origin` is the placed centre of the cylinder:

```
parameters["model"]["cylinder_1"] = {
    "mesh": "cylinder_origin.h5",
    "transforms": {
        "transform": [
            [1.0, 0.0, 0.0, 0.0],
            [0.0, 1.0, 0.0, 1.2],
            [0.0, 0.0, 1.0, 1.2],
            [0.0, 0.0, 0.0, 1.0],
        ],
        "motion": {
            "type": "rotating",
            "axis": [1.0, 0.0, 0.0],
            "origin": [0.0, 1.2, 1.2],
            "omega": 104.72,
            "moving mesh": True,
        },
    },
    "boundary conditions": {"BC_1": {"type": "interface", "kind": "overset", "zones": [7]}},
}
```

Each model entry sits under a label of its own, so several entries may load the same mesh file, each with its own placement and motion. A rotor turning inside a stator across a sliding interface carries its motion in the same way; see Sliding (p. 289).

A frame

A steady rotor in a mesh of its own is the same block with `"moving mesh": False`. The flow is solved in the rotating frame and the mesh does not move:

```
parameters["model"]["rotor"]["transforms"] = {
    "motion": {
        "type": "rotating",
        "axis": [0.0, 0.0, 1.0],
        "origin": [0.0, 0.0, 0.0],
        "omega": 100.0,
        "moving mesh": False,
    }
}
```

A frame over only part of a mesh, such as a wheel or an impeller meshed as cell zones of their own, is a `rotating` fluid zone (p. 277) instead of a `motion`.

A sliding interface whose rotating side is a frame is a frozen rotor. The two meshes keep their relative position, the rotor is solved in its own frame, and the interface takes its axis from the frame. The run may be steady.

A translating body that starts from rest

A train, placed 110 m along x, accelerated from rest to 69.4 m/s over the first 1.5 s:

```
parameters["model"]["train"] = {
    "mesh": "train.h5",
    "transforms": {
        "transform": [
            [1.0, 0.0, 0.0, 110.0],
            [0.0, 1.0, 0.0, 0.0],
            [0.0, 0.0, 1.0, 0.0],
            [0.0, 0.0, 0.0, 1.0],
        ],
        "motion": {
            "type": "translating",
            "velocity": [-69.4, 0.0, 0.0],
            "moving mesh": True,
            "ramp time": 1.5,
        },
    },
    "boundary conditions": {"BC_2": {"type": "interface", "kind": "overset", "zones": [9]}},
}
```

Without `ramp time` the mesh moves at its full rate from the start of the run. With it, the rate rises linearly from rest over `ramp time` seconds of physical time, which avoids the transient of an impulsive start. For a ramp time T the angle turned by time t is $\omega t^2/(2T)$ during the ramp and $\omega(t - T/2)$ after it, and a translation follows the same law.

A scaled and placed mesh

A mesh built in inches, converted to metres and then turned 10 degrees about y and moved 1 m upstream. The matrix acts on the scaled coordinates, so its translation is in metres:

```
import zutil.transform

placement = zutil.transform.rotate([0.0, 1.0, 0.0], 10.0)
placement = zutil.transform.translate([0.0, 0.0, 0.0], [-1.0, 0.0, 0.0], placement)

parameters["model"]["wing"]["transforms"] = {
    "scale": [0.0254, 0.0254, 0.0254],
    "transform": placement,
}
```

The `zutil.transform` helpers and the conventions of an affine matrix are described in Initial placement (p. 210).

A placement function

`transform` may instead be a function. It is called with each node's scaled position as the keyword argument `location`, and returns the placed position as three numbers. Here the mesh is wrapped about the z axis:

```
import math


def place(*, location):
    x, y, z = location
    radius = 5.0 + y
    angle = x / 5.0
    return (radius * math.sin(angle), radius * math.cos(angle), z)


parameters["model"]["duct"]["transforms"] = {"transform": place}
```

A function is evaluated node by node in Python, so it is slower than a matrix on a large mesh. It cannot place a mesh that carries non-conformal interface faces; give such a mesh a matrix.

Each time step of a moving mesh

At each real time step a moving mesh is placed where its motion puts it at the end of the step. Its velocity over the step is the distance moved divided by the time step, so the volumes swept by the moving faces are consistent with the motion. A restart is described in the next section.

Restarting a run with a moving mesh

A restart carries on from the mesh as the stored solution left it. The results file records where the mesh was, and the run resumes from that position, whatever has been changed in the deck. A restart therefore continues the run as an uninterrupted one would, or stops at start-up with an error that names the model, the file, both values and what to do.

Changing the motion

You may change `omega` or `velocity`, reverse the direction, or change or remove `ramp time`. The mesh moves on from its stored position at the new rate. With `rotate nodes`, you may also change `axis` or `origin`: the new rotation is applied to the mesh where it stands. A rotating motion may be replaced by a translating one, and the reverse, in the same way.

`ramp time` counts from the time the motion first appears. A motion present from the start of the run ramps from time zero. A motion added at a restart ramps from the restart time. A motion removed at a restart leaves the mesh where it was.

The rate of the first step after a restart is the mean over the step that produced the stored solution, so a mesh in the middle of a ramp resumes at the rate it had reached.

```
parameters["model"]["cylinder_1"]["transforms"]["motion"]["omega"] = 62.83
```

A steady run may be restarted as an unsteady run with a moving mesh. The clock starts at zero and the mesh keeps the position stored in the file. An unsteady run restarted as a steady one holds the mesh where it was.

Changing the time step

A restart may use a different `time step`. The first step at the new size is taken at first order, and the order of accuracy then rises as usual. The same applies when the step is edited during a run. A step that was stopped part way is finished at its own size before the new one takes over. Dual and global time stepping may be exchanged.

Restarting the clock

`restart_ignore_history` (p. 49) restarts the clock at zero and ignores the cycle history, but the mesh stays where the file left it. `ramp time` then counts from zero.

Changes that stop the run

The run stops at start-up, before the solution is read, when a restart would otherwise be inconsistent. Each error names the file and both values.

WHAT WAS DONE	WHAT TO DO INSTEAD
`total time` set below the time stored in the file	Raise `total time`. A value equal to the stored time ends the run at once, with a note.
A moving mesh restarted from a file that holds no physical time	Restart from a results file written by this version of zCFD, or start afresh with `restart` off.
`axis` or `origin` changed on a rotating mesh that carries its rotation as an angle	Keep the stored `axis` and `origin`, or start afresh with `restart` off.
A rotating mesh that carried its rotation as an angle given `rotate nodes`, another motion, or none	Keep the rotating motion with `moving mesh` on and `rotate nodes` off, or start afresh with `restart` off.
`moving mesh` switched between `True` and `False`	Keep the setting of the original run.
`scale` or `transform` changed since the file was written	Restore the original placement. Changes of position belong in `motion`.
A model of a several-model case restarted from a file written at a different cycle or time	Restart every model from files of the same step.

Results files written before these records were kept are accepted. The mesh is taken to have moved by the current motion from time zero, and a warning names the file.

PART 3

Tutorials

Worked cases to follow step by step.

Tutorials

The tutorials teach the fundamentals of using zCFD, starting with a simple case and building up the complexity of the simulation. They provide a good starting point for setting up your own simulations. The input files required for each tutorial are available online, linked from that tutorial's page.

1. Simple aerofoil (p. 373)
2. Steady-state RANS (p. 400)
3. Time-accurate solution (p. 414)
4. Aeroacoustics (FWH) (p. 426)
5. Overset meshes (p. 442)
6. Incompressible flow (p. 462)
7. Rotor in hover (p. 473)
8. Polars and trimming to a target (p. 489)

Simple aerofoil

What is in this tutorial?

This tutorial runs your first zCFD simulation: air flowing past an aerofoil, the cross-section of a wing. The aerofoil is the NACA 0012, a symmetrical shape that has been measured in wind tunnels and simulated so often that it is the standard first test for a flow solver, so there are good results to compare with.

You download a ready-made mesh and control file. The mesh divides the space around the aerofoil into small cells, and zCFD works out the flow in every cell; the control file tells zCFD what to simulate. You then run zCFD, watch the run settle to an answer, and look at the results in Python and ParaView. The run takes under two minutes on an NVIDIA RTX 3060.

What you need?

To run the tutorial you will need an installation of zCFD and a valid licence (the free licence is sufficient for this tutorial). For instructions on installing zCFD please see installation and licensing (p. 197).

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

Step 1. Download the files

The tutorial bundle can be downloaded here. It contains a mesh file (`naca0012.h5`) and a zCFD control file (`naca0012.py`).

This tutorial assumes you unpack the bundle in your home directory, which creates a directory called `tutorial_1_aerofoil`.

```
cd ~
wget https://zcfd.zenotech.com/v2026.11.16174/tutorial_1_aerofoil.tgz
tar xzf tutorial_1_aerofoil.tgz
cd tutorial_1_aerofoil
```

Step 2. Walk through the input

The control file, `naca0012.py`, is a short Python script. All it does is build one Python dictionary, called `parameters`, that tells zCFD everything about the run: the physics, the numerical method, what to write out, and the mesh with its boundaries. This step walks through that dictionary from top to bottom. Each heading below is the name of a key, written exactly as it appears in the file, and the headings nest the same way the dictionary does: `numerical scheme`, for example, sits inside `solver`. Under each heading you'll find what that part controls, what each setting means, and why it has the value it has here. Each section ends with a link to that part's reference page, which lists every setting it accepts and its default; you don't need it to follow the tutorial. On a wide screen the whole control file sits on the right, with the

part being described highlighted as you scroll.

```
parameters = {
    "config version": 2,
    "solver": {
        "solver settings": {"type": "compressible"},
        "equations": {
            "type": "rans",
            "turbulence": {"model": "sst"},
            "precondition": True,
        },
        "fluid properties": {
            "material": "air",
            "gamma": 1.4,
            "gas constant": 287.0,
            "sutherlands const": 110.4,
            "prandtl no": 0.72,
            "turbulent prandtl no": 0.9,
        },
        "reference conditions": {
            "IC_1": {
                "temperature": 300,
                "pressure": 101325.0,
                "v": {"vector": [1.0, 0.0, 0.0], "mach": 0.15},
                "viscosity": 1.02145e-05,
                "turbulence intensity": 5.2e-2,
                "eddy viscosity ratio": 1.0,
            },
        },
        "initialisation": {"initial conditions": "IC_1", "restart": False},
        "numerical scheme": {
            "order": "euler_second",
            "inviscid flux scheme": "roe",
        },
        "time settings": {"type": "steady"},
        "convergence control": {
            "scheme": {"name": "implicit euler"},
            "cfl": {
                "cfl": 50,
                "cfl ramp": [{"type": "exponential", "factor": 1.1, "initial": 1.0}],
            },
            "cycles": 1500,
        },
        "output settings": {
            "report": {
                "frequency": 10,
                "forces": {
                    "FR_1": {"name": "wall", "zones": [4]},
                },
            },
            "solution": {
                "surface variables": [
                    "V",
                    "p",
```

```
                    "mach",
                    "cp",
                    "cf",
                    "pressureforce",
                    "frictionforce",
                ],
                "volume variables": [
                    "V",
                    "p",
                    "T",
                    "rho",
                    "mach",
                    "cp",
                    "eddy",
                ],
            },
        },
    },
    "model": {
        "naca0012": {
            "mesh": "naca0012.h5",
            "boundary conditions": {
                "BC_1": {
                    "zones": [1],
                    "type": "farfield",
                    "condition": "IC_1",
                    "kind": "riemann",
                },
                "BC_2": {
                    "zones": [4],
                    "type": "wall",
                    "kind": "no slip",
                },
                "BC_3": {
                    "zones": [2, 3],
                    "type": "symmetry",
                },
            },
        },
    },
}
```

`config version` comes first: it is always `2`, and tells zCFD which form of control file to expect.

solver

`solver` holds every setting shared by the whole simulation: the physics, the numerics, how the run marches to convergence, and what it writes.

solver settings

`solver settings` chooses which solver runs the simulation.

- `type`: `"compressible"` selects the density-based solver, which suits air at any speed. The alternative,

`"incompressible"`, suits liquids and very slow flows.

See `solver settings` (p. 11).

`equations`

`equations` sets which equations are solved, and so what physics the simulation includes.

- `type`: `"rans"` solves the Reynolds-averaged Navier–Stokes (RANS) equations. The Navier–Stokes equations describe how a viscous fluid moves. Most real flows near a surface are turbulent: full of small, fast, chaotic eddies, far too fine to capture on a mesh like this one. RANS solves for the average flow instead, and adds a turbulence model to represent the effect of the eddies on it. How turbulent a flow is depends on its Reynolds number, which compares the fluid's momentum with its viscosity. Here it is 6 million, and at that value the thin layer of air next to the aerofoil, the boundary layer, is turbulent, so RANS is the natural choice.
- `turbulence`: picks the turbulence model. `"model": "sst"` is Menter's SST model, a robust general-purpose choice for flow that stays attached to an aerofoil.
- `precondition`: `True` switches on low-Mach-number preconditioning. The Mach number is the flow speed divided by the speed of sound; here it is 0.15. At such low Mach numbers sound waves travel far faster than the flow, which makes the equations zCFD solves slow to converge and less accurate. Preconditioning rescales them to remove the problem, and is recommended below about Mach 0.3.

See `equations` (p. 18) and Choosing a turbulence model (p. 226).

`fluid properties`

`fluid properties` describes the fluid: here, air treated as an ideal gas.

- `material`: `"air"` names the fluid.
- `gamma`: 1.4, the ratio of specific heats of air.
- `gas constant`: 287.0 J/kg/K, the specific gas constant of air.
- `sutherlands const`: 110.4 K, the constant in Sutherland's law, which sets how the viscosity varies with temperature.
- `prandtl no`: 0.72, the laminar Prandtl number, which relates heat conduction to viscosity.
- `turbulent prandtl no`: 0.9, the same ratio for the turbulent transport the SST model adds.

These are the standard values for air. See `fluid properties` (p. 37).

`reference conditions`

`reference conditions` defines named flow states. Boundary conditions, the initial flow and the force coefficients all refer to these states by name.

`IC_1` `IC_1` is the freestream: the undisturbed flow far from the aerofoil.

- `temperature`: 300 K.
- `pressure`: 101325 Pa, standard atmospheric pressure.
- `v`: the freestream velocity. `vector` gives its direction, along the mesh's x axis, and `mach` its size, Mach 0.15: about 52 m/s at this temperature.
- `viscosity`: 1.02145×10^{-5} kg m^{-1} s^{-1}. With the aerofoil's 1 m chord, this gives a Reynolds number of 6 million, matching the experiments this case is usually compared with.

- `turbulence intensity`: 0.052, the size of the velocity fluctuations in the incoming flow as a fraction of its speed.
- `eddy viscosity ratio`: 1.0, the turbulent viscosity of the incoming flow relative to its laminar viscosity.

The last two set the turbulence entering the domain; these values suit a flow with little freestream turbulence. See `reference conditions` (p. 39) and Choosing initialisation and reference state (p. 293).

`initialisation`

`initialisation` sets the flow the run starts from.

- `initial conditions`: `"IC_1"` fills the whole domain with the freestream at the first cycle. The same state is also the reference for the reported force coefficients.
- `restart`: `False` starts from that initial flow. `True` would continue from the solution an earlier run saved, as Steady-state RANS (p. 400) shows.

See `initialisation` (p. 47).

`numerical scheme`

`numerical scheme` sets how the equations are discretised in space: how the flow in each cell is turned into the flux between neighbouring cells.

- `order`: `"euler_second"` treats the flow variables to second order and the turbulence variables to first order. First order assumes the flow is constant inside each cell; second order lets it vary linearly across the cell, which is more accurate on the same mesh. A limiter, van Albada by default, stops the second-order values overshooting near sharp changes.
- `inviscid flux scheme`: `"roe"` uses Roe's scheme, a standard method for working out how much mass, momentum and energy flows between neighbouring cells.

See `numerical scheme` (p. 74) and Choosing a spatial scheme (p. 254).

`time settings`

`time settings` says what kind of run this is in physical time.

- `type`: `"steady"` asks for a steady solution: one that does not change with time. Real flows are rarely truly steady, but an attached flow round a streamlined aerofoil is close enough, and a steady run is far cheaper than a time-accurate one.

See `time settings` (p. 51) and Choosing a time-marching scheme (p. 237).

`convergence control`

`convergence control` sets how the solver reaches that steady solution. It starts from the initial flow and improves it a little on each pass, called a cycle. When further cycles stop changing the results you care about, the run has converged.

`scheme`

- `name`: `"implicit euler"` takes each step by solving a linear system, which allows large steps and so few cycles.

`cfl` The CFL number sets how big a step each cycle takes, relative to the size of each cell and the speed of the flow through it. Too large and the run becomes unstable and fails; too small and it needs many more cycles.

- `cfl`: 50, the CFL number the run works at.
- `cfl ramp`: how the CFL number climbs to 50. The single ramp here is `"exponential"`: it starts at the `initial` value of 1 and multiplies the CFL number by the `factor` 1.1 each cycle, reaching 50 after about 40 cycles. The first cycles, far from the answer, therefore take small steps.

See CFL control (p. 245).

`cycles`

- `cycles`: 1500, the number of cycles the run takes before it stops. This case has settled well before then.

See `convergence control` (p. 56).

`output settings`

`output settings` sets what zCFD writes during and after the run. It has two parts: the report, a small table written as the run goes, and the solution, the full flow field written to files for viewing.

`report` `report` writes one row of residuals and forces to `naca0012_report.csv` at a set interval, and prints the same values in the log. A residual measures how far the current solution is from satisfying one of the equations; the residuals fall as the run converges. zMon plots this file.

- `frequency`: 10, so a row every 10 cycles.
- `forces`: the force blocks. Each one sums the forces on a set of boundary zones and adds them to the report. The single block here, `FR_1`:
  - `name`: `"wall"`, which prefixes the columns it adds: `wall_Fx`, `wall_Fy` and `wall_Fz` for the total force, `wall_Fpx`… for its pressure part and `wall_Ffx`… for its friction part, and `wall_Mx`… for the moments about the origin.
  - `zones`: `[4]`, the aerofoil surface.

  Forces are reported as coefficients, divided by the dynamic pressure of `IC_1` and a reference area that defaults to 1. With the mesh's 1 m chord and unit span, `wall_Fx` is the drag coefficient here.

See `report` (p. 84).

`solution` `solution` lists the variables written to the flow-field files, for viewing in ParaView.

- `surface variables`: written on the boundaries, one file per boundary type: `naca0012_wall.vtkhdf`, `naca0012_farfield.vtkhdf` and `naca0012_symmetry.vtkhdf`. Here the velocity `V`, pressure `p`, Mach number, pressure and skin-friction coefficients `cp` and `cf`, and the pressure and friction forces on each face.
- `volume variables`: written in every cell, to `naca0012.vtkhdf`. Here the velocity, pressure, temperature `T`, density `rho`, Mach number, `cp`, and the eddy viscosity `eddy`.

The control file sets no `frequency` for the solution, so the files are written once, at the end of the run, along with the checkpoint `naca0012_results.h5` used for restarts. The report frequency and the solution frequency are separate: the report is cheap and written often, the flow-field files are large and usually written rarely. See `solution` (p. 84) and Output variables (p. 336).

`model`

`model` holds one entry per mesh. Each entry is a model: a mesh with its own boundary conditions.

`naca0012`

The key, `naca0012`, names the model. It also names the model's output folder and files: `naca0012_OUTPUT/naca0012/`, with `naca0012_report.csv`, `naca0012.vtkhdf` and the rest inside. The folder above, `naca0012_OUTPUT`, takes its name from the control file.

`mesh`

- `mesh`: `"naca0012.h5"`, the mesh file, in zCFD's own HDF5 format, read from the directory zCFD runs in. Meshes from other tools are converted to this format first; see Preparing a mesh (p. 201).

`boundary conditions` `boundary conditions` assigns a condition to each group of boundary faces. The groups are the mesh's zones, numbered when the mesh was made; each condition lists its zones by number in `zones`. A mesh converted with zCFD's tools comes with a zone helper file that maps the zone names from the meshing tool to these numbers; see Zone identifiers (p. 208). This mesh has four zones: the outer boundary, the two side faces, and the aerofoil.

The keys `BC_1`, `BC_2` and `BC_3` are names; any name will do.

`BC_1` The outer boundary of the domain.

- `zones`: `[1]`.
- `type`: `"farfield"`, a boundary far enough from the aerofoil to see the undisturbed flow.
- `condition`: `"IC_1"`, the flow state it imposes.
- `kind`: `"riemann"` lets flow pass both in and out without reflecting waves back into the domain.

`BC_2` The aerofoil surface.

- `zones`: `[4]`.
- `type`: `"wall"`.
- `kind`: `"no slip"`: the flow velocity is zero at the surface, as for any viscous flow. Note the space in the name.

`BC_3` The two side faces of the mesh, normal to y.

- `zones`: `[2, 3]`.
- `type`: `"symmetry"`: no flow passes through these faces, and the flow on each side mirrors the other. This makes the aerofoil behave as if it had an infinite span, so the mesh needs only one cell across it.

See Choosing boundary conditions (p. 285).

Step 3. Run

1. Activate your zCFD environment, which puts zCFD's commands, and the Python it uses, on your path:

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

2. Check the control file for errors with `validate_input`:

```
cd ~/tutorial_1_aerofoil/
validate_input naca0012.py
```

3. Run zCFD in the case directory:

```
run_zcfd -c naca0012.py
```

zCFD will attempt to detect the number of CPU cores and any GPUs present and make use of them. If you want to manually configure this see invocation and flags (p. 179).

zCFD writes its progress to the screen and to the log file `naca0012.log`. These are extracts from a run of this tutorial, with `...` marking lines left out:

```
Parameters validated successfully
...
 Reading file naca0012.h5
 Global Mesh Stats: Cells: 57344 Faces: 229856
...
 Dimensional Conditions
 ----------------------
  Pressure:     101325 Pa
  Temperature:  300 K
  Density:      1.17683 kg/m^3
  Viscosity:    1.02145e-05 Ns/m^2 @ 300 K
  Speed:        52.0783 m/s
...
  Mach:         0.15
  Reynolds No:  6.00003e+06 @ 300 K
...
Starting solve loop
...
Cycle     10 | Step    0 t=0s | CFL   2.358 | MG 1 |    17.8 ms | Total     0.271s
+-----------------------------------------------------------------------------------------+
| Report: Cycle 10  Wall time: 0.271s                                                     |
...
| Residuals:                                                                              |
|   rho       1.4099E+01  rhoV[0]   1.5372E+02  rhoV[1]   1.0380E-10  rhoV[2]   6.7992E+01 |
|   rhoE      4.3532E+01  rhok      1.2966E-05  rhoOmega  2.7047E+08                        |
+-----------------------------------------------------------------------------------------+
...
```

```
 Solve Complete: 1500 cycles in 41.162s (avg 27.4 ms/cycle)
...
zCFD finished: all requested cycles are complete (1500 in total).
```

- First the control file is checked, and the mesh read: 57,344 cells.
- The reference conditions are printed in dimensional form. Check them against what you meant to set up; here the speed is 52 m/s and the Reynolds number 6 million.
- Each cycle then prints one line: the cycle number, the current CFL number (still ramping up at cycle 10), and the time the cycle took. Every 10 cycles, the report frequency, a box gives the residual of each equation.
- Between the cycle lines, the linear solver prints a short summary of each implicit solve.
- The run ends with a timing summary and a line saying why it stopped.

To stop a run early, see Stopping a run (p. 326).

The run takes under two minutes on an NVIDIA RTX 3060, and longer on a CPU.

Output files

A successful zCFD run creates the following files and directories. They are named after the control file, `naca0012.py`, and each model's output is named after its key under `model`, which in this deck is also `naca0012`.

FILE/DIRECTORY	DESCRIPTION
`naca0012.log`	A copy of the output to screen during the run.
`naca0012_status.yaml`	A YAML file recording how the run was started: the zCFD version, the date, the control file, the mesh and its checksum, and the nodes and devices used.
`naca0012_OUTPUT/naca0012/naca0012_report.csv`	A .csv format table which shows the variation of flow statistics with cycle number.
`naca0012_OUTPUT/naca0012/naca0012_results.h5`	The checkpoint: the flow solution in zCFD format, used to restart the run.
`naca0012_OUTPUT/naca0012/naca0012.vtkhdf`	A VTK HDF file containing the volume data from the simulation, for visualisation.
`naca0012_OUTPUT/naca0012/naca0012_wall.vtkhdf` etc	One VTK HDF file for each boundary condition type, containing the surface data for that boundary condition.
`naca0012_OUTPUT/LOGGING/`	A separate log for each MPI rank, used for debugging.

The status file is described in The status file (p. 214).

Step 4. Monitor

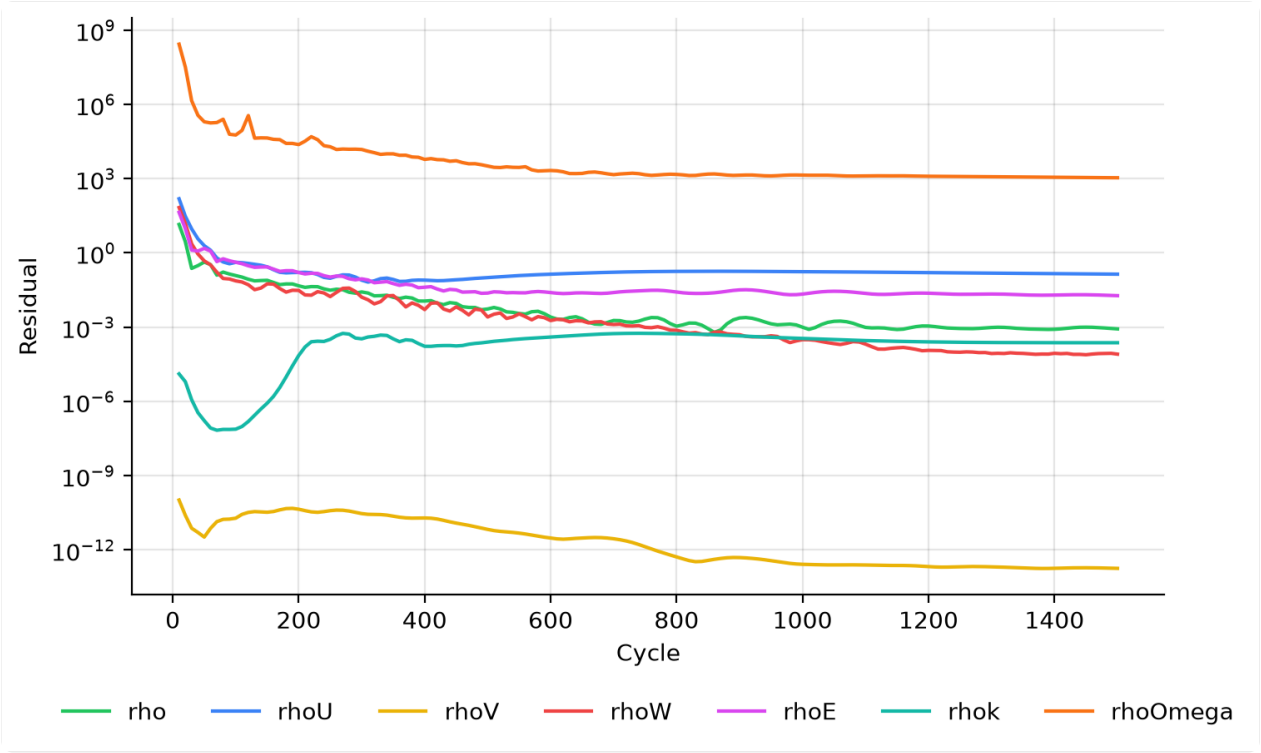
zMon shows the residuals and forces of a run as it goes, and of a finished run. Open it on the control file:

```
source <ZCFD_INSTALL_PATH>/bin/activate
cd ~/tutorial_1_aerofoil/
zmon naca0012.py
```

A browser opens on the zMon dashboard; Monitoring a run (p. 324) describes each panel. The plots below are the residual and force panels as zMon draws them; Step 5 shows how to plot the same data yourself.

Residuals

The residual of each equation measures how far the current solution is from satisfying it: the lower the residual, the more converged the solution. There is one for each equation: density `rho`, momentum in each direction, energy `rhoE`, and the two turbulence variables `rhok` and `rhoOmega`. The report, the log and the script write the momentum residuals `rhoV[0]`, `rhoV[1]` and `rhoV[2]`, for x, y and z; zMon names them `rhoU`, `rhoV` and `rhoW`.



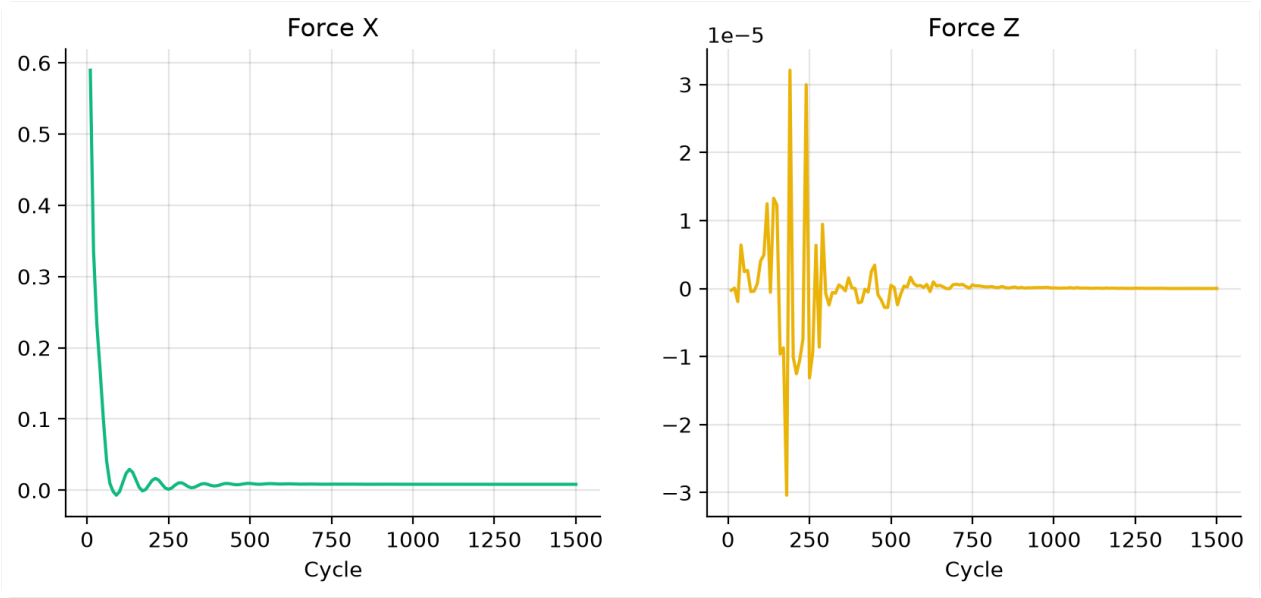
Residuals of the NACA0012 run against cycle

The residuals fall fastest in the first few hundred cycles, as the flow develops from the uniform initial state. The density residual falls about four orders of magnitude before levelling off. `rhoV[1]`, the momentum across the span, stays near round-off throughout, because the flow has no velocity across the span. The turbulence residuals behave differently: `rhok` rises as the boundary layer forms, and `rhoOmega` falls from a large initial value. By 1200 cycles every residual is flat, which, with the forces below, shows the solution has converged.

Forces

The force panels show the x and z forces on the wall. The aerofoil is symmetrical and meets the flow at zero angle of attack, so the x force is the drag and the z force the lift. The lift should be zero and the

drag positive.



Force on the aerofoil in x and z against cycle

The drag falls quickly from its initial value and settles after a few damped oscillations; the lift oscillates about zero at around 10^{-5} before dying away. Both traces are flat from 1200 cycles onwards. The drag coefficient settles at 0.0082, against 0.00828 from NASA's CFL3D code on a finer grid of the same family (897 × 257 points).

Step 5. Post-process

First activate your zCFD environment, which provides Python with zutil, h5py and ParaView:

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

Python

This script draws the residual and force plots shown in Step 4. `get_zcfd_result` finds the run's output from the control file, and `report.data` is the report as a table, one column per quantity. Save the script as `plot_convergence.py` in the case directory:

```
import matplotlib.pyplot as plt

from zutil.fileutils import get_zcfd_result

report = get_zcfd_result("naca0012.py").report.data

fig, ax = plt.subplots()
for column in ["rho", "rhoV[0]", "rhoV[1]", "rhoV[2]", "rhoE", "rhok", "rhoOmega"]:
    ax.semilogy(report["Cycle"], report[column], label=column)
ax.set_xlabel("Cycle")
ax.set_ylabel("Residual")
ax.legend()
```

```
fig.savefig("residuals.png", dpi=200)

fig, (drag, lift) = plt.subplots(1, 2, figsize=(10, 4))
drag.plot(report["Cycle"], report["wall_Fx"])
drag.set_title("Drag coefficient (wall_Fx)")
lift.plot(report["Cycle"], report["wall_Fz"])
lift.set_title("Lift coefficient (wall_Fz)")
for ax in (drag, lift):
    ax.set_xlabel("Cycle")
fig.savefig("forces.png", dpi=200)
```

Run it:

```
python plot_convergence.py
```

It writes `residuals.png` and `forces.png`. Batch post-processing with Python scripts (p. 340) lists what `get_zcfd_result` provides.

ParaView

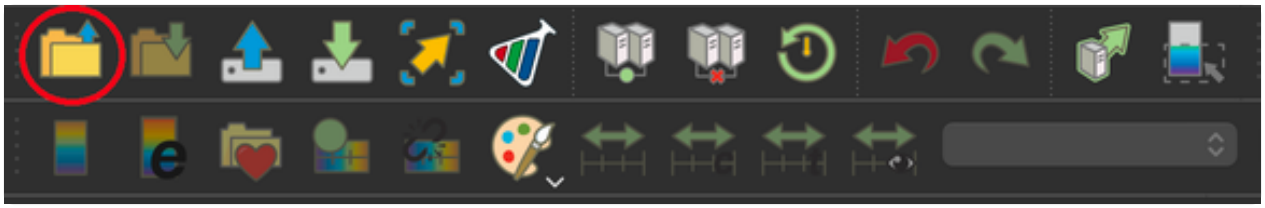
ParaView is a free, general-purpose tool for viewing simulation results, and the recommended way to look at zCFD's flow fields. See ParaView.org to download and install it.

> **NOTE**
>
> The .vtkhdf output format requires ParaView 6.1.1 or later.

Loading your data

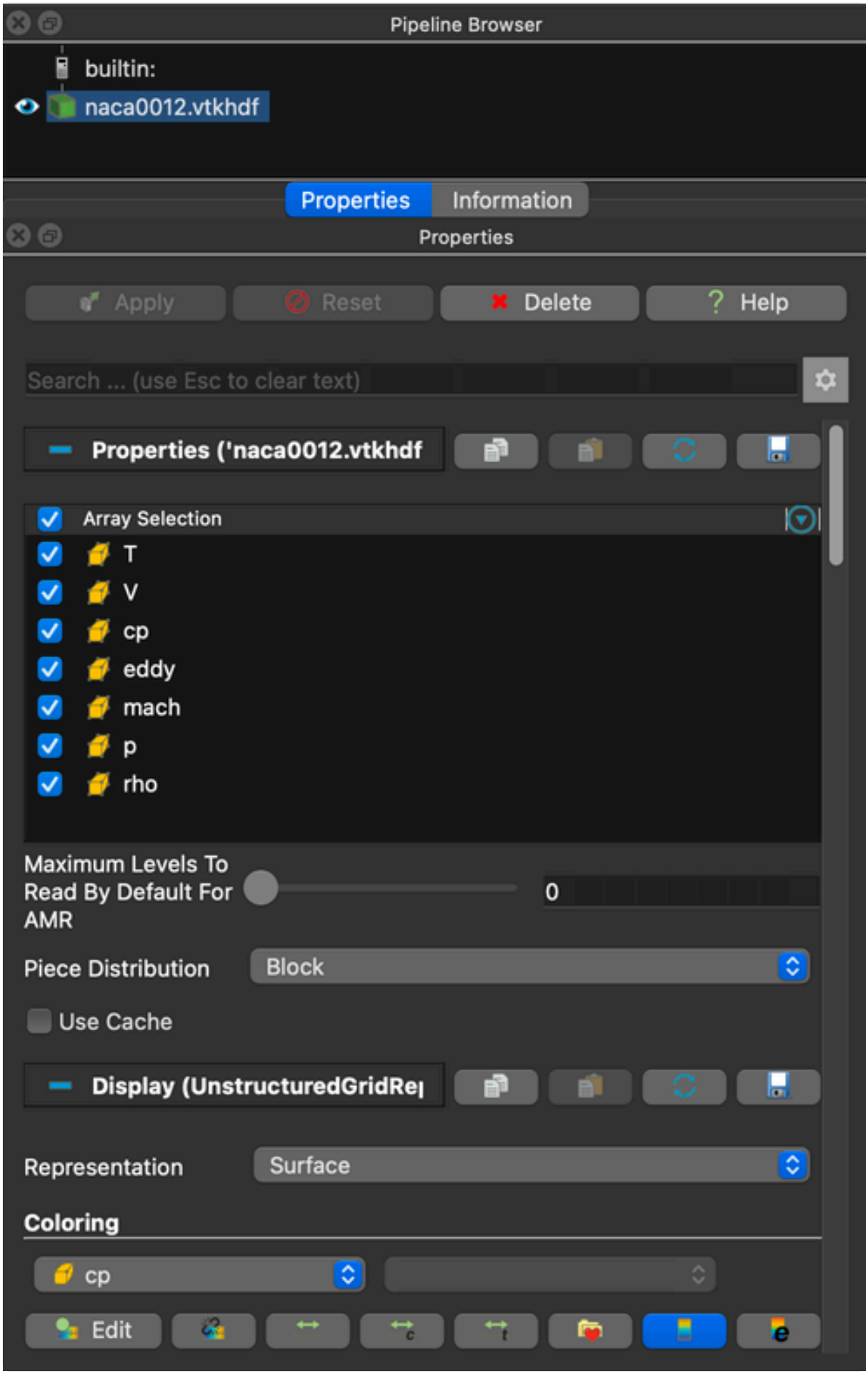
The run wrote the flow in every cell to a single file, `naca0012_OUTPUT/naca0012/naca0012.vtkhdf`.

After opening ParaView, click the "Open" icon shown below or go into File›Open or press Ctrl+O. Locate the directory in which you have stored your data and open `naca0012.vtkhdf`.



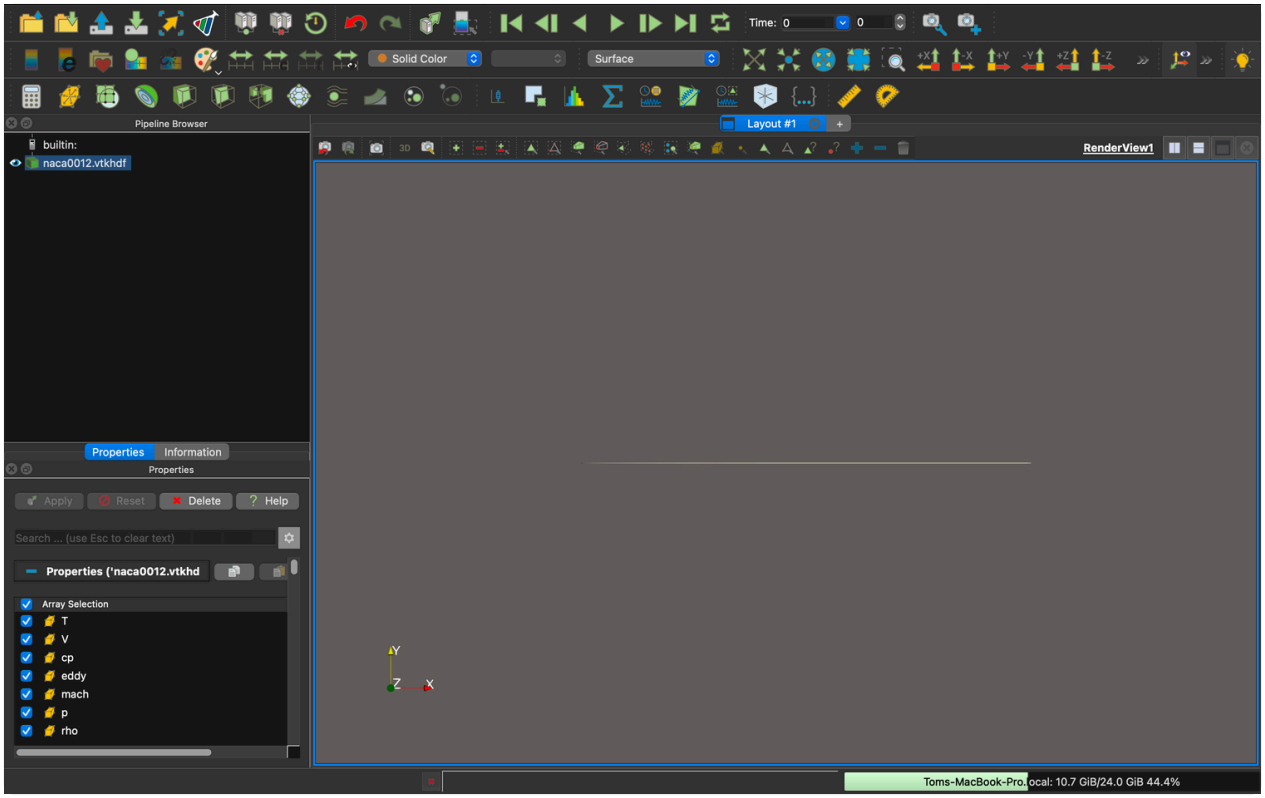
The Open icon on ParaView's main toolbar

Once the file is open, the Pipeline Browser on the left lists it, and the Properties panel below lists the variables the file holds under Array Selection:



The Pipeline Browser and the Properties panel after opening the file, before Apply

Click Apply to read them in, which leads to the following layout:



ParaView after loading `naca0012.vtkhdf`

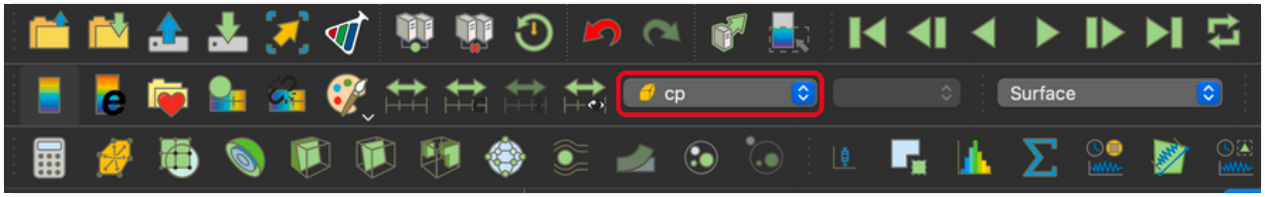
To look at the aerofoil side-on, set the view direction to +Y with this button:



The +Y view direction button

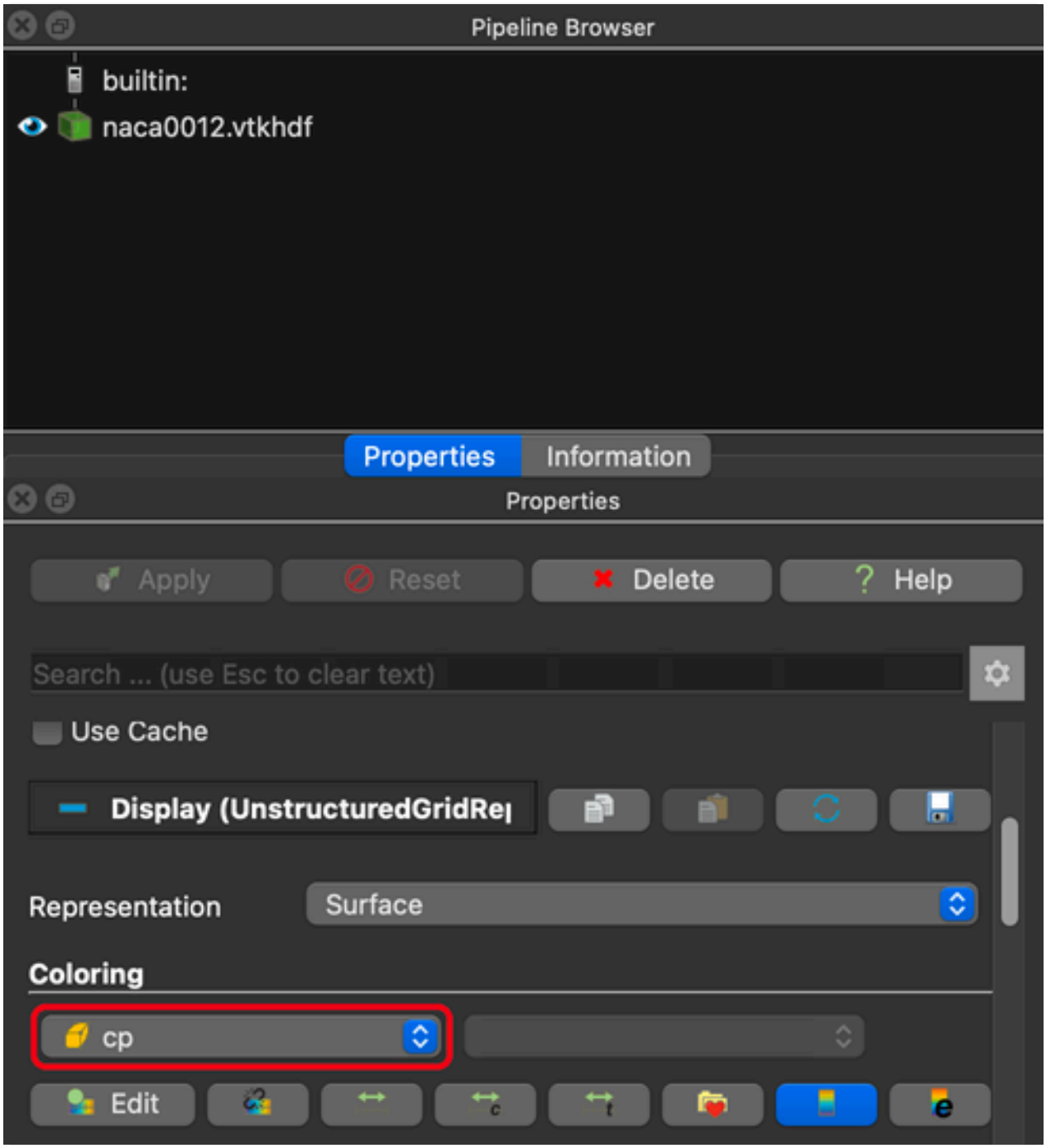
Visualising variables

ParaView colours the mesh by one of the output variables. You can choose which in two places, and both do the same thing: the drop-down on the toolbar,



The colouring drop-down on the toolbar

or the Coloring section of the Properties panel:



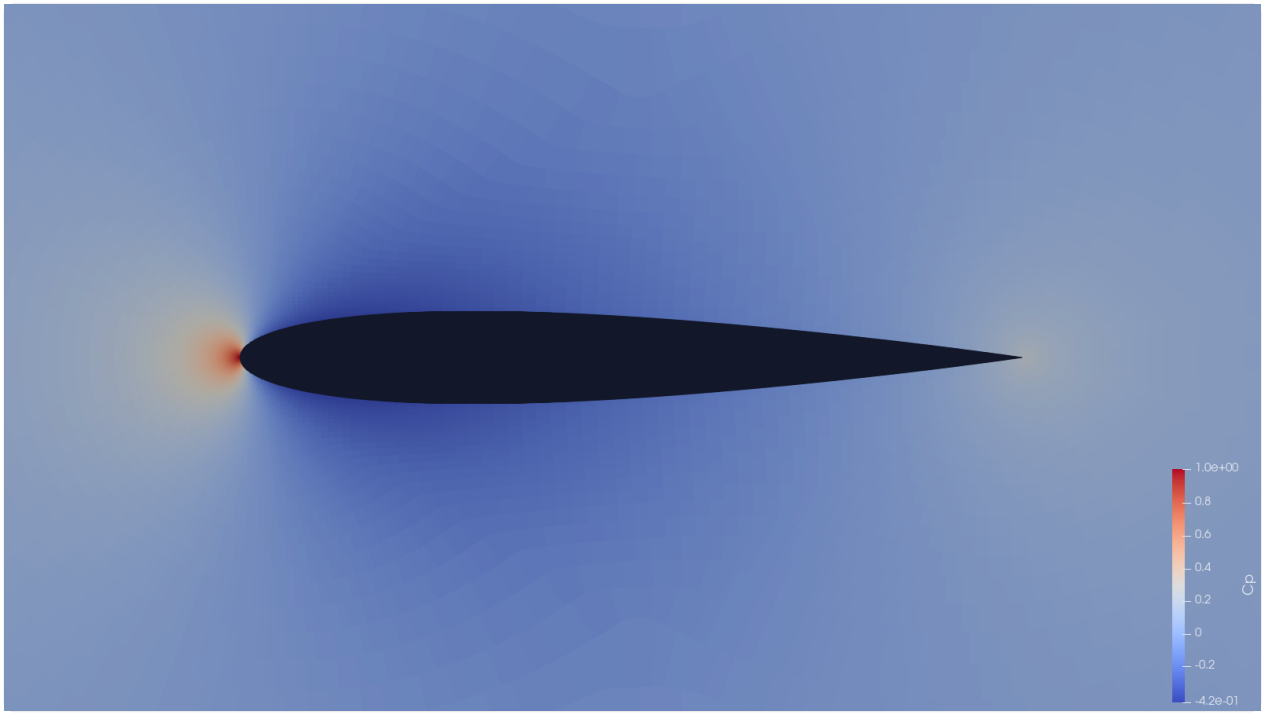
The Coloring drop-down in the Properties panel

The Properties panel holds every setting for whichever object is selected in the Pipeline Browser, so as you get used to ParaView you will find yourself working there more and more.

Select “cp” to colour the mesh by the pressure coefficient. It compares the local pressure with the freestream pressure, scaled by the dynamic pressure of the oncoming flow: it is negative where the flow speeds up and the pressure drops, as it does over the front of the aerofoil, and reaches 1 where the flow comes to rest at the nose.

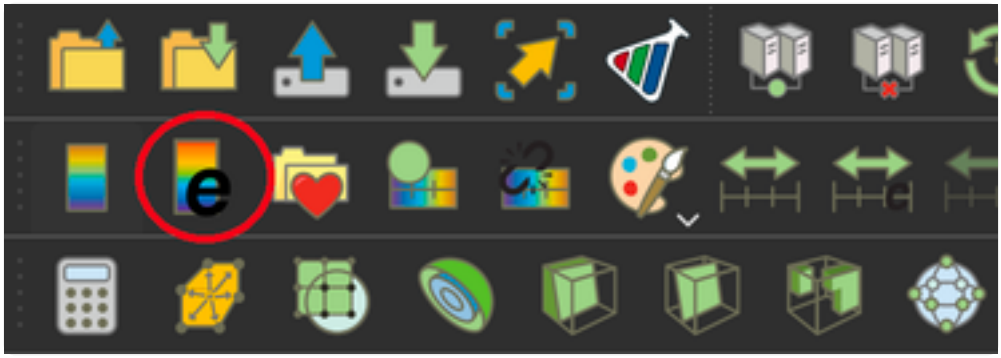
> **NOTE**
>
> The pressure coefficient is given by the following formula $C_p = \frac{p - p_\infty}{\frac{1}{2}\rho_\infty V_\infty^2}$

The aerofoil is small compared with the mesh, so zoom in on the centre to see it:



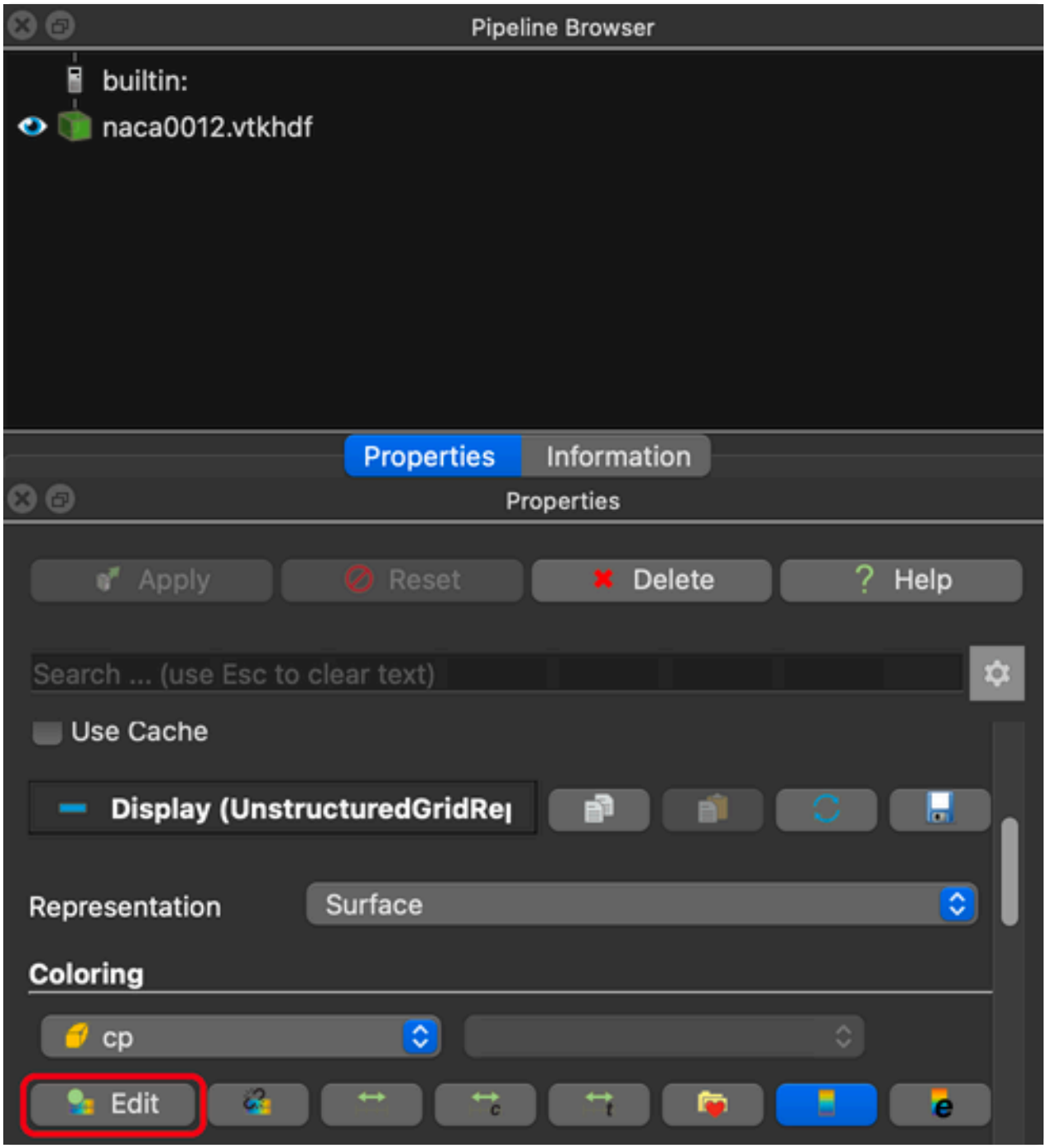
Pressure coefficient around the aerofoil, in ParaView's default colour map

To change the colour map, open the Color Map Editor, again from either place: the Edit Color Map icon on the toolbar, the colour bar marked "e",



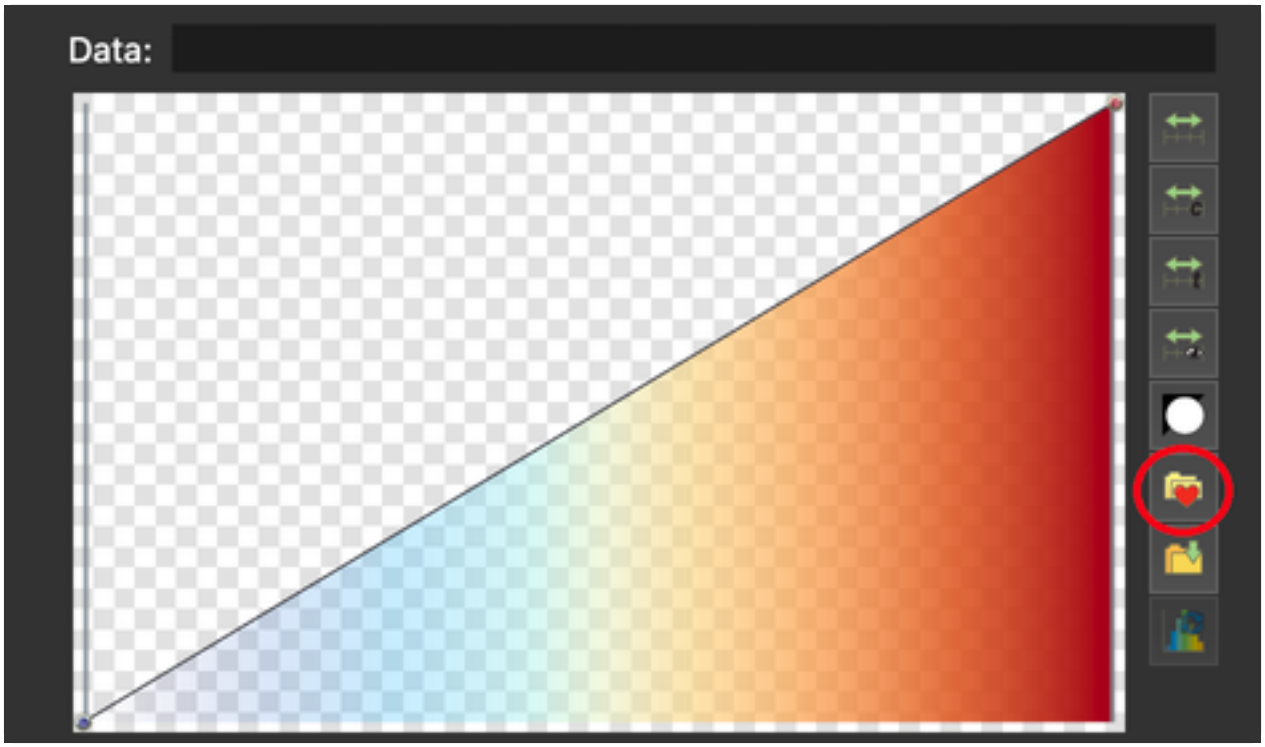
The Edit Color Map icon on the toolbar

or Edit in the Coloring section of the Properties panel:



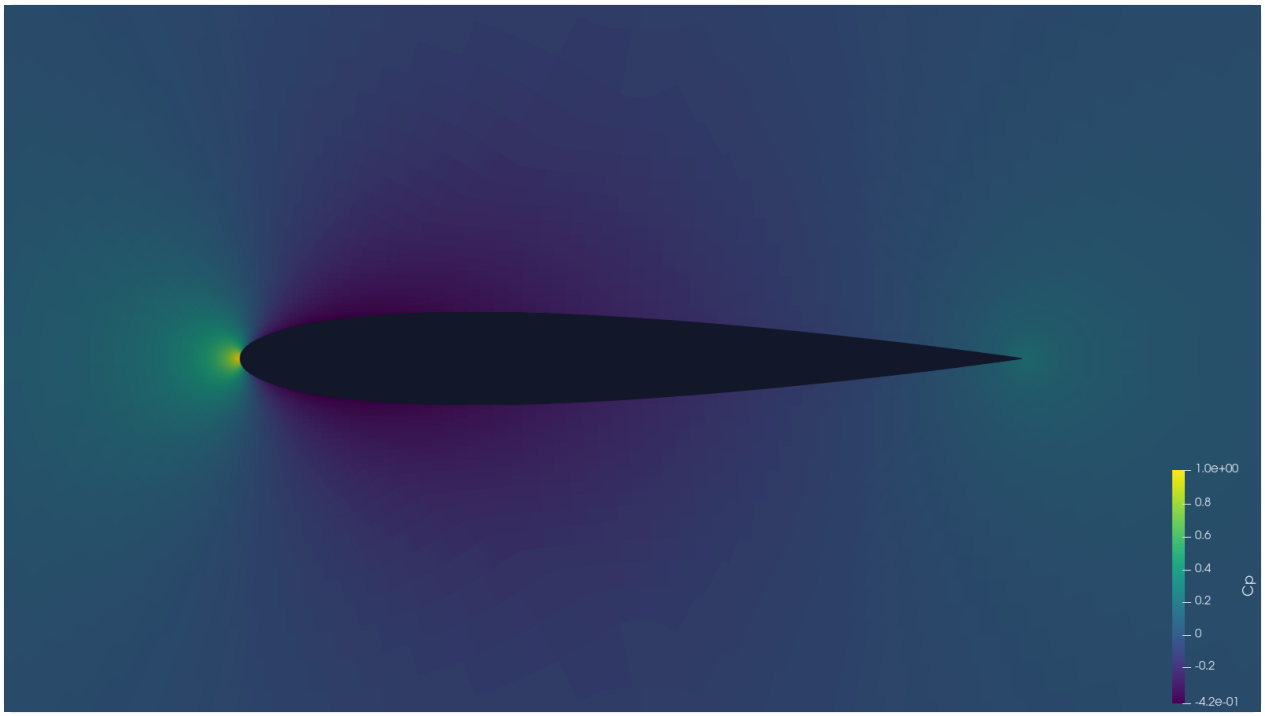
Edit in the Coloring section of the Properties panel

In the Color Map Editor, click Choose Preset:

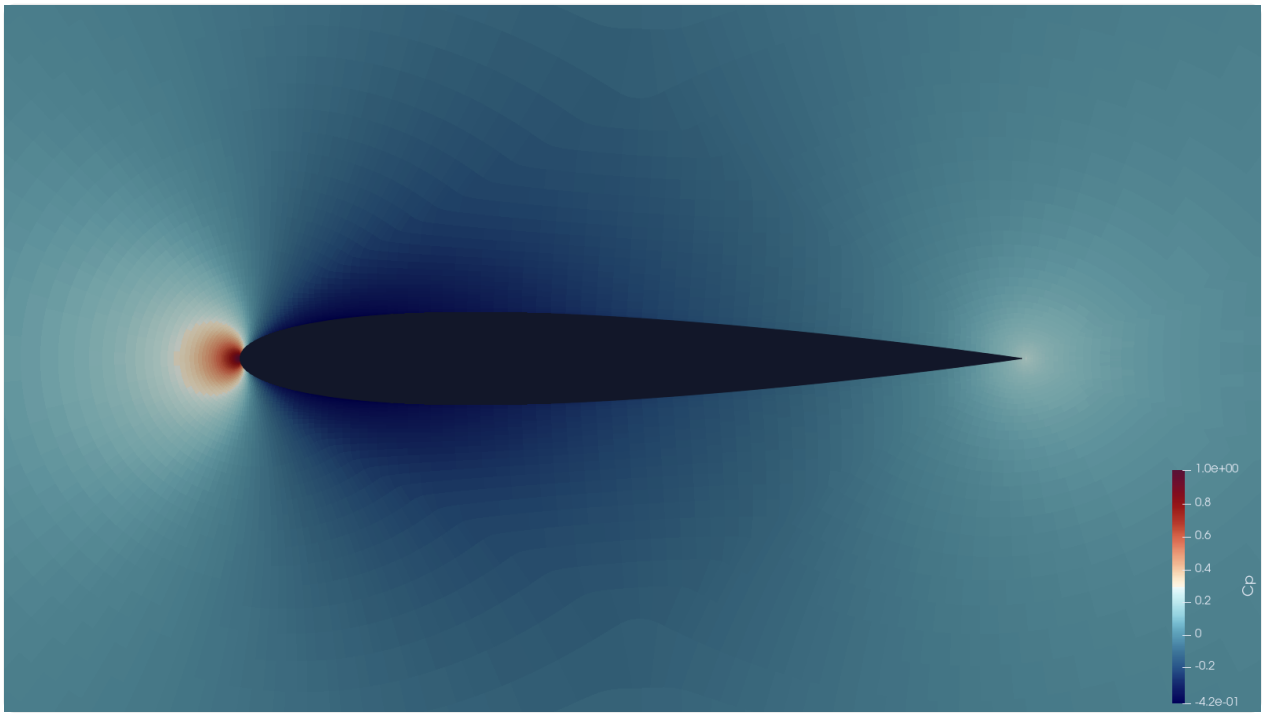


Choose Preset in the Color Map Editor

Here are the same results shown with two other presets, Viridis and Cool to Warm (Extended):



The same view in Viridis

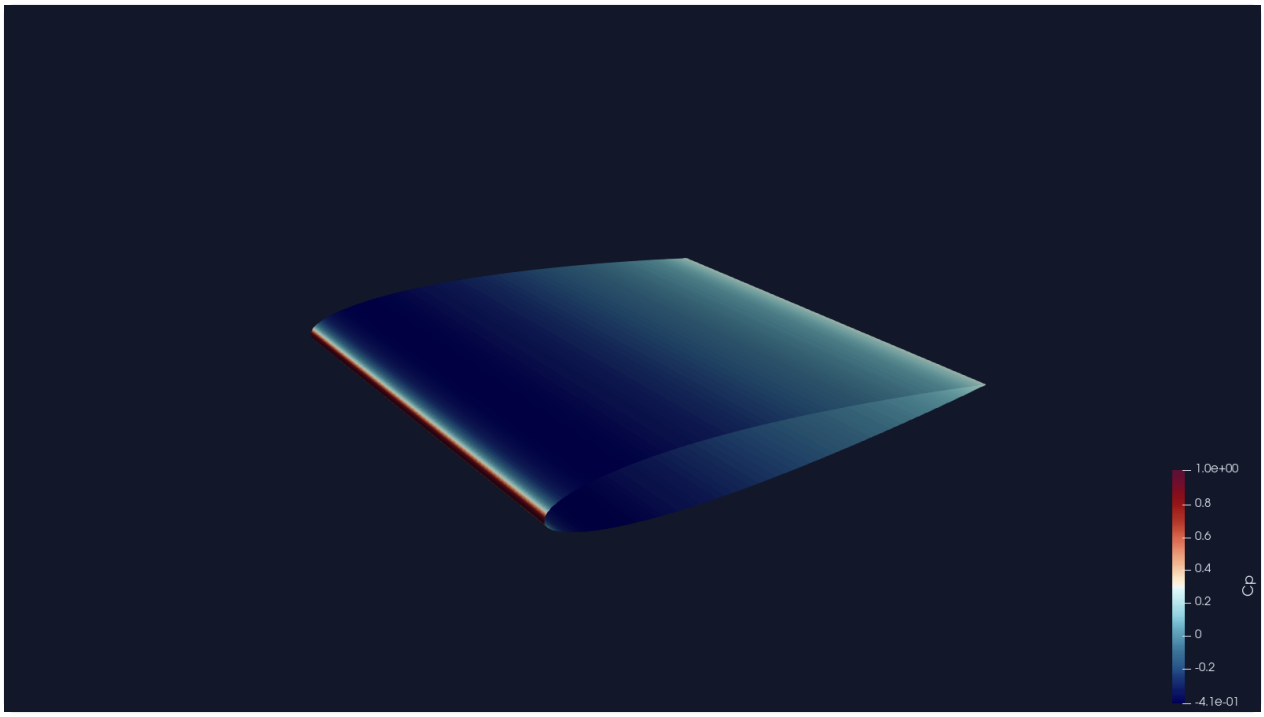


The same view in Cool to Warm (Extended)

A picture shows where the pressure is high or low. To compare with experiments you usually want numbers: most often, the pressure coefficient along the chord, the straight line from the front of the aerofoil to its back.

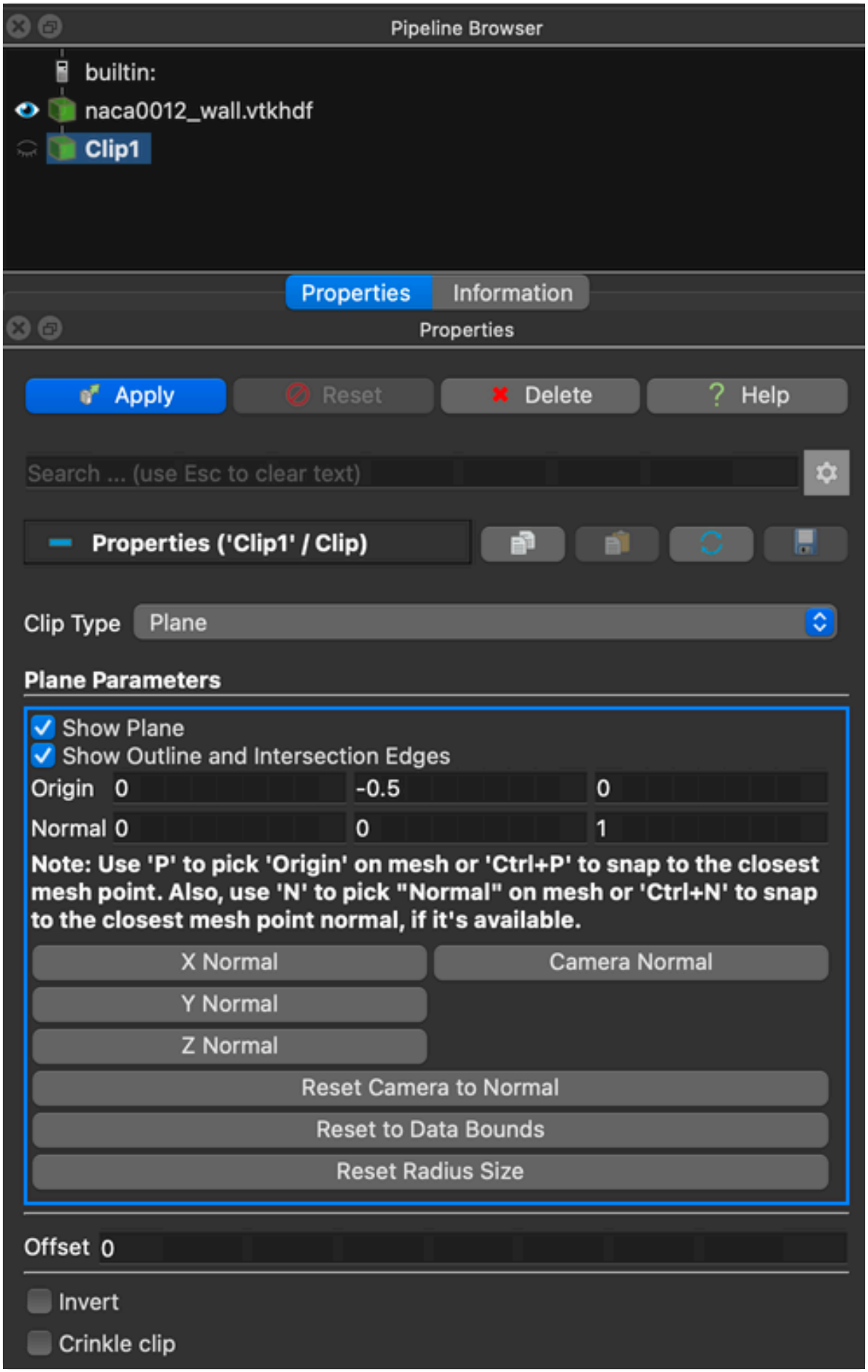
Pressure coefficient plot

Open `naca0012_wall.vtkhdf`. It holds only the boundaries with a `wall` condition, which here is the aerofoil surface.



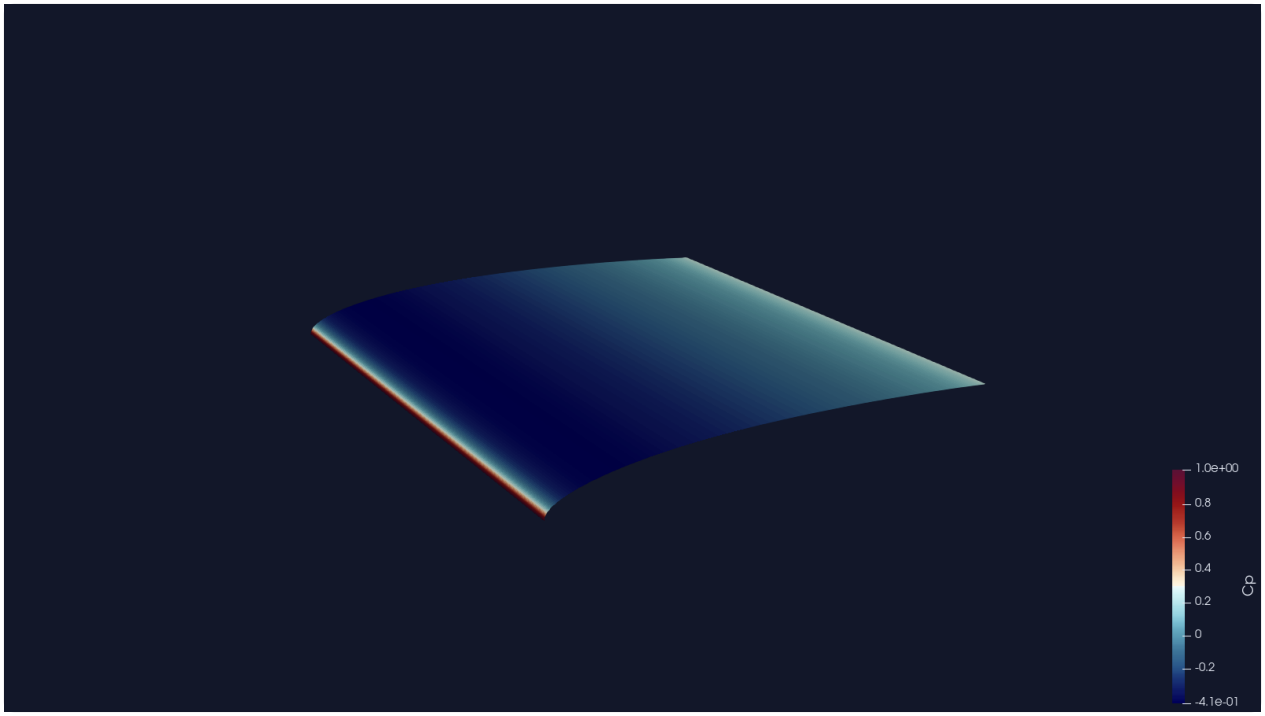
Wall.vtkhdf

The upper and lower surfaces are usually plotted as separate lines, so first separate them with a “Clip” filter.Right-click on the `naca0012_wall.vtkhdf` object in the Pipeline Browser, select Add Filter → Alphabetical → Clip. The properties of the “Clip” filter are displayed on the left. Firstly, untick the “Invert” option and then you should set them to the following combination:



Clip parameters

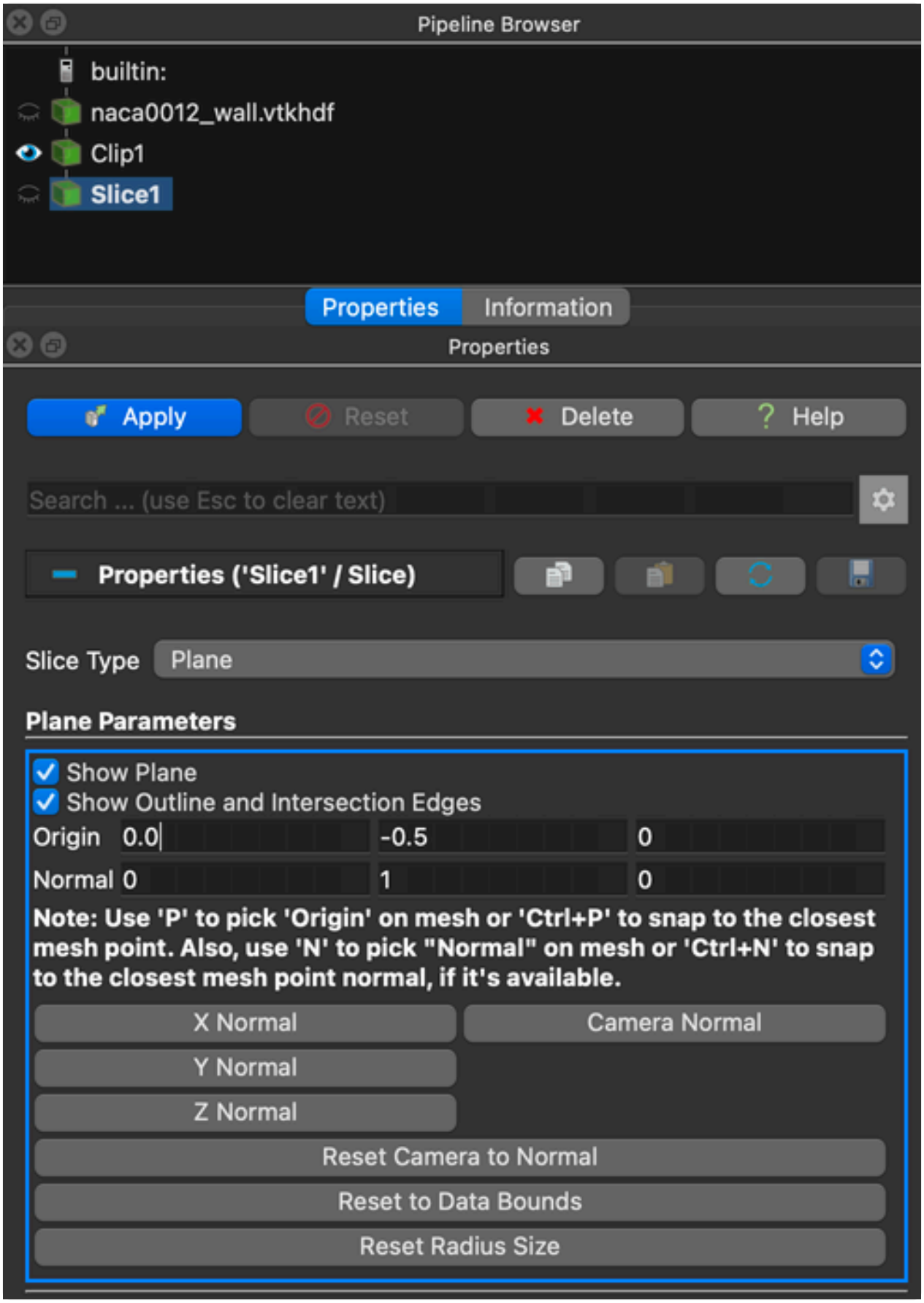
This removes the lower surface, leaving the upper one:



Aerofoil upper surface

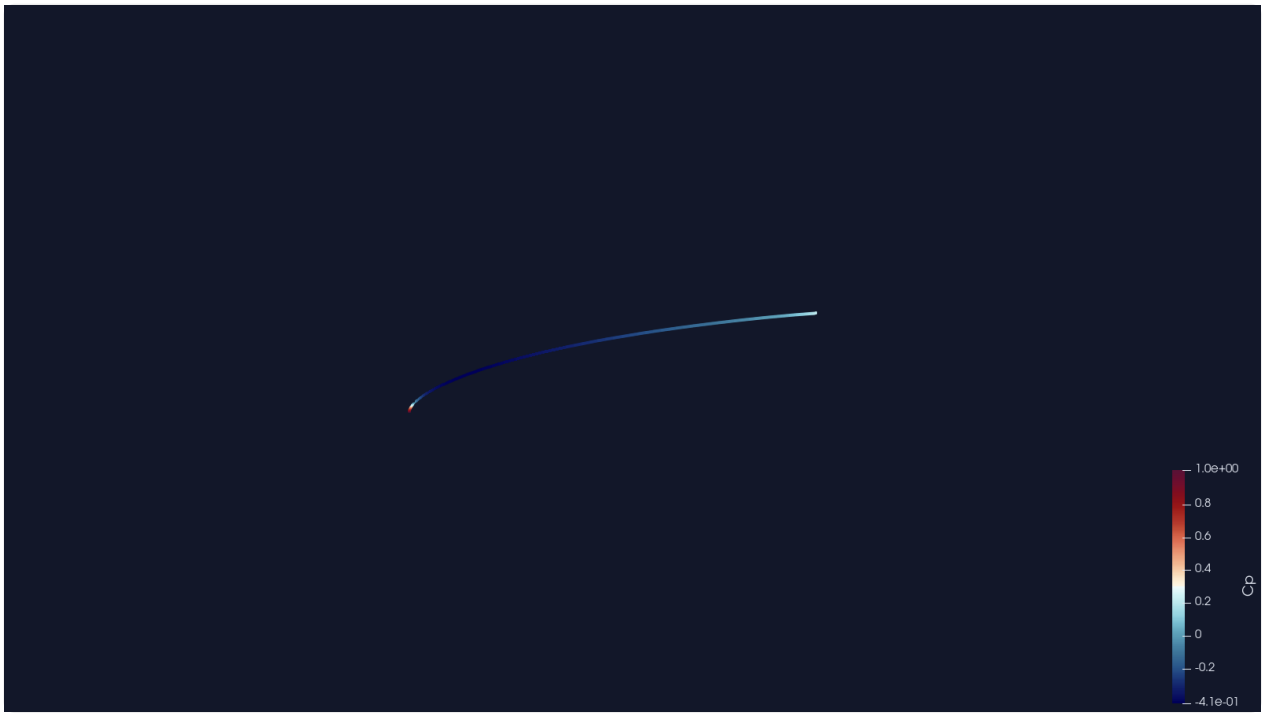
> **NOTE**
>
> To remove the upper surface instead, tick the “Invert” option in the “Clip” filter’s properties.

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



Slice parameters

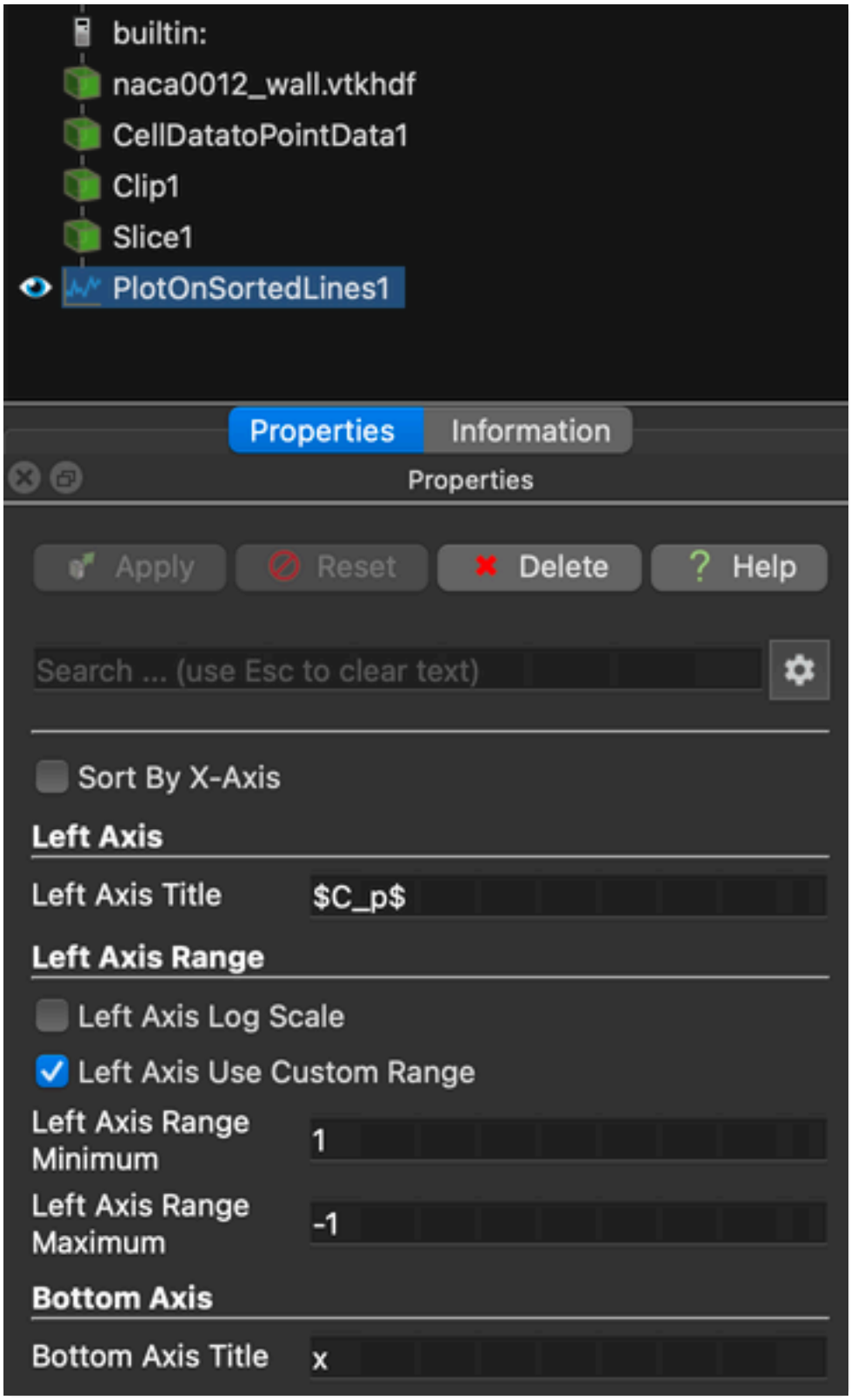
You should end up with this:



Upper aerofoil slice

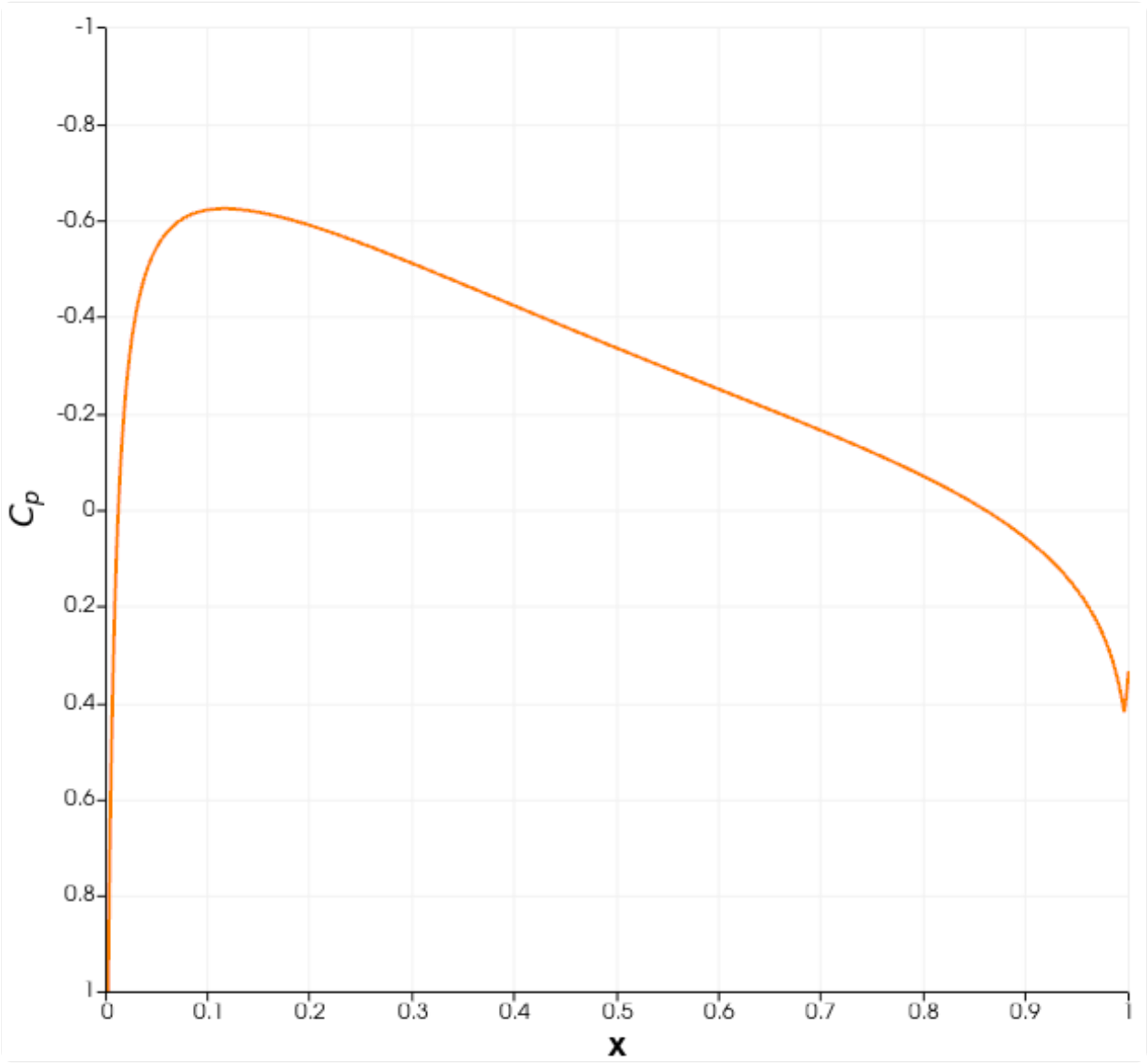
zCFD writes one value per cell face ("cell data"). Line plots need values at points, so convert them with the "CellDatatoPointData" filter: right-click on the "Slice" object and choose it under "Add Filter".

Once this is done, a plot can be produced in ParaView using the "PlotOnSortedLines" filter. Right-click on the "CellDatatoPointData1" object created previously and add the "PlotOnSortedLines" filter. A plot should appear on the right. You can select the variables to plot in the "Properties → Series Parameters" window, as well as changing the settings of the axis. Select the "cp" variable in "Series Parameters', and use `Points_X` as the 'X Array Name'. Untick 'Show Legend', set 'Left Axis Title' to 'C_p' and set the 'Bottom Axis Title' to 'x'. You can also use the"Left Axis Use Custom Range" option to change the y-axis range:



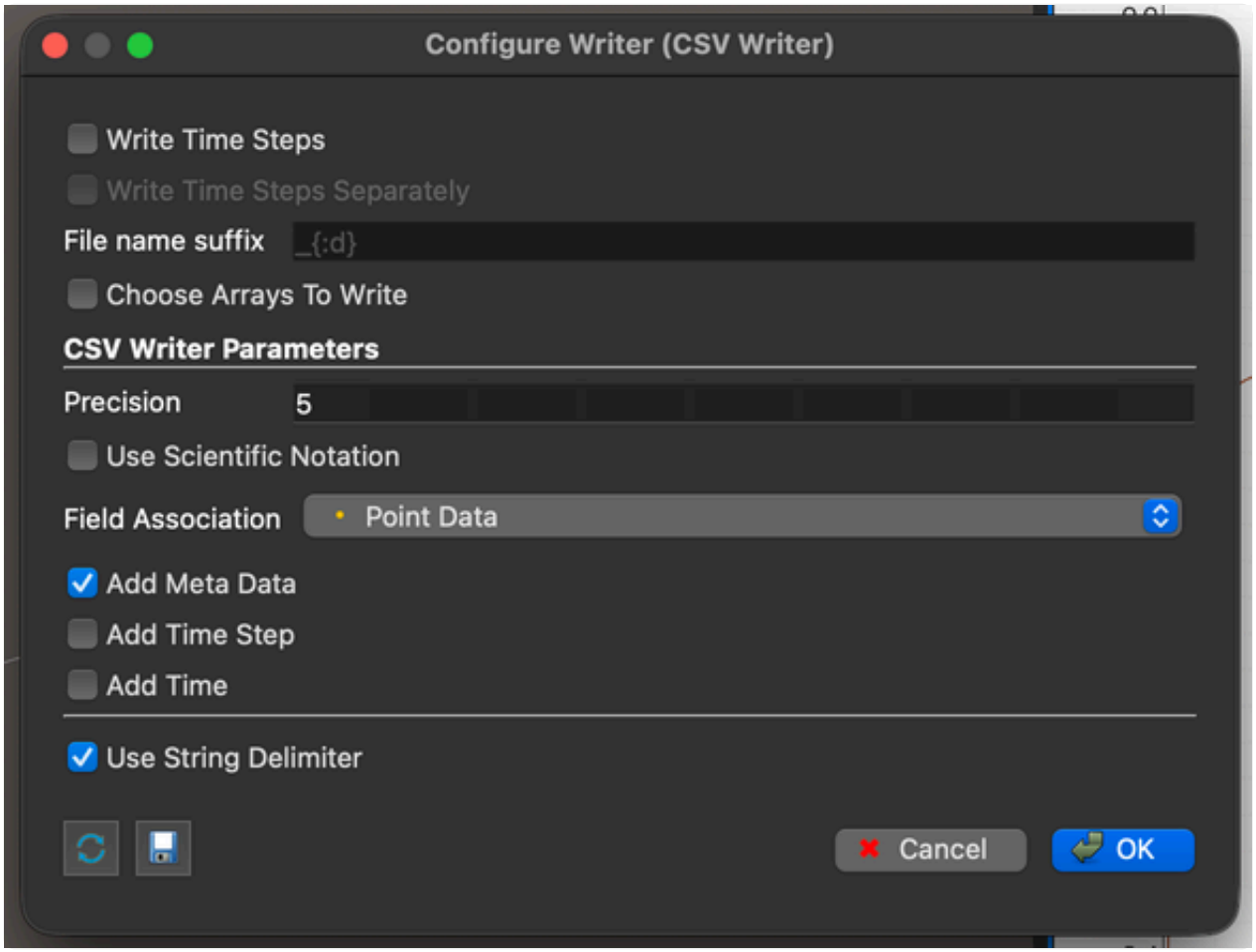
Left axis parameters

Setting the maximum as a negative number inverts the y-axis and the negative numbers will be shown "at the top".



Cp vs Chord

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



CSV Writer tool

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

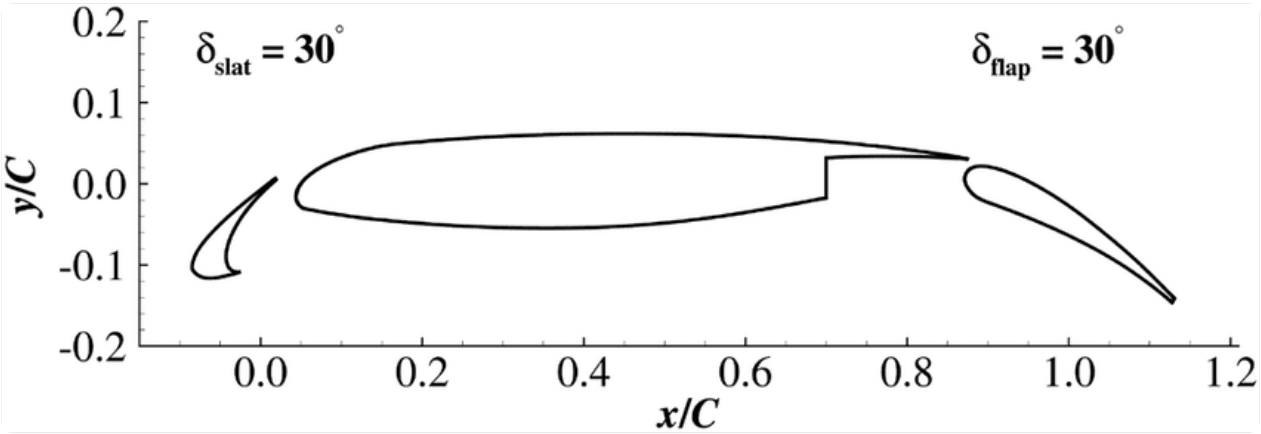
Repeat these steps for the lower surface, with “Invert” ticked in the clip, to plot both surfaces for comparison with experiment.

Steady-state RANS

What is in this tutorial?

Simple aerofoil (p. 373) showed how to run a case in zCFD and look at the results. This tutorial uses a harder case to show more of the solver: the 30P30N, a high-lift aerofoil of the kind a wing uses for take-off and landing. It has three parts: a slat at the front, the main element, and a flap at the back, with gaps between them. It is a well-known test case, first developed to study the noise these wings make [1].

You run it at an angle of attack, the angle between the oncoming flow and the aerofoil's chord, and report the lift (the force across the flow) and the drag (the force along it). The first run stops before the answer has settled, so you then restart it from where it stopped and carry it on.



30P 30N

What you need?

Work through Simple aerofoil (p. 373) first: this tutorial relies on it for activating zCFD, running a case and reading its output. The case has 64,000 cells. On an NVIDIA RTX 3060 the first run takes under two minutes and the restart about a minute.

Step 1. Download the files

The tutorial bundle can be downloaded here. It contains a mesh file (`30p30n.h5`) and a zCFD control file (`30p30n.py`).

```
cd ~
wget https://zcfd.zenotech.com/v2026.11.16174/tutorial_2_high_lift_multi_element_
↪  aerofoil.tgz
tar xzf tutorial_2_high_lift_multi_element_aerofoil.tgz
cd tutorial_2_high_lift_multi_element_aerofoil
```

Step 2. Walk through the input

The control file, `30p30n.py`, is a short Python script. All it does is build one Python dictionary, called `parameters`, that tells zCFD everything about the run: the physics, the numerical method, what to write out, and the mesh with its boundaries. This step walks through that dictionary from top to bottom. Each heading below is the name of a key, written exactly as it appears in the file, and the headings nest the same way the dictionary does: `numerical scheme`, for example, sits inside `solver`. Under each heading you'll find what that part controls, what each setting means, and why it has the value it has here. Each section ends with a link to that part's reference page, which lists every setting it accepts and its default; you don't need it to follow the tutorial. On a wide screen the whole control file sits on the right, with the part being described highlighted as you scroll.

```
import math

import zutil

reynolds = 1.71e6
mach = 0.17
T = 295.56
p = 101325
R = 287.6
gamma = 1.4
reference_length = 0.457
alpha = 5.5

U = math.sqrt(gamma * R * T) * mach

wall = [6]
symmetry = [4, 5]
farfield = [7]


def my_transform(*, vector):
    return zutil.rotate_vector(vector, alpha, 0.0)


parameters = {
    "config version": 2,
    "solver": {
        "solver settings": {"type": "compressible"},
        "equations": {
            "type": "rans",
            "turbulence": {
                "model": "sst",
                "limit mut": False,
                "production": "strain-compressible",
            },
            "precondition": True,
        },
        "fluid properties": {
            "material": "air",
            "gamma": gamma,
```

```
            "gas constant": R,
            "sutherlands const": 110.4,
            "prandtl no": 0.72,
            "turbulent prandtl no": 0.9,
        },
        "reference conditions": {
            "IC_1": {
                "temperature": T,
                "pressure": p,
                "v": {"vector": zutil.vector_from_angle(alpha, 0.0, U)},
                "reynolds no": reynolds,
                "reference length": reference_length,
                "turbulence intensity": 0.01,
                "eddy viscosity ratio": 0.1,
            },
        },
        "initialisation": {"initial conditions": "IC_1", "restart": False},
        "numerical scheme": {
            "order": "euler_second",
            "inviscid flux scheme": "roe",
        },
        "time settings": {"type": "steady"},
        "convergence control": {
            "scheme": {"name": "implicit euler"},
            "cfl": {
                "cfl": 50.0,
                "cfl ramp": [{"type": "exponential", "factor": 1.1, "initial": 1.0}],
            },
            "cycles": 1000,
        },
        "output settings": {
            "report": {
                "frequency": 50,
                "forces": {
                    "FR_1": {
                        "name": "wall",
                        "zones": wall,
                        "transform": my_transform,
                        "reference area": 0.4572,
                    },
                },
            },
            "solution": {
                "frequency": 500,
                "surface variables": ["p", "cp", "V"],
                "volume variables": [
                    "V",
                    "p",
                    "T",
                    "rho",
                    "var_6",
                    "eddy",
                    "turbulencekineticenergy",
```

```
                    "turbulenceeddyfrequency",
                    "walldist",
                ],
            },
        },
    },
    "model": {
        "30p30n": {
            "mesh": "30p30n.h5",
            "transforms": {"scale": [0.0254, 0.0254 / 0.127, 0.0254]},
            "boundary conditions": {
                "BC_1": {"zones": symmetry, "type": "symmetry"},
                "BC_2": {"zones": wall, "type": "wall", "kind": "no slip"},
                "BC_3": {
                    "zones": farfield,
                    "type": "farfield",
                    "condition": "IC_1",
                    "kind": "riemann",
                },
            },
        },
    },
}
```

Unlike the first tutorial's file, this one runs a little ordinary Python before the dictionary, to set the flow conditions once and work out values from them. The first two headings cover that code.

`reynolds`, `mach`, `T`, `p`, `R`, `gamma`, `reference_length`, `alpha`, `U`, `wall`, `symmetry`, `farfield`

The control file is Python, so it can do more than list settings. After importing `math` and `zutil`, its first lines set the key values for the case once, by name, and the dictionary refers to them:

- `reynolds`: 1.71 million, the Reynolds number based on the chord: how strongly the flow's momentum outweighs its viscosity.
- `mach`: 0.17, the freestream Mach number: the flow speed as a fraction of the speed of sound.
- `T` and `p`: 295.56 K and 101325 Pa, the freestream temperature and pressure.
- `R` and `gamma`: the gas constant, 287.6 J/kg/K, and the ratio of specific heats, 1.4.
- `reference_length`: 0.457 m, the aerofoil's chord, which the Reynolds number is based on.
- `alpha`: 5.5°, the angle of attack.
- `U`: the freestream speed, worked out from the Mach number and the speed of sound, $\sqrt{\gamma RT}$: 58.6 m/s.
- `wall`, `symmetry` and `farfield`: the mesh zones that the forces and boundary conditions use.

To run a variation of the case, change one value here, and every setting that uses it follows.

`my_transform`

A function that turns a force from the mesh's axes into the flow's. Lift and drag are defined relative to the freestream, lift normal to it and drag along it. `zutil.rotate_vector(vector, alpha, 0.0)` rotates a vector through the angle of attack, so that its x component is the drag and its z component the lift. The force report below applies it.

`config version`

The version of the control-file format. `2` is the current format.

`solver`

Everything shared by the whole simulation.

`solver settings`

Which solver runs the case. Reference (p. 11).

- `type`: `"compressible"`, the density-based solver.

`equations`

Which equations are solved. Reference (p. 18).

- `type`: `"rans"`, the Reynolds-averaged Navier–Stokes equations, which solve for the average flow and use a turbulence model for the effect of turbulent eddies, as in Simple aerofoil (p. 376).
- `turbulence`: the turbulence model and its options.
  - `model`: `"sst"`, Menter's SST model. The mesh is fine enough next to the surfaces to resolve the boundary layer all the way down to the wall.
  - `limit mut`: `False` switches off the SST model's limit on the eddy viscosity, which is on by default. The eddy viscosity is the extra viscosity the turbulence model adds to represent the mixing done by turbulent eddies.
  - `production`: `"strain-compressible"` sets how the model works out how fast turbulence is generated: from how quickly the flow is being stretched (the strain rate), instead of the default, from how fast it is spinning (the vorticity).
- `precondition`: `True` switches on low-Mach-number preconditioning, recommended below about Mach 0.3.

Choosing a turbulence model (p. 226) compares the models and options.

`fluid properties`

The gas the flow is made of. Reference (p. 37).

- `material`: `"air"`.
- `gamma` and `gas constant`: taken from the reference variables.
- `sutherlands const`: 110.4 K, for Sutherland's law of viscosity.
- `prandtl no` and `turbulent prandtl no`: 0.72 and 0.9, which relate heat conduction to viscosity in the laminar flow and the turbulence.

`reference conditions`

Named flow states, which the initialisation and the boundary conditions refer to. Reference (p. 39).

`IC_1` The freestream, built from the reference variables.

- `temperature` and `pressure`: 295.56 K and 101325 Pa.
- `v`: the velocity. Its `vector` comes from `zutil.vector_from_angle(alpha, beta, U)`, which gives the velocity for an angle of attack `alpha` and a sideslip `beta`, in degrees, at speed `U`. Following the ESDU convention for aerodynamic axes, the angle of attack turns the flow from x towards z, so the flow

meets the aerofoil at 5.5°, with no sideslip. The aerofoil stays where it is in the mesh, and the flow is turned instead.
- `reynolds no` and `reference length`: a Reynolds number of 1.71 million on the 0.457 m chord. zCFD works out the viscosity from them.
- `turbulence intensity`: 0.01, so the velocity fluctuations in the incoming flow are 1% of its speed.
- `eddy viscosity ratio`: 0.1, the ratio of eddy viscosity to molecular viscosity in the incoming flow. Together with the intensity, it sets the turbulence the SST model starts from.

`initialisation`

How the flow field starts. Reference (p. 47).

- `initial conditions`: `"IC_1"`, the freestream everywhere.
- `restart`: `False`, a fresh start. Step 3 turns it on to carry the run on.

`numerical scheme`

How the equations are discretised in space. Reference (p. 74).

- `order`: `"euler_second"`, second order for the flow variables and first order for the turbulence variables, with the default limiter, as in Simple aerofoil (p. 377).
- `inviscid flux scheme`: `"roe"`, Roe's scheme for the flow between cells.

`time settings`

Reference (p. 51).

- `type`: `"steady"`, a solution that does not change with time.

`convergence control`

How the solver marches towards the steady solution, once per cycle. Reference (p. 56).

- `scheme`: `implicit euler`, which solves a linear system each cycle and allows large steps.
- `cfl`: the CFL number ramps from 1 by a factor of 1.1 each cycle up to 50.
- `cycles`: 1000. As Step 4 shows, that is not quite enough, and Step 3 carries the run on with a restart.

`output settings`

What zCFD writes, and how often. Reference (p. 84).

`report` The report is the CSV file `30p30n_report.csv`: one row of residuals and forces at a time, which zMon plots.

- `frequency`: 50, a row every 50 cycles.
- `forces`: force blocks, each summing the pressure and friction forces on a set of zones. This deck has one, `FR_1`:
  - `name`: `"wall"`. The report names the block's columns after it: `wall_Fx`, `wall_Fy` and `wall_Fz` are the forces in mesh axes, split into pressure (`wall_Fpx`, …) and friction (`wall_Ffx`, …) parts, and `wall_Mx` and so on are the moments.
  - `zones`: the `wall` zones from the top of the file, zone 6: the slat, main element and flap together.
  - `transform`: `my_transform`. The report gains the transformed forces and moments, in columns starting `wall_Ft` and `wall_Mt`: `wall_Ftx` is the drag and `wall_Ftz` the lift.

  - `reference area`: 0.4572 m², the chord times the 1 m span. Forces are reported as coefficients against `IC_1` and this area, so `wall_Ftx` and `wall_Ftz` are the drag and lift coefficients.

`solution` The flow fields written for viewing in ParaView.

- `frequency`: 500. In a steady run, zCFD writes the fields every 500 cycles, each time replacing the last, so a run that stops early still leaves a recent solution.
- `surface variables`: on the boundaries: pressure `p`, pressure coefficient `cp` and velocity `V`.
- `volume variables`: in every cell: velocity, pressure, temperature and density, the sixth solved variable `var_6` (ρk), the eddy viscosity `eddy`, the turbulence kinetic energy and eddy frequency, and `walldist`, the distance to the nearest wall, which the SST model uses.

Output variables (p. 336) lists every variable.

`model`

One entry per mesh, each with its own boundary conditions. Reference (p. 96).

`30p30n`

The only model. Its name, `30p30n`, names its output folder, `30p30n_OUTPUT/30p30n/`, and its output files.

`mesh`

- `"30p30n.h5"`, a zCFD mesh with 64,000 cells and a single cell across the span. It follows the ESDU convention for aerodynamic axes: x points downstream along the chord, z points up and y runs along the span. It was built in inches.

`transforms` Changes made to the mesh as it is read, without editing the file. Reference (p. 169).

- `scale`: multiplies the x, y and z coordinates of the file. 0.0254 turns inches into metres. The y factor, 0.0254 / 0.127, stretches the 5 inch span to 1 m: there is no spanwise flow, and a 1 m span makes the reported forces per unit span.

Placing and moving a mesh (p. 365) describes the other ways to place a mesh.

`boundary conditions` A condition for each group of boundary faces, named by the zone lists at the top of the file. Choosing boundary conditions (p. 285) describes every type.

`BC_1`

- `zones`: `symmetry`, zones 4 and 5, the two faces normal to the span.
- `type`: `"symmetry"`, so the flow behaves as if the aerofoil were infinitely long and stays 2D.

`BC_2`

- `zones`: `wall`, zone 6, the surfaces of the slat, main element and flap.
- `type`: `"wall"`, with `kind "no slip"`: the flow velocity is zero at the surface.

BC_3

- `zones`: `farfield`, zone 7, the outer boundary.
- `type`: `"farfield"`, with `condition` `"IC_1"`, the freestream outside.
- `kind`: `"riemann"`, which passes waves out without reflecting them back into the domain.

Step 3. Run

Activate zCFD as in Simple aerofoil (p. 380), then check and run the case:

```
cd ~/tutorial_2_high_lift_multi_element_aerofoil/
validate_input 30p30n.py
run_zcfd -c 30p30n.py
```

These are extracts from the log of a run of this tutorial, with `...` marking lines left out:

```
Parameters validated successfully
...
 Reading file 30p30n.h5
 Global Mesh Stats: Cells: 63957 Faces: 256413
...
  Viscosity:    1.86824e-05 Ns/m^2 @ 295.56 K
  Speed:        58.6449 m/s
...
  Mach:         0.17
  Reynolds No:  1.71e+06 @ 295.56 K
...
Starting solve loop
...
Cycle     50 | Step    0 t=0s | CFL  50.000 | MG 1 |    39.9 ms | Total     1.328s
+-----------------------------------------------------------------------------------------+
| Report: Cycle 50  Wall time: 1.328s                                                     |
...
| Residuals:                                                                              |
|   rho       5.0013E-01  rhoV[0]   1.0090E+00  rhoV[1]   1.7333E-11  rhoV[2]   8.5874E-01 |
|   rhoE      1.7070E+00  rhok      4.4082E-07  rhoOmega  1.4652E+05                      |
+-----------------------------------------------------------------------------------------+
...
 Solve Complete: 1000 cycles in 36.943s (avg 36.9 ms/cycle)
...
zCFD finished: all requested cycles are complete (1000 in total).
```

The lines are the same as in the first tutorial. The viscosity is worked out from the Reynolds number, and the speed, 58.6 m/s, from the Mach number. In the report box every 50 cycles, `rhoV[1]`, the momentum along the span (y), stays below 10⁻¹⁰: with a single cell across the span, no flow develops in that direction.

Restarting the run

Step 4 shows that after 1000 cycles the residuals are still falling. Rather than start again, restart the run from where it stopped. In `30p30n.py`, turn on `restart` under `initialisation`, and raise `cycles` under

`convergence control` to 2000, the total the run should reach:

```
"initialisation": {"initial conditions": "IC_1", "restart": True},
```

```
"cycles": 2000,
```

With `restart` on, zCFD reads the checkpoint the previous run of the same control file wrote, `30p30n_OUTPUT/30p30n/30p30n_results.h5`, and carries on from cycle 1001. Run it with the same command:

```
run_zcfd -c 30p30n.py
```

The log says where it restarted from:

```
 Restarting solution from results file 30p30n_OUTPUT/30p30n/30p30n
...
Cycle   1001 | Step    0 t=0s | CFL   1.100 | MG 1 |    97.1 ms | Total     0.097s
...
Cycle   2000 | Step    0 t=0s | CFL  50.000 | MG 1 |    23.1 ms | Total    25.001s
+-----------------------------------------------------------------------------------------+
| Report: Cycle 2000  Wall time: 25.001s                                                  |
...
| Residuals:                                                                              |
|   rho       4.4183E-04  rhoV[0]   5.1424E-03  rhoV[1]   1.6681E-14  rhoV[2]   1.3771E-03 |
|   rhoE      1.0785E-03  rhok      8.6709E-06  rhoOmega  1.5141E+03                      |
+-----------------------------------------------------------------------------------------+
...
zCFD finished: all requested cycles are complete (2000 in total).
```

The CFL ramp starts again from its initial value, so the first restarted cycles take small steps. Choosing initialisation and reference state (p. 293) covers restarts in full.

Output files

The output follows the same pattern as the first tutorial, named after the control file and the model. The restarted run appends to the same report.

FILE/DIRECTORY	DESCRIPTION
`30p30n.log`	A copy of the output to screen during the run.
`30p30n_status.yaml`	A YAML file recording how the run was started.
`30p30n_OUTPUT/30p30n/30p30n_report.csv`	The report: residuals and forces, including the `wall_Ftx` and `wall_Ftz` columns.
`30p30n_OUTPUT/30p30n/30p30n_results.h5`	The checkpoint, which the restart reads.
`30p30n_OUTPUT/30p30n/30p30n.vtkhdf`	The volume data.
`30p30n_OUTPUT/30p30n/30p30n_wall.vtkhdf` etc	One file for each boundary condition type, with the surface data.

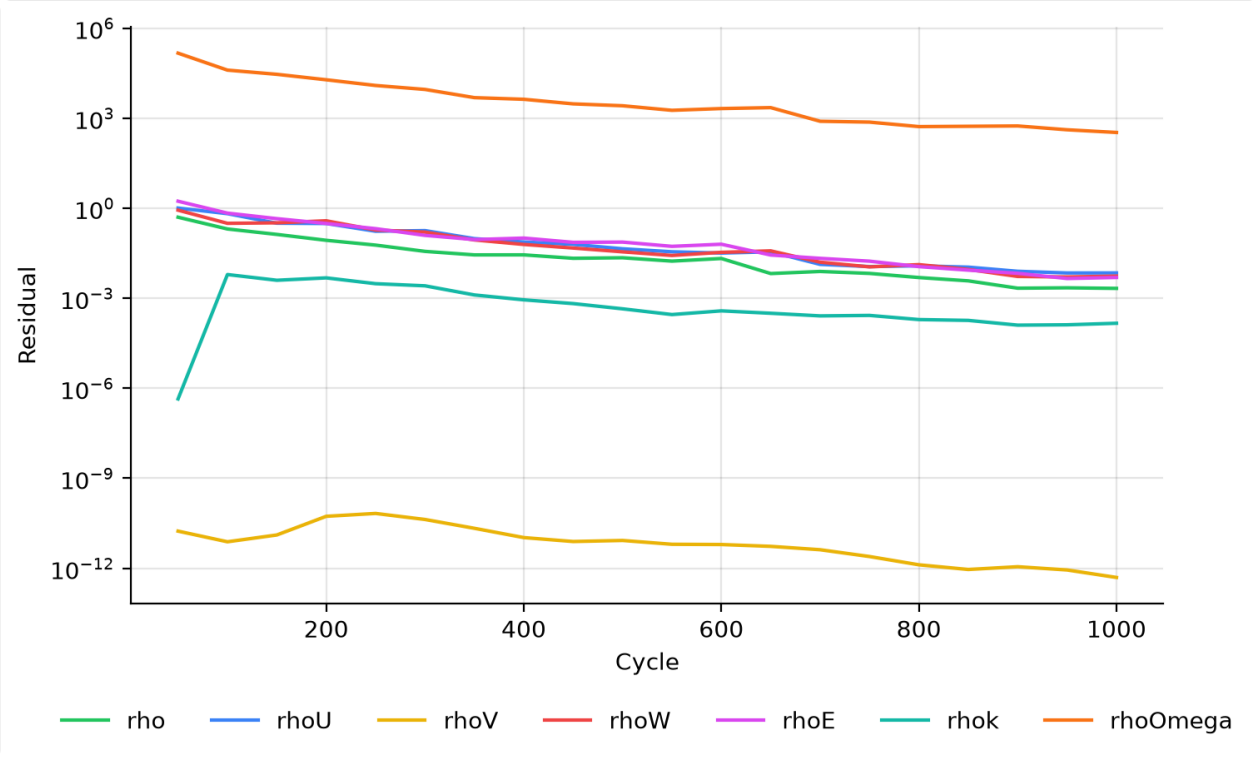
Step 4. Monitor

Open the case in zMon to follow the residuals and forces as the solver runs:

```
zmon 30p30n.py
```

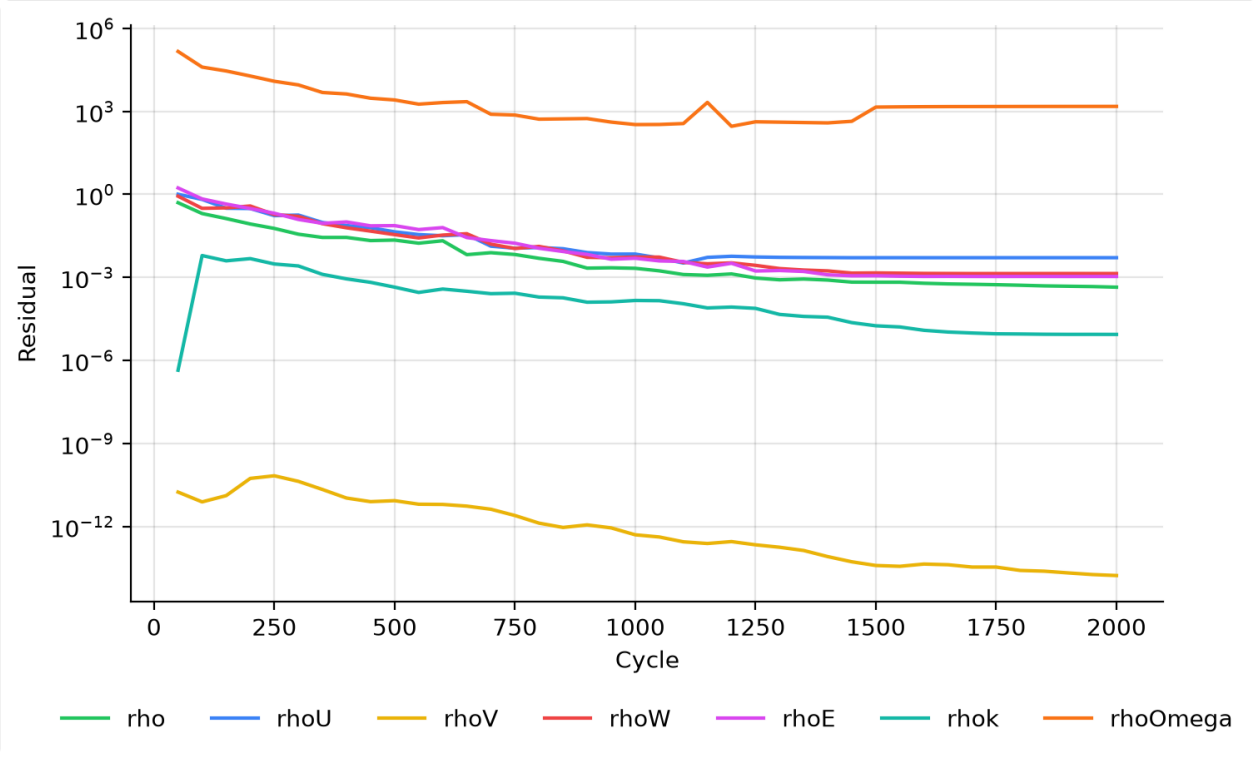
zMon follows the restarted run too, which appends to the same report. The plots below are the panels as zMon draws them; Step 5 shows how to plot the same data yourself.

Residuals



Residuals over the first 1000 cycles

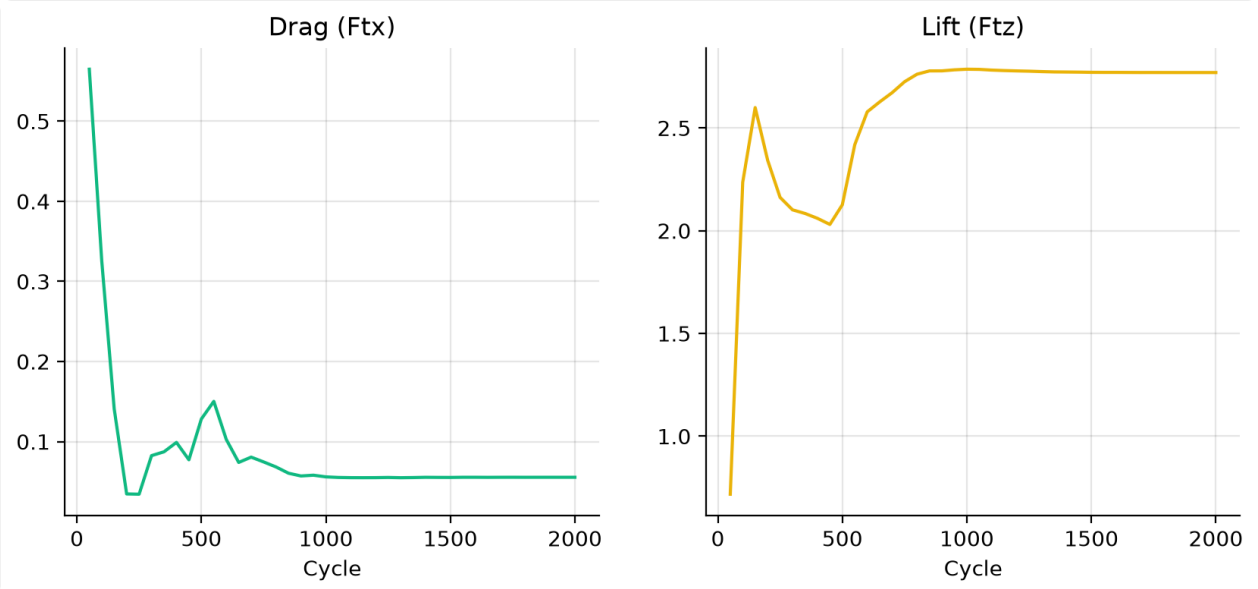
After the first 1000 cycles the residuals are still falling, which is why the run is restarted. By the end of the restarted run they have levelled off:



Residuals over the whole 2000 cycles, after the restart

Forces

The forces of interest are the transformed ones, `wall_Ftx` (drag) and `wall_Ftz` (lift), in zMon's forces panel:



Drag (Ftx) and lift (Ftz) against cycle

Both have converged by the end of the restarted run, with a lift coefficient of 2.77 and a drag coefficient of 0.055. The grid for this case is still extremely coarse, so these lift and drag coefficients are not a validation of the 30P30N.

Step 5. Post-process

First activate your zCFD environment, which provides Python with zutil, h5py and ParaView:

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

Python

This script draws the plots in Step 4 and prints the final lift and drag coefficients. It is the first tutorial's script with the transformed force columns, `wall_Ftx` and `wall_Ftz`. Save it as `plot_convergence.py` in the case directory:

```
import matplotlib.pyplot as plt

from zutil.fileutils import get_zcfd_result

report = get_zcfd_result("30p30n.py").report.data

fig, ax = plt.subplots()
for column in ["rho", "rhoV[0]", "rhoV[1]", "rhoV[2]", "rhoE", "rhok", "rhoOmega"]:
    ax.semilogy(report["Cycle"], report[column], label=column)
ax.set_xlabel("Cycle")
ax.set_ylabel("Residual")
ax.legend()
fig.savefig("residuals.png", dpi=200)

fig, (drag, lift) = plt.subplots(1, 2, figsize=(10, 4))
drag.plot(report["Cycle"], report["wall_Ftx"])
drag.set_title("Drag coefficient (wall_Ftx)")
lift.plot(report["Cycle"], report["wall_Ftz"])
lift.set_title("Lift coefficient (wall_Ftz)")
for ax in (drag, lift):
    ax.set_xlabel("Cycle")
fig.savefig("forces.png", dpi=200)

final = report.iloc[-1]
print(f"Lift coefficient {final['wall_Ftz']:.2f}, drag coefficient {final['wall_Ftx']:.3f}")
```

Run it:

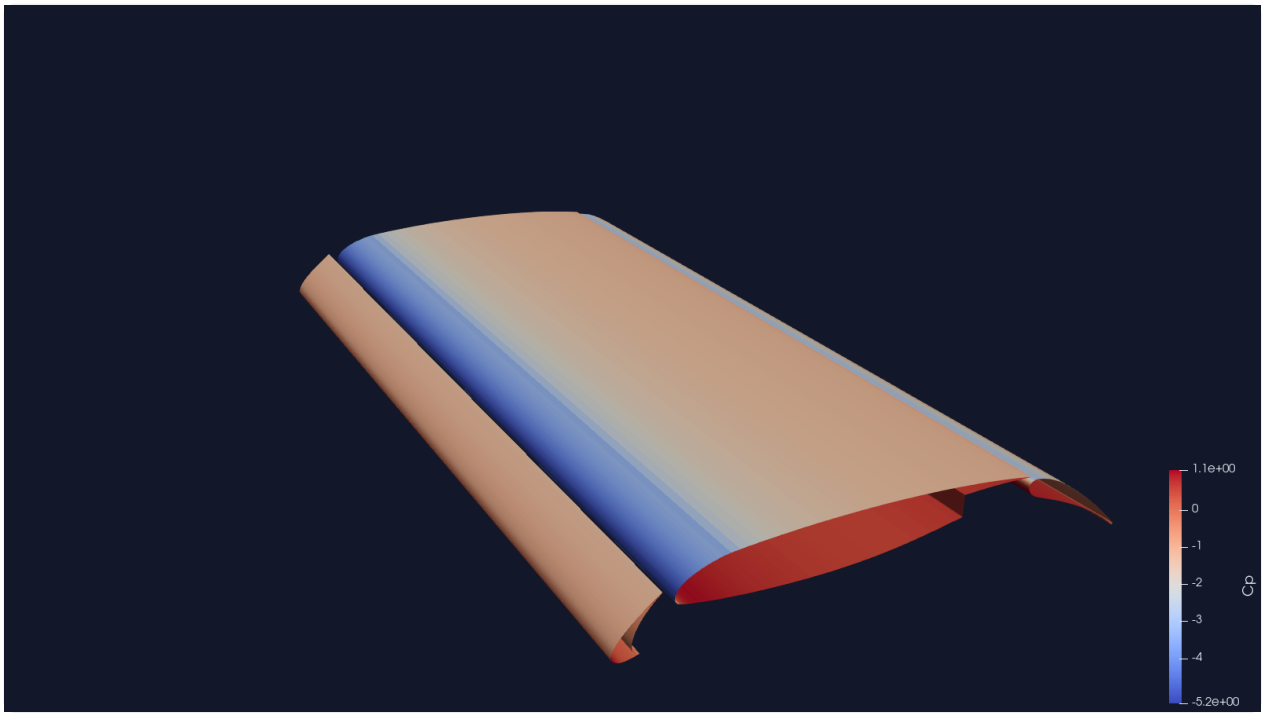
```
python plot_convergence.py
```

It writes `residuals.png` and `forces.png`, and prints:

```
Lift coefficient 2.77, drag coefficient 0.055
```

ParaView

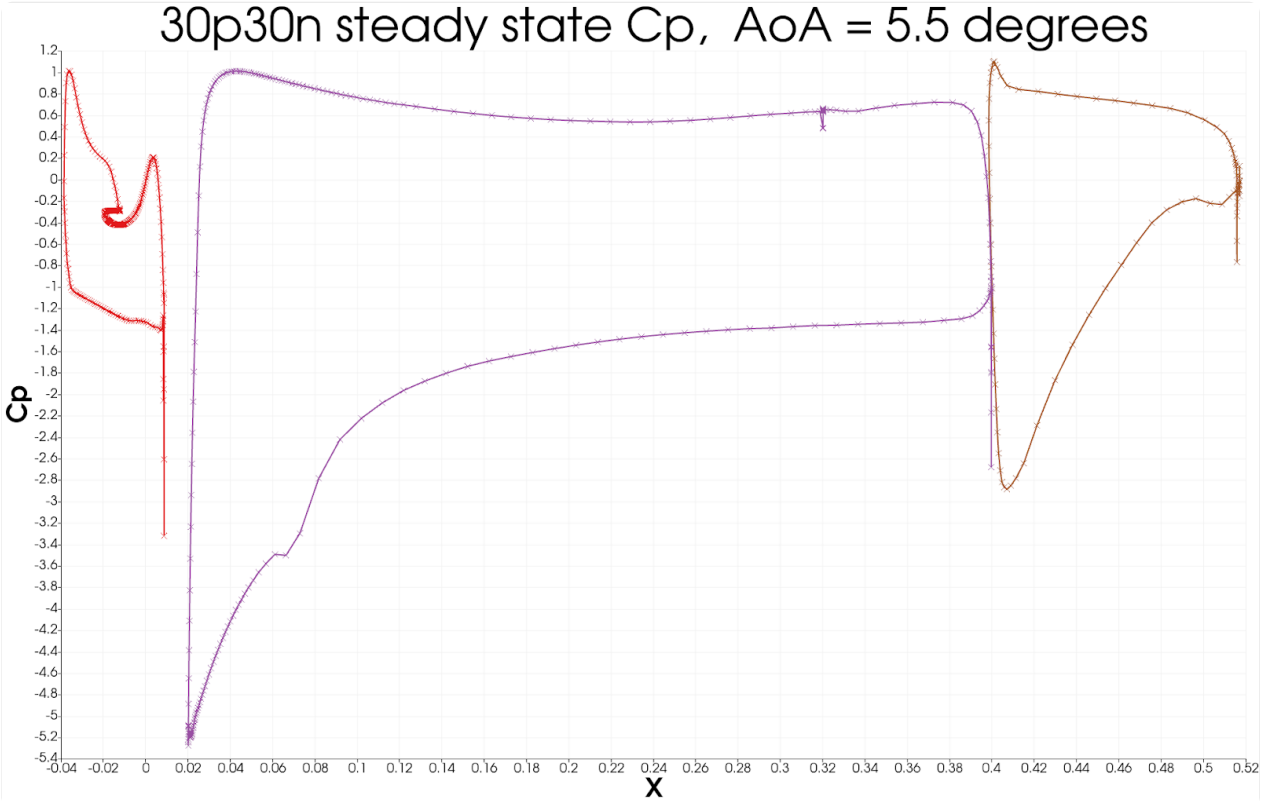
Open `30p30n_OUTPUT/30p30n/30p30n_wall.vtkhdf` in ParaView to see the results on the aerofoil surfaces, here coloured by the pressure coefficient:



30p30n C_p shaded

To plot the pressure coefficient along the chord (x/c, the distance from the front divided by the chord), use a "Plot On Sorted Lines" filter:

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



cp against x/c

Citations

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

Time-accurate solution

What is in this tutorial?

Some flows never settle to a steady state. Behind a cylinder, swirling vortices peel off each side in turn and are carried away downstream, and the flow repeats over and over at a definite frequency. This is vortex shedding, the same effect that makes wires hum in the wind.

The previous tutorials looked for a steady answer. Here the answer changes with time, so the run has to follow it step by step in real time: a time-accurate run. The flow is a cylinder in a slow, smooth (laminar) flow at a Reynolds number of 150. You record the pressure at one point in the wake and measure the shedding frequency from it.



Pressure coefficient round the cylinder and in its wake at 1.6 s

What you need?

Work through Simple aerofoil (p. 373) first: this tutorial relies on it for activating zCFD, running a case and reading its output. The run takes about four minutes on an NVIDIA RTX 3060.

Step 1. Download the files

The tutorial bundle can be downloaded here. It contains a mesh file (`cylinder.h5`) and a zCFD control file (`cylinder.py`).

```
cd ~
wget https://zcfd.zenotech.com/v2026.11.16174/tutorial_3_cylinder_shedding.tgz
tar xzf tutorial_3_cylinder_shedding.tgz
cd tutorial_3_cylinder_shedding
```

Step 2. Walk through the input

The control file, `cylinder.py`, is a short Python script. All it does is build one Python dictionary, called `parameters`, that tells zCFD everything about the run: the physics, the numerical method, what to write out, and the mesh with its boundaries. This step walks through that dictionary from top to bottom. Each heading below is the name of a key, written exactly as it appears in the file, and the headings nest the same way the dictionary does: `numerical scheme`, for example, sits inside `solver`. Under each heading you'll find what that part controls, what each setting means, and why it has the value it has here. Each section ends with a link to that part's reference page, which lists every setting it accepts and its default; you don't need it to follow the tutorial. On a wide screen the whole control file sits on the right, with the part being described highlighted as you scroll.

```
parameters = {
    "config version": 2,
    "solver": {
        "solver settings": {"type": "compressible"},
        "equations": {
            "type": "viscous",
            "precondition": True,
        },
        "fluid properties": {
            "material": "air",
            "gamma": 1.4,
            "gas constant": 287.0,
            "sutherlands const": 110.4,
            "prandtl no": 0.72,
        },
        "reference conditions": {
            "IC_1": {
                "temperature": 277.7777,
                "pressure": 101325.0,
                "v": {"vector": [0.0, 1.0, 0.0], "mach": 0.2},
                "reynolds no": 150,
                "reference length": 1.0,
            },
        },
        "initialisation": {"initial conditions": "IC_1", "restart": False},
        "numerical scheme": {
            "order": "second",
            "inviscid flux scheme": "roe",
        },
        "time settings": {
            "type": "unsteady",
            "kind": "dual time stepping",
            "total time": 1.6,
```

```
            "time step": 0.002,
            "order": "second",
            "start": 0,
        },
        "convergence control": {
            "scheme": {"name": "implicit euler"},
            "cfl": {
                "cfl": 200,
                "cfl viscous factor": 1.0e-6,
                "cfl ramp": [{"type": "exponential", "factor": 1.05, "initial": 0.1}],
            },
            "cycles": 5,
        },
        "output settings": {
            "report": {
                "frequency": 5,
                "monitor": {
                    "MR_1": {
                        "name": "probe",
                        "point": [0.25, 1.07, 0.313],
                        "variables": ["cp"],
                    },
                },
            },
            "solution": {
                "format": "vtk",
                "surface variables": ["V", "p", "T", "rho", "cp"],
                "volume variables": ["V", "p", "T", "rho", "m", "cp"],
                "frequency": {
                    "volume data": 5,
                    "surface data": 5,
                    "checkpoint": 20,
                },
            },
        },
    },
    "model": {
        "cylinder": {
            "mesh": "cylinder.h5",
            "boundary conditions": {
                "BC_1": {"zones": [2], "type": "symmetry"},
                "BC_2": {"zones": [3], "type": "wall", "kind": "no slip"},
                "BC_3": {
                    "zones": [1],
                    "type": "farfield",
                    "condition": "IC_1",
                    "kind": "riemann",
                },
            },
        },
    },
}
```

If you have worked through Simple aerofoil (p. 373), the parts that are new here are `time settings`,

`convergence control` and `output settings`.

`config version`

The version of the control-file format. `2` is the current format, which zCFD checks before it reads anything else.

`solver`

Everything shared by the whole simulation: the equations, the flow conditions, how the run marches in time and what it writes.

`solver settings`

Which solver runs the case. Reference (p. 11).

- `type`: `"compressible"`, the density-based solver. At Mach 0.2 the flow is nearly incompressible, but the compressible solver with preconditioning handles it well.

`equations`

Which equations are solved. Reference (p. 18).

- `type`: `"viscous"`, the Navier–Stokes equations with no turbulence model. At a Reynolds number of 150 the flow round the cylinder is laminar: smooth and orderly, apart from the vortices themselves, so no turbulence model is needed.
- `precondition`: `True` switches on low-Mach-number preconditioning, which keeps the solver efficient and accurate at low Mach numbers. It is recommended below about Mach 0.3.

`fluid properties`

The gas the flow is made of. Reference (p. 37).

- `material`: `"air"`.
- `gamma`: 1.4, the ratio of specific heats.
- `gas constant`: 287 J/kg/K.
- `sutherlands const`: 110.4 K, the constant in Sutherland's law, which gives the viscosity's change with temperature.
- `prandtl no`: 0.72, the laminar Prandtl number, which relates heat conduction to viscosity. There is no turbulent Prandtl number, as there is no turbulence model.

`reference conditions`

Named flow states, which the initialisation and the boundary conditions refer to. Reference (p. 39).

`IC_1` The freestream.

- `temperature`: 277.78 K.
- `pressure`: 101325 Pa.
- `v`: the velocity, as a direction and a Mach number: Mach 0.2 along y, which is the flow direction in this mesh.
- `reynolds no`: 150. Instead of a viscosity, the state gives a Reynolds number, and zCFD works out the viscosity that produces it.

- `reference length`: 1 m, the cylinder's diameter, which the Reynolds number is based on.

`initialisation`

How the flow field starts. Reference (p. 47).

- `initial conditions`: `"IC_1"` fills the domain with the freestream. The uniform start is symmetric, so the wake takes a while to break symmetry and begin shedding.
- `restart`: `False`, a fresh start rather than a continuation of an earlier run.

Choosing initialisation and reference state (p. 293) covers both.

`numerical scheme`

How the equations are discretised in space. Reference (p. 74).

- `order`: `"second"` reconstructs every flow variable to second order, with the default limiter keeping it free of overshoots.
- `inviscid flux scheme`: `"roe"`, Roe's scheme for the flow between cells.

Choosing a spatial scheme (p. 254) covers both.

`time settings`

What kind of run this is, and how it advances in physical time. This is what makes the run time-accurate. Reference (p. 51).

- `type`: `"unsteady"`, a solution that changes with time.
- `kind`: `"dual time stepping"`. The run advances in real (physical) time steps. At each one it has to find the flow at the new time, and it does that the same way a steady run finds its answer: by repeating cycles, called inner cycles, that home in on it. The inner cycles march in an artificial time, pseudo-time, that only matters inside the step. `convergence control`, below, sets that inner loop.
- `total time`: 1.6 s of physical time.
- `time step`: 0.002 s per step, so the run takes 1.6 / 0.002 = 800 time steps. The step must be small enough to follow the flow's fastest changes; the shedding period here is about 0.09 s, more than 40 steps.
- `order`: `"second"`, a second-order backward difference for the time derivative. The first steps build up to it from first order.
- `start`: 0, the number of steady cycles to run before time stepping begins. None here: the run marches from the uniform start straight away.

Dual time stepping (p. 238) describes the options.

`convergence control`

The inner pseudo-time loop that converges each physical time step, set just as it sets the whole solve in a steady run. Reference (p. 56).

`scheme`

- `name`: `"implicit euler"`, which takes each pseudo-time step by solving a linear system, allowing large steps.

`cfl` The CFL number sets the size of each pseudo-time step.

- `cfl`: 200, the CFL number the ramp climbs to.
- `cfl viscous factor`: scales the viscous part of each cell's pseudo-time step estimate. At 10⁻⁶ it leaves the convective wave speeds alone to set the step.
- `cfl ramp`: starts at 0.1 and multiplies the CFL by 1.05 each cycle until it reaches 200, so the first cycles, far from the answer, take small steps.

`cycles`

- 5 inner cycles per physical time step, so the run makes 800 × 5 = 4000 cycles in all.

The time step and the number of inner cycles together set the cost and accuracy of the run. Too large a time step, or too few inner cycles, and the faster unsteady features are lost; too small a step, or too many cycles, and the run costs more for no gain. Check a new case by repeating it with a smaller time step or more inner cycles, and making sure the result you care about, here the shedding frequency, does not change. CFL control (p. 245) describes the ways to set the CFL.

`output settings`

What zCFD writes, and how often. There are two kinds of output: the `report`, a table that grows as the run goes, and the `solution`, flow fields for viewing. Reference (p. 84).

`report` The report is the CSV file `cylinder_report.csv`: one row of residuals, and of any forces and monitors, at a time. zMon plots it as the run goes.

- `frequency`: 5, a row every 5 cycles. With 5 inner cycles per time step, that is one row per time step, at the end of its inner loop.
- `monitor`: monitor points, each of which records flow variables at a point. This deck has one, `MR_1`:
  - `name`: `"probe"`. The report names its columns after it.
  - `point`: (0.25, 1.07, 0.313), just behind the cylinder, where the shed vortices pass.
  - `variables`: `["cp"]`, the pressure coefficient. The monitor adds a `probe_cp` column to the report.

The pressure at the probe rises and falls as each vortex passes, so its history gives the shedding frequency. Monitor points (p. 90) lists the options.

`solution` The flow fields written for viewing in ParaView, and the checkpoint for restarting.

- `format`: `"vtk"`, VTK HDF files.
- `surface variables`: written on the boundaries: velocity `V`, pressure `p`, temperature `T`, density `rho` and pressure coefficient `cp`.
- `volume variables`: written in every cell: the same, plus the Mach number `m`.
- `frequency`: in a time-accurate run it counts physical time steps, and each output is kept rather than overwritten, so the files hold the flow's history. A dictionary sets each kind separately:
  - `volume data`: every 5 time steps, or every 0.01 s of physical time.
  - `surface data`: every 5 time steps.
  - `checkpoint`: every 20 time steps, the file a restart reads.

Volume output is the slowest and largest, so on a large case write it rarely and rely on surface data, monitor points and interpolated volume data (p. 88) for high-frequency data. Output variables (p. 336) lists every variable.

`model`

One entry per mesh, each with its own boundary conditions. Reference (p. 96).

`cylinder`

The only model. Its name, `cylinder`, names its output folder, `cylinder_OUTPUT/cylinder/`, and its output files.

`mesh`

- `"cylinder.h5"`, a zCFD mesh with 80,000 cells. The cylinder has a diameter of 1 m and is centred on the origin, with its span along x and a single cell across it.

`boundary conditions` A condition for each group of boundary faces. The groups are the mesh's zones, numbered when the mesh was made, and each condition names its zones in a `zones` list. Choosing boundary conditions (p. 285) describes every type.

`BC_1`

- `zones`: `[2]`, the two faces normal to the span.
- `type`: `"symmetry"`, which makes the flow behave as if the cylinder were infinitely long, so it stays 2D.

`BC_2`

- `zones`: `[3]`, the cylinder surface.
- `type`: `"wall"`, with `kind` `"no slip"`: the flow velocity is zero at the surface.

`BC_3`

- `zones`: `[1]`, the outer boundary.
- `type`: `"farfield"`, which lets flow pass in and out of the domain.
- `condition`: `"IC_1"`, the freestream state outside.
- `kind`: `"riemann"`, which passes waves out without reflecting them back into the domain.

Step 3. Run

Activate zCFD as in Simple aerofoil (p. 380), then check and run the case:

```
cd ~/tutorial_3_cylinder_shedding/
validate_input cylinder.py
run_zcfd -c cylinder.py
```

These are extracts from the log of a run of this tutorial, with `...` marking lines left out:

```
Parameters validated successfully
...
 Reading file cylinder.h5
 Global Mesh Stats: Cells: 80000 Faces: 320400
...
  Viscosity:    0.566148 Ns/m^2 @ 277.778 K
  Speed:        66.8165 m/s
...
  Mach:         0.2
  Reynolds No:  150 @ 277.778 K
...
Starting solve loop
...
Cycle      5 | Step    1 t=0.004s | CFL 200.000 | MG 1 |    33.3 ms | Total     0.325s
+-----------------------------------------------------------------------------------+
| Report: Cycle 5  Wall time: 0.325s                                                |
...
| Residuals:                                                                        |
|   rho      4.6237E-03  rhoV[0]  2.5934E-12  rhoV[1]  2.4954E-03  rhoV[2]  1.7882E-03 |
|   rhoE     1.1779E-02                                                             |
| Inner convergence:                                                                |
|   Inner 5/5: 0.89 orders (ratio 1.36e-03, residual 1.19 orders, 11596 cells)      |
| Reports:                                                                          |
|   probe_cp -5.8776E-02                                                            |
+-----------------------------------------------------------------------------------+
Real time cycle 1 complete: 5 inner cycles, 0.89 orders dropped
...
Real time cycle 799 complete: 5 inner cycles, 0.84 orders dropped
...
 Solve Complete: 4000 cycles in 1m 10.6s (avg 17.6 ms/cycle)
...
zCFD finished: all requested cycles are complete (4000 in total).
```

- The reference conditions are printed in dimensional form. The viscosity is very high because the Reynolds number is only 150 on a 1 m cylinder at 67 m/s.
- The cycle line now shows the time step (`Step`) and the physical time it has reached (`t`), as well as the inner cycle.
- The report box gains an `Inner convergence` line, saying how far the inner cycles dropped the residual in that time step, and a `Reports` section with the monitor point's value.
- At the end of each time step, a line sums up its inner loop. Here five inner cycles drop the residual by under one order of magnitude per step.
- The run ends after 800 time steps of 5 cycles each.

Output files

The output follows the same pattern as the first tutorial, named after the control file and the model:

FILE/DIRECTORY	DESCRIPTION
`cylinder.log`	A copy of the output to screen during the run.
`cylinder_status.yaml`	A YAML file recording how the run was started.

FILE/DIRECTORY	DESCRIPTION
`cylinder_OUTPUT/cylinder/cylinder_report.csv`	The report: residuals and the `probe_cp` monitor, one row per time step.
`cylinder_OUTPUT/cylinder/cylinder_results.h5`	The checkpoint, for restarting the run.
`cylinder_OUTPUT/cylinder/cylinder.vtkhdf`	The volume data. One file holds every time step written, 160 in this run.
`cylinder_OUTPUT/cylinder/cylinder_wall.vtkhdf` etc	One file for each boundary condition type, holding the surface data at every time step.

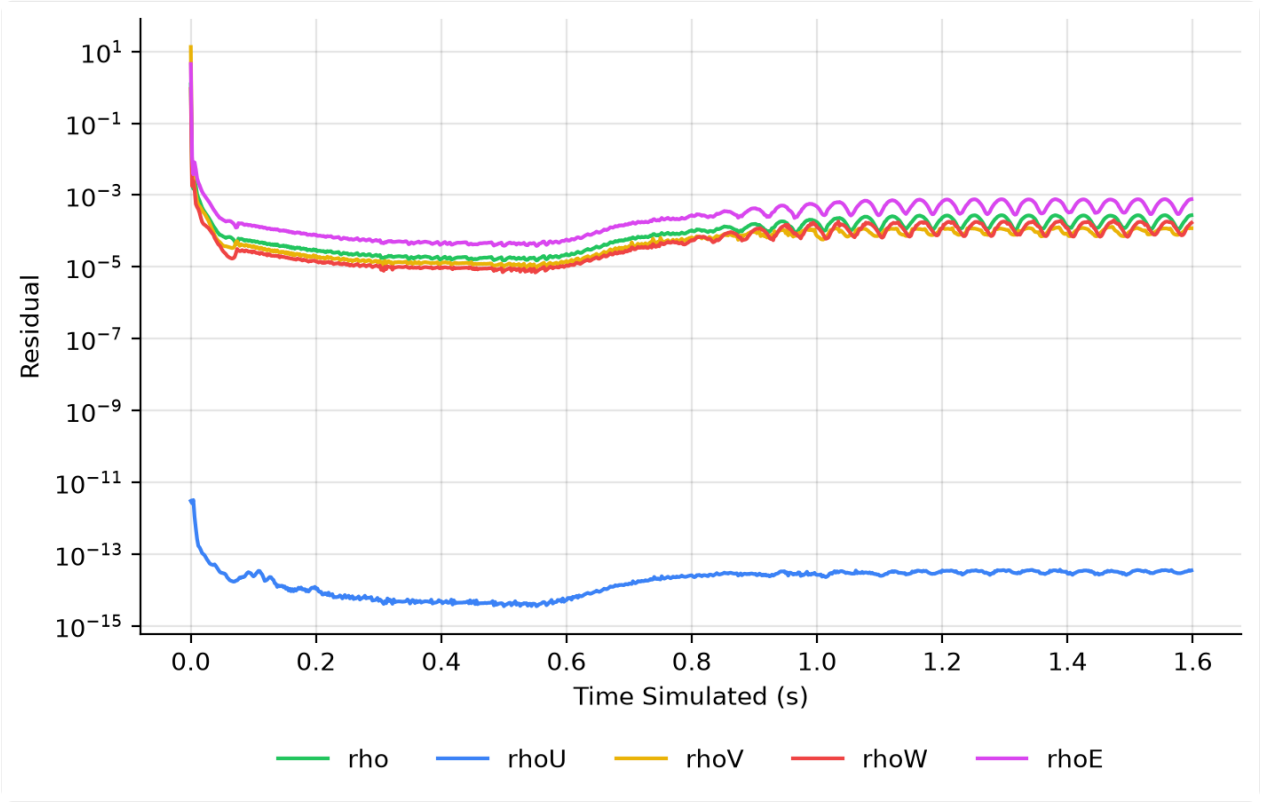
Step 4. Monitor

Open the run in zMon, as in the previous tutorials:

```
zmon cylinder.py
```

For an unsteady run, zMon can plot against physical time instead of cycle, showing one point per time step. The plots below are zMon's panels drawn that way; Step 5 shows how to plot the same data yourself.

Residuals



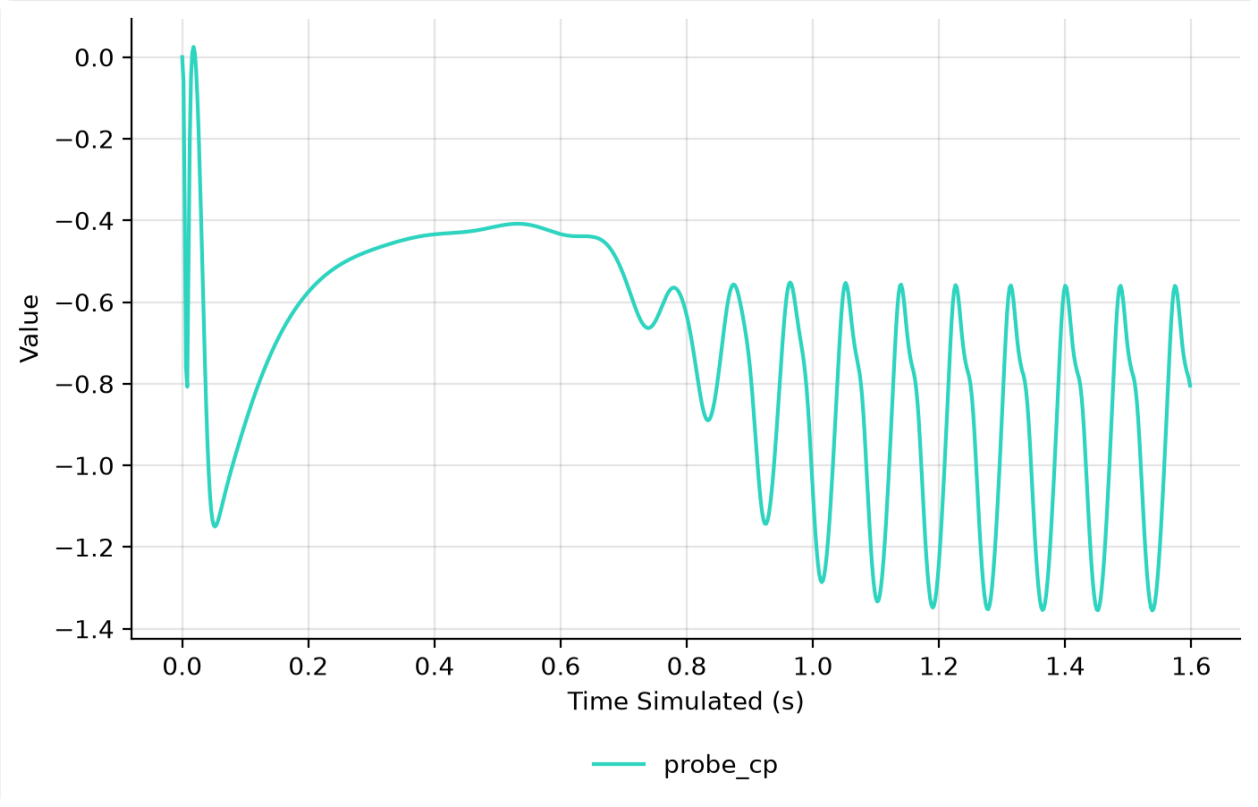
Residuals of the cylinder run against physical time

Each point is the residual at the end of a time step. They fall quickly from the uniform start, then sit between 10⁻⁵ and 10⁻³. Once shedding begins, from about 0.7 s, they rise a little and oscillate at the shedding frequency: the flow is changing all the time, so there is no steady

state for the residuals to fall towards. `rhoV[0]`, the momentum along the span (x), stays near round-off throughout, because the flow has no velocity in that direction.

Monitor points

zMon's monitor points panel shows the `probe_cp` column the monitor adds to the report:



Pressure coefficient at the probe against physical time

The pressure behind the cylinder settles at first, then from about 0.7 s starts to oscillate as the wake breaks symmetry and vortices are shed. By 1.2 s the oscillation is regular. Its frequency gives the shedding frequency: the cylinder sheds at 11.5 Hz. Engineers usually quote it as a Strouhal number, the frequency times the diameter divided by the flow speed, which lets cylinders of any size and speed be compared: here it is 0.17.

Step 5. Post-process

First activate your zCFD environment, which provides Python with zutil, h5py and ParaView:

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

Python

This script draws the plots in Step 4 and measures the shedding frequency. It keeps the last report row of each time step (`groupby("RealTimeStep").last()`) and turns the step number into time with the 0.002 s time step. It then finds the peaks of `probe_cp` after 1.2 s, once the shedding is regular: the frequency is the number of periods between the first and last peak divided by the time between them. The Strouhal number divides by the flow speed, 66.82 m/s, which the log prints. Save the script as

`shedding_frequency.py` in the case directory:

```
import matplotlib.pyplot as plt

from zutil.fileutils import get_zcfd_result

report = get_zcfd_result("cylinder.py").report.data
steps = report.groupby("RealTimeStep").last()
time = steps.index * 0.002

fig, ax = plt.subplots()
for column in ["rho", "rhoV[0]", "rhoV[1]", "rhoV[2]", "rhoE"]:
    ax.semilogy(time, steps[column], label=column)
ax.set_xlabel("Time (s)")
ax.set_ylabel("Residual")
ax.legend()
fig.savefig("residuals.png", dpi=200)

fig, ax = plt.subplots()
ax.plot(time, steps["probe_cp"])
ax.set_xlabel("Time (s)")
ax.set_ylabel("probe_cp")
fig.savefig("probe.png", dpi=200)

cp = steps["probe_cp"].to_numpy()
peaks = [time[i] for i in range(1, len(cp) - 1) if time[i] > 1.2 and cp[i - 1] < cp[i] > cp[i
↪  + 1]]
frequency = (len(peaks) - 1) / (peaks[-1] - peaks[0])
print(f"Shedding frequency {frequency:.2f} Hz, Strouhal number {frequency * 1.0 /
↪  66.82:.3f}")
```

Run it:

```
python shedding_frequency.py
```

It writes `residuals.png` and `probe.png`, and prints the shedding frequency and the Strouhal number:

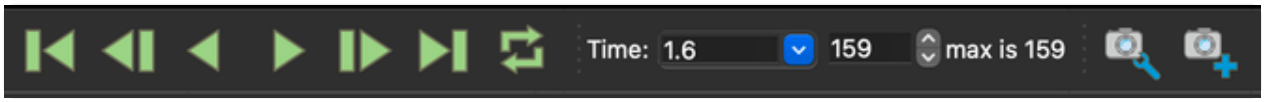
```
Shedding frequency 11.49 Hz, Strouhal number 0.172
```

ParaView

Flow field

The solution output frequency is 5, so the solver writes visualisation data every 5 time steps. All of them are stored in a single `cylinder.vtkhdf` file in the `cylinder_OUTPUT/cylinder/` directory.

Launch ParaView and load `cylinder_OUTPUT/cylinder/cylinder.vtkhdf`. ParaView shows one time step at a time, and you can move through them with the icons at the top of the screen:

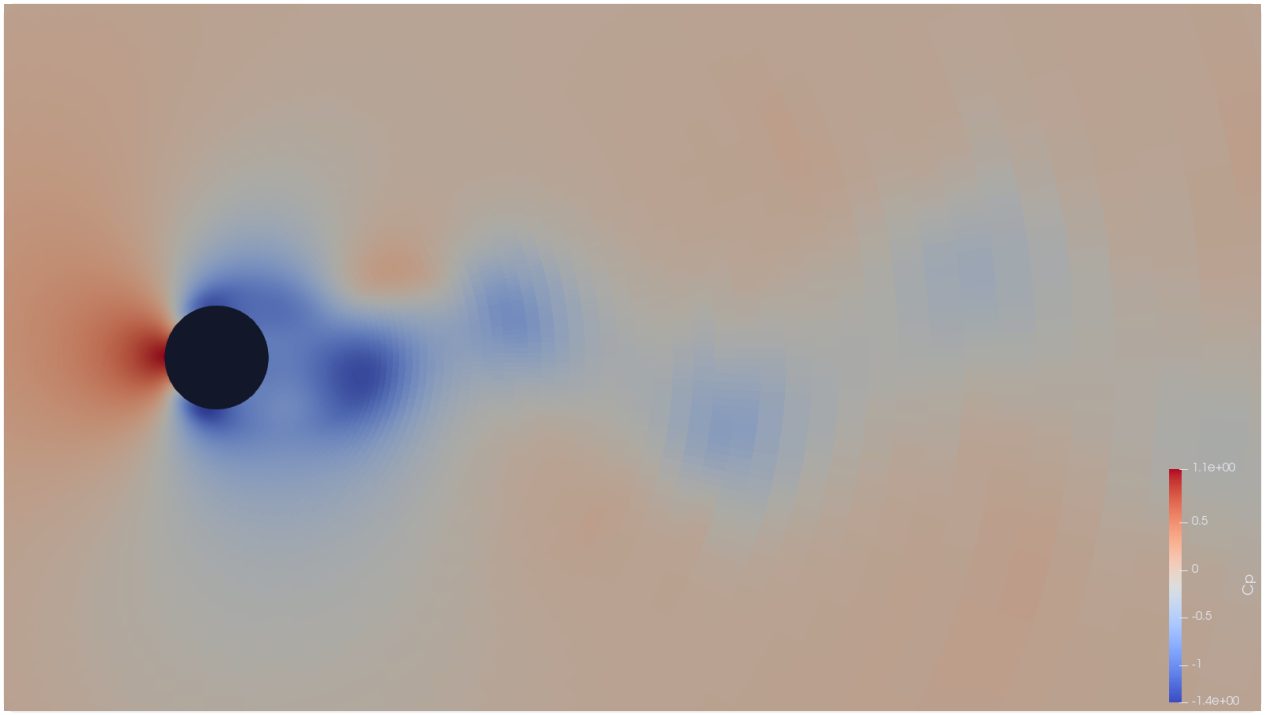


ParaView's time controls

Animation

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

1. Launch ParaView and load `cylinder_OUTPUT/cylinder/cylinder.vtkhdf`.
2. Set the view up to shade the output by *"cp"* and zoom/move the image until you are happy with the view:



Cp shaded

3. Select *'File→Save Animation'* in ParaView. To export as video select your preferred format, the format will depend on your Operating System but for Windows you should be able to select "MP4 files" and on Linux "FFMPEG AVI". Give the file a name and click OK.
4. On the next screen you can set the target resolution, frame rate and the timesteps to export. Set the Image resolution to 1920x1080 (FHD) and the frame rate to 24. To export only the end of the simulation (where the oscillation is established), set the Frame Window from 80 to 160.
5. Click OK to export the animation. This will take some time.

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

Aeroacoustics (FWH)

What is in this tutorial?

The vortices shed by the cylinder in Time-accurate solution (p. 414) make a tone that can be heard far away. This tutorial predicts that sound at two microphones, or *observers*, with the Ffowcs Williams–Hawkings (FWH) solver (p. 305) that comes with zCFD.

Simulating the air all the way out to a distant microphone would need an enormous mesh. An FWH solver avoids that. zCFD solves the flow only near the cylinder, where the sound is made, and records the flow, as the run goes, on a closed *FWH surface* round the cylinder. After the run, the FWH solver uses those recordings to work out the sound that reaches each observer, without solving the flow in between. You will run the CFD (computational fluid dynamics) simulation, run the FWH solver in two ways, and plot and listen to what each observer hears.

What you need?

Work through Time-accurate solution (p. 414) first: this tutorial runs the same cylinder, and relies on it for running an unsteady case and viewing it in ParaView. The CFD run takes about ten minutes on an NVIDIA RTX 3060, and the FWH solver a few seconds.

Step 1. Download the files

The tutorial bundle can be downloaded here.

```
cd ~
wget https://zcfd.zenotech.com/v2026.11.16174/tutorial_4_cylinder_noise.tgz
tar xzf tutorial_4_cylinder_noise.tgz
cd tutorial_4_cylinder_noise
```

It contains:

- `cylinder.h5`, the mesh, and `cylinder_noise.py`, the zCFD control file;
- `fwh_surface_small.stl`, the FWH surface this tutorial uses, and `fwh_surface_medium.stl` and `fwh_surface_large.stl`, two larger ones for the extensions;
- `observers.csv`, the observer used by the `run_fwh` utility;
- `fwh_python.py`, which runs the FWH solver from Python;
- `plot_run_fwh.py` and `plot_fwh_python.py`, which plot what the observers hear.

Step 2. Walk through the input

The control file, `cylinder_noise.py`, is a short Python script. All it does is build one Python dictionary, called `parameters`, that tells zCFD everything about the run: the physics, the numerical method, what

to write out, and the mesh with its boundaries. This step walks through that dictionary from top to bottom. Each heading below is the name of a key, written exactly as it appears in the file, and the headings nest the same way the dictionary does: `numerical scheme`, for example, sits inside `solver`. Under each heading you'll find what that part controls, what each setting means, and why it has the value it has here. Each section ends with a link to that part's reference page, which lists every setting it accepts and its default; you don't need it to follow the tutorial. On a wide screen the whole control file sits on the right, with the part being described highlighted as you scroll.

`cylinder_noise.py` has the same settings as `cylinder.py` from Time-accurate solution (p. 414), with two changes: the run lasts 4 seconds instead of 1.6, and the output also records the flow on the FWH surface.

```
parameters = {
    "config version": 2,
    "solver": {
        "solver settings": {"type": "compressible"},
        "equations": {
            "type": "viscous",
            "precondition": True,
        },
        "fluid properties": {
            "material": "air",
            "gamma": 1.4,
            "gas constant": 287.0,
            "sutherlands const": 110.4,
            "prandtl no": 0.72,
            "turbulent prandtl no": 0.9,
        },
        "reference conditions": {
            "IC_1": {
                "temperature": 277.7777,
                "pressure": 101325.0,
                "v": {"vector": [0.0, 1.0, 0.0], "mach": 0.2},
                "reynolds no": 150,
                "reference length": 1.0,
            },
        },
        "initialisation": {"initial conditions": "IC_1", "restart": False},
        "numerical scheme": {
            "order": "second",
            "inviscid flux scheme": "roe",
        },
        "time settings": {
            "type": "unsteady",
            "kind": "dual time stepping",
            "total time": 4.0,
            "time step": 0.002,
            "order": "second",
            "start": 0,
        },
        "convergence control": {
            "scheme": {"name": "implicit euler"},
```

```
            "cfl": {
                "cfl": 200,
                "cfl viscous factor": 1.0e-6,
                "cfl ramp": [{"type": "exponential", "factor": 1.05, "initial": 0.1}],
            },
            "cycles": 5,
        },
        "output settings": {
            "report": {
                "frequency": 5,
                "monitor": {
                    "MR_1": {
                        "point": [0.25, 1.07, 0.313],
                        "variables": ["cp"],
                    },
                },
            },
            "solution": {
                "format": "vtk",
                "surface variables": ["V", "p", "T", "rho", "cp"],
                "volume variables": ["V", "p", "T", "rho", "m", "cp"],
                "fwh interpolate": ["fwh_surface_small.stl"],
                "frequency": {
                    "fwh interpolate": 1,
                    "volume data": 1000,
                    "surface data": 1000,
                },
            },
        },
    },
    "model": {
        "cylinder": {
            "mesh": "cylinder.h5",
            "boundary conditions": {
                "BC_1": {"zones": [2], "type": "symmetry"},
                "BC_2": {"zones": [3], "type": "wall", "kind": "no slip"},
                "BC_3": {
                    "zones": [1],
                    "type": "farfield",
                    "condition": "IC_1",
                    "kind": "riemann",
                },
            },
        },
    },
}
```

`config version`

`"config version": 2` says which layout of the control file this is. Every current control file declares it.

solver

`solver` holds the settings shared by the whole simulation: the physics, the numerics, time marching and output. Its blocks are walked below in the order the deck gives them.

solver settings

`solver settings` (p. 11) chooses the solver.

- `type`: `compressible`, the density-based solver, which suits air at Mach 0.2.

equations

`equations` (p. 18) sets which equations are solved.

- `type`: `viscous`, the laminar Navier–Stokes equations. At a Reynolds number of 150 the flow round the cylinder is laminar, so no turbulence model is needed.
- `precondition`: `True` switches on low-Mach-number preconditioning, recommended below about Mach 0.3.

fluid properties

`fluid properties` (p. 37) describes the gas.

- `material`: `air`, an ideal gas.
- `gamma`: 1.4, the ratio of specific heats.
- `gas constant`: 287.0 J/kg/K.
- `sutherlands const`: 110.4 K, the constant in Sutherland's law, which gives the viscosity's change with temperature.
- `prandtl no`: 0.72, the laminar Prandtl number, which sets the heat conduction.
- `turbulent prandtl no`: 0.9. It plays no part in a laminar run.

reference conditions

`reference conditions` (p. 39) defines named flow states, which the initialisation and the boundary conditions refer to by name.

IC_1

`IC_1` is the freestream.

- `temperature`: 277.7777 K.
- `pressure`: 101325 Pa.
- `v`: the velocity, given as a direction, `vector` [0, 1, 0], and a `mach` number, 0.2. The flow runs along +y at 66.8 m/s, the speed the log prints.
- `reynolds no`: 150. The Reynolds number compares the flow's inertia with its viscosity; at 150 the flow is slow and viscous enough to stay laminar. zCFD meets it by setting the viscosity.
- `reference length`: 1.0 m, the length the Reynolds number is based on: the cylinder's diameter.

The `run_fwh` utility in Step 5 takes its speed of sound and density from this block too.

initialisation

`initialisation` (p. 47) sets the state the run starts from.

- `initial conditions`: `IC_1`, so the whole domain starts at the freestream.

- `restart`: `False`, a fresh start rather than one from an earlier run's checkpoint.

`numerical scheme`

`numerical scheme` (p. 74) sets how the equations are discretised in space.

- `order`: `second`, a second-order reconstruction of the flow in each cell, with the default limiter.
- `inviscid flux scheme`: `roe`, Roe's approximate Riemann solver for the flux between cells.

Choosing a limiter (p. 258) covers the order, the flux scheme and the limiter.

`time settings`

`time settings` (p. 51) says how the run advances in physical time.

- `type`: `unsteady`, a time-accurate run.
- `kind`: `dual time stepping`. Each physical time step is solved like a small steady problem: an inner loop of iterations, in a made-up *pseudo-time*, drives the flow towards its state at the end of the step. That inner loop is set by `convergence control` below.
- `total time`: 4.0 s of flow. The cylinder tutorial runs 1.6 s; the FWH solver needs a long, settled signal. The shedding takes about a second to settle, and the observers then need several seconds of it to resolve the tone.
- `time step`: 0.002 s, so 2000 time steps in all.
- `order`: `second`, second-order accuracy in time.
- `start`: 0, so no steady cycles are run before the time stepping begins.

Choosing a time-marching scheme (p. 238) explains dual time stepping.

`convergence control`

`convergence control` (p. 56) sets the inner loop that converges each time step.

- `scheme`: `name implicit euler`, which takes each inner step by solving a linear system, allowing large steps.
- `cfl`: the CFL number, which sets the size of each inner step.
  - `cfl`: 200.
  - `cfl viscous factor`: 1.0e-6. It scales the viscous part of each cell's pseudo-time step estimate, so at this value the convective wave speeds alone set the step, as in Time-accurate solution (p. 414).
  - `cfl ramp`: an `exponential` ramp that starts at 0.1 and multiplies the CFL by 1.05 each cycle. It starts the very first time step gently, from a flow that has not yet seen the cylinder; the log shows every inner cycle at 200 from the second time step on.
- `cycles`: 5 inner cycles per time step. They don't fully converge each step: the residuals, which measure how far the equations are from being satisfied, fall by a median of about 0.8 orders of magnitude (a factor of about 6) within a step. The Strouhal number in Step 5 shows the shedding is still right.

CFL control (p. 245) describes the other ways to set the CFL number.

`output settings`

`output settings` (p. 84) sets what zCFD writes during and after the run. It has two parts: the `report`, a table written as the run goes, and the `solution`, the flow fields written for viewing and post-processing.

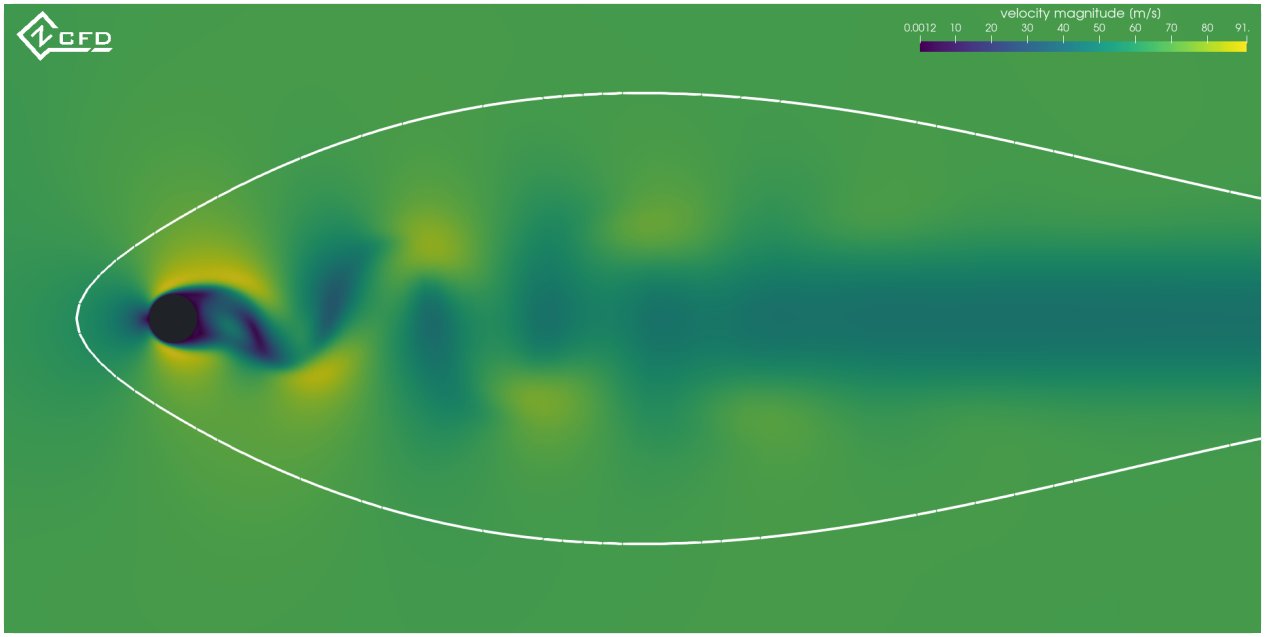
`report` The report is `cylinder_noise_OUTPUT/cylinder/cylinder_report.csv`, the table zMon plots.

- `frequency`: 5 cycles. In this run that writes a row at the first and the last inner cycle of every time step, each with the residual of every equation.
- `monitor`: monitor points, each recording flow variables at a fixed point. `MR_1` has a `point` in the wake, [0.25, 1.07, 0.313], and records its `variables`, the pressure coefficient `cp`. Without a `name`, its report column is named after its key: `mr_1_cp`.

`solution` The solution output is written to files under `cylinder_noise_OUTPUT/cylinder/`.

- `format`: `vtk`, VTK HDF files (*.vtkhdf*) for ParaView.
- `surface variables`: the variables written on the boundary surfaces: velocity `V`, pressure `p`, temperature `T`, density `rho` and pressure coefficient `cp`.
- `volume variables`: the variables written throughout the domain: the same, with the Mach number `m`.
- `fwh interpolate`: the FWH surface files to record the flow on, here `fwh_surface_small.stl`. The flow is interpolated to the centre of each of the surface's panels and written to `ACOUSTIC_DATA/fwh_surface_small_FWHData.h5`, the FWH solver's input.
- `frequency`: how often each kind of output is written, in time steps.
  - `fwh interpolate`: 1, every time step, because the FWH solver needs every step.
  - `volume data` and `surface data`: 1000, so only at time steps 1000 and 2000. The cylinder tutorial already gives you the flow field in detail.

The FWH surface is made of 1000 flat panels. It surrounds the cylinder and runs far downstream alongside the wake. Here its outline, in white, is drawn over the velocity magnitude on the mid-span plane at the end of the run. In ParaView, show the surface as *Feature Edges* with a thicker *Line Width* to see it beside the volume solution:



The FWH surface's outline in white over the velocity magnitude at mid-span, with the cylinder's wake inside it

It encloses the region where the noise is made, but does not cut through the wake. Vortices crossing the surface would be read as sound that isn't really there, so-called pseudo-sound [1], [2].

`model`

`model` (p. 96) has one entry per mesh, with that mesh's boundary conditions.

`cylinder`

The model's name, `cylinder`, names its output folder, `cylinder_noise_OUTPUT/cylinder/`, and its files.

`mesh` `cylinder.h5`, the mesh file, in zCFD's native format: 80,000 cells round a cylinder of 1 m diameter centred at (0.25, 0, 0), with its span along x and one cell across it.

`boundary conditions` `boundary conditions` (p. 98) assign a condition to each group of boundary faces. The groups are the mesh's zones, numbered when the mesh was made, and each condition lists its zones in `zones`. Choosing boundary conditions (p. 285) describes every type.

`BC_1`

- `zones`: [2], the two faces normal to the span.
- `type`: `symmetry`. With symmetry on both, the cylinder behaves as if it were infinitely long, so the mesh needs only one cell across the span.

`BC_2`

- `zones`: [3], the cylinder's surface.
- `type`: `wall`, with `kind` `no slip`: the flow velocity is zero at the surface.

`BC_3`

- `zones`: [1], the outer boundary.
- `type`: `farfield` (p. 286).
- `condition`: `IC_1`, the state the farfield imposes.
- `kind`: `riemann`, which lets flow and waves leave the domain without reflecting back into it.

Step 3. Run

In the activated environment (see Simple aerofoil (p. 380)), check the control file and run it:

```
cd ~/tutorial_4_cylinder_noise/
validate_input cylinder_noise.py
run_zcfd -c cylinder_noise.py
```

These are extracts from the log of a run of this tutorial, `cylinder_noise.log`, with `...` marking lines left out:

```
Parameters validated successfully
...
 Reading file cylinder.h5
 Global Mesh Stats: Cells: 80000 Faces: 320400
...
  Speed:        66.8165 m/s
...
  Reynolds No:  150 @ 277.778 K
...
Starting solve loop
----------------------- Cycle      1  Step    0 t=0.002s -----------------------
...
Real time cycle 1000 complete: 5 inner cycles, 0.87 orders dropped
...
Inner convergence over 2000 real time cycles: median 0.82 orders dropped, 5-5 inner cycles
 ↪  used
Only 0% of real time cycles reached 2.0 orders - consider increasing 'cycles' or 'cfl'
 Solve Complete: 10000 cycles in 2m 59.1s (avg 17.9 ms/cycle)
...
zCFD finished: all requested cycles are complete (10000 in total).
```

- Each time step runs 5 inner cycles and ends with a line saying how far the residuals fell within it.
- A line starting `FWH Interpolate Field Output` marks each write of the FWH surface data.
- At the end, a summary gives the median fall in the residuals per time step, here 0.82 orders, with a hint that no step reached 2 orders. The Strouhal check in Step 5 shows that is enough for the shedding in this case. This extract is from a run on a faster GPU than the RTX 3060. For your own cases, Choosing a time-marching scheme (p. 238) discusses the balance between inner cycles and cost.

Output files

As in the earlier tutorials, the output is named after the control file and the model:

FILE/DIRECTORY	DESCRIPTION
`cylinder_noise.log`	A copy of the output to screen during the run.
`cylinder_noise_status.yaml`	How the run was started: version, date, control file, mesh, nodes.
`cylinder_noise_OUTPUT/cylinder/cylinder_report.csv`	The residuals and the monitor point, row by row.
`cylinder_noise_OUTPUT/cylinder/cylinder_results.h5`	The checkpoint, used to restart the run.
`cylinder_noise_OUTPUT/cylinder/cylinder.vtkhdf` etc	Volume and surface data, at time steps 1000 and 2000.
`cylinder_noise_OUTPUT/cylinder/ACOUSTIC_DATA/fwh_surface_small_FWHData.h5`	The flow on the FWH surface at every time step: the FWH input.
`cylinder_noise_OUTPUT/cylinder/ACOUSTIC_DATA/fwh_surface_small_FWHData.xdmf`	Opens the FWH surface data in ParaView.
`cylinder_noise_OUTPUT/LOGGING/`	A separate log for each MPI rank, used for debugging.

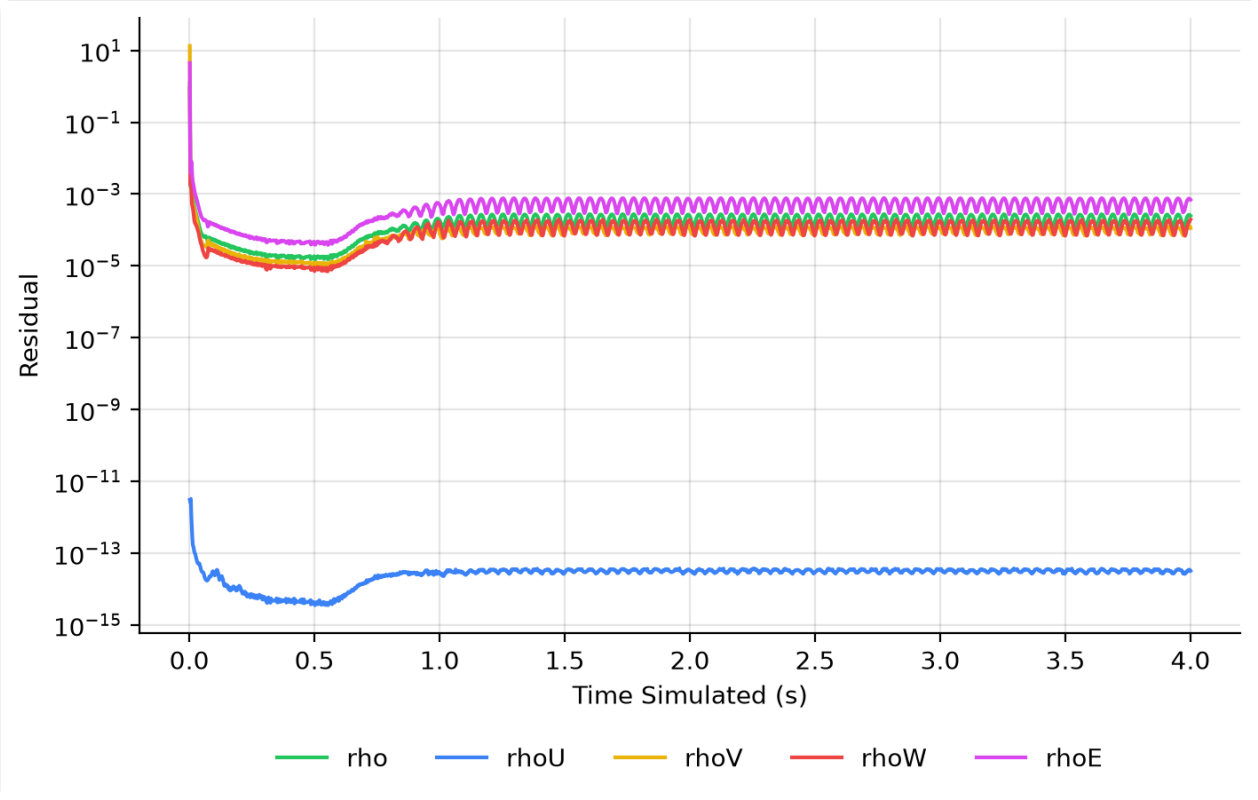
Step 4. Monitor

Open the run in zMon, during or after it:

```
zmon cylinder_noise.py
```

Monitoring a run (p. 324) describes each panel. The plot below shows the residuals at the end of each time step against simulated time, as zMon draws them with its x axis set to time; it was drawn in zMon's style with a script like the one in Simple aerofoil (p. 383).

Residuals



Residuals at the end of each time step against simulated time

The residuals fall over the first half second as the flow settles round the cylinder, then rise as the wake starts to shed, and from about 1 s they oscillate steadily, twice per shedding cycle: once for each vortex shed. That steady oscillation shows the shedding has settled. `rhoU`, the momentum along the span, stays at round-off level throughout, because the flow has no velocity along the span. The `mr_1_cp` monitor in zMon's reports panel oscillates at the shedding frequency itself.

Step 5. Post-process

First activate your zCFD environment, which provides Python with zutil, h5py and ParaView:

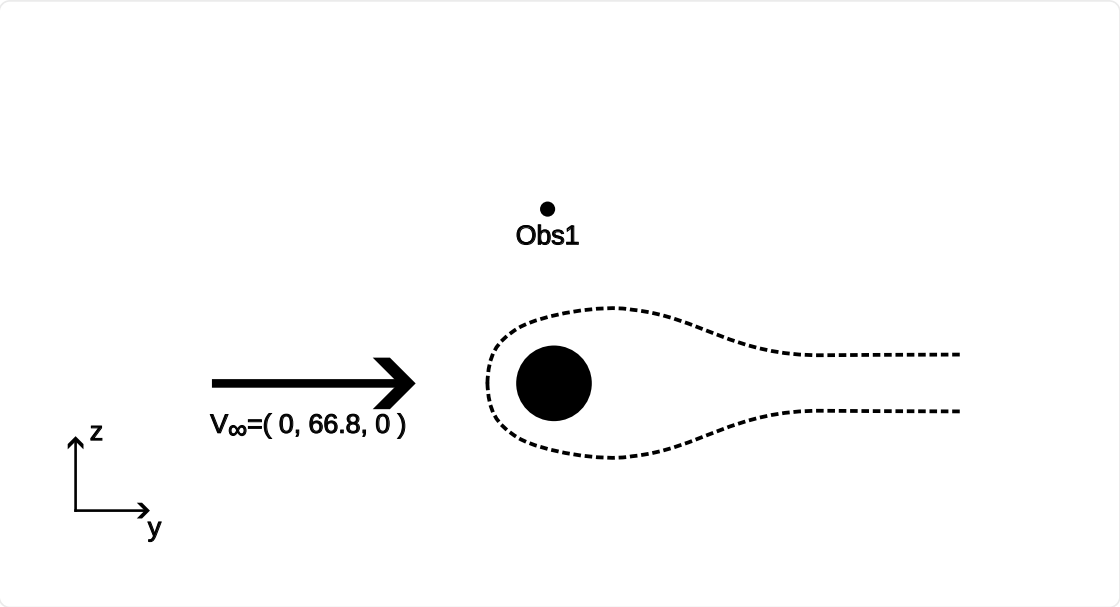
```
source <ZCFD_INSTALL_PATH>/bin/activate
```

The FWH solver post-processes the surface data the run wrote. It can be run in two ways: the run_fwh

utility (p. 312), which needs no code, and the fwh Python module (p. 308), which handles more cases.

FWH with the run_fwh utility

The `run_fwh` utility calculates the noise at an observer `Obs1`, which stays a fixed distance from the cylinder: for a cylinder in a wind tunnel, `Obs1` would be a microphone on the tunnel wall. `run_fwh` handles only observers that don't move relative to the FWH surface, which it calls being in the same translational reference frame (p. 313). `Obs1` is one.



FWH Observer and surface motions - CFD frame of reference

`run_fwh` needs three inputs:

- a zCFD control file (p. 312): `cylinder_noise.py`, from which it finds the FWH surface and its data, `cylinder_noise_OUTPUT/cylinder/ACOUSTIC_DATA/fwh_surface_small_FWHData.h5`;
- a reference condition (p. 313): the freestream, which is `IC_1` in the control file;
- an observers .csv file (p. 314) naming each observer and its position. The cylinder's centre is at (0.25, 0, 0) and `Obs1` stays 20 m from it along z, so `observers.csv` holds:

```
Obs1, 0.25, 0.0,  20.0
```

Run it in the activated environment:

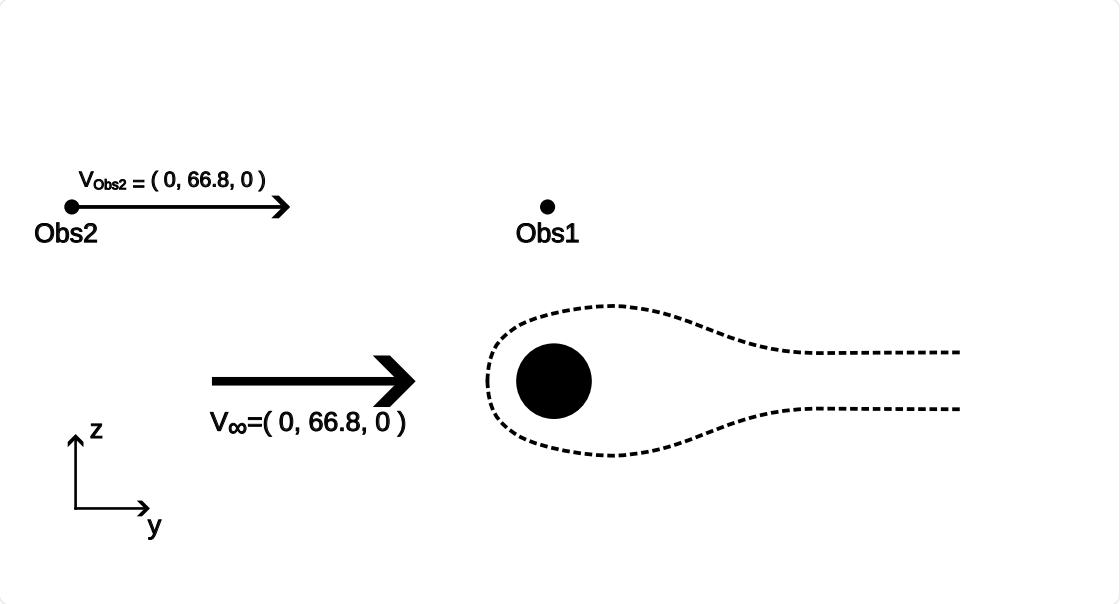
```
run_fwh -c cylinder_noise.py --reference_condition IC_1 --observer_locations observers.csv
```

It writes its progress to the screen and to `cylinder_noise.fwh.log`, and the pressure history at `Obs1` to `cylinder_noise.fwh.Obs1.csv`. Where there are several noise sources, the contribution of each is broken out, with the total alongside.

FWH with the fwh Python module

`run_fwh` is limited to observers and surfaces in the same translational reference frame, and to cases where no two FWH surfaces enclose the same source, since that would count the source twice (p. 314). The fwh Python module (p. 308) has neither limit. This section uses it to calculate the pressure at both

`Obs1` and `Obs2`, a 'fly-by' observer that `run_fwh` cannot handle.

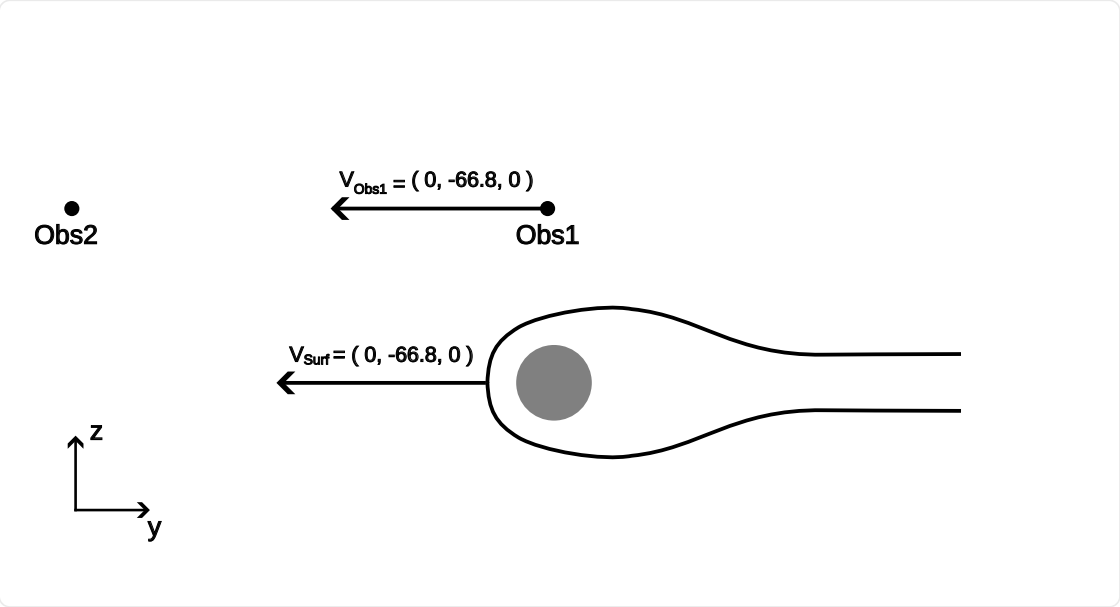


FWH surface and observer motions - CFD frame of reference

Observer and surface motions

The FWH equations assume a *quiescent medium*: still air. The CFD freestream is (0, 66.8, 0) m/s, so the FWH calculation is carried out in a frame moving with the air. `run_fwh` makes that change of frame for you; with the Python module you give the motions of the surface and observers through the still air yourself.

In the still-air frame, the cylinder and `Obs1` fly through the air together. `Obs2` stands still, like a microphone on the ground as the cylinder flies overhead.



FWH surface and observer motions - FWH (quiescent medium) frame of reference

The script

`fwh_python.py` defines the surface, the observers and the solver settings, then calls `fwh.solve()`:

```
import json
import math

from fwh import fwh, motion

c = math.sqrt(1.4 * 277.7777 * 287)
v_surf = [0, -0.2 * c, 0]
surfaceCentrePoint = motion.ConstantVelocityPoint([0.0, 0.0, 0.0], v_surf)
surfaceMotion = motion.NonrotatingSurface(surfaceCentrePoint)

obs1_CFD_frame_posn = [0.25, 0, 20]
obs1Motion = motion.ConstantVelocityPoint(obs1_CFD_frame_posn, v_surf)

obs2_posn = [obs1_CFD_frame_posn[i] + 2.5 * v_surf[i] for i in range(3)]
obs2Motion = motion.StationaryPoint(obs2_posn)

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

surfaces = {
    "surf1": {
        "motion": surfaceMotion,
        "fileName": "./cylinder_noise_OUTPUT/cylinder/ACOUSTIC_DATA/fwh_surface_small_
        ↪  FWHData.h5",
    }
}
observers = {"Obs1": obs1Motion, "Obs2": obs2Motion}

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

with open("./FWH_data.json", "w") as f:
    json.dump({"p": pOut, "t": tOut}, f)
```

The most important idea is that *motion classes* define how observers and surfaces move. The point (observer) motions (p. 311) are `OriginPoint`, `StationaryPoint` and `ConstantVelocityPoint`; the surface motions (p. 311), which take a point motion for their centre, are `NonrotatingSurface` and `RotatingSurface`. Running the zCFD FWH solver (p. 307) gives the full set-up.

- The speed of sound, density and pressure in `solverSettings` follow from `IC_1` in `cylinder_noise.py`, and `dt` is the 0.002 s between FWH surface outputs.
- A point on the FWH surface moves as $\mathbf{x}(t) = \mathbf{x}_0 + \mathbf{V}_{surf}\, t$ in the still-air frame, where $\mathbf{x}_0$ is its position in `fwh_surface_small.stl` and $\mathbf{V}_{surf}$ is the surface velocity, the freestream reversed.
- `Obs1` stays 20 m from the cylinder's centre along z, so it moves as $\mathbf{x}(t) = (0.25, 0, 20) + \mathbf{V}_{surf} t$.
- `Obs2` is stationary, and placed where `Obs1` is at t = 2.5 s.

Run it in the activated environment:

```
python3 fwh_python.py
```

It prints the *valid min and max times* for each surface and observer pair: the window in which the observer hears every point of the surface. Sound takes time to travel: sound from a surface point at time *t* reaches an observer *r/c* seconds later, where *r* is the distance between them and *c* the speed of sound. So the window starts later than the 0.002 s to 4.0 s of surface data, and a larger surface gives a narrower window. The pressure histories are saved to `FWH_data.json`.

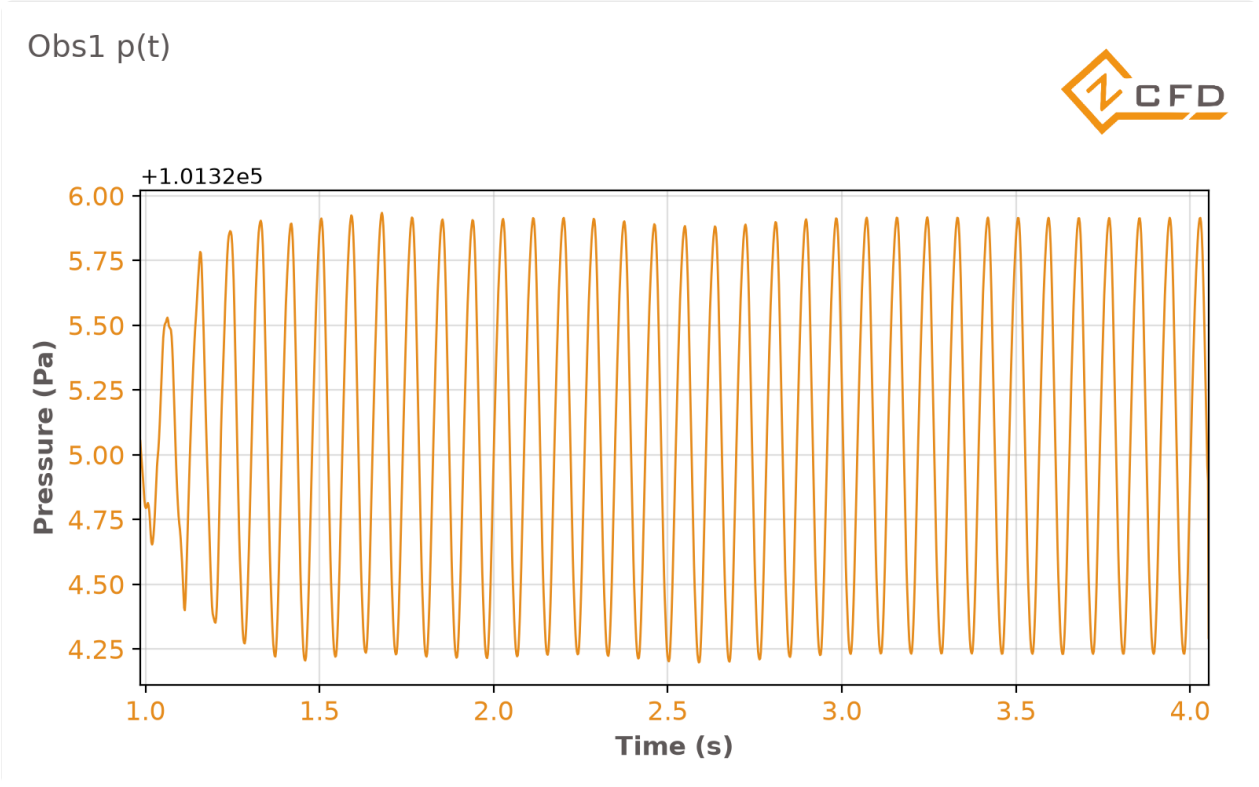
Python

`plot_fwh_python.py` plots the data from the fwh Python module, and `plot_run_fwh.py` does the same for the run_fwh utility. Run either, after the step whose data it reads:

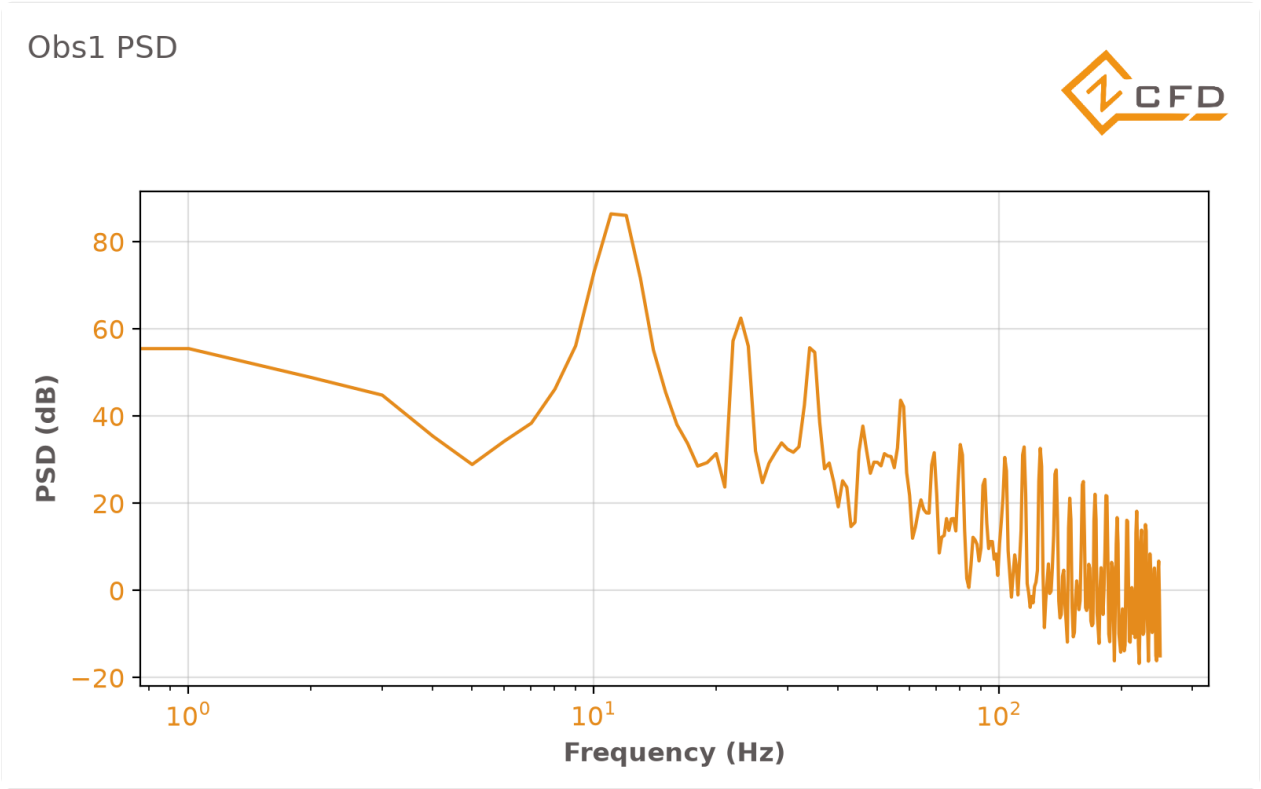
```
python3 plot_fwh_python.py
```

They use the *zutil* acoustic functions, which may be useful for your own acoustic post-processing, to plot the pressure history *p(t)* and the power spectral density (PSD), which shows how the sound's energy is spread across frequencies, at each observer: `Obs1_p.png` and `Obs1_psd.png`, and from the Python module data also `Obs2_p.png` and `Obs2_psd.png`. Both also write *.wav* files, `Obs1_playback.wav` and `Obs2_playback.wav`, so you can listen to what each observer hears.

`Obs1` moves with the cylinder. The noise it hears is dominated by a steady tone at 11.5 Hz, the frequency at which the cylinder sheds vortices. As a Strouhal number, the frequency times the diameter divided by the flow speed, that is 0.17. Harmonics, whole-number multiples of the tone, show as smaller peaks in the PSD.

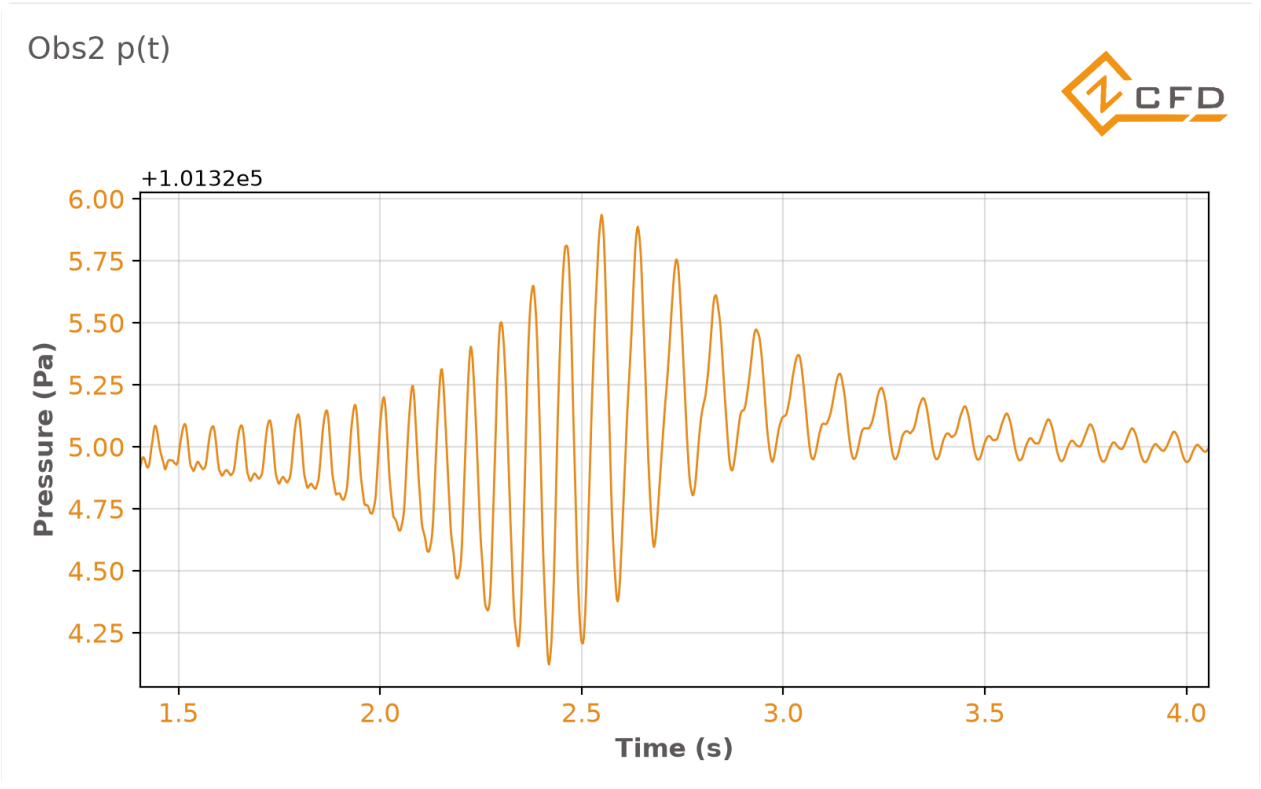


Observer 1 p(t)

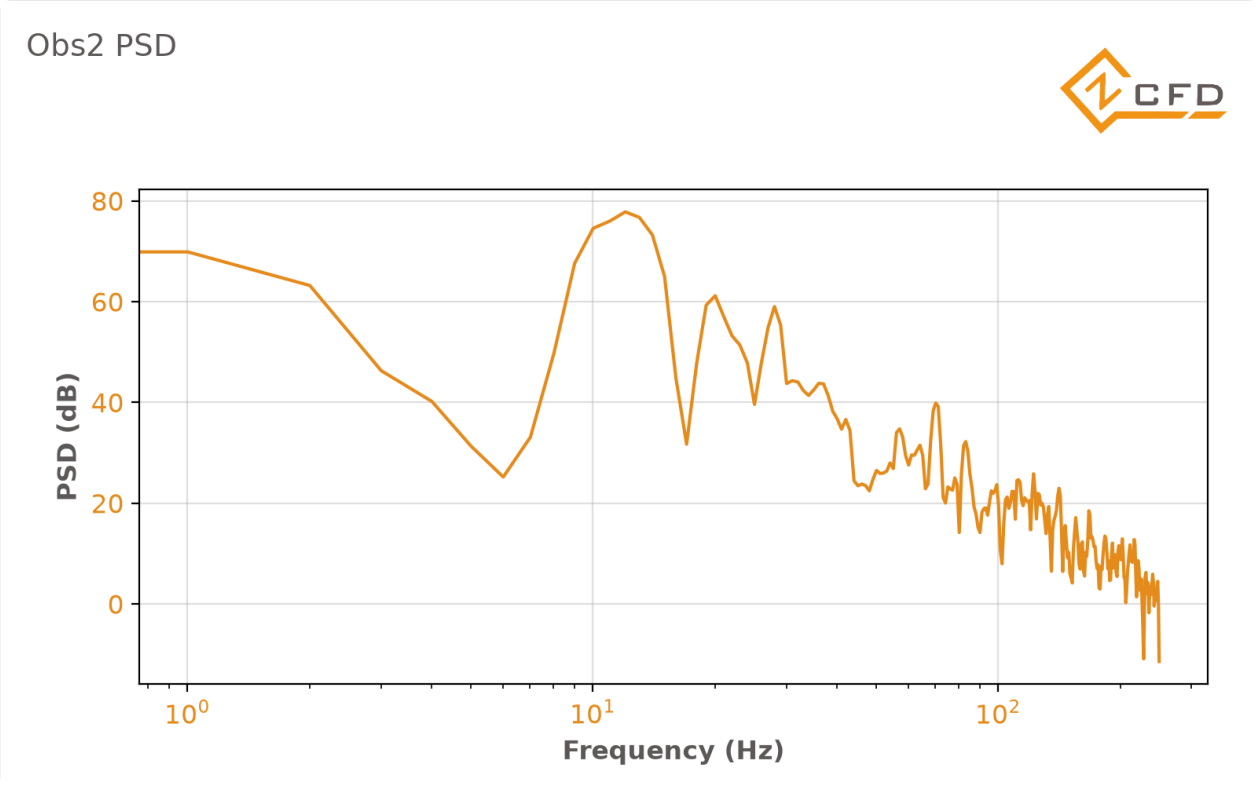


Observer 1 PSD

The cylinder approaches `Obs2` until 2.5 s and then moves away. In the `Obs2` *p(t)* plot the peaks are closer together before 2.5 s than after it: the pitch is higher while the source approaches and lower as it recedes, the Doppler shift you hear as a siren passes. It also spreads the shedding peak in the PSD, compared with `Obs1`'s.



Observer 2 p(t)



Observer 2 PSD

ParaView

`cylinder_noise_OUTPUT/cylinder/ACOUSTIC_DATA/fwh_surface_small_FWHData.xdmf` opens the FWH surface data in ParaView, with the pressure, velocity and density at each panel centre changing with time. Use the *Time* controls to step through it; a *Point Gaussian* view style, with a larger *Gaussian Radius* in the *Properties* panel, makes the panel values easier to see. The volume and surface output views as in Time-accurate solution (p. 414).

Extensions

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

1. Re-run the zCFD solver, but this time extract FWH data on the cylinder wall (p. 84) instead of `fwh_surface_small.stl`. Run the FWH solver (using the wall data as an impermeable (p. 305) FWH surface, one the flow cannot pass through) and compare the results with the `fwh_surface_small` surface, a permeable one that the flow passes through.
2. Re-run the zCFD solver, but this time also extract FWH data on the `fwh_surface_medium.stl` and `fwh_surface_large.stl` surfaces. These are like `fwh_surface_small.stl` but slightly larger. Run the FWH solver with these surfaces and investigate the effect of using different surfaces on the FWH results. You will have to do this using the fwh python module to avoid double accounting (p. 314).
3. Add extra observers of your choice to the FWH solver.
4. Use the file utility writeFWHSurfaceFrequenciesFile (p. 315) to visually inspect the spatial distribution of different frequencies in the FWH surface.
5. Use the file utility writeBlankedOffFwhSurfaceFile (p. 315) to 'blank off' all FWH surface cells more than 5 cylinder diameters downstream of the cylinder's centre. Run the FWH solver using this

modified surface, and based on this estimate how much of the shedding noise originates more than 5 diameters downstream of the cylinder.

6. Use the `fwh.fwh_helper_functions` module to rewrite `fwh_python.py` so that the `solverSettings` dictionary is automatically set based on the zCFD control file name and reference condition which were used in the `run_fwh` utility.

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

Citations

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

Overset meshes

What is in this tutorial?

A CFD solver divides the space round an object into small cells, the *mesh*, and solves the flow equations in each cell. An *overset* mesh is a second mesh placed over the first. Where it lies, its cells take over: the cells of the mesh beneath are switched off, which is called *hole cutting*. Along the edge of the overlap, the *interface*, each mesh takes the values it needs from the nearest active cells of the other, its *donor* cells. This is useful when:

- a region needs a finer or different mesh than the one around it;
- a component is added to an existing mesh, such as a propeller on an aircraft, or a new building in a cityscape, without regenerating the whole mesh;
- parts move relative to each other, such as a train entering a tunnel.

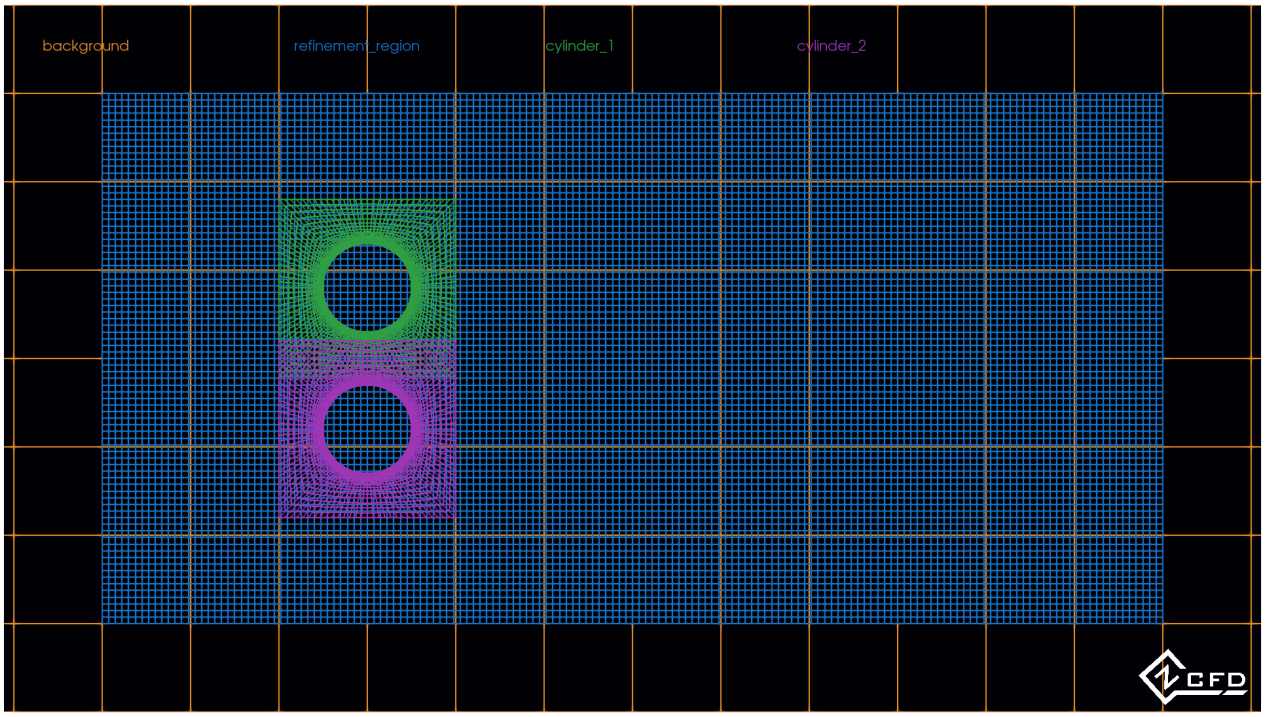
zCFD solves the flow separately on each mesh, finds where each mesh is active, and handles the exchange between all of them, including when a run is split across several GPUs. Each overset boundary names the meshes it takes its data from. Overset meshes (p. 328) describes how this works.

This tutorial has two examples. The first places two cylinders over a refinement region over a background mesh, to show how overset meshes are set up and viewed. The second moves a train into a tunnel and checks the pressure wave it makes against a European standard.

What you need?

Work through Simple aerofoil (p. 373) first. The cylinders take about two and a half minutes on an NVIDIA RTX 3060. The train in a tunnel is far larger, about 8 million cells, and needs about 48 GB of GPU memory.

Example 1. Overset cylinders



The four meshes at mid-span: the coarse background (orange), the regular Cartesian refinement region (blue), and the two cylinder meshes (green and purple). The two cylinder meshes overlap each other, both overset the refinement region, and the refinement region in turn oversets the background.

Four meshes make up this case: a coarse background, a finer refinement region over it, and a mesh round each of two cylinders over that.

Step 1. Download the files

The bundle can be downloaded here.

```
cd ~
wget https://zcfd.zenotech.com/v2026.11.16174/tutorial_5_overset_cylinders.tgz
tar xzf tutorial_5_overset_cylinders.tgz
cd tutorial_5_overset_cylinders
```

It contains the control file `overset_cylinders.py` and three meshes: `background.h5`, `refinement_region.h5` and `cylinder.h5`, which both cylinders use.

Step 2. Walk through the input

The control file, `overset_cylinders.py`, is a short Python script. All it does is build one Python dictionary, called `parameters`, that tells zCFD everything about the run: the physics, the numerical method, what to write out, and the mesh with its boundaries. This step walks through that dictionary from top to bottom. Each heading below is the name of a key, written exactly as it appears in the file, and the headings nest the same way the dictionary does: `numerical scheme`, for example, sits inside `solver`. Under each heading you'll find what that part controls, what each setting means, and why it has the value it has here. Each section ends with a link to that part's reference page, which lists every setting it accepts and its default; you don't need it to follow the tutorial. On a wide screen the whole control file

sits on the right, with the part being described highlighted as you scroll.

Here `model` has one entry for each of the four meshes.

```
parameters = {
    "config version": 2,
    "solver": {
        "solver settings": {"type": "compressible"},
        "equations": {"type": "viscous", "precondition": True},
        "fluid properties": {
            "material": "air",
            "gamma": 1.4,
            "gas constant": 287.0,
            "sutherlands const": 110.4,
            "prandtl no": 0.72,
            "turbulent prandtl no": 0.9,
        },
        "reference conditions": {
            "IC_1": {
                "temperature": 277.7777,
                "pressure": 101325.0,
                "v": {"vector": [1.0, 0.0, 0.0], "mach": 0.2},
                "reynolds no": 20.0,
                "reference length": 1.0,
            }
        },
        "initialisation": {"initial conditions": "IC_1"},
        "numerical scheme": {"order": "second"},
        "time settings": {"type": "steady"},
        "convergence control": {
            "scheme": {"name": "implicit euler", "stage": 1},
            "cfl": {
                "cfl": 20,
                "cfl viscous factor": 1.0e-6,
                "cfl ramp": [{"type": "exponential", "factor": 1.05, "initial": 0.1}],
            },
            "cycles": 2000,
        },
        "output settings": {
            "report": {
                "frequency": 5,
                "monitor": {
                    "MR_1": {
                        "name": "probe",
                        "point": [1.07, 0.25, 0.313],
                        "variables": ["cp"],
                    }
                },
            },
            "solution": {
                "format": "vtk",
                "surface variables": ["V", "p", "T", "rho", "cp"],
                "volume variables": ["V", "p", "T", "rho", "mach", "cp", "overset"],
            },
```

```
        },
    },
    "model": {
        "background": {
            "mesh": "background.h5",
            "transforms": {
                "transform": [
                    [1.0, 0.0, 0.0, 0.0],
                    [0.0, 1.0, 0.0, 0.0],
                    [0.0, 0.0, 1.0, -0.5],
                    [0.0, 0.0, 0.0, 1.0],
                ]
            },
            "boundary conditions": {
                "BC_1": {"zones": [9, 11], "type": "symmetry"},
                "BC_2": {
                    "zones": [10],
                    "type": "farfield",
                    "condition": "IC_1",
                    "kind": "riemann",
                },
            },
        },
        "refinement_region": {
            "mesh": "refinement_region.h5",
            "boundary conditions": {
                "BC_1": {"zones": [9, 11], "type": "symmetry"},
                "BC_2": {
                    "zones": [10],
                    "type": "interface",
                    "kind": "overset",
                    "connection": {"model": ["background"]},
                    "additional active layers": 3,
                },
            },
        },
        "cylinder_2": {
            "mesh": "cylinder.h5",
            "transforms": {
                "transform": [
                    [1.0, 0.0, 0.0, 0.0],
                    [0.0, 1.0, 0.0, -0.8],
                    [0.0, 0.0, 1.0, 0.0],
                    [0.0, 0.0, 0.0, 1.0],
                ]
            },
            "boundary conditions": {
                "BC_1": {"zones": [11, 12], "type": "symmetry"},
                "BC_2": {
                    "zones": [13],
                    "type": "interface",
                    "kind": "overset",
                    "connection": {"model": ["refinement_region"]},
```

```
                    "additional active layers": 3,
                },
                "BC_3": {"zones": [10], "type": "wall", "kind": "no slip"},
            },
        },
        "cylinder_1": {
            "mesh": "cylinder.h5",
            "transforms": {
                "transform": [
                    [1.0, 0.0, 0.0, 0.0],
                    [0.0, 1.0, 0.0, 0.8],
                    [0.0, 0.0, 1.0, 0.0],
                    [0.0, 0.0, 0.0, 1.0],
                ]
            },
            "boundary conditions": {
                "BC_1": {"zones": [11, 12], "type": "symmetry"},
                "BC_2": {
                    "zones": [13],
                    "type": "interface",
                    "kind": "overset",
                    "connection": {"model": ["refinement_region"]},
                    "additional active layers": 3,
                },
                "BC_3": {"zones": [10], "type": "wall", "kind": "no slip"},
            },
        },
    },
}
```

`config version`

`"config version": 2` says which layout of the control file this is. It may be written with a space or an underscore.

`solver`

`solver` holds the settings shared by all four meshes.

`solver settings`

`solver settings` (p. 11) chooses the solver.

- `type`: `compressible`, the density-based solver.

`equations`

`equations` (p. 18) sets which equations are solved.

- `type`: `viscous`, the laminar Navier–Stokes equations; the flow round each cylinder is laminar at a Reynolds number of 20. The Reynolds number compares the flow's inertia with its viscosity.
- `precondition`: `True`, low-Mach-number preconditioning, recommended below about Mach 0.3.

`fluid properties`

`fluid properties` (p. 37) describes the gas.

- `material`: `air`, an ideal gas, with `gamma` 1.4 and `gas constant` 287.0 J/kg/K.
- `sutherlands const`: 110.4 K, for the viscosity's change with temperature.
- `prandtl no` 0.72 and `turbulent prandtl no` 0.9; the turbulent one plays no part in a laminar run.

`reference conditions`

`reference conditions` (p. 39) defines named flow states.

`IC_1` `IC_1` is the freestream.

- `temperature` 277.7777 K and `pressure` 101325 Pa.
- `v`: `vector` [1, 0, 0] and `mach` 0.2, so the flow runs along +x at 66.8 m/s.
- `reynolds no`: 20, based on the `reference length`, 1.0 m, the diameter of each cylinder. Above a Reynolds number of about 47, a single cylinder sheds vortices: they peel off alternately behind it, so the flow never settles to a steady state. Two cylinders 1.6 diameters apart behave more like one wider body, so they start to shed sooner; 20 is chosen to stay below that, so the flow settles to a steady state. The run confirms it by converging.

`initialisation`

`initialisation` (p. 47) sets the starting state.

- `initial conditions`: `IC_1` in every cell of every mesh.

`numerical scheme`

`numerical scheme` (p. 74) sets the spatial discretisation.

- `order`: `second`, with the default Roe flux and van Albada limiter. Choosing a limiter (p. 258) covers both.

`time settings`

`time settings` (p. 51) says how the run advances.

- `type`: `steady`. The flow at this Reynolds number does not change in time, so the solver simply iterates, in a made-up *pseudo-time*, until the solution stops changing. Choosing a time-marching scheme (p. 237) explains the choice.

`convergence control`

`convergence control` (p. 56) sets how the pseudo-time iterations converge to the steady state.

`scheme`

- `name`: `implicit euler`. An overset case needs an implicit pseudo-time scheme; see Practical consequences (p. 334).
- `stage`: 1. It has no effect under implicit Euler, which has only one stage.

`cfl`

- `cfl`: 20, the pseudo-time CFL number. This overset stack diverges in its first cycles at 100 or more.
- `cfl viscous factor`: 1.0e-6. It scales the viscous part of each cell's pseudo-time step estimate, so at this value the convective wave speeds alone set the step.
- `cfl ramp`: starts at 0.1 and grows by 1.05 a cycle, so the first cycles, from a flow that has not yet seen the cylinders, are gentle.

`cycles`

- `cycles`: 2000, enough for the residuals, which measure how far the equations are from being satisfied, to fall several orders of magnitude, and for the probe to settle.

`output settings`

`output settings` (p. 84) sets the report, written as the run goes, and the solution files. Each model gets its own report and solution files, in its own folder.

`report`

- `frequency`: a row every 5 cycles in each model's report, with that model's residuals.
- `monitor`: `MR_1`, named `probe`, records `cp` at the `point` [1.07, 0.25, 0.313], between and behind the cylinders. Every model's report has the column `probe_cp`, but only the model whose active cell holds the point fills it: here the refinement region. In the others it stays zero.

`solution`

- `format`: `vtk`, VTK HDF files for ParaView.
- `surface variables`: `V`, `p`, `T`, `rho` and `cp` on each model's boundaries.
- `volume variables`: the same, with `mach`, and `overset`: 0 in a cell another mesh has taken over, 1 in a cell that is active, and 2 in a cell kept active only by `additional active layers`, in the band where two meshes overlap. Step 5 uses it to show only the active cells.
- With no `frequency` set, the solution files and the checkpoint are written once, at the end of the run.

`model`

`model` (p. 96) has one entry for each mesh. Each name, such as `background`, names that model's output folder, `overset_cylinders_OUTPUT/background/`, and its files. The order matters: each mesh oversets meshes before it in the list. `background` is walked in full below; the others differ only as each says.

`background`

The coarse mesh beneath the others, 100 m square.

`mesh` `background.h5`, 10,000 cells.

`transforms` `transforms` (p. 169) places the mesh when it is read.

- `transform`: a 4 × 4 matrix whose last column translates the mesh. Here it moves the mesh 0.5 m down z, so that its span, built from z = 0.5 to 1.5, lines up with the other meshes' span, z = 0 to 1. Placing and moving a mesh (p. 365) describes the matrix.

`boundary conditions` `boundary conditions` (p. 98) assign a condition to each group of the mesh's zones.

BC_1

- `zones` [9, 11], the two faces normal to the span, `type symmetry`, so one cell across the span is enough.

BC_2

- `zones` [10], the outer boundary, `type farfield` (p. 286) with `condition IC_1` and `kind riemann`.

`refinement_region`

A finer Cartesian mesh round both cylinders, 12,800 cells in `refinement_region.h5`, with no transform.

`boundary conditions` Its `BC_1` is the same symmetry. Its `BC_2` is where it meets the background:

BC_2

- `zones`: [10], the outer boundary of the refinement region.
- `type interface` and `kind overset`: the boundary of an overset mesh. The background cells that lie under the refinement region are not solved, and the two meshes exchange the solution across this boundary.
- `connection`: `model ["background"]`, the mesh it takes its data from. Naming it tells the solver exactly which pair of meshes to map; without it, the solver maps against every model before this one.
- `additional active layers`: 3. The hole the refinement region cuts in the background would otherwise match its own outline exactly, leaving no band where both meshes are solved, and the interface faces would find no active cell to take their data from. Three extra layers of background cells are kept active round the hole, so the two meshes overlap. Donor search (p. 332) explains why this matters.

Overset interface (p. 328) lists its other settings.

`cylinder_2`

The lower cylinder: `cylinder.h5`, 5040 cells round a cylinder of 1 m diameter.

- `transforms`: `transform` moves it 0.8 m down y.
- `BC_1`: symmetry on zones [11, 12].
- `BC_2`: an overset interface on zone 13, its outer boundary, with `connection model ["refinement_region"]` and, for the same reason as the refinement region's, `additional active layers` 3.
- `BC_3`: zone 10, the cylinder surface, a no-slip `wall`.

`cylinder_1`

The upper cylinder: the same mesh and conditions as `cylinder_2`, moved 0.8 m up y instead. One mesh file can serve several models, each placed by its own transform.

The two cylinder meshes overlap each other between the cylinders. That is allowed, because each names its own donor, the refinement region; in each place the solution is taken from the topmost mesh.

Step 3. Run

In the activated environment, check the control file and run it:

```
cd ~/tutorial_5_overset_cylinders/
validate_input overset_cylinders.py
run_zcfd -c overset_cylinders.py
```

All four meshes are read from the `mesh` key of each model. These are extracts from the log of a run of this tutorial, `overset_cylinders.log`, with `...` marking lines left out:

```
Parameters validated successfully
...
 Reading file background.h5
 Global Mesh Stats: Cells: 10000 Faces: 40200
...
 Reading file refinement_region.h5
 Global Mesh Stats: Cells: 12800 Faces: 51440
...
 Reading file cylinder.h5
 Global Mesh Stats: Cells: 5040 Faces: 20280
...
Overset hole cut: blanked 168 background cells enclosed by the hole (inside the overset
 ↪  mesh's solid bodies)
...
Overset mapping: 104 of 288 (36.1111%) background halo centres found no active donor cell in
 ↪  the overset mesh. Those faces fall back to the unmapped face treatment instead of
 ↪  interpolated data. The meshes do not overlap deeply enough: increase 'additional active
 ↪  layers' on the overset interface.
...
============================================================
 Overset Mesh Hierarchy (4 models, 6 pairs)
============================================================
 background                  [root]
 └── refinement_region       [overset]
     ├── cylinder_2              [overset]
     └── cylinder_1              [overset]
 Peer overlaps: background ↔ cylinder_2, background ↔ cylinder_1, cylinder_2 ↔ cylinder_1
============================================================
...
 Solve Complete: 2000 cycles in 2m 27.7s (avg 73.8 ms/cycle)
...
zCFD finished: all requested cycles are complete (2000 in total).
```

- Each mesh is read in turn; `cylinder.h5` is read twice, once for each cylinder.
- The hole cut blanks the background cells inside each cylinder.
- The hierarchy shows which mesh oversets which, as the `connection` entries ask.
- As the run goes, a report box every few cycles gives each model's residuals separately.
- The log may print a few lines starting `Overset mapping:`. Each reports the share of an interface's halo cells, the extra cells along the interface that hold the other mesh's values, that found no active donor cell to take data from; those faces use the unmapped-face treatment instead of interpolated data.

They are diagnostics, not errors. The three `additional active layers` set above reduce them, and this case converges cleanly with the few that remain. Donor search (p. 332) explains these lines.

- The run ends after its 2000 cycles.

Output files

FILE/DIRECTORY	DESCRIPTION
`overset_cylinders.log`	A copy of the output to screen during the run.
`overset_cylinders_status.yaml`	How the run was started: version, date, control file, meshes.
`overset_cylinders_OUTPUT/background/`	The background's report, checkpoint, volume and surface files.
`overset_cylinders_OUTPUT/refinement_region/`	The same for the refinement region.
`overset_cylinders_OUTPUT/cylinder_1/`	The same for the upper cylinder.
`overset_cylinders_OUTPUT/cylinder_2/`	The same for the lower cylinder.
`overset_cylinders_OUTPUT/LOGGING/`	A separate log for each MPI rank, used for debugging.

Each model folder holds files named after the model, such as `cylinder_1/cylinder_1_report.csv` and `cylinder_1/cylinder_1.vtkhdf`.

Step 4. Monitor

Open the run in zMon:

```
zmon overset_cylinders.py
```

zMon shows each model's residuals and monitors against cycle; Monitoring a run (p. 324) describes the panels.

- Each model has its own residuals. The cylinder meshes start with the largest, as the flow first meets the cylinders, and fall more than four orders of magnitude. The refinement region starts from the uniform freestream, so its residual starts near zero, rises as the disturbance reaches it, then falls. The background's residual stays at round-off level, around 10⁻¹⁸, throughout.
- The `probe_cp` monitor sits in the wake between and behind the cylinders. Choose the refinement region's report: it is the mesh whose active cell holds the point, and the other models show zero. Once the run has converged the probe holds a steady value.

Step 5. Post-process

First activate your zCFD environment, which provides Python with zutil, h5py and ParaView:

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

Python

This script prints each model's largest and final density residual, and the final value at the probe. Save it as `print_convergence.py` beside the control file:

```
from zutil.fileutils import get_zcfd_result

result = get_zcfd_result("overset_cylinders.py")
for model, report in result.model_reports.items():
    rho = report.data["rho"]
    print(f"{model}: density residual fell from {rho.max():.1e} to {rho.iloc[-1]:.1e}")
probe = result.model_reports["refinement_region"].data["probe_cp"]
print(f"probe cp at the end of the run: {probe.iloc[-1]:.3f}")
```

```
python print_convergence.py
```

ParaView

Overset meshes keep their solutions separate, so you can load only the meshes covering the region you want. To view the whole domain as one, combine them in ParaView:

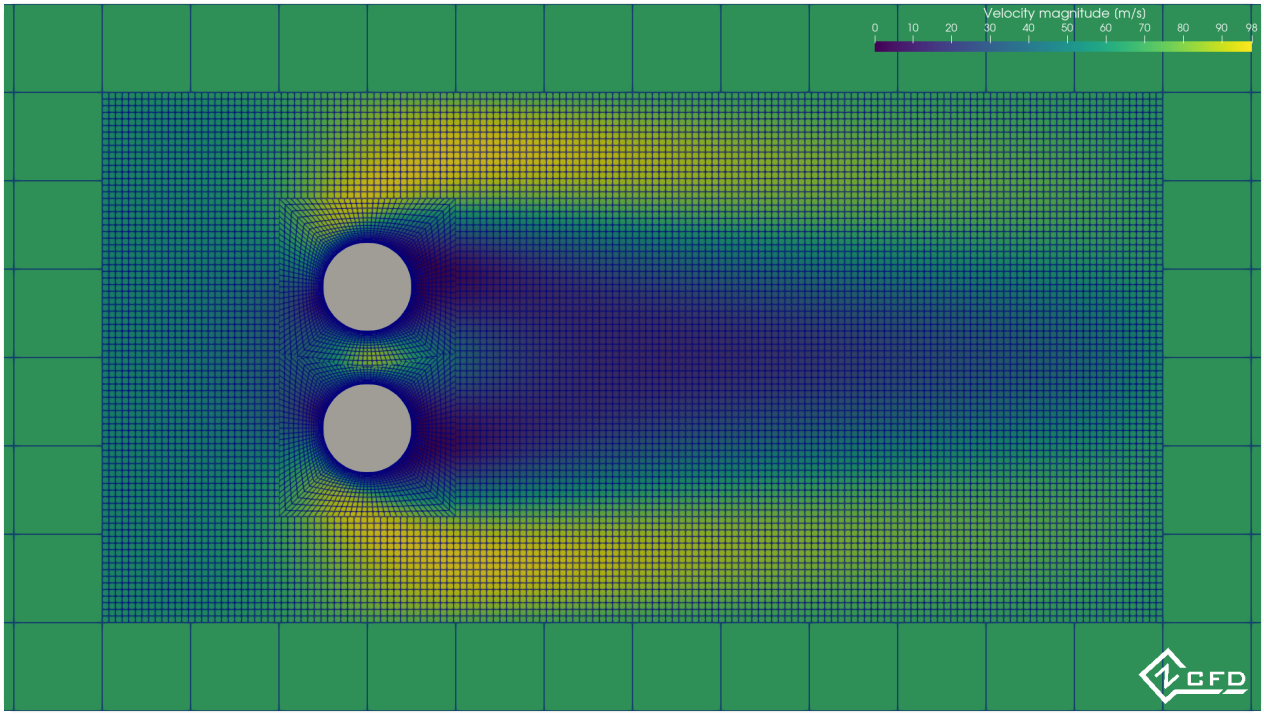
- Load the four files `overset_cylinders_OUTPUT/background/background.vtkhdf`, `.../refinement_region/refinement_region.vtkhdf`, `.../cylinder_1/cylinder_1.vtkhdf` and `.../cylinder_2/cylinder_2.vtkhdf`.

> **NOTE**
>
> To make this easier, `generate_paraview_helpers` writes ParaView scripts that load every model's output for you. Run it, then run the volume script once with `pvpython` to save its state:
>
> ```
> (zCFD)> generate_paraview_helpers overset_cylinders.py
> (zCFD)> pvpython paraview_view_volumes.py
> ```
>
> Open `paraview_view_volumes.pvsm` with *File › Load State…* to load all four volumes, each in its own colour. The overset cells are still there; the steps below remove them. See Opening a run in ParaView (p. 337).

- Select all four in the pipeline browser and apply *Filters › Group Datasets*. The new *GroupDatasets1* holds all four.
- Apply *Filters › Merge Blocks* to *GroupDatasets1*. ParaView reads each .vtkhdf file as one block per partition, which *Append Datasets* cannot take; *MergeBlocks1* is a single dataset holding the cells of all four meshes.
- Apply *Filters › Threshold* to *MergeBlocks1* on the `overset` variable, with the lower threshold raised from 0 to 1. *Threshold1* keeps only the active cells.
- Apply *Filters › Cell Data to Point Data* to *Threshold1*, which makes later interpolation more accurate.
- Apply *Filters › Slice* with its normal along z. The slice holds data from all four meshes.
- Colour *Slice1* by `V` *Magnitude*, with *Surface With Edges* in the representation toolbar.

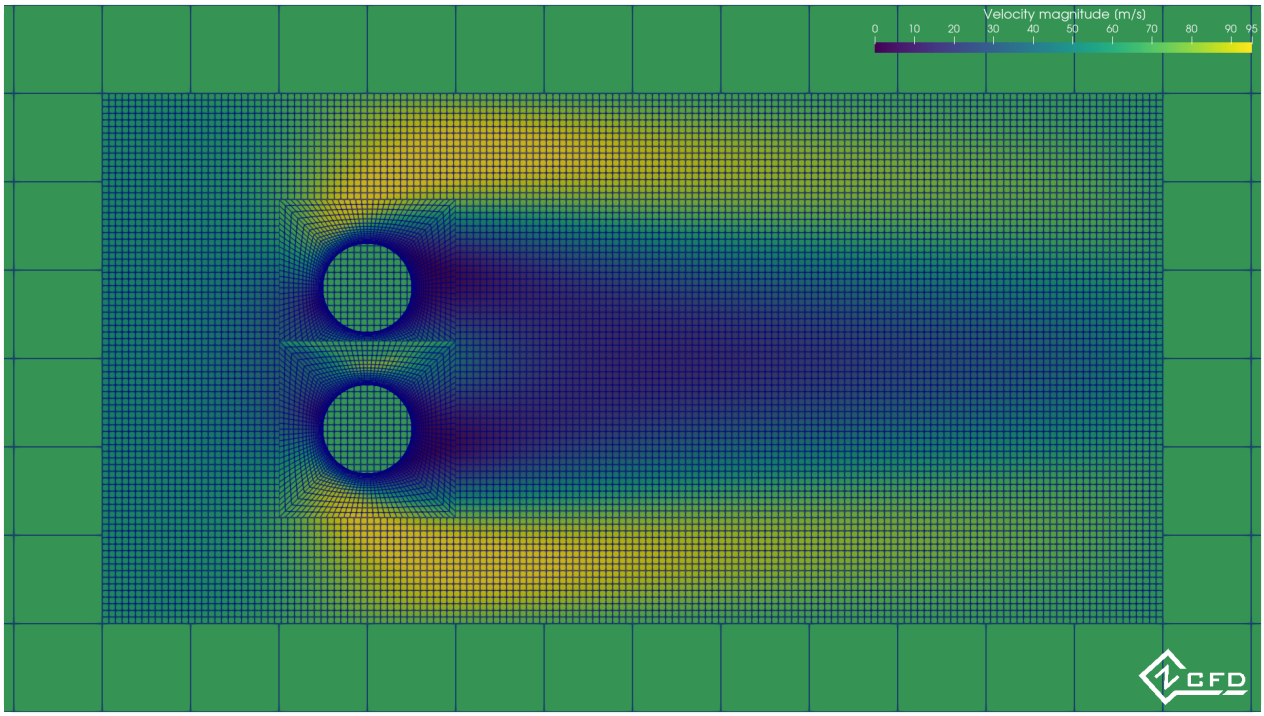
Streamlines and other filters work the same way. Where meshes move during an unsteady run, ParaView updates their positions.

The cylinder meshes have holes where the cylinders are, so the refinement region shows through behind them. To hide it, add a cylinder of unit diameter with *Sources › Cylinder* at [0, 0, 0], rotate it 90° about x and translate it by −0.8 in y; repeat with +0.8 for the other cylinder. The view then looks like this:



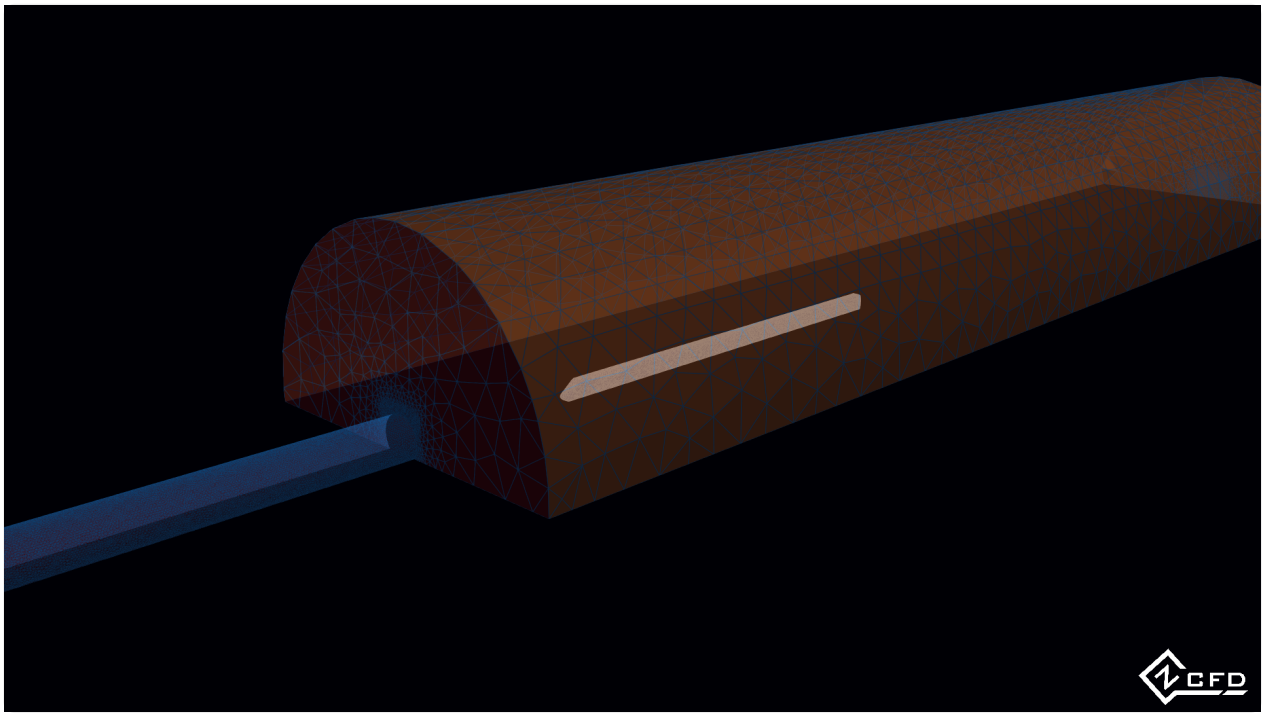
Velocity magnitude on a slice through the two cylinders. The background solution shows through the holes the cylinder meshes cut, so blocky cells surround each cylinder.

To bring the cylinder meshes to the front instead, view each mesh separately and give each cylinder mesh a small offset towards the viewer, normal to the slice (0.0, 0.001, 0.0015, 0.002). That is closer to what the solver sees, since the covered cells of the meshes beneath are not solved.



The same slice with each cylinder mesh offset slightly towards the viewer, so the overset solutions sit in front of the background mesh.

Example 2. Train in a tunnel



The train (white) inside the tunnel mesh: the open-air region round the portal (orange, drawn see-through) and the tunnel bore (blue). The mesh round the train is overset on the tunnel mesh, so moving it through the tunnel simulates the train's motion without making a new mesh at each physical time step.

A train entering a tunnel pushes a pressure wave ahead of it, which leaves the far end as a micro pressure wave: a boom. This example predicts the wave, with the set-up of Section 7.6 of EN 14067-5, *Railway applications — Aerodynamics — Part 5: Requirements and test procedures for aerodynamics in tunnels* [1]. For the reference train entering the reference tunnel at 250 km/h, the maximum entry pressure gradient dp/dt should lie between 8800 Pa/s and 9700 Pa/s.

Step 1. Download the files

The bundle can be downloaded here.

```
cd ~
wget https://zcfd.zenotech.com/v2026.11.16174/traintunnel.tgz
tar xzf traintunnel.tgz
cd traintunnel
```

It contains the control file `traintunnel.py` and two meshes: `tunnel.h5`, 7.07 million cells, and `train.h5`, 1.06 million.

Step 2. Walk through the input

The control file, `traintunnel.py`, is a short Python script. All it does is build one Python dictionary, called `parameters`, that tells zCFD everything about the run: the physics, the numerical method, what to write out, and the mesh with its boundaries. This step walks through that dictionary from top to bottom. Each heading below is the name of a key, written exactly as it appears in the file, and the headings nest the same way the dictionary does: `numerical scheme`, for example, sits inside `solver`.

Under each heading you'll find what that part controls, what each setting means, and why it has the value it has here. Each section ends with a link to that part's reference page, which lists every setting it accepts and its default; you don't need it to follow the tutorial. On a wide screen the whole control file sits on the right, with the part being described highlighted as you scroll.

Here `model` has two entries, the tunnel and the train. Where a block works as in the cylinders above, it is covered briefly.

```
parameters = {
    "config version": 2,
    "solver": {
        "solver settings": {"type": "compressible"},
        "initialisation": {"initial conditions": "IC_1"},
        "equations": {"type": "euler"},
        "reference conditions": {
            "IC_1": {
                "temperature": 292.7777,
                "pressure": 101325.0,
                "v": {"vector": [1.0, 0.0, 0.0], "mach": 1e-10},
                "reference length": 1.0,
            }
        },
        "time settings": {
            "type": "unsteady",
            "kind": "dual time stepping",
            "total time": 3.6,
            "time step": 0.001,
            "start": 0,
            "order": 1,
        },
        "convergence control": {
            "scheme": {"name": "implicit euler", "stage": 1},
            "cfl": {"cfl": 30},
            "cycles": 5,
        },
        "numerical scheme": {
            "order": "second",
        },
        "fluid properties": {
            "material": "ideal gas",
            "gamma": 1.4,
            "gas constant": 287.0,
            "sutherlands const": 110.4,
            "prandtl no": 0.72,
        },
        "output settings": {
            "report": {
                "frequency": 10,
                "monitor": {
                    "MR_1": {
                        "name": "probe",
                        "point": [-140, 1.5, 0],
                        "variables": ["p"],
```

```
                    }
                },
            },
            "solution": {
                "format": "vtk",
                "surface variables": ["overset", "V", "p", "T", "rho"],
                "volume variables": ["overset", "V", "p", "T", "rho", "m", "sponge"],
            },
        },
    },
    "model": {
        "background": {
            "mesh": "tunnel.h5",
            "boundary conditions": {
                "BC_1": {"zones": [11, 13], "type": "symmetry"},
                "BC_2": {
                    "zones": [12],
                    "type": "farfield",
                    "kind": "riemann",
                    "condition": "IC_1",
                    "sponge layer damping": {
                        "distance": 20.0,
                        "damping factor": 2.037495,
                        "condition": "IC_1",
                    },
                },
            },
        },
        "train": {
            "mesh": "train.h5",
            "transforms": {
                "transform": [
                    [1.0, 0.0, 0.0, 110.0],
                    [0.0, 1.0, 0.0, 0.0],
                    [0.0, 0.0, 1.0, 0.0],
                    [0.0, 0.0, 0.0, 1.0],
                ],
                "motion": {
                    "type": "translating",
                    "velocity": [-69.4, 0.0, 0.0],
                    "moving mesh": True,
                    "ramp time": 1.5,
                },
            },
            "boundary conditions": {
                "BC_1": {"zones": [10], "type": "symmetry"},
                "BC_2": {
                    "zones": [9],
                    "type": "interface",
                    "kind": "overset",
                    "create overset halos": False,
                    "interpolation method": "nearest cell",
                },
```

```
                "BC_3": {"zones": [8], "type": "wall", "kind": "slip"},
            },
        },
    },
}
```

`config version`

`"config version": 2`, the layout of the control file.

`solver`

`solver` holds the settings shared by both meshes.

`solver settings`

- `type`: `compressible`, the density-based solver.

`initialisation`

- `initial conditions`: `IC_1` in every cell: still air everywhere, with the train at rest.

`equations`

- `type`: `euler`, the inviscid Euler equations. The pressure wave is a compressible, inviscid effect, so viscosity is left out, which keeps a case of this size affordable.

`reference conditions`

`IC_1`

- `temperature` 292.7777 K (19.6 °C) and `pressure` 101325 Pa.
- `v`: `vector` [1, 0, 0] and `mach` 1e-10: still air. The train moves; the air does not. The Mach number is not set to exactly zero, because the solver scales by it.
- `reference length`: 1.0 m.

`time settings`

- `type unsteady`, `kind dual time stepping`: the train moves, so the run follows the flow in time. Each physical time step is solved like a small steady problem, by the inner iterations set in `convergence control`.
- `total time`: 3.6 s, long enough for the train to accelerate and enter the tunnel.
- `time step`: 0.001 s, 3600 steps.
- `start`: 0, no steady cycles first.
- `order`: 1, first order in time.

`convergence control`

- `scheme`: `implicit euler`. Its `stage` of 1 has no effect.
- `cfl`: 30, without a ramp.
- `cycles`: 5 inner cycles per time step.

`numerical scheme`

- `order`: `second`, with the default Roe flux and van Albada limiter.

`fluid properties`

- `material`: `ideal gas`, with `gamma` 1.4 and `gas constant` 287.0 J/kg/K.
- `sutherlands const` and `prandtl no` play no part in an Euler run, which has no viscosity or heat conduction.

`output settings`

`report`

- `frequency`: a row every 10 cycles.
- `monitor`: `MR_1`, named `probe`, records the pressure `p` at the `point` [-140, 1.5, 0], inside the tunnel. It falls in the tunnel mesh, so its column `probe_p` is filled in `traintunnel_OUTPUT/background/background_report.csv`. Its history gives the entry pressure gradient in Step 5.

`solution`

- `format`: `vtk`, VTK HDF files for ParaView.
- `surface variables`: `overset`, `V`, `p`, `T` and `rho`.
- `volume variables`: the same, with the Mach number `m` and `sponge`, the damping applied by the sponge layer below.

`model`

`background`

The tunnel and the air round it. Its name gives the output folder, `traintunnel_OUTPUT/background/`.

`mesh` `tunnel.h5`, 7.07 million cells.

`boundary conditions`

`BC_1`

- `zones` [11, 13], `type symmetry`.

`BC_2`

- `zones`: [12], the outer boundary.
- `type`: `farfield`, with `kind riemann` and `condition IC_1`.
- `sponge layer damping`: within `distance` 20 m of the farfield, the solution is damped towards its `condition`, `IC_1`, at a rate rising to the `damping factor`, 2.037495, at the boundary. The sponge absorbs pressure waves before they reach the farfield, where they would reflect back into the domain.

`train`

The train, with its output in `traintunnel_OUTPUT/train/`.

`mesh` `train.h5`, 1.06 million cells.

`transforms`

- `transform`: a 4 × 4 matrix whose last column places the train 110 m along x before it starts to move.
- `motion`: how the train moves during the run.
  - `type`: `translating`, a straight-line motion.
  - `velocity`: [-69.4, 0, 0] m/s, 250 km/h towards −x and the tunnel.
  - `moving mesh`: `True`, so the train mesh itself moves through the tunnel mesh. Every time step it is moved to the position the motion gives at the end of the step, and its overset mappings are recalculated.
  - `ramp time`: 1.5 s. The speed rises linearly from rest to the full velocity over the first 1.5 s, which avoids the transients of an impulsive start.

A moving mesh is allowed only in an unsteady compressible run, on a mesh that also has an overset or sliding interface. Placing and moving a mesh (p. 365) describes `transforms`.

`boundary conditions`

`BC_1`

- `zones` [10], `type symmetry`.

`BC_2`

- `zones` [9], the outer boundary of the train mesh, `type interface`, `kind overset`. It names no `connection`, so it is mapped against every model before it: here only the tunnel.
- `create overset halos`: `False`. Donor values are taken at the boundary face centres. The default samples at the halo cell centres, just outside the train mesh. On this moving train, some of those points have no donor in the tunnel mesh.
- `interpolation method`: `nearest cell`: each point takes the value of the donor cell that contains it.

`BC_3`

- `zones`: [8], the train's surface.
- `type wall` with `kind slip`: in the Euler equations the flow slides along a wall rather than stopping at it.

Step 3. Run

The case needs about 48 GB of GPU memory. Run it in the activated environment:

```
validate_input traintunnel.py
run_zcfd -c traintunnel.py
```

Step 4. Monitor

```
zmon traintunnel.py
```

zMon shows each model's residuals, and the `probe_p` monitor in the reports panel of the background model: the pressure inside the tunnel rising as the train enters.

Step 5. Post-process

First activate your zCFD environment, which provides Python with zutil, h5py and ParaView:

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

Python

This script plots the entry pressure gradient: how fast the pressure inside the tunnel rises, the time derivative of `probe_p`. Save it as `plot_entry_pressure_gradient.py` in the case directory:

```
import matplotlib.pyplot as plt
import numpy as np

from zutil.fileutils import get_zcfd_result

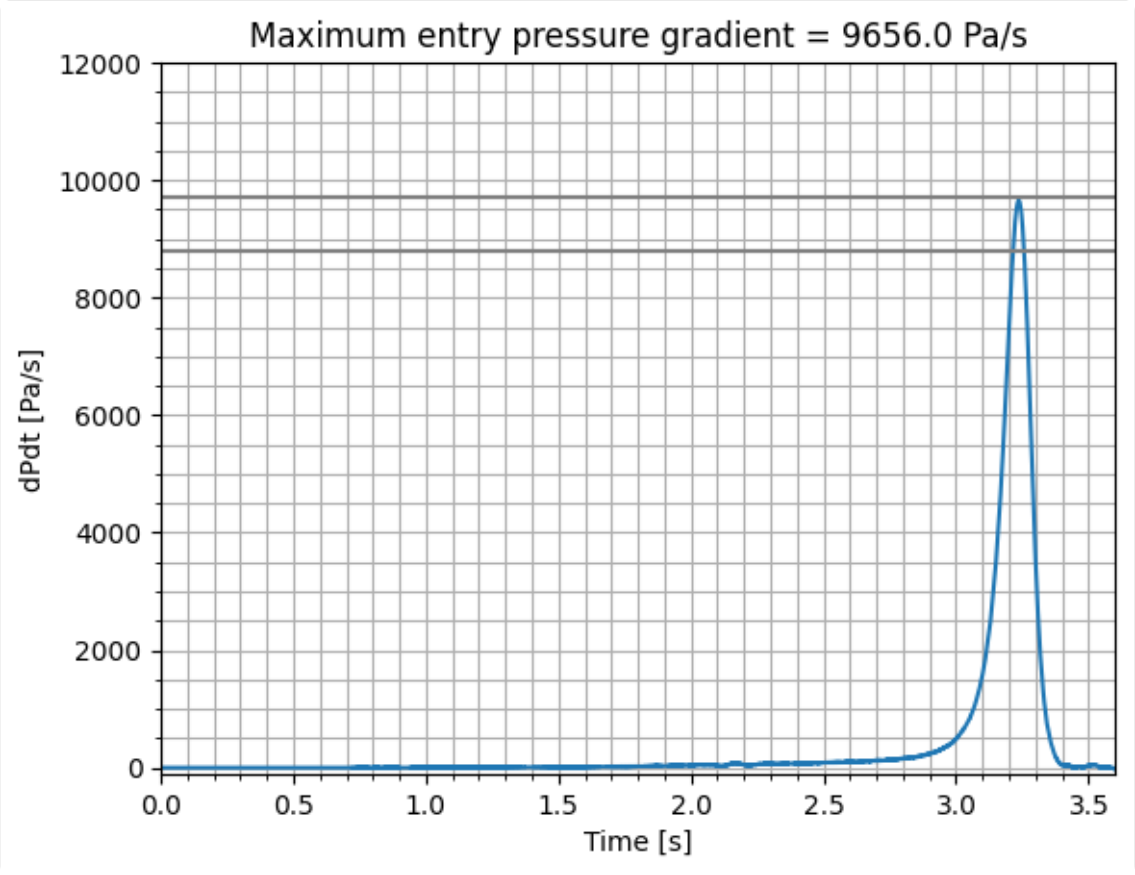
time_step = 0.001
result = get_zcfd_result("traintunnel.py")
p = np.array(result.model_reports["background"].get_array("probe_p"))
dpdt = np.diff(p) / time_step
time = time_step * np.arange(len(dpdt))

fig, ax = plt.subplots()
ax.minorticks_on()
ax.grid(which="both")
ax.set_title(f"Maximum entry pressure gradient = {dpdt.max():.1f} Pa/s")
ax.set_xlabel("Time [s]")
ax.set_ylabel("dPdt [Pa/s]")
ax.set_xlim(0, 3.6)
ax.set_ylim(-100, 12000)
ax.plot(time, dpdt, label="250 km/h")
ax.axhline(8800, label="min valid dP/dT", c="grey")
ax.axhline(9700, label="max valid dP/dT", c="grey")
ax.legend()
fig.savefig("EntryPressureGradient.png")
```

Run it:

```
python plot_entry_pressure_gradient.py
```

It plots the entry pressure gradient against time. The maximum is 9590 Pa/s, within the 8800 Pa/s to 9700 Pa/s the standard specifies.



Entry Pressure Gradient.

Citations

1. EN 14067-5:2006 - Railway applications - Aerodynamics - Part 5: Requirements and test procedures for aerodynamics in tunnels, see

Incompressible flow

What is in this tutorial?

This tutorial solves the flow in a lid-driven cavity: a closed square box of fluid whose top wall, the lid, slides sideways at a constant speed. The moving lid drags the fluid beneath it round into one large swirl, or vortex, with two small vortices turning the other way in the bottom corners. The case is one of the most widely used checks of a CFD code, because accurate published results exist for it: here, the velocities that Ghia, Ghia and Shin (1982) tabulate along the two lines through the middle of the box, the centrelines.

The flow is set by its Reynolds number, the ratio of the fluid's inertia to its viscosity: low values give slow, smooth, syrupy flow, high values fast, turbulent flow. At the Reynolds number used here, 400, the flow settles to a steady pattern that stays the same over time.

The tutorial uses zCFD's incompressible solver. An incompressible solver assumes the density of the fluid never changes, and solves only for its velocity and pressure. That holds for liquids such as water, and for gases moving well below the speed of sound: as a rule of thumb, below Mach 0.3, where the Mach number is the flow speed divided by the speed of sound. The compressible solver, used in Simple aerofoil (p. 373), lets the density change. Use it instead where a flow is mostly slow but the density changes somewhere, through heating or a fast region.

What you need?

Work through Simple aerofoil (p. 373) first: this tutorial relies on it for running zCFD, reading the log and monitoring a run in zMon. The run takes about fifteen seconds on an NVIDIA RTX 3060.

Step 1. Download the files

The tutorial bundle can be downloaded here. It contains a mesh file (`cavity.h5`), a zCFD control file (`cavity.py`), and Ghia's tabulated centreline velocities at Re 400 (`ghia_re400.csv`).

```
cd ~
wget https://zcfd.zenotech.com/v2026.11.16174/tutorial_6_lid_driven_cavity.tgz
tar xzf tutorial_6_lid_driven_cavity.tgz
cd tutorial_6_lid_driven_cavity
```

The cavity is 1 m square. The mesh has 127 × 127 cells, clustered towards the walls where the velocity changes fastest, and is one cell thick: the flow is two-dimensional, with symmetry conditions on the front and back faces.

Step 2. Walk through the input

The control file, `cavity.py`, is a short Python script. All it does is build one Python dictionary, called `parameters`, that tells zCFD everything about the run: the physics, the numerical method, what to write out, and the mesh with its boundaries. This step walks through that dictionary from top to bottom. Each heading below is the name of a key, written exactly as it appears in the file, and the headings nest the same way the dictionary does: `numerical scheme`, for example, sits inside `solver`. Under each heading you'll find what that part controls, what each setting means, and why it has the value it has here. Each section ends with a link to that part's reference page, which lists every setting it accepts and its default; you don't need it to follow the tutorial. On a wide screen the whole control file sits on the right, with the part being described highlighted as you scroll.

```
parameters = {
    "config version": 2,
    "solver": {
        "solver settings": {"type": "incompressible"},
        "equations": {"type": "viscous"},
        "fluid properties": {
            "material": "ideal gas",
            "gamma": 1.4,
            "gas constant": 287.0,
            "sutherlands const": 110.4,
            "prandtl no": 0.72,
            "turbulent prandtl no": 0.9,
        },
        "reference conditions": {
            "IC_1": {
                "temperature": 273.15,
                "pressure": 101325.0,
                "v": {"vector": [1.0, 0.0, 0.0]},
                "reynolds no": 400.0,
                "reference length": 1.0,
            },
            "IC_2": {
                "temperature": 273.15,
                "pressure": 101325.0,
                "v": {"vector": [0.0, 0.0, 0.0]},
                "reynolds no": 400.0,
            },
        },
        "initialisation": {"reference": "IC_1", "initial conditions": "IC_2"},
        "numerical scheme": {"order": "second"},
        "time settings": {"type": "steady"},
        "convergence control": {
            "scheme": {"name": "coupled", "implicit relax": 0.9},
            "cycles": 500,
        },
        "output settings": {
            "report": {"frequency": 1},
            "solution": {
                "format": "vtk",
                "surface variables": ["V", "p"],
```

```
                "volume variables": ["V", "p", "cell_centre"],
            },
        },
    },
    "model": {
        "cavity": {
            "mesh": "cavity.h5",
            "boundary conditions": {
                "BC_1": {
                    "zones": [14],
                    "type": "wall",
                    "kind": "no slip",
                    "velocity": {"type": "linear", "vector": [1.0, 0.0, 0.0]},
                },
                "BC_2": {
                    "zones": [12],
                    "type": "wall",
                    "kind": "no slip",
                },
                "BC_3": {
                    "zones": [11, 13],
                    "type": "symmetry",
                },
            },
        }
    },
}
```

`config version`

Says which layout of control file this is, so zCFD knows how to read the rest.

- `config version`: `2`, the current layout, with `solver` and `model` at the top level.

`solver`

The settings shared by the whole simulation: which solver runs, what it solves, the fluid, the flow states, and how the run marches and what it writes.

`solver settings`

Chooses the solver.

- `type`: `"incompressible"`: the solver that holds the density constant and solves for velocity and pressure. The lid moves at 1 m/s, far below the speed of sound, so the density does not change.

See `solver settings` (p. 11).

`equations`

Sets the equations being solved.

- `type`: `"viscous"`: the Navier–Stokes equations, which describe a fluid with viscosity, solved without a turbulence model. At a Reynolds number of 400 the flow is laminar: it moves in smooth layers, with none of the chaotic eddies of turbulence.

See `equations` (p. 18), and Choosing a turbulence model (p. 226) for flows that are turbulent.

`fluid properties`

Describes the fluid.

- `material`: `"ideal gas"`: pressure, density and temperature are related by the ideal gas law.
- `gamma`: `1.4`, the ratio of specific heats, a property of the gas.
- `gas constant`: `287.0` J/kg/K. With the reference pressure and temperature it gives the density, 1.29 kg/m³, which the solver then holds constant.
- `sutherlands const`: `110.4` K, for Sutherland's law, which varies the viscosity with temperature. The temperature does not change in this run, and the viscosity is set by the Reynolds number instead.
- `prandtl no` and `turbulent prandtl no`: `0.72` and `0.9`, which relate heat conduction to viscosity. This run solves no energy equation, so they play no part.

See `fluid properties` (p. 37).

`reference conditions`

Defines named flow states, each a set of values such as temperature, pressure and velocity. Other parts of the file refer to them by name: to fill the domain at the start, to set conditions at boundaries, and to scale the equations.

`IC_1` The reference state, moving at the lid's speed.

- `temperature` and `pressure`: `273.15` K and `101325.0` Pa, which with the gas constant fix the density.
- `v`: `{"vector": [1.0, 0.0, 0.0]}`, 1 m/s along x, the lid speed.
- `reynolds no`: `400.0`, the Reynolds number of the flow, based on the reference length.
- `reference length`: `1.0` m, the side of the cavity. With the speed and density, Re 400 sets the viscosity to 0.00323 Ns/m².

`IC_2` The fluid at rest.

- `temperature` and `pressure`: as `IC_1`.
- `v`: `{"vector": [0.0, 0.0, 0.0]}`, still fluid.
- `reynolds no`: `400.0`, as `IC_1`.

See `reference conditions` (p. 39) and Viscosity (p. 295).

`initialisation`

Chooses which states start the run and scale it.

- `reference`: `"IC_1"`. The solver scales its equations by this state, so the lid speed sets the scale.
- `initial conditions`: `"IC_2"`. The fluid starts at rest, and the moving lid sets it in motion.

See `initialisation` (p. 47) and Choosing initialisation and reference state (p. 293).

`numerical scheme`

Sets how the equations are discretised in space: how the smooth flow is represented by one value in each cell of the mesh.

- `order`: `"second"`: the solver estimates how the velocity varies across each cell, not just its average, which is more accurate than treating it as constant (first order).

See `numerical scheme` (p. 74) and Choosing a spatial scheme (p. 254).

`time settings`

Says what kind of run this is.

- `type`: `"steady"`: at Re 400 the cavity flow settles to a pattern that does not change with time, so the run looks for that final state directly, rather than following the flow through time.

See `time settings` (p. 51).

`convergence control`

Sets how the solver reaches the steady solution. It starts from a guess and improves it once per cycle, or iteration, until the flow stops changing; the run has then converged. The incompressible solver needs no CFL number, the step-size setting the compressible tutorials use.

`scheme` In an incompressible flow the pressure is whatever it must be to keep the flow from piling up or emptying out anywhere, so velocity and pressure have to be found together. The scheme says how.

- `name`: `"coupled"`: each cycle solves for velocity and pressure together, as one large set of linear equations. It is the scheme to start from. `"simple"` instead solves for them one after the other and corrects the pressure in between.
- `implicit relax`: `0.9`: each cycle applies 90 % of the change it computes, which keeps the iterations stable.

The linear solvers, the routines that solve those sets of equations each cycle, keep their defaults, which are tuned for the incompressible equations; Incompressible solver (p. 264) lists them.

`cycles`

- `cycles`: `500`: the run stops after 500 cycles. Step 4 shows the residuals, the measure of how far the solution is from satisfying the equations, have fallen by more than six orders of magnitude, a factor of a million, by then.

See `convergence control` (p. 56) and Incompressible convergence control (p. 251).

`output settings`

Sets what zCFD writes, and how often. See `output settings` (p. 84).

`report` The report is a table with one row per report cycle, written to `cavity_OUTPUT/cavity/cavity_report.csv` and plotted by zMon.

- `frequency`: `1`: a row every cycle. The file asks for no forces, so each row holds only the residuals: one for each component of momentum, `rhoV[0]`, `rhoV[1]` and `rhoV[2]`, and one for pressure, `p`.

`solution` The flow solution, written for visualisation and post-processing.

- `format`: `"vtk"`: VTK HDF files, `.vtkhdf`, which ParaView opens.
- `surface variables`: `V` and `p`, written to one file for each type of boundary condition.
- `volume variables`: `V`, `p` and `cell_centre`, the coordinates of each cell centre, which Step 5 uses to pick out the cells along the centrelines.

No `frequency` is given, so the solution is written once, at the end of the run. Output variables (p. 336) lists the others.

`model`

One entry per mesh, each with its own boundary conditions. See `model` (p. 96).

`cavity`

The name of the only model. It names the output folder, `cavity_OUTPUT/cavity/`, and the files in it.

`mesh`

- `mesh`: `"cavity.h5"`, the mesh file in zCFD's format, read from the case directory. Its boundary faces are grouped into numbered zones, which the boundary conditions refer to.

`boundary conditions` Assigns a condition to each zone of boundary faces. See `boundary conditions` (p. 98) and Choosing boundary conditions (p. 285).

`BC_1` The lid.

- `zones`: `[14]`, the top face of the cavity.
- `type` and `kind`: `"wall"`, `"no slip"`: the fluid at the wall moves with the wall.
- `velocity`: `{"type": "linear", "vector": [1.0, 0.0, 0.0]}`: a translating wall, sliding at 1 m/s along x. A `"rotating"` velocity would spin it about an axis instead.

`BC_2` The other three walls.

- `zones`: `[12]`, the bottom and the two sides.
- `type` and `kind`: `"wall"`, `"no slip"`, with no velocity: the walls are still.

`BC_3` The front and back faces.

- `zones`: `[11, 13]`.
- `type`: `"symmetry"`: a mirror plane. No flow passes through it, so the flow stays two-dimensional across the single cell of depth.

The cavity is closed, so this file has no inflow or outflow. For a flow through a domain, the incompressible solver needs an inflow and an outflow boundary rather than the farfield used by the compressible tutorials.

Step 3. Run

1. Activate your environment:

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

2. Check the control file for errors:

```
cd ~/tutorial_6_lid_driven_cavity/
validate_input cavity.py
```

3. Run zCFD in the case directory:

```
run_zcfd -c cavity.py
```

These are extracts from the log of a run of this tutorial, with `...` marking lines left out:

```
Parameters validated successfully
...
 Reading file cavity.h5
 Global Mesh Stats: Cells: 16129 Faces: 64770
...
 Dimensional Conditions
 ----------------------
  Pressure:     101325 Pa
  Temperature:  273.15 K
  Density:      1.29251 kg/m^3
  Viscosity:    0.00323127 Ns/m^2 @ 273.15 K
  Speed:        1 m/s
...
  Reynolds No:  400 @ 273.15 K
...
Starting solve loop
...
Cycle     10 | Step    0 t=0s | CFL   1.000 | MG 1 |    16.6 ms | Total     0.212s
+-------------------------------------------------------------------------------------+
| Report: Cycle 10  Wall time: 0.212s                                                 |
...
| Residuals:                                                                          |
|   rhoV[0]  1.3464E-07  rhoV[1]  1.6890E-07  rhoV[2]  0.0000E+00  p        1.8761E-08 |
+-------------------------------------------------------------------------------------+
...
 Solve Complete: 500 cycles in 9.573s (avg 19.1 ms/cycle)
...
zCFD finished: all requested cycles are complete (500 in total).
```

- The mesh has 16,129 cells, 127 × 127 across one cell of depth.
- The dimensional conditions show the viscosity the Reynolds number set, 0.00323 Ns/m², for a lid speed of 1 m/s.
- The report box lists three momentum residuals and the pressure residual. `rhoV[2]`, across the depth,

is exactly zero, because the flow has no velocity across the symmetry planes.
- The run takes about fifteen seconds on an NVIDIA RTX 3060.

Output files

FILE/DIRECTORY	DESCRIPTION
`cavity.log`	A copy of the output to screen during the run.
`cavity_status.yaml`	A record of how the run was started: version, control file, mesh and devices.
`cavity_OUTPUT/cavity/cavity_report.csv`	The residuals at every cycle.
`cavity_OUTPUT/cavity/cavity_results.h5`	The checkpoint: the flow solution in zCFD format, used to restart the run.
`cavity_OUTPUT/cavity/cavity.vtkhdf`	The volume output, for visualisation and post-processing.
`cavity_OUTPUT/cavity/cavity_wall.vtkhdf`	The surface output on the four walls.
`cavity_OUTPUT/cavity/cavity_symmetry.vtkhdf`	The surface output on the front and back symmetry planes.

Step 4. Monitor

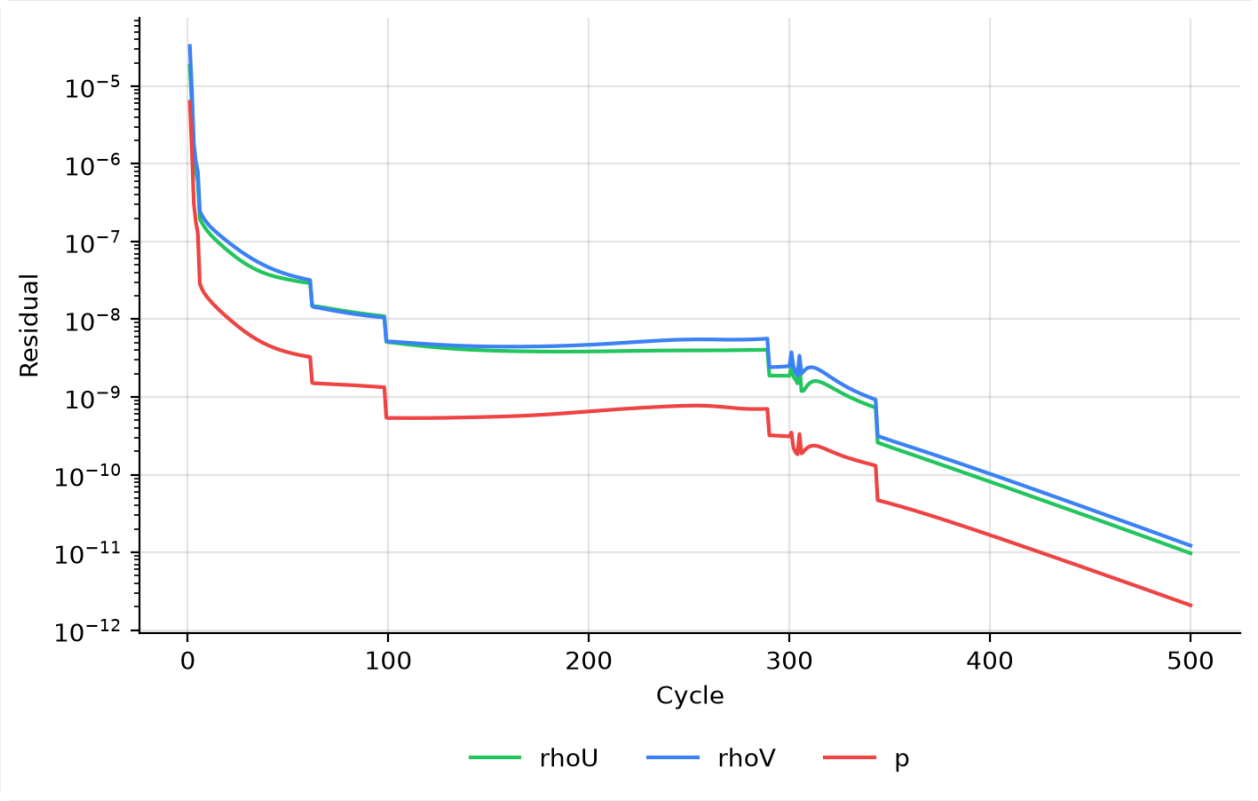
Open the run in zMon to watch the residuals, while it runs or after it has finished:

```
zmon cavity.py
```

Monitoring a run (p. 324) describes each panel. The plot below is the residual panel as zMon draws it.

Residuals

A residual measures how far the current solution is from satisfying one of the equations; as the run converges, the residuals fall. The report holds the momentum residuals, named `rhoU`, `rhoV` and `rhoW` in zMon, and `p`, the pressure residual, which measures how well the flow conserves mass. `rhoW` is zero throughout, so it does not appear on the logarithmic scale.



Residuals of the cavity run against cycle

The residuals fall steeply in the first few cycles, as the lid sets the fluid moving, then in steps as the vortex develops, and steadily from about cycle 350. All three fall by more than six orders of magnitude over the 500 cycles, and the solution is converged.

Step 5. Post-process

Python

First activate your zCFD environment, which provides Python with zutil, h5py and ParaView:

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

Ghia, Ghia and Shin tabulate two profiles: the x velocity u along the vertical centreline, x = 0.5, and the y velocity v along the horizontal centreline, y = 0.5. The mesh has an odd number of cells across, so a column of cell centres lies exactly on each centreline, and the profiles can be read straight from the cells. This script reads the volume output, picks out those cells, and plots them against Ghia's values. Save it as `compare_with_ghia.py` beside the control file:

```
import h5py
import matplotlib.pyplot as plt
import numpy as np

with h5py.File("cavity_OUTPUT/cavity/cavity.vtkhdf", "r") as f:
    centre = f["VTKHDF/CellData/cell_centre"][:]
    velocity = f["VTKHDF/CellData/V"][:]
```

```
# The cells on the vertical centreline (x = 0.5), bottom to top,
# and on the horizontal centreline (y = 0.5), left to right
column = np.flatnonzero(np.isclose(centre[:, 0], 0.5, atol=1e-5))
column = column[np.argsort(centre[column, 1])]
row = np.flatnonzero(np.isclose(centre[:, 1], 0.5, atol=1e-5))
row = row[np.argsort(centre[row, 0])]

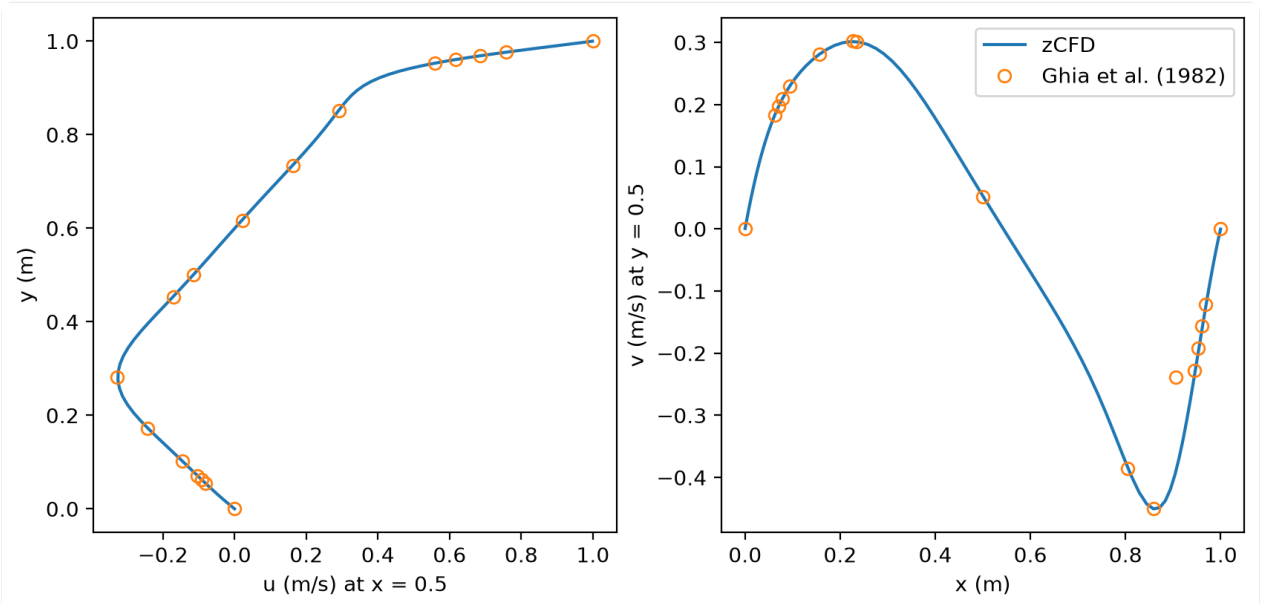
ghia = np.genfromtxt("ghia_re400.csv", delimiter=",", names=True)

fig, (left, right) = plt.subplots(1, 2, figsize=(10, 4.5))
left.plot(velocity[column, 0], centre[column, 1], label="zCFD")
left.plot(ghia["u"], ghia["y"], "o", mfc="none", label="Ghia et al. (1982)")
left.set_xlabel("u (m/s) at x = 0.5")
left.set_ylabel("y (m)")
right.plot(centre[row, 0], velocity[row, 1], label="zCFD")
right.plot(ghia["x"], ghia["v"], "o", mfc="none", label="Ghia et al. (1982)")
right.set_xlabel("x (m)")
right.set_ylabel("v (m/s) at y = 0.5")
right.legend()
fig.savefig("compare_with_ghia.png", dpi=200, bbox_inches="tight")
```

and run it:

```
python compare_with_ghia.py
```

It writes `compare_with_ghia.png`:



Centreline velocities from zCFD against Ghia et al. (1982)

zCFD matches Ghia's values to within 0.01 m/s, 1 % of the lid speed, at every tabulated point. The minimum u on the vertical centreline is −0.33 m/s, near y = 0.27, where the main vortex carries the fluid back beneath its centre. Along the horizontal centreline, v runs from 0.30 m/s on the upward side of the vortex to −0.45 m/s on the downward side, close to the right-hand wall.

NOTE

One of Ghia's v values, −0.238 at x = 0.906, lies off the curve traced by its neighbours, −0.228 at x = 0.945 and −0.450 at x = 0.859. zCFD gives −0.389 there, on the curve. Leave that point out when you compare the two.

ParaView

NOTE

The .vtkhdf output format requires ParaView 6.1.1 or later.

Open `cavity_OUTPUT/cavity/cavity_symmetry.vtkhdf` in ParaView and click "Apply". The symmetry plane spans the whole cavity, so it shows the flow in a single view. Colour it by `V`, magnitude, to see the fast-moving layer beneath the lid. To see the vortices, add a "Stream Tracer" filter, which draws the paths the fluid follows, with a "Line" seed type across the cavity; the main vortex fills most of the cavity, with the two smaller corner vortices below it.

Next steps

- Raise the Reynolds number to 1000 in both reference conditions. Ghia also tabulates Re 1000 and above; the primary vortex moves towards the centre of the cavity. Check that the residuals have converged before you compare.
- Change the coupling scheme to `"simple"`, following Incompressible convergence control (p. 251), and compare how many cycles each scheme needs to converge.

Citations

Ghia, U., Ghia, K. N. and Shin, C. T. (1982). High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics*, 48(3), 387–411.

Rotor in hover

What is in this tutorial?

This tutorial simulates a helicopter-style rotor hovering in still air, and compares the pressure on its blades with wind-tunnel measurements. Along the way it shows three things you will need for your own cases: converting a mesh made by another program, referring to boundaries by name, and two ways of cutting the cost of a rotor run. The first is a rotating frame, which lets a spinning rotor be solved as a flow that does not change with time. The second is a periodic boundary, which lets you model only one of the identical blades.

The rotor is the one Caradonna and Tung tested at NASA in 1981[1]. It has two straight blades, each the same NACA 0012 aerofoil shape from root to tip, six times as long as they are wide, set at a collective pitch of 8°: every blade meets the air at the same 8° angle. The tips move at 0.877 times the speed of sound (Mach 0.877), fast enough for a shock wave, a sudden jump in pressure, to form near the tip. The experiment measured the pressure on the blade surface at five distances from the shaft, and those measurements are the reference here.

What you need?

Work through Simple aerofoil (p. 373) first: this tutorial relies on it for running zCFD, reading the log and monitoring a run in zMon. The mesh has 447,468 cells, and the run takes about nine and a half minutes on an NVIDIA RTX 3060. The Python step in Step 5 also uses ParaView's Python modules, which come with zCFD.

Step 1. Download the files

The tutorial bundle can be downloaded here.

```
cd ~
wget https://zcfd.zenotech.com/v2026.11.16174/tutorial_7_rotor_in_hover.tgz
tar xzf tutorial_7_rotor_in_hover.tgz
cd tutorial_7_rotor_in_hover
```

FILE	CONTENTS
`caradonna_tung.msh.h5`	the mesh as Fluent wrote it
`caradonna_tung.h5`	the same mesh, converted for zCFD
`caradonna_tung_zone.py`	the zone helper written by the conversion
`caradonna_tung.py`	the control file

[1] Caradonna, F. X. and Tung, C., *Experimental and Analytical Studies of a Model Helicopter Rotor in Hover*, NASA TM-81232, 1981.

FILE	CONTENTS
`data/cp-0p50.txt` …	the measured surface pressure at r/R = 0.50, 0.68, 0.80, 0.89 and 0.96

Converting the mesh

zCFD reads its own mesh format. A mesh from another program is converted once, before the first run:

```
convert_mesh caradonna_tung.msh.h5 caradonna_tung.h5
```

The converter recognises the Fluent CFF format from the file name, and writes two files: the zCFD mesh, `caradonna_tung.h5`, and a zone helper, `caradonna_tung_zone.py`. The bundle already holds both, so this step can be skipped; running it replaces them with identical copies. Preparing a mesh (p. 201) lists the converters for the other formats.

A mesh's boundary faces are grouped into zones, each with a number. The zone helper maps the names the boundaries were given in the mesh generator to those numbers:

```
zone_name_to_id={
    "farfield": 48,
    "periodic2": 50,
    "core": 51,
    "lower_surface": 52,
    "upper_surface": 53,
    "tip": 54,
    "periodic1": 732,
    "interior--enclosure-enclosure": 1909,
}
```

Its `to_id` function turns a list of names into a list of numbers. The control file imports it, so every boundary condition below names its boundaries as the mesh generator did. See Zone identifiers (p. 208).

Step 2. Walk through the input

The control file, `caradonna_tung.py`, is a short Python script. All it does is build one Python dictionary, called `parameters`, that tells zCFD everything about the run: the physics, the numerical method, what to write out, and the mesh with its boundaries. This step walks through that dictionary from top to bottom. Each heading below is the name of a key, written exactly as it appears in the file, and the headings nest the same way the dictionary does: `numerical scheme`, for example, sits inside `solver`. Under each heading you'll find what that part controls, what each setting means, and why it has the value it has here. Each section ends with a link to that part's reference page, which lists every setting it accepts and its default; you don't need it to follow the tutorial. On a wide screen the whole control file sits on the right, with the part being described highlighted as you scroll.

Two parts are new since Simple aerofoil (p. 373): the rotating frame, under the model's `transforms`, and the pair of periodic boundaries among its boundary conditions. The file also starts with two Python import lines, which get their own heading.

```
import math

from caradonna_tung_zone import to_id

parameters = {
    "config version": 2,
    "solver": {
        "solver settings": {
            "type": "compressible",
        },
        "equations": {
            "type": "rans",
            "turbulence": {"model": "sst"},
            "precondition": False,
        },
        "fluid properties": {
            "material": "air",
            "gamma": 1.4,
            "gas constant": 287.0,
            "sutherlands const": 110.4,
            "prandtl no": 0.72,
            "turbulent prandtl no": 0.9,
        },
        "reference conditions": {
            "IC_1": {
                "temperature": 289.75,
                "pressure": 103037.0,
                "v": {"vector": [0.0, 0.0, -1.0], "mach": 0.877},
                "reynolds no": 3.93e6,
                "reference length": 1.0,
                "turbulence intensity": 0.01,
                "eddy viscosity ratio": 0.1,
            },
            "IC_2": {
                "temperature": 289.75,
                "pressure": 103037.0,
                "v": {"vector": [0.0, 0.0, -1.0], "mach": 0.00001},
                "reynolds no": 3.93e6 * 0.00001 / 0.87,
                "reference length": 1.0,
                "turbulence intensity": 0.01,
                "eddy viscosity ratio": 0.1,
            },
        },
        "initialisation": {"initial conditions": "IC_2", "reference": "IC_1"},
        "numerical scheme": {
            "order": "second",
            "limiter": "van albada",
        },
        "time settings": {"type": "steady"},
        "convergence control": {
            "scheme": {"name": "lu-sgs"},
            "cfl": {
                "cfl": 20.0,
```

```
                "cfl ramp": [{"type": "exponential", "factor": 1.05, "initial": 0.1}],
            },
            "cycles": 1500,
        },
        "output settings": {
            "report": {
                "frequency": 10,
                "forces": {
                    "FR_1": {
                        "name": "blade",
                        "zones": to_id(["lower_surface", "upper_surface", "tip"]),
                    },
                },
            },
            "solution": {
                "format": "vtk",
                "surface variables": ["V", "p", "T", "rho", "mach", "yplus"],
                "volume variables": ["V", "p", "T", "rho", "mach", "cell_velocity"],
            },
        },
    },
    "model": {
        "rotor": {
            "mesh": "caradonna_tung.h5",
            "transforms": {
                "motion": {
                    "type": "rotating",
                    "omega": 49.873,
                    "axis": [0.0, 0.0, 1.0],
                    "origin": [0.0, 0.0, 0.0],
                    "moving mesh": False,
                }
            },
            "boundary conditions": {
                "BC_1": {
                    "zones": to_id(["lower_surface", "upper_surface", "tip"]),
                    "type": "wall",
                    "kind": "wall function",
                },
                "BC_2": {
                    "zones": to_id(["farfield"]),
                    "type": "farfield",
                    "kind": "riemann",
                    "condition": "IC_2",
                },
                "BC_3": {"zones": to_id(["core"]), "type": "symmetry"},
                "BC_4": {
                    "zones": to_id(["periodic2"]),
                    "type": "interface",
                    "kind": "periodic",
                    "connection": {"boundary": "BC_5"},
                    "transform": {
                        "type": "rotated",
```

```
                        "theta": math.radians(180.0),
                        "axis": [0.0, 0.0, 1.0],
                        "origin": [0.0, 0.0, 0.0],
                    },
                },
                "BC_5": {
                    "zones": to_id(["periodic1"]),
                    "type": "interface",
                    "kind": "periodic",
                    "connection": {"boundary": "BC_4"},
                    "transform": {
                        "type": "rotated",
                        "theta": math.radians(-180.0),
                        "axis": [0.0, 0.0, 1.0],
                        "origin": [0.0, 0.0, 0.0],
                    },
                },
            },
        }
    },
}
```

`from caradonna_tung_zone import to_id`

The control file is Python, so it can import helpers before it defines `parameters`.

- `import math` supplies `math.radians`, which turns the periodic angles below from degrees to radians.
- `from caradonna_tung_zone import to_id` imports the zone helper written by the mesh conversion in Step 1. `to_id(["farfield"])` returns the zone numbers of the boundaries the mesh generator named `farfield`, so the boundary conditions name their boundaries rather than number them.

See Zone identifiers (p. 208).

`config version`

Says which layout of control file this is, so zCFD knows how to read the rest.

- `config version`: `2`, the current layout, with `solver` and `model` at the top level.

`solver`

The settings shared by the whole simulation: which solver runs, what it solves, the fluid, the flow states, and how the run marches and what it writes.

`solver settings`

Chooses the solver.

- `type`: `"compressible"`: the solver that lets the air's density change. The tip moves at Mach 0.877, fast enough to squeeze the air noticeably.

See `solver settings` (p. 11).

`equations`

Sets the equations being solved.

- `type`: `"rans"`, the Reynolds-averaged Navier–Stokes equations. These solve for the average flow and represent the effect of turbulence, the small chaotic eddies, with a turbulence model rather than resolving every eddy.
- `turbulence`: `{"model": "sst"}`, Menter's SST model, a widely used general-purpose turbulence model.
- `precondition`: `False`: low-speed preconditioning, a technique that helps the compressible solver with slow flows, is off. The flow the blades see is fast, up to Mach 0.877 at the tip.

See `equations` (p. 18) and Choosing a turbulence model (p. 226).

`fluid properties`

Describes the gas.

- `material`: `"air"`, as an ideal gas.
- `gamma` and `gas constant`: `1.4` and `287.0` J/kg/K, the standard values for air.
- `sutherlands const`: `110.4` K, for Sutherland's law, which sets how the viscosity changes with temperature.
- `prandtl no` and `turbulent prandtl no`: `0.72` and `0.9`, which relate how fast heat spreads to how fast momentum does, in smooth and turbulent flow.

See `fluid properties` (p. 37).

`reference conditions`

Defines named flow states, each a set of values such as temperature, pressure and velocity. Other parts of the file refer to them by name: to fill the domain at the start, to set conditions at boundaries, and to scale the equations.

`IC_1` The conditions at the blade tip, used as the reference.

- `temperature` and `pressure`: `289.75` K and `103037.0` Pa, the test conditions.
- `v`: `{"vector": [0.0, 0.0, -1.0], "mach": 0.877}`: the tip Mach number, 299 m/s.
- `reynolds no`: `3.93e6`, the Reynolds number at the tip: the ratio of the air's inertia to its viscosity, based on the blade's width (its chord). The solver sets the viscosity to give this value.
- `reference length`: `1.0` m, the blade chord.
- `turbulence intensity` and `eddy viscosity ratio`: `0.01` and `0.1`, the turbulence in the incoming air: very little here.

`IC_2` The still air the rotor turns in.

- `temperature` and `pressure`: as `IC_1`.
- `v`: `{"vector": [0.0, 0.0, -1.0], "mach": 0.00001}`: a speed so small that the air is, in effect, still.
- `reynolds no`: `3.93e6 * 0.00001 / 0.87`: `IC_1`'s Reynolds number scaled down with the speed, so that both states have very nearly the same viscosity.
- `reference length`, `turbulence intensity` and `eddy viscosity ratio`: as `IC_1`.

See `reference conditions` (p. 39).

`initialisation`

Chooses which states start the run and scale it.

- `initial conditions`: `"IC_2"`: the air starts still.
- `reference`: `"IC_1"`: internally the solver works with values divided by a reference velocity. Still air would give it almost none to divide by, so the tip conditions provide one that matches the flow the blades see.

See `initialisation` (p. 47) and Reference state (p. 295).

`numerical scheme`

Sets how the equations are discretised in space: how the smooth flow is represented by one value in each cell of the mesh.

- `order`: `"second"`: the solver estimates how each variable, the turbulence variables included, varies across each cell, which is more accurate than treating it as constant.
- `limiter`: `"van albada"`: near a sharp change such as the shock on the outer blade, that estimate can overshoot and produce unphysical wiggles; the limiter damps them.

See `numerical scheme` (p. 74) and Choosing a limiter (p. 258).

`time settings`

Says what kind of run this is.

- `type`: `"steady"`: the run looks for a flow that does not change with time. In hover that is true for an observer turning with the blades, which `transforms` sets up.

See `time settings` (p. 51).

`convergence control`

Sets how the solver reaches the steady solution. It starts from a guess and improves it once per cycle, or iteration, until the flow stops changing; the run has then converged.

`scheme`

- `name`: `"lu-sgs"`: each cycle updates the solution with a cheap sweep through the mesh (lower–upper symmetric Gauss–Seidel). A cycle costs less than with the `"implicit euler"` scheme of Simple aerofoil (p. 373), but it needs a smaller CFL number, so more cycles.

`cfl`

- `cfl`: `20.0`, the CFL number the run ramps up to. The CFL number sets how big a step each cycle takes towards the answer: too large and the run becomes unstable, too small and it needs many cycles.
- `cfl ramp`: `[{"type": "exponential", "factor": 1.05, "initial": 0.1}]`: start at 0.1 and multiply by 1.05 each cycle, so the early cycles, far from the answer, take small steps.

`cycles`

- `cycles: 1500`: the run stops after 1500 cycles. Step 4 shows the thrust, the upward force on the rotor, has settled by then.

See `convergence control` (p. 56) and CFL control (p. 245).

`output settings`

Sets what zCFD writes, and how often. See `output settings` (p. 84).

`report` The report is a table with one row per report cycle, written to `caradonna_tung_OUTPUT/rotor/rotor_report.csv` and plotted by zMon.

- `frequency: 10`: a row every 10 cycles, holding the residuals and the forces.
- `forces`: one block, `FR_1`, which sums the forces on a set of zones:
  - `name: "blade"`, which prefixes its report columns: `blade_Fx`, `blade_Fy` and `blade_Fz` for the force, `blade_Mx` to `blade_Mz` for the moment, and their pressure (`Fp`, `Mp`) and friction (`Ff`, `Mf`) parts.
  - `zones`: the lower surface, upper surface and tip of the blade, by name through `to_id`.

  On a rotating model the block also reports the thrust (the force along the shaft), the torque (the turning resistance about it) and the power: `blade_rotor_Thrust`, `blade_rotor_Tq` and `blade_rotor_Tqp`, named after the block and the model.

`solution` The flow solution, written for visualisation and post-processing.

- `format: "vtk"`: VTK HDF files, `.vtkhdf`, which ParaView opens.
- `surface variables`: velocity `V`, pressure `p`, temperature `T`, density `rho`, Mach number `mach` and `yplus`, which measures how close the first cell is to the wall. The pressure coefficient `cp` is left out. It compares the local pressure with the pressure of the oncoming flow, and in hover there is no oncoming flow, so Step 5 works out the experiment's version from `p` instead.
- `volume variables`: the same flow variables, and `cell_velocity`, the velocity of the turning frame at each cell.

No `frequency` is given, so the solution is written once, at the end of the run. Output variables (p. 336) lists the others.

`model`

One entry per mesh, each with its own placement, motion and boundary conditions. See `model` (p. 96).

`rotor`

The name of the only model. It names the output folder, `caradonna_tung_OUTPUT/rotor/`, and the files in it, and appears in the thrust column.

`mesh`

- `mesh: "caradonna_tung.h5"`, the mesh converted in Step 1. It holds one blade in a half-cylinder.

`transforms` Places and moves the mesh. Here it holds only a `motion`, which sets up a rotating frame.

In hover the air far from the rotor is still, and each blade always sees the same flow. To an observer standing still the flow keeps changing as the blades sweep past, but to an observer turning with the blades it does not change at all. Solving the flow as that turning observer sees it, in a rotating frame, turns the rotor into a steady problem, which is far cheaper than following the blades round in time.

- `motion`:
  - `type`: `"rotating"`.
  - `omega`: `49.873` rad/s, the rotor speed: 476 revolutions a minute, which at a radius of 6 m gives the tip Mach number of `IC_1`.
  - `axis`: `[0.0, 0.0, 1.0]`, the rotor shaft, along z.
  - `origin`: `[0.0, 0.0, 0.0]`, a point on the shaft.
  - `moving mesh`: `False`: the mesh stays where it is. Instead, the solver adds to every cell the extra forces a turning observer feels, the centrifugal and Coriolis forces.

The frame is valid while the flow is steady in it. A rotor in forward flight, or one interacting with a body that does not turn with it, sees a flow that changes as it goes round, and needs a moving mesh in a time-accurate run. See `transforms` (p. 169) and Placing and moving a mesh (p. 365).

`boundary conditions` Assigns a condition to each zone of boundary faces. See `boundary conditions` (p. 98) and Choosing boundary conditions (p. 285).

`BC_1` The blade.

- `zones`: the lower surface, upper surface and tip.
- `type` and `kind`: `"wall"`, `"wall function"`: the air sticks to the blade, and the thin layer next to the surface, where the speed drops to zero, is represented by a formula, a wall function, rather than resolved with very fine cells. See Choosing a wall model (p. 231).

`BC_2` The outer boundary.

- `zones`: `farfield`.
- `type` and `kind`: `"farfield"`, `"riemann"`: the outer edge of the domain, far from the rotor. Air can pass in and out, and pressure waves leave without bouncing back in.
- `condition`: `"IC_2"`, the still air.

`BC_3` The hub.

- `zones`: `core`.
- `type`: `"symmetry"`: no flow passes through it.

`BC_4` One flat face of the half-cylinder.

The two blades are identical and half a turn apart, so the flow round one blade is a copy of the flow round the other, turned through 180°. The mesh therefore holds only half the rotor, one blade in a half-cylinder, and two periodic boundaries join its two flat faces: flow leaving through one comes back in through the other, turned through 180°. Modelling one blade halves the cost of the run.

- `zones`: `periodic2`.
- `type` and `kind`: `"interface"`, `"periodic"`.
- `connection`: `{"boundary": "BC_5"}`, its partner.
- `transform`: the rotation that maps this face onto its partner: `type` `"rotated"`, `theta` 180° in radians, about `axis` z through `origin`.

See Periodic (p. 291).

`BC_5` The other flat face.

- `zones`: `periodic1`.
- `type` and `kind`: `"interface"`, `"periodic"`.
- `connection`: `{"boundary": "BC_4"}`.
- `transform`: the same rotation the other way, `theta` −180°: each side carries the rotation that maps it onto its partner, so the two angles are opposite.

Step 3. Run

1. Activate your environment:

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

2. Check the control file for errors:

```
cd ~/tutorial_7_rotor_in_hover/
validate_input caradonna_tung.py
```

3. Run zCFD in the case directory:

```
run_zcfd -c caradonna_tung.py
```

These are extracts from the log of a run of this tutorial, with `...` marking lines left out:

```
Parameters validated successfully
...
 Reading file caradonna_tung.h5
 Global Mesh Stats: Cells: 447468 Faces: 2529579
...
 Setup periodic boundaries
...
 Dimensional Conditions
 ----------------------
  Pressure:     103037 Pa
  Temperature:  289.75 K
  Density:      1.23905 kg/m^3
  Viscosity:    9.43434e-05 Ns/m^2 @ 289.75 K
  Speed:        299.238 m/s
```

```
...
  Mach:         0.877
  Reynolds No:  3.93e+06 @ 289.75 K
...
LU-SGS: rotating-frame Coriolis diagonal enabled (omega = 0.172947, cells = 447468)
...
Starting solve loop
...
Cycle     10 | Step    0 t=0s | CFL   0.155 | MG 1 |   375.3 ms | Total     3.779s
+-----------------------------------------------------------------------------------------+
| Report: Cycle 10  Wall time: 3.779s                                                     |
...
| Residuals:                                                                              |
|   rho       1.1218E+02  rhoV[0]   2.6131E+01  rhoV[1]   9.9169E-01  rhoV[2]   2.1170E+01 |
|   rhoE      4.1126E+02  rhok      5.6513E-01  rhoOmega  7.7031E+06                      |
+-----------------------------------------------------------------------------------------+
...
 Solve Complete: 1500 cycles in 9m 24.5s (avg 376.4 ms/cycle)
...
zCFD finished: all requested cycles are complete (1500 in total).
```

- The mesh is read and its periodic faces paired.
- The dimensional conditions are those of the reference state, the blade tip: 299 m/s, Mach 0.877.
- The LU-SGS line confirms the rotating frame is active; its `omega` is the control file's 49.873 rad/s in the solver's non-dimensional units.
- The CFL number is still ramping up at cycle 10, from 0.1 towards 20.
- The run takes about nine and a half minutes on an NVIDIA RTX 3060.

Output files

FILE	CONTENTS
`caradonna_tung.log`	the solver log
`caradonna_tung_status.yaml`	the run status
`caradonna_tung_OUTPUT/rotor/rotor_report.csv`	residuals and forces every 10 cycles
`caradonna_tung_OUTPUT/rotor/rotor.vtkhdf`	the volume solution
`caradonna_tung_OUTPUT/rotor/rotor_wall.vtkhdf`	the blade surface solution

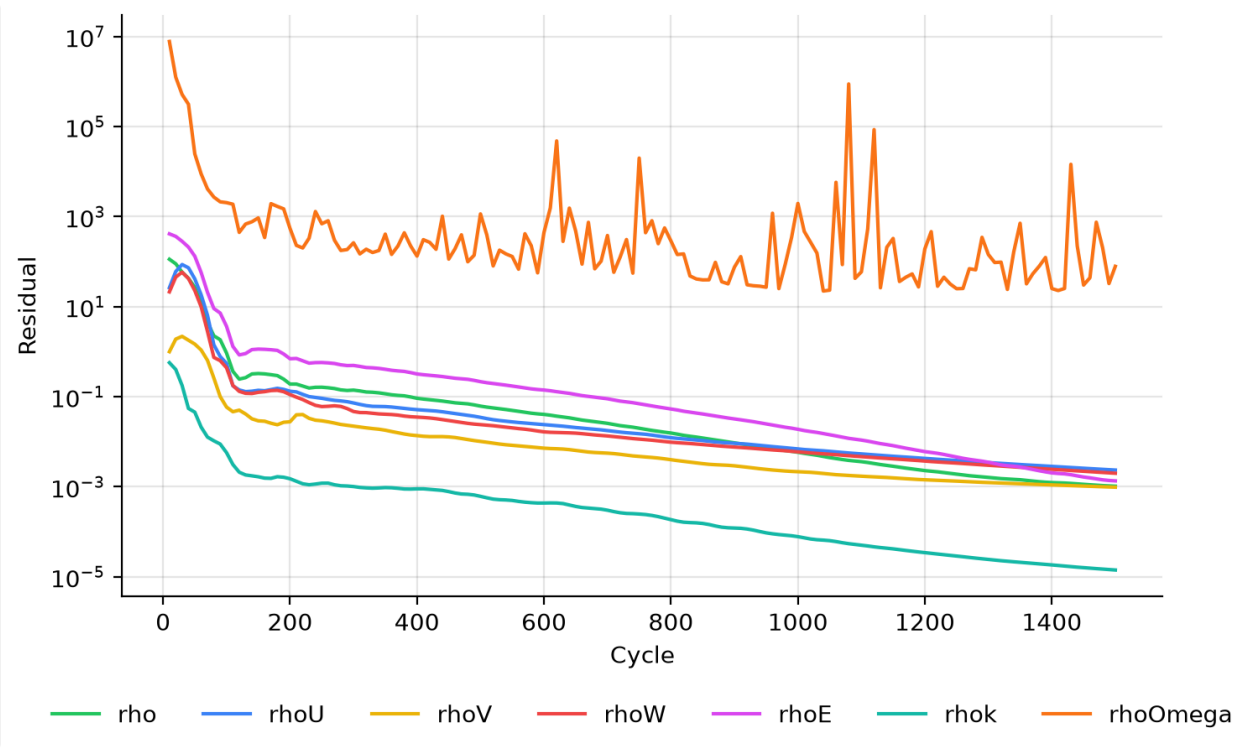
Step 4. Monitor

Open the run in zMon to watch the residuals and forces, while it runs or after it has finished:

```
zmon caradonna_tung.py
```

Monitoring a run (p. 324) describes each panel. The plots below are the residual and thrust panels as zMon draws them.

Residuals

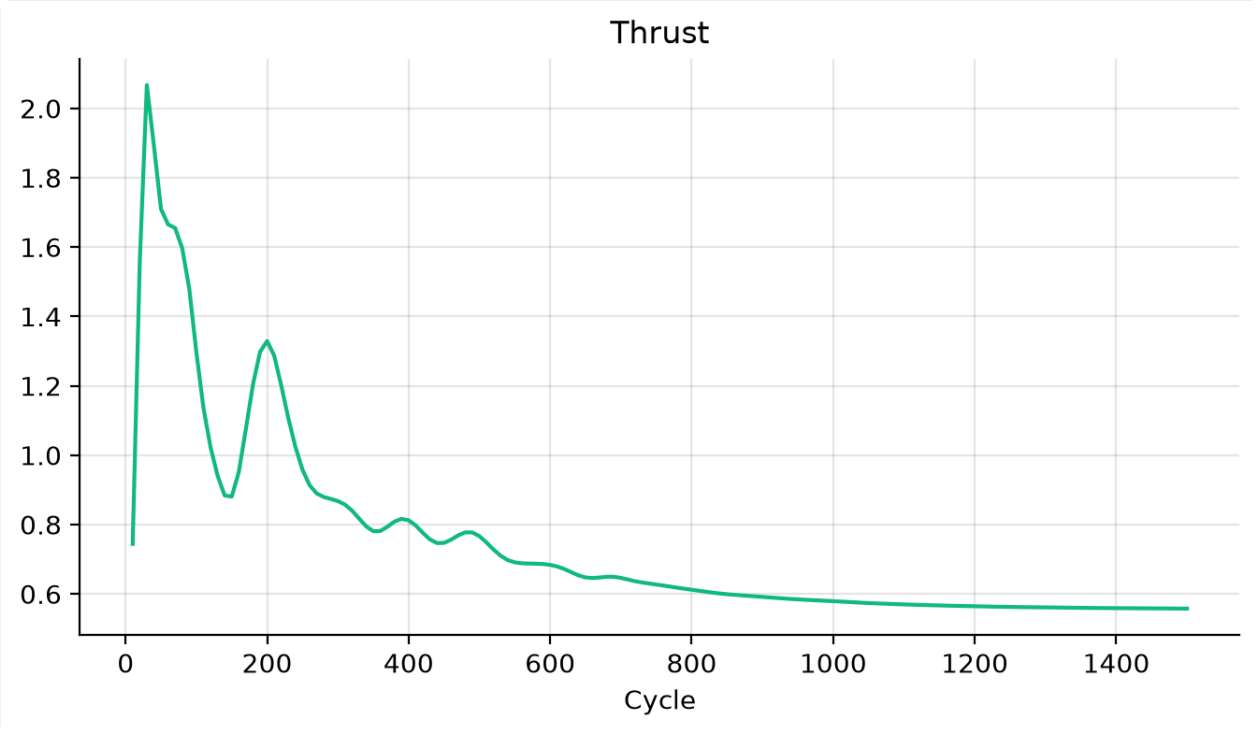


Residuals of the rotor run against cycle

A residual measures how far the current solution is from satisfying one of the equations; as the run converges, the residuals fall. The mean-flow residuals fall by between three and five and a half orders of magnitude (factors of a thousand to a few hundred thousand) over the 1500 cycles, and the turbulence residual `rhok` by about four and a half. The `rhoOmega` residual jumps about throughout and is no guide to convergence here; the thrust is.

Thrust

The force block is named `blade` and the model `rotor`, so the thrust is in the column `blade_rotor_Thrust`.



Thrust on the blade against cycle

The thrust overshoots as the flow starts up, then settles. It is within 1% of its final value by cycle 1300, and changes by less than 1% over the last 200 cycles.

Step 5. Post-process

Python

First activate your zCFD environment, which provides Python with zutil, h5py and ParaView:

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

The pressure coefficient, Cp, compares the pressure at a point on the blade with a reference pressure, divided by the dynamic pressure, ½ × density × speed², which measures how hard the air is pushing. The experiment refers each station at radius r to the speed of the blade section there, Ωr, where Ω is the rotor speed. This script cuts the blade surface at r/R = 0.80, works out Cp round the section that way, and plots it against the measured values. It uses ParaView's Python modules to make the cut. Save it as `plot_blade_cp.py` beside the control file:

```
import matplotlib.pyplot as plt
import numpy as np
import paraview.simple as pvs

from zutil.post import clean_vtk
from zutil.post.post import slice_metric

# Values from the control file
tip_radius = 6.0  # m
```

```
omega = 49.873  # rad/s
p_ref = 103037.0  # Pa
rho_ref = p_ref / (287.0 * 289.75)  # kg/m^3, the still air

# The experiment refers Cp to the blade section's own dynamic pressure
radius = 0.80 * tip_radius
q_local = 0.5 * rho_ref * (omega * radius) ** 2

# Cut the blade surface at r/R = 0.80 and read the pressure round the section
wall = clean_vtk(pvs.VTKHDFReader(FileName="caradonna_tung_OUTPUT/rotor/rotor_wall.vtkhdf"))
profile = slice_metric(wall, [0.0, 1.0, 0.0], [0.0, radius, 0.0], "p")

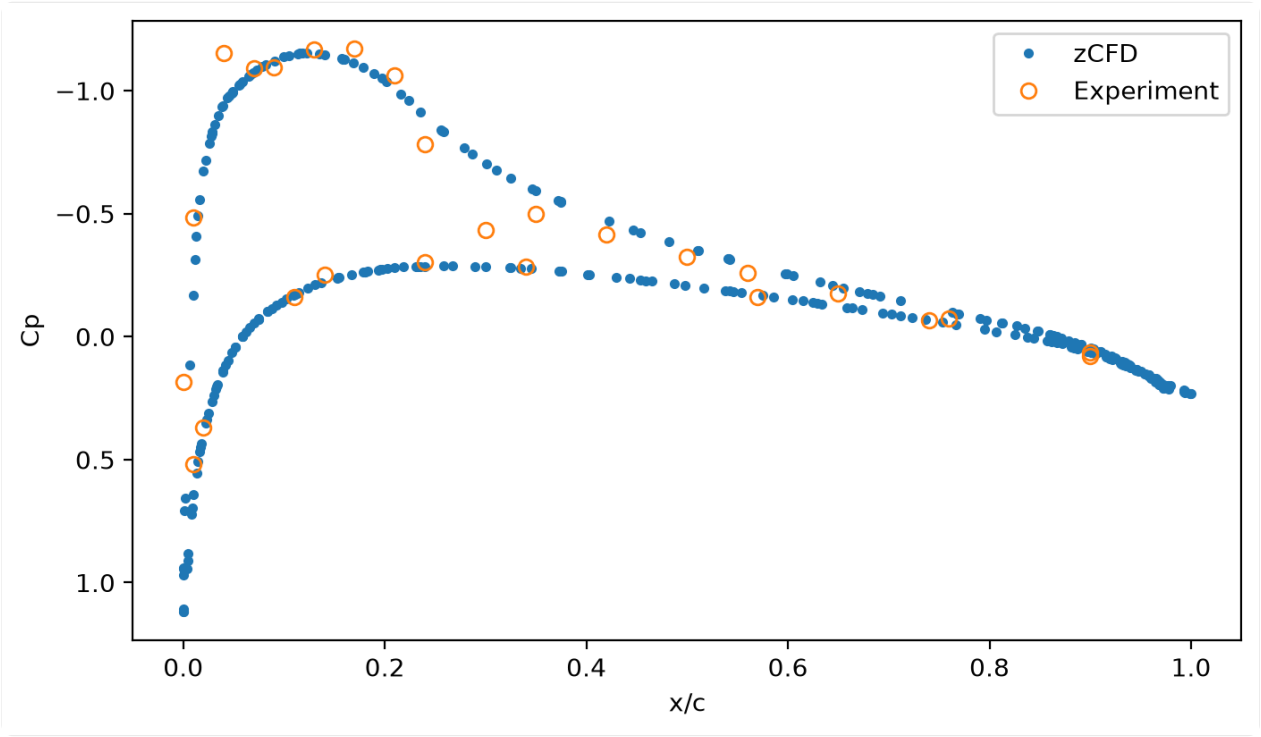
# The measured file stores -Cp against x/c
measured = np.loadtxt("data/cp-0p80.txt", skiprows=1)

fig, ax = plt.subplots(figsize=(8, 4.5))
for block in profile.values():
    cp = (np.asarray(block["p"]) - p_ref) / q_local
    ax.plot(block["chord"], cp, ".", label="zCFD")
ax.plot(measured[:, 0], -measured[:, 1], "o", mfc="none", label="Experiment")
ax.invert_yaxis()
ax.set_xlabel("x/c")
ax.set_ylabel("Cp")
ax.legend()
fig.savefig("blade_cp.png", dpi=200, bbox_inches="tight")
```

and run it:

```
python plot_blade_cp.py
```

It writes `blade_cp.png`:



Pressure coefficient at r/R = 0.80 from zCFD against the experiment

Cp is plotted with negative values upwards, as is usual for aerofoils, so the low-pressure upper surface is the top curve. zCFD follows the experiment closely on the lower surface and in the suction peak near the leading edge.

Adding up the pressure all round each section gives the normal-force coefficient, C_n: the force pushing the section up, relative to its dynamic pressure. The table compares C_n from zCFD with C_n worked out the same way from the measured pressures:

R/R	ZCFD	EXPERIMENT
0.50	0.21	0.20
0.68	0.27	0.28
0.80	0.33	0.30
0.89	0.31	0.32
0.96	0.23	0.28

The loading is predicted to within 0.03 at the inner stations. The station nearest the tip, r/R = 0.96, is furthest from the experiment. Each blade trails a swirling tip vortex, and the vortex of the blade ahead passes close to this station. This mesh captures that vortex for only about a quarter turn behind the blade before its cells grow too coarse, so the tutorial makes no claim for the wake.

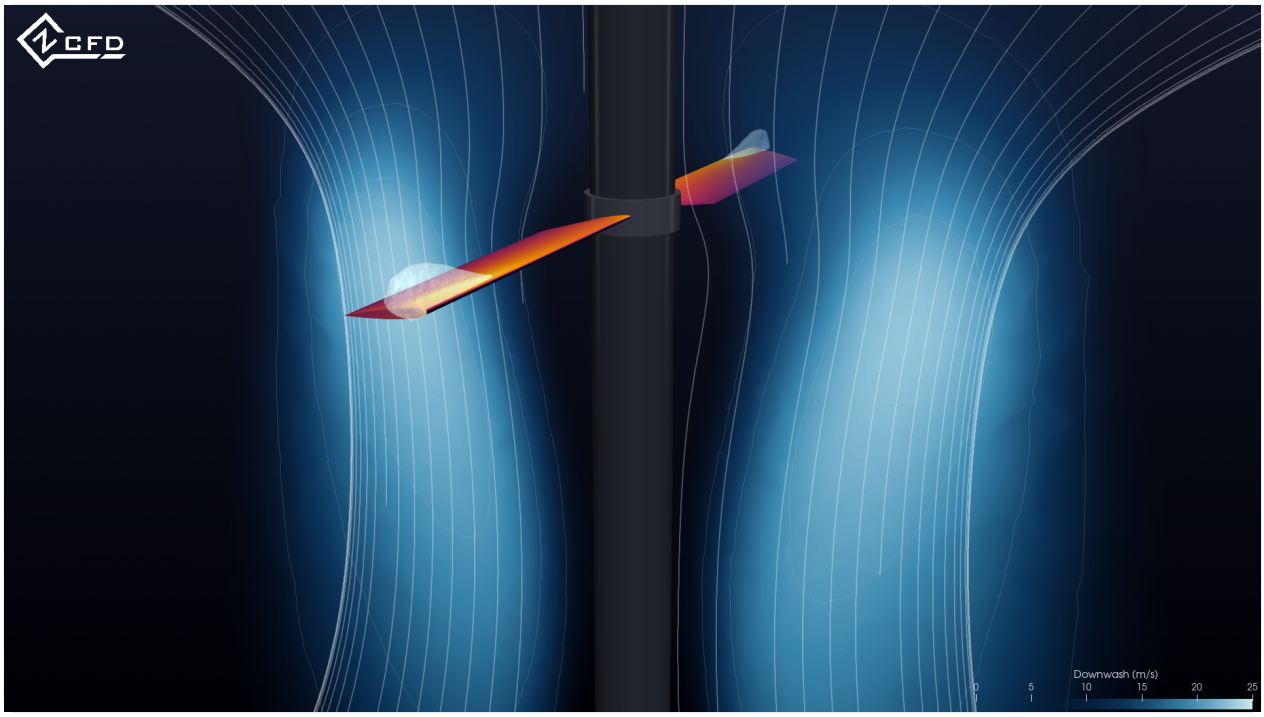
ParaView

> **NOTE**
>
> The .vtkhdf output format requires ParaView 6.1.1 or later.

Open `caradonna_tung_OUTPUT/rotor/rotor_wall.vtkhdf` in ParaView to see the blade. To colour it by the pressure coefficient of the experiment, apply a **Calculator** with the expression

```
(p-103037)/(0.5*1.2390*49.873^2*(coordsX^2+coordsY^2))
```

where 1.2390 kg/m³ is the density of the still air. To see both blades, apply a **Transform** that rotates the surface by 180° about z, and show it alongside the original. The volume solution, `caradonna_tung_OUTPUT/rotor/rotor.vtkhdf`, holds the velocity, from which a slice through the rotor axis shows the air driven down through the disc.



The Caradonna–Tung rotor in hover, coloured by local-section Cp, with the air driven down below it

Citations

Polars and trimming to a target

What is in this tutorial?

Two questions come up on almost every aerofoil or wing. How do the lift (the force at right angles to the oncoming air) and the drag (the force along it) change with the angle of attack, the angle between the aerofoil's chord line and the oncoming air? And what angle gives the lift you need? This tutorial answers both on the NACA 0012 aerofoil from Simple aerofoil (p. 373).

First you run the aerofoil at three angles and plot its polar: the lift against the angle, and the lift against the drag. Then you let zCFD trim the aerofoil: it turns the oncoming flow, a step at a time, until the lift reaches a value you ask for. Lift and drag are given as coefficients, CL and CD, the force divided by the dynamic pressure of the oncoming air and a reference area, so that they do not depend on the size or speed of the test.

What you need?

Work through Simple aerofoil (p. 373) first: this tutorial uses the same mesh, and relies on it for running zCFD, reading the log and monitoring a run in zMon. On an NVIDIA RTX 3060 each polar run takes about four minutes, and the trim about five.

Step 1. Download the files

The tutorial bundle can be downloaded here. It contains the mesh (`naca0012.h5`) and four control files: `naca0012.py` at zero angle of attack, `naca0012_alpha5.py` and `naca0012_alpha10.py` for the polar, and `naca0012_target_cl.py` for the trim.

```
cd ~
wget https://zcfd.zenotech.com/v2026.11.16174/tutorial_8_aerofoil_polar_and_trim.tgz
tar xzf tutorial_8_aerofoil_polar_and_trim.tgz
cd tutorial_8_aerofoil_polar_and_trim
```

Step 2. Walk through the input

The control file, `naca0012.py`, is a short Python script. All it does is build one Python dictionary, called `parameters`, that tells zCFD everything about the run: the physics, the numerical method, what to write out, and the mesh with its boundaries. This step walks through that dictionary from top to bottom. Each heading below is the name of a key, written exactly as it appears in the file, and the headings nest the same way the dictionary does: `numerical scheme`, for example, sits inside `solver`. Under each heading you'll find what that part controls, what each setting means, and why it has the value it has here. Each section ends with a link to that part's reference page, which lists every setting it accepts and its default; you don't need it to follow the tutorial. On a wide screen the whole control file sits on the right, with the

part being described highlighted as you scroll.

`naca0012.py` is the base control file: the other three load it and change only a few keys, so they are described after it, by the keys they change. The base file is the one from Simple aerofoil (p. 373) with two parts changed for an aerofoil that makes lift: `numerical scheme` and `convergence control`.

```
parameters = {
    "config version": 2,
    "solver": {
        "solver settings": {"type": "compressible"},
        "equations": {
            "type": "rans",
            "turbulence": {"model": "sst"},
            "precondition": True,
        },
        "fluid properties": {
            "material": "air",
            "gamma": 1.4,
            "gas constant": 287.0,
            "sutherlands const": 110.4,
            "prandtl no": 0.72,
            "turbulent prandtl no": 0.9,
        },
        "reference conditions": {
            "IC_1": {
                "temperature": 300,
                "pressure": 101325.0,
                "v": {"vector": [1.0, 0.0, 0.0], "mach": 0.15},
                "viscosity": 1.02145e-05,
                "turbulence intensity": 5.2e-2,
                "eddy viscosity ratio": 1.0,
            },
        },
        "initialisation": {"initial conditions": "IC_1", "restart": False},
        "numerical scheme": {
            "order": "second",
            "limiter": "van albada",
            "inviscid flux scheme": "roe",
        },
        "time settings": {"type": "steady"},
        "convergence control": {
            "scheme": {"name": "implicit euler"},
            "cfl": {
                "cfl": 100,
                "cfl ramp": [{"type": "exponential", "factor": 1.1, "initial": 0.05}],
            },
            "cycles": 2000,
        },
        "output settings": {
            "report": {
                "frequency": 10,
                "forces": {
                    "FR_1": {"name": "wall", "zones": [4]},
```

```
                },
            },
            "solution": {
                "surface variables": [
                    "V",
                    "p",
                    "mach",
                    "cp",
                    "cf",
                    "pressureforce",
                    "frictionforce",
                ],
                "volume variables": [
                    "V",
                    "p",
                    "T",
                    "rho",
                    "mach",
                    "cp",
                    "eddy",
                ],
            },
        },
    },
    "model": {
        "naca0012": {
            "mesh": "naca0012.h5",
            "boundary conditions": {
                "BC_1": {
                    "zones": [1],
                    "type": "farfield",
                    "condition": "IC_1",
                    "kind": "riemann",
                },
                "BC_2": {
                    "zones": [4],
                    "type": "wall",
                    "kind": "no slip",
                },
                "BC_3": {
                    "zones": [2, 3],
                    "type": "symmetry",
                },
            },
        },
    },
}
```

`config version`

Says which layout of control file this is, so zCFD knows how to read the rest.

- `config version`: `2`, the current layout, with `solver` and `model` at the top level.

`solver`

The settings shared by the whole simulation: which solver runs, what it solves, the fluid, the flow state, and how the run marches and what it writes.

`solver settings`

Chooses the solver.

- `type`: `"compressible"`: the solver that lets the air's density change, as in Simple aerofoil.

See `solver settings` (p. 11).

`equations`

Sets the equations being solved.

- `type`: `"rans"`, the Reynolds-averaged Navier–Stokes equations. These solve for the average flow and represent the effect of turbulence, the small chaotic eddies, with a turbulence model.
- `turbulence`: `{"model": "sst"}`, Menter's SST model, a widely used general-purpose turbulence model.
- `precondition`: `True`: low-speed preconditioning, which helps the compressible solver converge for slow flows. It is recommended below about Mach 0.3, 0.3 times the speed of sound; this flow is at Mach 0.15.

See `equations` (p. 18) and Choosing a turbulence model (p. 226).

`fluid properties`

Describes the gas.

- `material`: `"air"`, as an ideal gas.
- `gamma` and `gas constant`: `1.4` and `287.0` J/kg/K, the standard values for air.
- `sutherlands const`: `110.4` K, for Sutherland's law, which sets how the viscosity changes with temperature.
- `prandtl no` and `turbulent prandtl no`: `0.72` and `0.9`, which relate how fast heat spreads to how fast momentum does, in smooth and turbulent flow.

See `fluid properties` (p. 37).

`reference conditions`

Defines named flow states, each a set of values such as temperature, pressure and velocity. Other parts of the file refer to them by name: to fill the domain at the start, to set conditions at boundaries, and to scale the equations.

`IC_1` The freestream: the undisturbed air far from the aerofoil.

- `temperature` and `pressure`: `300` K and `101325.0` Pa.
- `v`: `{"vector": [1.0, 0.0, 0.0], "mach": 0.15}`: Mach 0.15 along x, along the chord, so the aerofoil is at zero angle of attack. The polar control files turn this vector.
- `viscosity`: `1.02145e-05` kg/m/s, which gives a Reynolds number, the ratio of the air's inertia to its viscosity, of 6 million on the 1 m chord.

- `turbulence intensity` and `eddy viscosity ratio`: `5.2e-2` and `1.0`, the turbulence in the incoming flow.

See `reference conditions` (p. 39).

`initialisation`

Chooses which state starts the run.

- `initial conditions`: `"IC_1"`: the whole domain starts at the freestream.
- `restart`: `False`: start afresh rather than from a previous solution.

See `initialisation` (p. 47).

`numerical scheme`

Sets how the equations are discretised in space: how the smooth flow is represented by one value in each cell of the mesh. This is the first of the two parts that differ from Simple aerofoil (p. 373).

- `order`: `"second"`: the solver estimates how every variable, the turbulence variables included, varies across each cell, which is more accurate than treating it as constant. Simple aerofoil's `"euler_second"` does this for the flow but treats the turbulence variables as constant in each cell.
- `limiter`: `"van albada"`: where the flow changes sharply, as it does accelerating round the leading edge, that estimate can overshoot; the limiter damps it. It is also the default, so Simple aerofoil uses it too; here it is written out.
- `inviscid flux scheme`: `"roe"`: Roe's method for working out how much mass, momentum and energy flows between neighbouring cells.

See `numerical scheme` (p. 74) and Choosing a limiter (p. 258).

`time settings`

Says what kind of run this is.

- `type`: `"steady"`: at these angles the flow stays attached to the aerofoil and does not change with time, so the run looks for that final state directly.

See `time settings` (p. 51).

`convergence control`

Sets how the solver reaches the steady solution. It starts from a guess and improves it once per cycle, or iteration, until the flow stops changing; the run has then converged. This is the second part that differs from Simple aerofoil.

`scheme`

- `name`: `"implicit euler"`: each step solves a linear system, which allows large steps.

`cfl`

- `cfl`: `100`, the CFL number the run ramps up to, twice Simple aerofoil's. The CFL number sets how big a step each cycle takes towards the answer: too large and the run becomes unstable, too small and it needs many cycles.

- `cfl ramp`: `[{"type": "exponential", "factor": 1.1, "initial": 0.05}]`: start at 0.05, lower than Simple aerofoil's 1, and multiply by 1.1 each cycle, so the first cycles, far from the answer, take very small steps.

`cycles`

- `cycles`: `2000`, enough for each angle in this tutorial to settle: the 10° lift changes by less than 0.0004 over the last 500 cycles.

See `convergence control` (p. 56) and CFL control (p. 245).

`output settings`

Sets what zCFD writes, and how often. See `output settings` (p. 84).

`report` The report is a table with one row per report cycle, written to `<control file>_OUTPUT/naca0012/naca0012_report.csv` and plotted by zMon.

- `frequency`: `10`: a row every 10 cycles.
- `forces`: one block, `FR_1`, which sums the forces on a set of zones:
  - `name`: `"wall"`, which prefixes its report columns: `wall_Fx`, `wall_Fy` and `wall_Fz` for the force, in the axes of the mesh, with their pressure and friction parts.
  - `zones`: `[4]`, the aerofoil surface.

  Forces are reported as coefficients, against `IC_1` and a reference area of 1.

`solution` The flow solution, written once at the end of the run for visualisation.

- `surface variables`: velocity, pressure, Mach number, `cp`, skin friction `cf`, and the pressure and friction forces on each face.
- `volume variables`: velocity, pressure, temperature, density, Mach number, `cp` and the eddy viscosity `eddy`.

`model`

One entry per mesh, each with its own boundary conditions. See `model` (p. 96).

`naca0012`

The name of the only model. It names the folder under each run's output, `<control file>_OUTPUT/naca0012/`, and the files in it.

`mesh`

- `mesh`: `"naca0012.h5"`: the aerofoil, with its span along y and one cell across it.

`boundary conditions` Assigns a condition to each zone of boundary faces. See `boundary conditions` (p. 98) and Choosing boundary conditions (p. 285).

BC_1 The outer boundary.

- `zones`: `[1]`.
- `type` and `kind`: `"farfield"`, `"riemann"`: the outer edge of the domain, far from the aerofoil. Air passes in and out, and pressure waves leave without bouncing back in.
- `condition`: `"IC_1"`, the freestream.

BC_2 The aerofoil.

- `zones`: `[4]`.
- `type` and `kind`: `"wall"`, `"no slip"`: the flow velocity is zero at the surface.

BC_3 The two faces normal to y.

- `zones`: `[2, 3]`.
- `type`: `"symmetry"`: mirror planes. The aerofoil behaves as if it were infinitely long, so the flow is two-dimensional and the mesh needs only one cell across it.

`naca0012_alpha5.py` and `naca0012_alpha10.py`

The polar control files load `naca0012.py` with `get_parameters_from_file` and change one key.

`naca0012_alpha5.py`:

```
import zutil
from zutil.fileutils import get_parameters_from_file

parameters = get_parameters_from_file("naca0012.py")

parameters["solver"]["reference conditions"]["IC_1"]["v"]["vector"] = (
    zutil.vector_from_angle(5.0, 0.0)
)
```

`naca0012_alpha10.py`:

```
import zutil
from zutil.fileutils import get_parameters_from_file

parameters = get_parameters_from_file("naca0012.py")

parameters["solver"]["reference conditions"]["IC_1"]["v"]["vector"] = (
    zutil.vector_from_angle(10.0, 0.0)
)
```

`solver › reference conditions › IC_1 › v › vector`

- `zutil.vector_from_angle(alpha, beta)` returns the direction of the oncoming flow, as a vector of length 1, for an angle of attack `alpha` and a sideslip angle `beta` (sideways), both in degrees. It turns the flow from +x towards +z, so the aerofoil, whose chord lies along x, meets the flow at 5° or 10°. The Mach number is unchanged, and the mesh stays where it is.

`naca0012_target_cl.py`

The trim control file loads `naca0012.py` and gives its farfield a target.

```
from zutil.fileutils import get_parameters_from_file

parameters = get_parameters_from_file("naca0012.py")

parameters["model"]["naca0012"]["boundary conditions"]["BC_1"]["target"] = {
    "alpha": {"cl": 1.0778, "hold": 500}
}
parameters["solver"]["convergence control"]["cycles"] = 5000
```

`model › naca0012 › boundary conditions › BC_1 › target`

A farfield with a `target` trims the aerofoil: it turns the direction of the oncoming flow, and so the angle of attack, until the lift coefficient reaches the value you ask for.

- `alpha`: the setting the farfield turns, the angle of attack, and what it aims for:
  - `cl`: `1.0778`, the lift coefficient wanted. It is CFL3D's value at 10°, so the trimmed angle should come out close to 10°.
  - `hold`: `500`: once the settled lift is within 0.001 of the target, the farfield freezes its angle and the run goes on for 500 more cycles, then stops.

The run starts at the farfield's own angle, 0°, waits for the lift to settle, steps the angle towards the target, and repeats. See Trimming to a target (p. 360), which describes the other options, such as trimming for drag instead of lift.

`solver › convergence control › cycles`

- `cycles`: `5000`: the budget for the whole run, the trim steps and the hold together.

Step 3. Run

1. Activate your environment:

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

2. Check the control files for errors:

```
cd ~/tutorial_8_aerofoil_polar_and_trim/
validate_input naca0012.py
```

3. Run the three polar control files and the trim, one after the other:

```
run_zcfd -c naca0012.py
run_zcfd -c naca0012_alpha5.py
run_zcfd -c naca0012_alpha10.py
run_zcfd -c naca0012_target_cl.py
```

Each log reads like the one in Simple aerofoil (p. 380). The trim run also writes a line each time it steps the angle, and a last line with the angle it found:

```
grep "target BC_1" naca0012_target_cl.log
```

In a run of this tutorial it printed:

```
target BC_1: trimming alpha (x–z plane, forward +x, up +z) for cl 1.0778, then hold 500 cycles
target BC_1: cycle  300  alpha  0.000° → 4.000°   cl -0.0000   target 1.0778
target BC_1: cycle  700  alpha  4.000° → 8.000°   cl 0.4332 (heading for 0.4360)   target
↪  1.0778
target BC_1: cycle 1200  alpha  8.000° → 9.671°   cl 0.8208 (heading for 0.8916)   target
↪  1.0778
target BC_1: cycle 1600  alpha  9.671° → 9.927°   cl 1.0304 (heading for 1.0498)   target
↪  1.0778
target BC_1: cycle 2000  alpha  9.927° → 9.974°   cl 1.0710 (heading for 1.0733)   target
↪  1.0778
target BC_1: cycle 2300  cl 1.0776 met at alpha 9.974° — holding 500 cycles
target BC_1: alpha trimmed to 9.974° for cl 1.0778 (cl 1.0781 at end, cycle 2800); as the
↪  condition's vector: zutil.vector_from_angle(9.974, 0.000)
```

Each step waits for the lift to settle, then moves the angle towards the target, with large steps at first and smaller ones as it closes in. Once the settled lift is within 0.001 of the target, the angle is frozen for the 500-cycle hold, and the run stops well inside its 5000-cycle budget. The last line gives the inflow vector to use for a plain run at the trimmed angle.

Output files

Each run writes its own output, named after its control file:

FILE/DIRECTORY	DESCRIPTION
`<control file>.log`	A copy of the output to screen during the run.
`<control file>_status.yaml`	A record of how the run was started; the trim run adds its result.
`<control file>_OUTPUT/naca0012/naca0012_report.csv`	The residuals and forces every 10 cycles.
`<control file>_OUTPUT/naca0012/naca0012.vtkhdf`	The volume output, for visualisation.
`<control file>_OUTPUT/naca0012/naca0012_wall.vtkhdf`	The surface output on the aerofoil.

For example, the 10° run writes `naca0012_alpha10_OUTPUT/naca0012/naca0012_report.csv`.

Step 4. Monitor

Open each run in zMon, while it runs or after it has finished, to check it has converged:

```
zmon naca0012_alpha10.py
```

Monitoring a run (p. 324) describes each panel. Check each run before you trust its forces: the residuals, which measure how far the solution is from satisfying the equations, should have stopped falling, and the lift should be flat over the last few hundred cycles.

The trim run's report gains two columns, `BC_1_alpha` and `BC_1_cl`, which zMon plots alongside the forces: the angle steps up, the lift follows, and both go flat for the hold.

Step 5. Post-process

Python

First activate your zCFD environment, which provides Python with zutil, h5py and ParaView:

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

The force report is in the axes of the mesh: x along the chord and z normal to it. Once the flow is turned, lift and drag are the components normal to and along the freestream, so the x and z forces must be rotated through the angle of attack α:

$$C_L = C_z \cos\alpha - C_x \sin\alpha, \qquad C_D = C_x \cos\alpha + C_z \sin\alpha$$

This script reads the last report row of each polar run, rotates the forces, prints the polar and plots it. Save it as `plot_polar.py` beside the control files:

```
import math

import matplotlib.pyplot as plt

from zutil.fileutils import get_zcfd_result

runs = {"naca0012.py": 0.0, "naca0012_alpha5.py": 5.0, "naca0012_alpha10.py": 10.0}

lift, drag = [], []
for control_file, alpha in runs.items():
    result = get_zcfd_result(control_file)
    final = result.report.data.iloc[-1]
    prefix = result.force_prefix()
    cx, cz = final[f"{prefix}_Fx"], final[f"{prefix}_Fz"]
    a = math.radians(alpha)
    lift.append(cz * math.cos(a) - cx * math.sin(a))
    drag.append(cx * math.cos(a) + cz * math.sin(a))
    print(f"alpha {alpha:4.1f}  cl {lift[-1]:.4f}  cd {drag[-1]:.5f}")

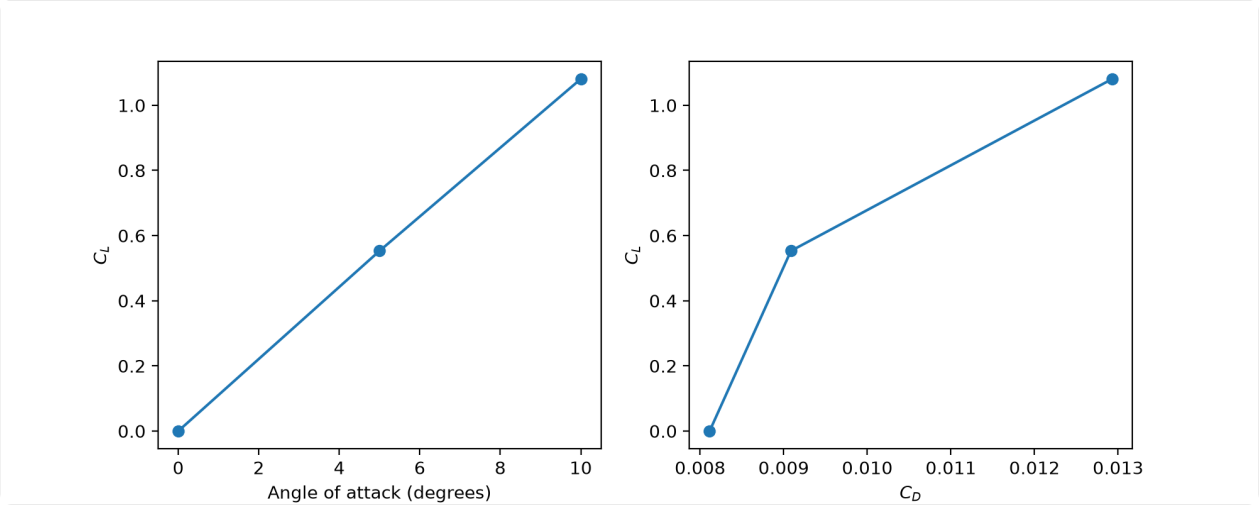
fig, (ax_lift, ax_polar) = plt.subplots(1, 2, figsize=(10, 4))
ax_lift.plot(list(runs.values()), lift, "o-")
ax_lift.set_xlabel("Angle of attack (degrees)")
ax_lift.set_ylabel("$C_L$")
ax_polar.plot(drag, lift, "o-")
ax_polar.set_xlabel("$C_D$")
ax_polar.set_ylabel("$C_L$")
```

```
fig.savefig("polar.png", dpi=200)
```

and run it:

```
python plot_polar.py
```

It prints the lift and drag coefficient at each angle and writes `polar.png`:



Lift against angle of attack, and the drag polar, at 0°, 5° and 10°

NASA's CFL3D code gives a lift coefficient of 1.0778 at 10°, and drag coefficients of 0.00828 at 0° and 0.01245 at 10°, on a finer grid of the same aerofoil. At 5°, Ladson's wind-tunnel measurements give a lift coefficient of about 0.54. On this grid zCFD's lift lies within a few hundredths of these values, and its drag within a thousandth of CFL3D's.

Closer to stall, the angle at which the flow separates from the upper surface and the lift stops rising, the flow is much harder to converge. Ladson's measured lift peaks near 17°, and at 15° a steady run on this grid needs many times more cycles than at 10° before its forces stop changing. Above about 10°, always check in zMon that the lift has stopped drifting before you add a point to the polar.

The trimmed angle comes out close to 10°, consistent with the polar. The status file `naca0012_target_cl_status.yaml` records whether the target was met and at which angle.

ParaView

> **NOTE**
>
> The .vtkhdf output format requires ParaView 6.1.1 or later.

Each run's output opens in ParaView as in Simple aerofoil (p. 384). Colour the volume output of the 10° run by `cp` to see the suction peak move forward onto the upper surface, and the stagnation point move round under the leading edge, as the angle rises.

PART 4

Release notes

What changed in each release of zCFD, newest first.

Unreleased

2026.11.16174

New features

- Added a sliding-plane interface for rotating meshes, and a low-memory implicit scheme, `mf-gmres`.
- Extended the incompressible solver to fluid zones and stratified atmospheric flows.
- Added compressible and thermal wall functions, trimming of the angle of attack to a target lift or drag, and adaptive inner-cycle convergence for dual time stepping.
- Added a mesh checker, `check_mesh`, Plot3D mesh import, more detailed residual reporting and more flexible restarts onto a different mesh. The user guide has been rewritten and published online.

Changes in behaviour

- Control files must use `config_version: 2`; convert older decks with `update_input`. Mesh placement and motion are set in a single `transforms` block.
- Decks using the `'barth jespersen'` limiter should change to `'symmetric ratio'` to keep their current behaviour. Overset cases require `implicit euler`, `lu-sgs` or `mf-gmres`.
- Overset, low-Mach preconditioned and DES-family results differ from earlier releases; preconditioned decks may need a gentler CFL ramp.

Removals

- Removed the `'immersed wall'` boundary condition and its output settings, the porous zone `porosity` and `direction` fields, and the actuator-disc controller type `none` (use `fixed`).

Fixes

- Improved robustness and accuracy of wall-function, overset and restart workflows.

Validation cases

- Added the DARPA SUBOFF fully appended submarine.

Platforms and packaging

- Built with CUDA 13.4; requires NVIDIA driver 580 or newer. ParaView 6.2.0 is bundled. Apptainer containers are provided separately.

2026.08.14860

New features

- The incompressible solver adds an energy equation with buoyancy, a segregated scheme (`scheme: "simple"`) and linear periodic boundaries.
- Added a low-memory implicit scheme, `lu-sgs`.
- A run stops cleanly, with final output and a restart checkpoint, when a `<case name>.stop` file appears in the run directory.
- `convert_mesh` converts OpenFOAM meshes, and overset re-mapping on moving meshes is faster.

Changes in behaviour

- Partitioned VTKHDF volume output opens in ParaView without Clean to Grid.
- The launch scripts ignore Python packages in the user's site-packages; set `ZCFD_ALLOW_USER_SITE=1` to include them.

Fixes

- Improved robustness of rotating-mesh and moving overset workflows.

Platforms and packaging

- zMon is installed with zCFD, and `run_zcfd --apptainer` launches the solver in an Apptainer container.

2026.07.14548

New features

- Added a low-dissipation HLLC scheme, `'HLLC low diss'`, for LES and hybrid RANS/LES.
- The CGNS converter handles structured and polyhedral meshes.
- The incompressible solver writes a wider range of output variables.
- Overset assembly can use the GPU.

Changes in behaviour

- The mesh conversion script `convert` is renamed `convert_mesh`.
- The low-dissipation settings are renamed `low dissipation sensor`, `low dissipation sensor parameter` and `dissipation sensor minimum`; the old `roe ...` names are still accepted. Results with the `'FD'` and `'DES'` sensors differ from earlier releases.

2026.06.14412

New features

- Added `convert`, a single mesh-conversion command for Fluent (including CFF), SU2, CGNS and UGRID meshes, and `meshstat`, which reports a mesh's size and the memory a run will need.
- Added `generate_paraview_helpers`, which writes ParaView state files for viewing overset results.
- Added a `"velocity"` option for `roe low dissipation sensor parameter`.

Changes in behaviour

- Volume and surface output is written in the VTKHDF format, one `.vtkhdf` file per volume and per boundary.
- Overset runs log compact mapping summaries; set `verbose logging: True` in the `mapping` block for full diagnostics.

Removals

- Removed the discontinuous Galerkin solver (including `dgcaa`), the legacy external fluid-structure coupling and the `lpumuscl` option; use `muscl` or `umuscl`.

Fixes

- Improved accuracy of hybrid RANS/LES and moving-mesh simulations.

Platforms and packaging

- ParaView 6.1.1 is bundled.

2025.11.13331

New features

- The user guide documents the incompressible solver settings for viscous, RANS and LES runs.

Changes in behaviour

- Reported incompressible residual levels differ from earlier releases.
- Incompressible decks must not set compressible-only time-marching keys, such as `cfl`, the CFL ramp and the multigrid settings.

Validation cases

- Added the NASA 3D bump-in-channel case for the compressible and incompressible solvers.

Platforms and packaging

- GPU builds support NVIDIA Blackwell and require an Ampere or newer GPU.

2025.09.12996

New features

- In-situ visualisation uses ParaView Catalyst 2.
- zutil adds acoustic post-processing: A-weighting, overall sound pressure level and spectrum plots.

Changes in behaviour

- `validate_input` normalises a rotation `axis` that is not a unit vector, with a warning.

Validation cases

- Added the Windsor square-back body for the incompressible solver, compared with wind-tunnel data.

Platforms and packaging

- ParaView 6.0 is bundled, and the bundled Python is 3.12.

2025.07.12237

New features

- Actuator discs report the mass flow through the disc.
- zutil can extract report columns, filtered to the final inner cycle of each real time step.

Platforms and packaging

- The `activate` script supports `bash` and `zsh`.

2025.06.11828

Changes in behaviour

- Reduced GPU memory use.

Fixes

- Improved robustness of overset and acoustics workflows.

2025.05.11605

New features

- Porous media zones and actuator discs work on moving meshes.
- Added an `entropy fix` setting to the numerical scheme.
- `run_zcfd` accepts an `https://` or `s3://` URL for the mesh or control file.
- zutil adds mesh transform helpers and acoustic post-processing, and the user guide gains an overset tutorial.

Changes in behaviour

- When unset, `turbulence intensity` defaults to 0.01 and `eddy viscosity ratio` to 0.1; set them explicitly to keep earlier values.
- `run_fwh` writes each surface's contribution separately; use `--sum_contributions` for summed output only.

Removals

- Removed the `mesh transform function` setting; use `scale` and `mesh transform matrix` instead.

2025.03.10415

Fixes

- Improved robustness of rotating fluid zones and GPU device selection.

Platforms and packaging

- The GPU build supports NVIDIA Hopper and Ada Lovelace GPUs and requires a driver compatible with CUDA 12.3.
- zCFD runs on Linux distributions with glibc 2.28 or later.

2023.next

- Added anisotropy to FRPM
- Added rotating terms to Jacobian
- Bug fix in fluent convert
- Added non-orthogonal correction to wall distance calculation
- Improved startup time of large jobs on GPUs
- Added force driver to alter boundary conditions to achieve a target lift
- Linear solver performance improvements

2025.02.10039

New features

- Immersed wall boundaries report viscous as well as pressure forces, and the new `immersed boundary friction velocity` output variable is available on immersed walls.

Changes in behaviour

- In dual time stepping runs, `first order cycles` counts cycles from the start of the simulation. Adjust the value if you relied on it applying at every real time step.

2025.01.9934

New features

- Added the `run_fwh` utility, which computes acoustic pressure histories at observer locations directly from a zCFD control file, including overset and rotating cases.
- Simpler acoustic set-up, with `dt` and `permeable` now optional, automatic summation over surfaces, and utilities to view surface data by frequency in ParaView and to exclude chosen faces.
- Time-averaged runs store averaged boundary data, which can be output and is kept across a restart.
- The `zmesh` mesher can generate surrogate STLs for post-processing.

Changes in behaviour

- Each initial condition block must give the velocity `V` as a `vector`.
- A porous fluid zone can be defined by `def` alone; `zone` is optional.
- Acoustic surface data from rotating meshes is written in the unrotated mesh frame.

Fixes

- Improved robustness of mesh transforms and acoustic workflows.

2024.11.9510

New features

- The mesher takes `--stlSpacingResolution` to control the sizing of refinement regions around each STL.
- zutil adds `ZCFD_Result`, which locates a run's report, log, output and checkpoint files from its control file.
- zutil's `include` can share variables from an included file with the calling control file.

Changes in behaviour

- In rotating-reference-frame cases, force report columns are given in the inertial frame by default. Set `'inertial frame': False` in the report entry to keep the fixed frame. The separate `_Fif`, `_Fpif`, `_Ffif`, `_Mif`, `_Mpif` and `_Mfif` columns are no longer written.
- Mesher refinement regions use the level nearest the requested spacing; check the spacing printed by the mesher.
- zutil's `cp_profile` and `cf_profile` take a surface, slice normal and slice location. `cp_profile_span`, `cp_profile_wall_from_file_span`, `get_chord_spanwise` and `residual_plot` are removed.

Fixes

- Improved immersed-boundary and acoustic workflows.

2024.10.9337

Fixes

- Improved robustness of rotating-frame, aeroelastic and acoustic workflows.

2024.09.9288

New features

- A mesh can be placed with a general affine transform through `mesh transform matrix`, and the new `affine transform` fluid zone moves meshes by a time-dependent matrix.
- Added mesh quality remediation, controlled by `mesh quality remediation angle threshold`, with a `badcells` output variable showing the affected cells.
- Overset rotating meshes support immersed boundaries, so overset blades can be run in the blade frame. Added the `immersed boundary yplus` output variable.
- zMesh takes `--spacingOnlySTL` for STLs that set mesh spacing without being meshed as geometry.
- Acoustic and checkpoint output are written without interrupting the solve.

Fixes

- Improved robustness of restart and unsteady workflows.

Validation cases

- Added the IEA 15 MW reference wind turbine.

2024.06.8798

New features

- Forces can be reported on immersed walls, and immersed-boundary solutions can be written on STL surfaces.
- Force reports in a rotating reference frame include thrust along the rotation axis.

Changes in behaviour

- Actuator-disc fluid zones (`type: "disc"`) are defined through `controller`, `geometry`, `discretisation` and `model` blocks and require `name` and `rotation direction`. Rewrite decks that use the previous flat keys. The turbine and propeller utilities have moved to `zutil.zWind`.
- `immersed boundary surface` in `write output` takes a list of `zone` and `stl` pairs; replace `immersed boundary surface file` with it.
- `sponge layer damping` requires `distance`, `damping factor` and `condition`.

Fixes

- Improved robustness of overset, ParaView reader and cluster launch workflows.

2024.05.8491

New features

- `run_zcfd -v` prints the zCFD version, and `run_zcfd -o <n>` sets the OpenMP threads per process.
- Acoustic observers can record over a chosen window with `observerStartTime` and `observerEndTime`.

Changes in behaviour

- The acoustic solver settings require `c`, `dt`, `rho0` and `p0`. Remove any per-surface `startTimeI` and `endTimeI` keys.
- Under Slurm, `run_zcfd` limits OpenMP threads to the job's allocation.

Fixes

- Improved robustness of acoustic workflows.

2024.04.8259

New features

- The incompressible solver supports moving meshes, with rotating and translating cell zones and moving walls.
- Force reports in rotating-frame cases include torque and power, and `dimensional: True` gives forces in dimensional units.
- Extended immersed-wall options, including surface output, and a `condition` key for `sponge layer damping`.
- The `zmesh` mesher accepts `--stlDistanceSpacings` to set spacing by distance from each STL.

Changes in behaviour

- The default `inviscid flux scheme` is now `"Roe"`. Set `"HLLC"` to keep previous results.
- `qcr` takes `2000`, `2020` or `2021` in place of `True`/`False`.

Removals

- Removed the `run_mycluster` and `run_myclusterui` scripts and zutil's remote ParaView server helpers.

Fixes

- Improved accuracy and robustness across the solvers, output and restart.

Platforms and packaging

- Added support for GPUs of compute capability 8.6.

2023.11.6459

New features

- Implicit time marching can be used with translational periodic boundaries.
- Added gradient and immersed-boundary output variables, and acoustic surface output can be viewed in ParaView.
- New zm3 mesher options for removing volumes inside STLs, global spacing scaling and recovering the generating command line. `run_zcfd` takes `--location` to give the installation path.
- New and updated user-guide tutorials, including acoustics and aero-acoustics.

Changes in behaviour

- Replace `double precision linear solve` in `time marching` with `double precision` in `linear solver options`.
- The default `partitioner` has changed; set it explicitly to reproduce earlier partitions.
- Rename the `velocitygradient` output variable to `velocitygrad`.

Fixes

- Improved robustness of overset, periodic and aeroelastic workflows.

Platforms and packaging

- zCFD runs on macOS 14.

2023.10.5890

New features

- The incompressible solver supports the SA-neg turbulence model.
- More choices are available for `limiter`.
- The zM3 mesh generator gains new refinement options and a page in the guide.
- Extended acoustic output.
- Faster start-up and solution mapping on large cases.
- The user guide gains new pages and quick-start tutorials.

Changes in behaviour

- An exponential CFL ramp continues until it reaches the maximum CFL, and a run stops with an error when its residuals become invalid.
- Validation rejects a callable `mach` in an initial condition that a boundary condition refers to.

2023.06.5001

New features

- Added incompressible LES, selected with the `incomp les` equation set.
- The incompressible solver can time-average the solution, hold a fixed `mass flow rate` at inflow and outflow boundaries, and report the mass flow through each boundary.
- Added FW-H surface data output for acoustic post-processing.
- Mesh morphing can run on the GPU, new output variables are available, and `vtkToEnsight` converts output to EnSight format.

Changes in behaviour

- `run_zcfd` takes the mesh with `-m`. The status file names the mesh under the key `mesh`.
- `cfl` defaults to 1.0 when it is not set, and the incompressible `relax` must lie between 0.1 and 1.
- Incompressible `pressureforce` output is relative to the reference pressure, and an inflow boundary's total pressure and total temperature ratios are relative to its own reference conditions.

Fixes

- Improved robustness and accuracy of incompressible, overset and FSI workflows.

2023.03.3383

New features

- Overset and sliding meshes can be combined with a rotating frame of reference.
- Modal-model FSI runs in steady as well as unsteady simulations.
- Added the `'immersed wall'` boundary condition.
- Added `cba_convert` for CBA structured multiblock meshes, and the `menterf1` output variable.

Changes in behaviour

- The overset key `sliding planes` is renamed `create overset halos`.
- `run_zcfd` stops with an error on an unknown option, and validation rejects reports that share a name.

Fixes

- Improved robustness of restart and FSI workflows.

Platforms and packaging

- ParaView 5.11 is bundled.

2023.01.3113

New features

- A farfield initial condition can be a driving function, re-evaluated at each report, so a run can steer itself towards a target such as a lift coefficient and stop when it is reached.
- A pressure inflow can take an optional `mach`.
- The acoustic solver offers `multi level` time marching on Cartesian meshes and further FRPM source controls.
- The Fluent converter accepts internal boundary zones and unrecognised boundary types.

Changes in behaviour

- Validation requires reports of each kind to have unique names, and rejects periodic boundary conditions with `implicit euler` time marching.
- A pressure outflow must set a static pressure ratio.

Fixes

- Improved robustness of overset and rotating-mesh workflows.

Platforms and packaging

- ParaView 5.10.1 is bundled.

2022.09.2514

New features

- Added a porous fluid zone (`type: 'porous'`).
- Added the quadratic constitutive relation for SST and SA-neg, with `qcr: True` in the `turbulence` block.
- The limiter can be frozen at a given cycle with `freeze limiter cycle`.
- Modal-model FSI gains further mode mapping options, and the high-order solver gains a `cartesian aligned mesh` option for Cartesian refined meshes.

Changes in behaviour

- `double precision linear solve` defaults to `False`; set it to `True` to keep the previous setting.
- The compressible solver does not write a checkpoint when the solution contains invalid values.

Fixes

- Improved robustness of overset, restart and mesh conversion workflows.

Platforms and packaging

- Python 3.9 is bundled, Kepler GPUs such as the K80 are supported, and the solver warns at start-up when the GPU driver is older than the CUDA runtime.

2022.05.1788

New features

- Added the `"Roe"` and `"Roe low diss"` inviscid flux schemes, recommended for implicit runs at large CFL. The inviscid flux scheme can also be chosen for LES runs.
- The `implicit euler` scheme supports turbulence models, sponge layers, moving meshes and overset boundaries.
- The incompressible solver supports unsteady runs and an energy equation, and writes `yplus` and skin friction.
- Added sliding meshes (`'sliding planes': True`) and accelerating translating meshes (`translation function`).
- New run controls: `first order cycles`, `number of additional overset active layers` and `cycles per overset cycle`.
- `fwensight` output includes boundary data, and `tauconvert` converts TAU meshes.

Fixes

- Improved robustness of restart, overset and acoustic workflows.

2022.03.1467

New features

- Setting the environment variable `ZCFD_DISABLE_PINNED` lets the GPU solver run on systems that restrict pinned host memory.

Fixes

- Improved robustness of moving-mesh workflows on the GPU.

2022.02.1380

New features

- Added the `implicit euler` time-marching scheme, with linear solves on the GPU for both the compressible and incompressible solvers.
- Overset cases run on the GPU, can translate as well as rotate, and can time-average the solution.
- Added a generic FSI coupling (`fsi` with `type: "generic"`) for external structural codes.
- The SST model gains `rotation correction` and a settable `a1` constant.

Changes in behaviour

- Abaqus FSI is configured through the `fsi` block with `type: "abaqus"`, in place of the `abaqus fsi` block.
- Moving overset meshes need `moving mesh: True` in `time marching`.

Removals

- Removed the `lu-sgs` time-marching scheme and its settings block; use `implicit euler` instead.

Fixes

- Improved robustness of restart and FSI workflows.

2021.11.765

New features

- Added `su2convert`, which converts 3D SU2 meshes.
- Unsteady runs can accumulate time-averaged mean and RMS fields with `compute average and rms` in `write output`.
- A solution can be restarted on a different mesh with `interpolate restart`, and a viscous run can restart from SST or SA-neg results.
- The (I)DDES constants can be set in the `turbulence` block, and the high-order LES solver supports the WALE subgrid model.
- FRPM acoustic sources gain further inflow and near-wall controls.
- `run_zcfd --no-env-override` keeps environment variables that are already set.

Fixes

- Improved robustness of overset, restart and mesh import workflows.

2021.08.269

New features

- Overset cases can include rotating meshes.
- The DG solver supports curved, high-order mesh geometry, and gains options for faster runs.
- Added a `separation correction` option to the `sst-transition` model, and `turbulencetimestep` as a RANS output variable.

Changes in behaviour

- In the `dgcaa` block, rename `no caa only frpm solver` to `frpm solver only`.
- Remove `multipoly cycle pattern` from DG control files; `multipolycfl` takes two values, for the highest and lowest orders.
- Monitor points in DGCAA runs must not request `forces` or `massflow`.

Fixes

- Improved robustness of the DG solver, overset runs and cluster launch.

Platforms and packaging

- InfiniBand runs are supported inside a container.

2021.07.123

- Deal with conformal and non-conformal interfaces in Fluent meshes
- Improve stability of SST and SA-neg
- Added Langtry-Menter SST transition model
- Additional dynamically updatable parameters
- Increased granularity of output control

2021.05.230

New features

- Overset runs can use multigrid.
- `run_zcfd` and `cluster_run` take a `-f` option naming a Python file that sets the meshes, cases and run options, which simplifies overset runs.
- The DGCAA solver accepts several FRPM domains.
- `volume interpolate` output accepts binary STL files.
- The modal model can be updated every n cycles through `update frequency`.

Changes in behaviour

- In the `dgcaa` block, name the FRPM dictionary `'FRPM_1'`, even when there is only one domain.

Fixes

- Improved robustness of overset and high-order workflows.

Platforms and packaging

- Added support for NVIDIA Ampere GPUs. ParaView 5.9.0 is bundled.

2021.04.122

- Implemented FSI with rotating zones
- Added specification of mass flow rate for FV solver inlet and outlet BCs
- RBF/FSI bug fixes
- Added RBF/FSI restarts
- Added distributed RBF solve
- Added check for duplicate zones in boundary conditions

2021.02.120

New features

- Added overset meshes for the finite-volume and DG solvers, set with the `overset` boundary condition type.
- Added a further mesh deformation method for mesh transforms and fluid-structure interaction, and remote coupling to Abaqus.
- Inflow and outflow conditions can vary in space, given as a field file or a Python function.
- Forces can be reported relative to a `reference pressure`.
- Extended the incompressible solver with force reporting, restarts and additional output variables, and the DG solver with low-Mach preconditioning for unsteady runs.

Changes in behaviour

- zCFD runs on Python 3; control files must be valid Python 3.
- In mesh transform blocks, replace the `multiscale` flag with `type` (`'rbf'`, `'rbf multiscale'` or `'idw'`). In a modal model, replace `rbf type` with `transform type`; the default `alpha` is 1.0.
- The unsteady `order` defaults to second order, and `FRPM length scale resolution factor` to 0.5; set them explicitly to keep earlier settings.

Fixes

- Improved robustness of restart and incompressible workflows.

Validation cases

- Added the Skyhunter UAV low-speed case to the release suite.

Platforms and packaging

- GPUs of compute capability 3.0 are no longer supported. ParaView is updated to 5.8.1.

2020.06.110

New features

- Added fluid-structure interaction with a modal structural model, set up through a `modal model` block using Nastran mode shapes, in the compressible finite-volume solvers on CPU and GPU.
- The modal model supports mode selection, per-mode damping and prescribed forcing, with output variables for mode shapes, deformation and mesh velocity.
- Added coupling to Abaqus through an `abaqus fsi` block and the `--cse-director` command-line option.
- Mesh checks report the location of any negative-volume cell.

Changes in behaviour

- A restarted dual time stepping run takes its first real time step at first order in time.

2020.04.108

New features

- The DG CAA solver can be initialised from a Python function through `initial: {'func': ...}`.

Fixes

- Improved robustness of parallel and multi-zone runs.

Platforms and packaging

- `run_zcfd` selects a working network fabric automatically; set `FI_PROVIDER` to choose one.

2020.03.106

New features

- Added a blade element propeller model for disc fluid zones, `model: 'blade element propellor'`, with tip loss correction options. The user guide gains a propeller modelling section.
- Turbine blade pitch control takes `blade pitch` and `blade pitch step`.
- Extended the FRPM acoustic source model with further turbulence and source-mapping options.
- `output real time cycle frequency` can be changed while a case is running.
- zCFD can run inside a container, including on GPUs.

Changes in behaviour

- The `up` vector of a disc fluid zone is optional and defaults to `[0.0, 0.0, 1.0]`.

Fixes

- Improved robustness of high-order restarts and mesh conversion.

2019.11.95

New features

- Added a blade element theory turbine model for actuator disc fluid zones, `model: 'blade element theory'`, described in the reference guide.
- Added the incompressible RANS solver, `equations: 'incomp rans'`.
- The DG solvers support isothermal and adiabatic walls, translational periodic boundaries and polynomial order 4, and the DG CAA solver gains further source and output controls.

Changes in behaviour

- `scale` no longer accepts a function; use the new `mesh transform function` instead.
- Several DG solver defaults have changed (`inviscid flux scheme`, `shock sensing variable`, `high order filter frequency`); set them explicitly to keep earlier settings. `ldg upwind parameter` must lie between 0 and 1.
- DG control files must not set low-Mach preconditioning or wall functions, and DG restarts must use the same number of partitions.
- In the `dgcaa` block, use `rans case name` and `rans mesh name` in place of `frpm rans case name` and `frpm rans mesh name`, and `src_x`, `src_y` and `src_z` in place of `acousticsrc`. Microphone data is written to an `ACOUSTIC_DATA` subdirectory.
- Runtime-updatable settings are checked per solver; remove any key that does not apply to the solver in use.

Removals

- The `dgeuler` equations, and the DG settings `cell order`, `approximate curved boundaries`, `use muscl reconstruction`, `br2 diffusive flux scheme`, `use rusanov for turbulence equations` and `c11 stability parameter transport`, have been removed; delete them from control files.

Platforms and packaging

- Added support for NVIDIA Turing GPUs.

2019.06.86

- Added multiscale RBF transformations

2019.01.68

- Added capability to restart SA solver from SST and SST from SA
- Improved validation of different solver/time stepping combinations in parameters dictionary
- Removed requirement for write permissions on restart file
- Improved handling and validation of dynamically updated parameters

2018.09.62

- Fixed viscous forces calculation in high order solver which were a factor of 2.0 too large
- Upgraded versions of CUDA, Intel MPI and MKL used. CUDA 9.2 is now required for GPU runs
- Added validate_input script to allow checking of control dictionary before run time
- Enforce input parameters that are either True or False to be booleans when validating the dictionary. i.e. 'true' is no longer valid
- Changed colour of timer breakdown output to improve readability

2018.07.60

- Optimised GPU execution with 2x performance improvement on V100
- Fixed bug in activate script. `zdeactivate` didn't restore PYTHONPATH correctly

2018.05.58

- Upgraded to Paraview 5.4
- Bug fix to node ordering for DG tetrahedra
- MUSCL reconstruction no longer demotes interface faces between FV and high order cells
- Improved low Mach number preconditioning for turbulent flow for SST and SA models
- DG Performance improvements
- Fixed wave speed calculation for Rusanov scheme for FV solver
- Fixes for P4 hexahedra
- Default parameter values output when solver is run without `controldict` present are now no longer merged with the supplied control dictionary
- Fix for launching using LSF job scheduler
- Added option to set cell orders in cylindrical regions for DG solver
- Correction to dynamic viscosity output and added output of turbulence kinetic energy
- Added sponge layer option for farfield boundary conditions
- Improved warning message for cell zone definition
- Set velocity to cell velocity on the farfield boundary for ALE cases
- Fix rotating cell zones partition face flux ale velocity

2018.02.2

New features

- Mesh transformations can select their basis function with `basis`, and `scale` can be a function of node position.
- Added an outflow `kind` of `'radial pressure gradient'` for rotating periodic cases.
- The high-order solver supports buoyancy and Spalart-Allmaras wall functions.
- Added time-step diagnostic output variables, and the report plotting tools can compare several cases.

Changes in behaviour

- In the `preconditioner` block, rename `factor` to `minimum mach number`.
- Boundary and initial condition dictionaries are validated before the run starts; an `initial` dictionary no longer requires `func`.

Removals

- The `'scotch'` partitioner has been removed; remove it from `partitioner` settings.

Platforms and packaging

- Added support for NVIDIA Volta GPUs.

2017.2.3

- Corrected DG WALE solver and exposed DG WALE solver to the user
- Correction to BR2 scheme
- Bug fix to velocity gradient vtk output for DG solver

2017.2.2

- Print correct release version number, 2017.2.1 release inadvertently reported 2017.2.0
- Improved memory allocation reservations outputting to VTK format
- Always sync gradients from GPU when outputting
- Fixed logic to detect lu-sgs
- Fix to allow P4 hexas
- Efficiency improvements to the BR2 diffusive flux scheme
- Wall functions corrected for HO solver

2017.2.1

- Added ability for use to select SST DES variants
- When requested Python dictionary is not available, write default
- Fix issue on Skylake based CPUs
- Improve performance of P0 cases
- Correction to force output extracted in surfaces

2017.2.0

- Added ability to override convolution function from `controldict`
- Python libraries now link against bundled `openssl`
- Performance improvements
- Added some validation of parameters before running the solver
- Corrected parsing of filtering epsilon and filtering strength values so they are treated as double instead of truncated to `int`
- Fix to roughness values when running in parallel
- Added BR2 scheme to High Order code
- Ensured halo conditions are correct for strong Boundary Condition formulations
- Added improved interpolation for temperature field input
- Ensured restricted `mut` is the average of the child cells
- Fixes to restart logic
- Timing output has been improved to report in days. hours, minutes and seconds
- Enabled CUDA optimisations for Pascal architecture GPUS
- Added support for P3 Tris
- Bug fix to prism elements when running in parallel
- Allow user to scale cfl number for viscous timestep
- Added update frequency parameter for turbine zone definitions
- Added a native implementation of the convolution function
- Added specific keywords for specifying the turbine fluid zone length factor and the reference point offset
- Added automatic check to make sure that the reference point is located inside the turbine region
- Added fix for case where only one turbine location is specified as this broke the numpy logic
- Added warning when users specify a thrust coefficient greater than 1 as this invalidates the induction factor calculation later
- Added turbine power curve to turbine fluid zones
- Added keyword to specify the model (induction or simple) to be used for modelling turbines
- Added function 'create_turbine_segments_simple' which uniformly applies thrust and torque to the actuator disc segments, based on the thrust coefficient, tip speed ratio and power curves supplied by the user
- Added thrust summation for simple turbine
- Added use of a search tree to speed up cell searching in turbine regions
- Turbine region function now use line and tube filters rather than cylinder
- Added dissipation to boundary fluxes
- Turned on extrapolation for Farfield Boundary conditions
- Added support for Cray XC systems. Contact us to get a specific version for this platform
- Added relocate option for when the executable on the compute nodes are in a different location to the submit node

- Fixes to WALE model for High Order solver
- Added SA Neg Model for Finite Volume solver
- Added SA Model for High Order solver
- Improved output of high order eddy viscosity
- Robustness fixes for reference plane cell identification
- Added option to output fluidic zone cell sets
- Removed warnings when TRBX file specifies thrust coefficient greater than 1
- Added warning when thrust coefficient > 1 is specified for the 'induction' model. CT is then scaled back to 0.8
- Added turbine power output for Betz limit turbine
- Allow user to set shock sensor variable
- multigrid_cycles variable can be use to turn off Polynomial Multigrid for the High Order solver
- Allow specifying the real time step frequency at which the restart H5 file gets written
- Added user option to scale residuals by volume
- Ensured dynamic viscosity and kinematic viscosity output total mu+mu_t
- Allow using Rusanov for turbulence when HLLC is chosen for High Order solver
- Added ABL fluidic zone type
- Added ability to specify the PMG cycle pattern from the python dictionary using the 'multipoly cycle pattern' key
- Ensure that the last solve cycle does a PMG cycle in the High Order Solver
- Added STL file option for turbine offset specification
- Improved parallel load balancing
- Added SAS to LES options
- Changed default ambient turbulence to off so that SA and SST results match
- Improve startup performance for High Order Solver
- Added option to turn on rotation correction for SA
- Fix to memory allocation for SAS
- Added Boussinesq buoyancy
- Write turbine power to file on update
- Added `libgfortran` to tarball for systems where it's not already installed

2017.1.1

- Moved cfl logic for coarse grids into the driver layer
- Allow overriding of RungeKutta scheme from users control dictionary
- Fixed bug with incorrect gradient values being calculated at boundaries
- Slip walls now extrapolate at the boundary
- Updated a number of third party dependencies

2017.1.0

- run_zcfd script now passes through the tasks per node parameter to the `smartlaunch` script which is needed when not running through a scheduler
- Made Intel MPI aware of the filesystem it is running on to improve performance on parallel file systems and address issues seen running on GPFS
- Fixed bug with periodic boundary condition
- Removed some code duplication in the python layer
- Fixed bug in `create_trbx_zcfd_input` that caused turbine regions not to be rotated.
- Bug fix to physical nodal locations in high order code
- Added curvature approximation for high order
- Added porous media support
- Fixed bug detecting actuator disc reference cell when running in parallel
- Ensure the generated turbine region contains the reference point
- Added thrust and tip speed interpolation functions
- Added screen output for actuator disc initialisation
- Performance improvements of file reading during restart
- Added eddy output for boundary faces for high order
- Bug fix to quad nodal location ordering
- Fixed bug in high order filtering
- Added MUSCL reconstruction option for P0
- Changed binding behaviour to bind to sockets if oversubscribing the number of processes on a node
- Added checks for existence of the Unix tool `numactl`
- Added checks for the presence of the mesh and control files
- Bug fix to viscous force calculation for high order
- Stability improvements to polynomial multigrid
- Upgraded to Python 2.7.13
- Removed requirement for a C++11 compiler for the foamtozCFD converter
- Added missing face area scaling for viscous friction calculation
- Only convect at lower polynomial orders for PMG
- Include pressure and friction forces in reports for High Order
- Fixes to support multiple moving reference frames
- Allow selection of Rusanov scheme for high order
- Added support for multiple cell zones in a single fluid zone entry
- Added translating mesh option
- Fix to ALE HLLC fluxes
- Added moment reporting
- Updated reporting notebook with plotting functionality
- Enable time history vtk output for global timestepping
- Various fixes to global timestepping scheme

- Added vorticity vector output for HO
- Fix support for mixed element CGNS files
- Fix dimensional Q criterion output
- Added helicity output for HO
- Added ability to choose between finite difference and analytic Jacobian for LU-SGS FV
- Ensure reported residual is factored by CFL
- Fixes to FV LU-SGS residual reporting
- Added option to use Rusanov scheme for FV LU-SGS Jacobian
- Added pressure limiter to implicit time marching
- Added generic finite difference boundary Jacobian calculator
- Added analytic Jacobians for Wall, Wall Function, Farfield, Inflow, Outflow, Canopy and Symmetry boundary conditions
- Added ability to freeze viscous fluxes for FV LU-SGS sweeps
- Added support to restart from LnOmega turbulence model
- Bug fixes to preconditioning
- Added support for P4 Hexes for HO
- Added scaling of turbulent CFL for FV LU-SGS
- Added shock sensor for HO
- Enabled FV LU-SGS with multigrid
- Added output of artificial viscosity for HO
- Added logic to supply a start and end real time cycle for outputting solution in vtk format
- Add momentum source Jacobians for FV LU-SGS
- Added capability to identify cells at a plane in actuator disc region
- Added actuator disc plane average reference velocity calculation
- Use diffusive flux on the boundary faces rather than applying penalty parameter
- Only rename the hdf5 file on the master process
- Added new functions for conserved gradient calculation
- Fix for implicit solver with farfield subzones
- Added in prolongation relaxation factor for PMG
- Allow user to scale artificial viscosity
- Changed behaviour or restart reporting. Will create a backup of the restarted report file and start a new report file with all of the variable being reported at the top
- Put all *vtu and* vtp in a VISUALISATION sub-directory to help tidy up our file output. *.pvd files remain where they are and point to the files in VISUALISATION
- Put log files in a LOGGING sub-directory
- Added OpenSSL libraries to release distribution

2016.3.0

- Initial public release

2016.2.0

New features

- Added an implicit time-marching scheme for the finite-volume solvers, `scheme: {'name': 'lu-sgs'}`.
- Added high-order SST solvers for detached-eddy simulation (`DGMenterDES`) and RANS (`DGRANS`), with pressure outflow boundaries and P3 hexahedra.
- Wind energy: canopy model constants can be set per fluid zone, turbine zones can be built from turbine definition files, and probe data can be exported for wind-farm analysis.
- Extended post-processing with friction moments, centres of action and spanwise profiles.
- Added scripts to start notebook and ParaView servers, plot residuals and submit cluster jobs.

Changes in behaviour

- Results near farfield boundaries may differ from earlier versions.
- Reported residuals are normalised differently, so convergence histories are not directly comparable with earlier versions.

Fixes

- Improved robustness of parallel and high-order runs.

2016.1.1

New features

- A parallel finite-volume solver for compressible inviscid, laminar and turbulent flow, with RANS, scale-adaptive and transition models.
- A high-order discontinuous Galerkin solver for inviscid, viscous and LES flows on mixed-element meshes, with variable polynomial order.
- Steady and unsteady time marching with convergence acceleration and low-Mach preconditioning.
- A full set of boundary conditions, with rotating frames, moving walls and fluid zones for canopies, momentum sources and actuator discs.
- Atmospheric boundary-layer flows with buoyancy and Coriolis effects.
- Mesh import from Fluent, CGNS, STAR-CCM+ and UGRID, with run-time partitioning and mesh morphing.
- Runs in parallel on CPUs and NVIDIA GPUs, with surface and volume output, in-situ processing through ParaView Catalyst, force reports and restarts.

Validation cases

- The verification guide covers the Drag Prediction Workshop Common Research Model and the Ahmed body.