Wake CFD Docs

CLEW, Inc.

2026-11-06

Table of contents

About this manual **3**
- Wake CFD at a glance . . . 3
- Choosing a solver . . . 4
- How to use this manual . . . 4
- One case model across interfaces . . . 5
- A reproducible first run . . . 5
- Validation and engineering use . . . 6
- Document conventions . . . 6

I 1. Getting started **7**
- Install and run . . . 8
- Related guides . . . 8

1 Installation **10**
- 1.1 Distribution plan . . . 10
- 1.2 Requirements . . . 10
- 1.3 Windows: Installer . . . 11
- 1.4 Linux: Compressed binary and `install.sh` . . . 11
- 1.5 Verify installation . . . 12
- 1.6 Distribution layout . . . 13
- 1.7 Running Wake CFD . . . 13
  - 1.7.1 Approach 1 — Call using a relative or absolute path from the working directory . . . 13
  - 1.7.2 Approach 2 — Add `bin` to PATH . . . 15
- 1.8 Migrating existing YAML cases . . . 15
- 1.9 Related guides . . . 16

2 Quickstart: ONERA M6 wing **17**
- 2.1 What this example demonstrates . . . 17
- 2.2 1. Copy the example to a working directory . . . 18
- 2.3 2. Run the case . . . 18
- 2.4 3. Check the run . . . 19
- 2.5 4. View the results . . . 19
- 2.6 Related guides . . . 20

II 2. How to use Wake CFD **22**
- Choosing an interface . . . 23

Common workflows . . . 24
Settings shared across all interfaces . . . 24
Next steps . . . 25

GUI guide **26**
GUI chapters . . . 26
Find a topic . . . 27
GUI scope . . . 27
Related guides . . . 27

3 ONERA M6 GUI quickstart **28**
3.1 Planned workflow . . . 28

4 Create and configure a project **30**
4.1 Create a new project . . . 30
4.2 Save and export . . . 30

5 Mesh and boundary conditions **31**
5.1 Basic workflow . . . 31
5.2 Required checks . . . 31

6 Run and monitor **32**
6.1 Pre-run checks . . . 32
6.2 Check convergence . . . 32

7 Review and export results **33**
7.1 Result-file roles . . . 33

CLI guide **34**
CLI chapters . . . 34
Find a topic . . . 34
CLI scope . . . 35
Related guides . . . 35

8 Running Wake CFD **36**
8.1 Overview . . . 36
8.2 Basic execution . . . 36
8.3 Command-line options . . . 37
8.4 Input-file composition . . . 39
8.4.1 Starter template . . . 39
8.4.2 `$extends` (inheritance) . . . 40
8.4.3 `--overlay` (Merge Patch, RFC 7396) . . . 41
8.4.4 `--patch` (JSON Patch, RFC 6902) . . . 42
8.4.5 Composition order . . . 44
8.5 MPI execution . . . 44
8.5.1 Slurm (Linux HPC clusters) . . . 45
8.6 Pre-run verification . . . 45
8.7 Related guides . . . 46

9 Controlling a running job **47**
9.1 Stopping with a signal . . . 47
9.2 `wake.control.json` — scheduled stop and snapshots . . . 47
9.3 gRPC . . . 48

10 Environment variables **49**
10.1 Overview . . . 49
10.2 Setting environment variables . . . 49
10.2.1 Command-line (one-time use) . . . 49
10.2.2 `~/.wake/env` file (persistent defaults) . . . 50
10.3 Logging . . . 50
10.3.1 `WAKE_LOG_LEVEL` . . . 50
10.3.2 `WAKE_LOG_SCOPE` . . . 51
10.4 GPU selection . . . 52
10.4.1 `CUDA_VISIBLE_DEVICES` . . . 52
10.5 Summary . . . 53
10.6 Related guides . . . 53

11 CLI configuration-file reference **54**
11.1 Overview . . . 54
11.2 Input blocks . . . 54
11.3 JSONC syntax . . . 55
11.4 Configuration composition . . . 55

12 Troubleshooting **57**
12.1 Input file rejected . . . 57
12.2 Mesh names not found . . . 57
12.3 Convergence failure . . . 58
12.4 Solution divergence . . . 58
12.5 FR case errors . . . 58
12.6 Related guides . . . 59

III 3. Tutorials **60**

First CLI analysis: 2D external airfoil flow **62**
Prerequisites . . . 62
1. What does it calculate? . . . 62
2. Copy the example to a separate working directory . . . 63
3. Run the case . . . 63
4. What to check during the run . . . 63
Review the results . . . 64
First settings to change . . . 64
Next steps . . . 65

IV 4. Best Practices **66**

Recommended settings and troubleshooting **67**
Pre-run checklist . . . 67
Starting points by problem type . . . 67
Schemes and CFL . . . 68
Assessing convergence . . . 68
Mesh and wall treatment . . . 69
Output and restart operation . . . 69

V 5. Input and configuration (GUI/CLI) **71**
How to use this reference . . . 72
Find a topic . . . 72
Case-authoring sequence . . . 73
Related guides . . . 73

13 Mesh **75**
13.1 What this setting is . . . 75
13.2 Configure in the GUI . . . 75
13.3 Input keys . . . 76
13.4 Supported formats . . . 76
13.5 Multiple zones (multiple files) . . . 77
13.6 Boundary-name matching . . . 77
13.7 scale_factor . . . 79
13.8 JSONC examples . . . 79

14 Analysis **80**
14.1 What this setting is . . . 80
14.2 Configure in the GUI . . . 80
14.3 solver . . . 81
14.4 time . . . 81
14.5 dimension . . . 82
14.6 nondimensional . . . 82
14.7 JSONC examples . . . 82

15 Physical models **84**
15.1 What this setting is . . . 84
15.2 Configure in the GUI . . . 84
15.3 type . . . 85
15.4 regime . . . 85
15.5 axisymmetric . . . 86
15.6 turbulence . . . 87
15.6.1 model . . . 87
15.6.2 Model-constant overrides . . . 87
15.6.3 Wall treatment (global defaults) . . . 91
15.7 Four-axis solver identity . . . 92

15.8 Location of freestream turbulence quantities . . . 93
15.9 JSONC examples . . . 93

16 Materials **95**
16.1 What this setting is . . . 95
16.2 Configure in the GUI . . . 95
16.3 Material entry and type . . . 96
16.4 Equation of state (EOS) . . . 96
16.4.1 Property table . . . 97
16.5 Specific heat . . . 98
16.6 Viscosity . . . 99
16.7 Thermal conductivity . . . 99
16.8 Solid properties . . . 100
16.9 Reserved inputs . . . 100
16.10JSONC examples . . . 100

17 Reference values **103**
17.1 What this setting is . . . 103
17.2 Configure in the GUI . . . 103
17.3 Configure in JSONC . . . 103
17.4 Key definitions . . . 105
17.5 NACA0012 example (chord = 1 m) . . . 106
17.6 Reynolds-number reporting (not input) . . . 106

18 Domains and boundary conditions **108**
18.1 What this setting is . . . 108
18.2 Configure in the GUI . . . 108
18.3 Configure in JSONC . . . 109
18.4 domain.type . . . 110
18.5 mesh_zones . . . 110
18.6 motion (MRF, Multiple Reference Frame) . . . 110
18.7 Boundary conditions (nine-type tagged union) . . . 111
18.7.1 Wall . . . 112
18.7.2 Farfield . . . 114
18.7.3 Symmetry . . . 117
18.7.4 Internal . . . 117
18.7.5 `pressure_inlet`: total-pressure inlet . . . 118
18.7.6 `velocity_inlet`: velocity-magnitude inlet . . . 119
18.7.7 `mass_flow_inlet`: mass-flow inlet . . . 119
18.7.8 `pressure_outlet`: back-pressure outlet . . . 120
18.7.9 `mass_flow_outlet`: mass-flow outlet . . . 121

19 Interfaces **122**
19.1 What this setting is . . . 122
19.2 type . . . 123
19.3 GGI . . . 123
19.4 Periodic (single passage) . . . 124

19.5 Stage / Mixing plane . . . 124
19.6 Frozen rotor . . . 125
19.7 Sliding mesh (transient) . . . 126
19.8 CHT (Conjugate Heat Transfer) . . . 126

20 Numerical methods **128**
20.1 What this setting is . . . 128
20.2 Configure in the GUI . . . 128
20.3 Configure in JSONC . . . 129
20.4 Where step-size knobs live . . . 129
20.5 DBNS . . . 131
20.5.1 Convective terms . . . 131
20.5.2 Low-Mach dissipation and time stepping . . . 133
20.5.3 Time integration . . . 135
20.5.4 Mean-flow viscous flux . . . 135
20.5.5 CFL control . . . 136
20.5.6 Nonlinear control . . . 136
20.5.7 Advanced blocks . . . 138
20.6 PBNS . . . 138
20.6.1 Steady: `pseudo_time` . . . 138
20.6.2 Unsteady: CFL limits . . . 139
20.6.3 PBNS solver knobs . . . 139
20.7 FR . . . 141
20.7.1 time_integration . . . 141
20.7.2 Physical-Δt PI control . . . 142
20.7.3 Entropy-filter shock capturing . . . 142
20.7.4 Flux anti-aliasing . . . 143
20.7.5 dual time . . . 143
20.7.6 Artificial compressibility (AC) . . . 144
20.7.7 `source_terms` . . . 144
20.7.8 Boundary conditions and output . . . 144
20.8 Shared linear solver . . . 145
20.9 Jacobian refresh interval . . . 146
20.10 Solid conduction . . . 146

21 Run control and convergence **148**
21.1 Configure in the GUI . . . 148
21.2 Choose a setup . . . 149
21.2.1 DBNS steady . . . 150
21.2.2 PBNS steady . . . 150
21.2.3 DBNS unsteady: explicit . . . 152
21.2.4 DBNS unsteady: dual time . . . 152
21.2.5 PBNS unsteady . . . 152
21.2.6 FR unsteady: explicit . . . 153
21.2.7 FR unsteady: dual time . . . 153

21.3 Limits and restart . . . 154
21.3.1 `iteration`, `total_time` . . . 154
21.3.2 `resume` . . . 154
21.4 CFL . . . 154
21.4.1 CFL means something different for implicit and explicit schemes . . . 155
21.4.2 `schedule` . . . 155
21.4.3 `adaptive` . . . 156
21.5 `pseudo_time` . . . 158
21.6 `timestep_control` . . . 159
21.7 Convergence criteria . . . 160
21.7.1 residual . . . 161
21.7.2 monitor . . . 162
21.8 Unsteady physical time . . . 163
21.8.1 `timestep` and `total_time` . . . 163
21.9 Controlling a running job . . . 164
21.10 Restart numbering and FP32 convergence . . . 164

22 Initialization **165**
22.1 What this setting is . . . 165
22.2 Configure in the GUI . . . 165
22.3 type . . . 167
22.4 uniform . . . 167
22.4.1 Turbulence seeding . . . 168
22.4.2 Per-point fields . . . 169
22.5 FR preset . . . 170
22.6 restart . . . 170
22.7 Configure from the CLI . . . 171

23 File output **172**
23.1 What this setting is . . . 172
23.2 GUI configuration . . . 172
23.3 restart . . . 173
23.4 volumes . . . 175
23.5 surfaces and slices . . . 176
23.6 Choosing a `frequency` for volume/surface output . . . 178
23.7 time_average . . . 179
23.8 diagnostics . . . 179
23.9 Setting this from the CLI . . . 180
23.10 Rotating-frame energy field . . . 181

24 Monitoring **182**
24.1 What this setting is . . . 182
24.2 GUI configuration . . . 182
24.3 history_files . . . 183
24.3.1 Choosing `frequency` for post-processing (e.g. FFT) . . . 183
24.4 console . . . 183
24.5 print_inner_iter . . . 184

24.6 solver_state . . . 184
24.7 monitors . . . 185
24.7.1 `target` . . . 187
24.7.2 `function` . . . 187
24.8 Time-averaging a monitor . . . 188
24.9 Setting this from the CLI . . . 189

25 Expert parameters **190**
25.1 GPU edge accumulation . . . 190
25.2 Linear algebra . . . 190
25.3 Viscous spectral-radius factor . . . 192
25.4 Turbulence preconditioner . . . 192
25.5 MPI partitioner . . . 192
25.6 Other settings . . . 193
25.7 Precision and Krylov storage . . . 193
25.8 Mesh and material diagnostics . . . 194
25.8.1 Parallel mesh reading . . . 194

26 Expressions **195**
26.1 What this setting is . . . 195
26.2 Parameter table . . . 195
26.3 Constants and reusable expressions . . . 196
26.4 Reserved names and validation . . . 196

27 Complete input structure **197**
27.1 Format rules . . . 197
27.2 The 14 top-level blocks . . . 197
27.3 The 13 top-level blocks . . . 197
27.4 Minimal structure . . . 198
27.5 Four analysis axes . . . 199
27.6 Constants and expressions . . . 199
27.7 Inheritance and composition . . . 200

VI 6. Python API **202**
Related guides . . . 203

28 Python SDK **204**
28.1 When to use it . . . 204
28.2 Installation . . . 204
28.3 Shortest workflow . . . 205
28.4 Start from an existing JSONC case . . . 205
28.5 Parameter sweeps . . . 206
28.6 Results and runtime control . . . 206
28.7 Related documentation . . . 207

VII 7. Automation and integration **208**
Integration paths . . . 209
When to use each interface . . . 209

29 Slurm integration **210**
29.1 Basic job script . . . 210
29.2 Working directory . . . 211
29.3 Next steps . . . 211

30 Connect an AI agent **212**
30.1 What agents can do . . . 212
30.2 Connection method . . . 213
30.3 Recommended workflow . . . 214
30.4 Use with Wake CFD Desktop . . . 214
30.5 Engineer review points . . . 215

About this manual

Wake CFD is a GPU-based computational fluid dynamics (CFD) solver. Its primary documented engineering path is density-based Navier–Stokes (DBNS) for external aerodynamics. Pressure-based Navier–Stokes (PBNS) and Flux Reconstruction (FR) are in development. The formulations share a JSONC input model across the command line, desktop GUI, and automation workflows.

This manual explains how to install Wake CFD, select a solver, define a case, run it, and assess the result. It is written as a reference for engineers who need both a repeatable first run and enough detail to build production cases.

> **Note**
>
> **Version and scope.** The manual describes the capabilities and configuration schema of the Wake CFD build associated with the published documentation. Solver availability, platform support, and configuration keys can change between releases. Record the Wake CFD version with every engineering result.

Wake CFD at a glance

Use this table to find the part of the manual that matches the work you need to do.

Engineering task	Primary Wake CFD path	Start here
Install Wake CFD and run a known case	Desktop or CLI	Getting started
Compressible external aerodynamics	DBNS	Physical models
Low-Mach or incompressible flow	PBNS (in development)	Analysis setup
High-order, unsteady, or vortex-dominated flow	FR (in development)	Numerical methods
Turbomachinery and rotating domains	DBNS; PBNS path in development	Domains and Interfaces
Parameter studies, HPC, or external automation	CLI, Python, MPI, and Slurm	Automation and integration

Choosing a solver

DBNS, PBNS, and FR are different numerical formulations available through the same case model and execution environment. Choose the formulation from the flow physics and the required numerical behavior; do not choose it only from the mesh format or the user interface.

Formulation	Best suited to	Documented dimensional scope	Status
DBNS — density-based Navier–Stokes	Compressible external aerodynamics, transonic and hypersonic flow, turbomachinery	2D and 3D; Euler, laminar, and RANS paths	Production
PBNS — pressure-based Navier–Stokes	Low-Mach and pressure-based flow, including internal flow	3D; steady and unsteady controls depend on the case	In development
FR — Flux Reconstruction	High-order spatial discretization, unsteady and vortex-dominated flow	2D and 3D; polynomial order p=1–4	In development

The solver selection is independent of how a case is launched. A case can be created in the GUI, edited as JSONC for the CLI, or generated and monitored through the Python and automation interfaces.

How to use this manual

The chapters are organized by the decisions made during a typical analysis:

If you need to…	Read…
Install Wake CFD and complete a first run	Getting started
Use Wake CFD Desktop or the CLI	User interfaces
Learn from a guided engineering case	Learning and tutorials
Choose recommended settings or diagnose a run	Best practices
Define mesh, physics, domains, numerics, and outputs	Input and configuration
Integrate Wake CFD with Python, Slurm, or an AI agent	API and automation

Read the chapters in order for a first installation. For an experienced user, the input chapters can be used as a reference independently of the introductory chapters.

One case model across interfaces

The GUI, CLI, Python SDK, and automation workflows use the same JSONC case model. The model describes the mesh, analysis formulation, material and physical models, domains and boundaries, numerical method, run control, monitoring, and outputs. This common model lets a case move from interactive setup to a scripted or HPC run without translating the physics into a different input format. When an MCP agent attaches to Wake CFD Desktop, the same model becomes a live control surface: agent edits are reconciled into the open project for human review.

> **Note**
>
> **Figure placeholder — shared Wake CFD case workflow**
>
> ```
> Engineering intent
>         ↓
> Shared JSONC case model
>         ↓
> GUI  ·  CLI  ·  Python / automation
>         ↓
> DBNS  ·  PBNS  ·  FR solver paths
>         ↓
> Results, monitors, and validation review
> ```
>
> For the print release, replace this box with a final vector diagram showing the same relationships. The surrounding text defines the content and connections that the final figure must represent.

A reproducible first run

The recommended first run is the packaged ONERA M6 half-wing example. It exercises the installation, mesh loading, solver execution, output writing, and basic result review without requiring a new mesh or a case to be authored from scratch.

1. Install Wake CFD and confirm that the executable starts.
2. Run the ONERA M6 quickstart.
3. Inspect the residual history and the Mach, pressure, or Cp field.
4. Keep this result as a baseline before changing the mesh, boundary conditions, or physical model.

For a GUI-based version of the same workflow, see the GUI quickstart. For scripted execution, continue to the CLI running guide.

Validation and engineering use

Wake CFD results should be interpreted together with their mesh, boundary conditions, physical model, numerical settings, convergence history, and solver version. A successful process exit is not by itself evidence that the result is accurate.

Use the separate **Wake CFD Validation Reports** to review comparisons with analytical solutions, published reference computations, and experimental data. For a new engineering case:

- start from a comparable validated or verified case;
- change one major setting at a time and retain a baseline result;
- check conservation and convergence monitors before interpreting field data;
- record the solver, model, mesh, precision, and version with the result.

Document conventions

- Configuration keys and command-line arguments appear as `code`.
- Quantities are shown in SI units unless a section states otherwise.
- Tables summarize supported combinations; linked reference pages contain the full key descriptions and restrictions.
- A limitation or development status is stated where it affects solver selection or result interpretation.

The remainder of the manual provides the installation steps, interface workflows, input reference, numerical controls, automation hooks, and troubleshooting guidance needed to carry a case from setup to review.

Part I

1. Getting started

This section guides you through Wake CFD installation and initial verification. Complete the platform-specific installation, run the included ONERA M6 wing verification case, and review the results to confirm that Wake CFD is correctly installed and operational.

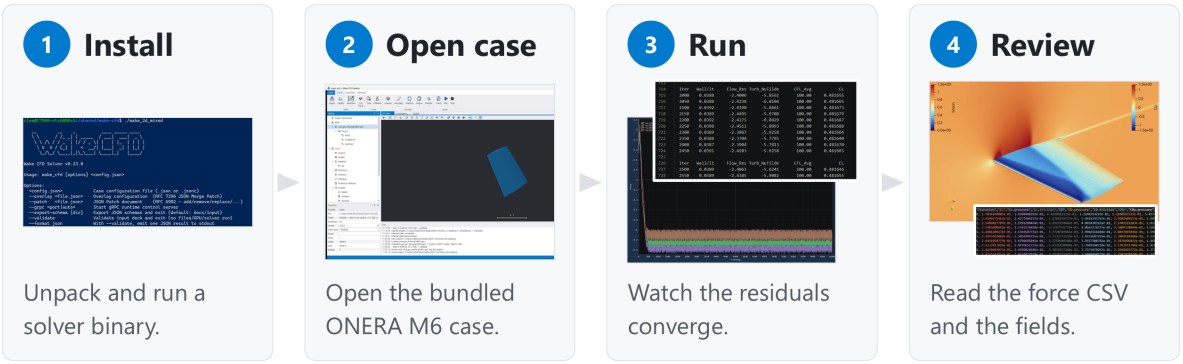


Figure 1: The first run, end to end: verify the installed solver, open the ONERA M6 case, run it, and review the results. Each step is covered in detail in the chapters that follow.

Install and run

Wake CFD is distributed in platform-specific formats for Windows and Linux. Complete the installation for your operating system, then execute the included ONERA M6 wing verification case.

1. **Install Wake CFD** — Follow the platform-specific installation procedure: Installation
2. **Run the verification case** — Execute the ONERA M6 example and review results: Quick-start: ONERA M6 wing

The installation procedure ensures that binaries and runtime libraries are correctly deployed. The verification case confirms that the solver executes and produces valid output. Completing both steps establishes a functional Wake CFD environment.

> Tip
>
> **Installation prerequisite**
> For new installations, refer to Installation for distribution formats, system requirements, and platform-specific installation procedures.

Related guides

- Workflows for the CLI, GUI, and API: How to use Wake CFD

- Input-file authoring and settings: Input and configuration reference
- Convergence or execution issues: CLI troubleshooting
- Automate with an AI agent: Connect an AI agent
- Learn through worked examples: Tutorials

1 Installation

Windows installer and Linux binary distribution

> **Note**
>
> **Wake CFD is distributed in a form appropriate for the operating system.**
> Windows provides an installer for Desktop and CLI components. Linux provides a compressed binary distribution with `install.sh` for user installation.

1.1 Distribution plan

Platform	Distribution	Recommended for
Windows 10 / 11	Installer (`WakeCFD-<version>-win-x64.exe`)	Individual workstations centered on Desktop
Linux x86_64	Compressed binary (`wake-<version>-linux-x64.tar.gz`) + `install.sh`	CLI, GPU servers, and MPI/HPC environments

Filenames and the list of supported distributions will be finalized for the first public release. The commands and folder structure on this page define the reference interface for that distribution.

1.2 Requirements

Item	Recommendation
OS	Windows 10 / 11, Linux (x86_64)
Memory	16 GB or more (32 GB+ for large meshes)
GPU	CUDA-capable NVIDIA GPU with CUDA 12.8 or later (recommended)
Disk	1 GB for binaries, plus workspace for each case

Item	Recommendation
Post-processing	ParaView (recommended for result visualization)

Wake CFD supports CPU execution without a GPU. Cases execute unchanged on CPU, but CPU-specific optimization has not been performed; CPU execution is slower than GPU execution. CPU parallelization uses MPI.

> ⚠ Warning
>
> **Windows supports one GPU only**
> The MPI stack in the Windows build does not support GPU-aware MPI. Multi-GPU execution is therefore unavailable on Windows; it runs on a single GPU only.
> For multi-GPU execution, use Linux with GPU-aware MPI (a CUDA-enabled OpenMPI or MVAPICH2 build, for example). See Running Wake CFD for parallel-execution details.

1.3 Windows: Installer

1. Run `WakeCFD-<version>-win-x64.exe`.
2. Select an installation location.
3. When installation finishes, start **Wake CFD Desktop** from the Start menu or use the CLI in the installation directory.

The installer includes Desktop, the 2D/3D solvers, examples, and required runtime libraries. After installation, verify the setup with the ONERA M6 quickstart.

1.4 Linux: Compressed binary and `install.sh`

Extract the archive and execute from the working directory, or install under a user path with `install.sh`.

```
tar -xzf wake-<version>-linux-x64.tar.gz
cd wake-<version>-linux-x64

# User installation: default path ~/.local
./install.sh

# Install to another location
./install.sh --prefix /opt/wake
```

`install.sh` installs executables and shared resources to the selected location, then prints PATH setup instructions. Open a new shell after installation or apply the indicated PATH setting.

1.5 Verify installation

To confirm the installed version, execute a solver binary without arguments. The solver prints version information and exits.

Windows PowerShell

```
.\wake\bin\wake_2d_mixed.exe
```



Figure 1.1: Version output from the 2D solver on Windows.

Linux

```
./wake/bin/wake_2d_mixed
./wake/bin/wake_3d_mixed
```

Each binary is verified separately, because a distribution may ship only one of them. This check confirms that the executable can start. To verify that cases execute and produce results, proceed to the ONERA M6 quickstart.

> ⚠ Warning
>
> **Do not relocate installed files**
> Wake CFD executables depend on runtime libraries in the distribution directory structure. Do not move solver binaries or shared libraries to other locations after installation.

1.6 Distribution layout

Example directory structure after extraction:

```
wake/
├─ bin/
│  ├─ wake_3d_mixed[.exe]    # 3D solver
│  └─ wake_2d_mixed[.exe]    # 2D solver
├─ examples/           # Runnable examples and meshes
│  └─ onera-m6-wing/
├─ install.sh          # Linux user-installation script
└─ ...                 # Windows DLLs or Linux shared libraries
```

Linux executables do not use the `.exe` extension.

1.7 Running Wake CFD

Two execution approaches are available. Select the approach appropriate for the workflow.

1.7.1 Approach 1 — Call using a relative or absolute path from the working directory

Place the JSONC case file in a working directory and reference the solver binary using a relative or absolute path.

```
clew@t7960-rtx6000x4:/shared/wake-cfd$ ./wake_2d_mixed


   __        __    _          ____ _____ ____
   \ \      / /_ _| | _____  / ___|  ___|  _ \
    \ \ /\ / / _` | |/ / _ \| |   | |_  | | | |
     \ V  V / (_| |   <  __/| |___|  _| | |_| |
      \_/\_/ \__,_|_|\_\___| \____|_|   |____/

Wake CFD Solver v0.23.0

Usage: wake_cfd [options] <config.json>

Options:
  <config.json>            Case configuration file (.json or .jsonc)
  --overlay <file.json>    Overlay configuration  (RFC 7396 JSON Merge Patch)
  --patch   <file.json>    JSON Patch document    (RFC 6902 - add/remove/replace/...)
  --grpc <port|auto>       Start gRPC runtime control server
  --export-schema [dir]    Export JSON schemas and exit (default: docs/input)
  --validate               Validate input deck and exit (no files/GPU/solver run)
  --format json            With --validate, emit one JSON result to stdout
  --capabilities           Print this binary's profile (dimension, precision,
                           device, features) as one JSON document and exit
  --iter N                 Override run_control.iteration (alias: --max-iter)
  --restart <dir|.wsol>    Continue a previous run: directory -> its latest
                           restart_flow_*.wsol, file -> that file. Injects
                           initialization.restart + run_control.resume=true
  --device N               Pin to CUDA device N (single-rank only; MPI uses
                           automatic rank->device binding). Indexes the
                           CUDA_VISIBLE_DEVICES subset when that env is set.
  --license <file|port@host>  Use this license instead of the usual lookup
                           (WAKECFD_LICENSE, wakecfd_license.ini, licenses/; see wake_lic where)
  -h, --help               Show this help and exit

Apply order: base + `$extends` chain -> --overlay (RFC 7396) -> --patch (RFC 6902).

Examples:
  wake_cfd naca0012.json
  wake_cfd naca0012.json --overlay fine_mesh.json
  wake_cfd naca0012.json --patch sweep_aoa.json   # RFC 6902 - parametric study
  wake_cfd naca0012.json --validate
  wake_cfd naca0012.json --grpc auto
  wake_cfd naca0012.json --device 1      # pin to GPU 1 (other jobs use GPU 0)
  mpirun -np 4 wake_cfd rotor.json
```

(a) 2D solver, `wake_2d_mixed`

```
clew@t7960-rtx6000x4:/shared/wake-cfd$ ./wake_3d_mixed


   __        __    _          ____ _____ ____
   \ \      / /_ _| | _____  / ___|  ___|  _ \
    \ \ /\ / / _` | |/ / _ \| |   | |_  | | | |
     \ V  V / (_| |   <  __/| |___|  _| | |_| |
      \_/\_/ \__,_|_|\_\___| \____|_|   |____/

Wake CFD Solver v0.23.0

Usage: wake_cfd [options] <config.json>

Options:
  <config.json>            Case configuration file (.json or .jsonc)
  --overlay <file.json>    Overlay configuration  (RFC 7396 JSON Merge Patch)
  --patch   <file.json>    JSON Patch document    (RFC 6902 - add/remove/replace/...)
  --grpc <port|auto>       Start gRPC runtime control server
  --export-schema [dir]    Export JSON schemas and exit (default: docs/input)
  --validate               Validate input deck and exit (no files/GPU/solver run)
  --format json            With --validate, emit one JSON result to stdout
  --capabilities           Print this binary's profile (dimension, precision,
                           device, features) as one JSON document and exit
  --iter N                 Override run_control.iteration (alias: --max-iter)
  --restart <dir|.wsol>    Continue a previous run: directory -> its latest
```

```
# Example working-directory layout
my_case/
└─ case.jsonc

# Run from the working directory
../../wake/bin/wake_3d_mixed ./case.jsonc
```

Or use an absolute path:

```
/path/to/wake/bin/wake_3d_mixed ./case.jsonc
```

> Tip
>
> **Recommended — Separate directory per case**
> Keeping each case in a separate directory isolates its output files (`*.csv`, `*.out`, `*.log`, restart files(`*.wsol`), volume dumps, and more).

1.7.2 Approach 2 — Add `bin` to PATH

For frequent use, add the `bin` directory to PATH to execute `wake_3d_mixed` from any directory. On Linux, `install.sh` prints PATH configuration instructions during installation.

Windows (PowerShell, persistent):

```
$env:Path += ";C:\path\to\wake\bin"
[Environment]::SetEnvironmentVariable("Path", $env:Path, "User")
```

Windows (cmd, current session only):

```
set PATH=%PATH%;C:\path\to\wake\bin
```

Linux / macOS (`~/.bashrc` or `~/.zshrc`):

```
export PATH="$PATH:/path/to/wake/bin"
```

After adding it:

```
cd my_case
wake_3d_mixed ./case.jsonc
```

1.8 Migrating existing YAML cases

> ⚠ Warning
>
> **YAML input is rejected by the solver before execution (d0125 Phase A, 2026-05).**
> Use the packaged `wake-migrate` utility to convert files in bulk, then use `.jsonc` or `.json`.

```
wake-migrate case.yaml -o case.jsonc
```

The conversion result is JSONC with comments preserved. After conversion, run `.jsonc` or `.json` files. For format and inheritance details, see Configuration files.

1.9 Related guides

- First run: ONERA M6 quickstart
- Create a case: start with the minimal example on Configuration files
- Command-line options: Running Wake CFD
- Complete key reference: CLI configuration-file reference

2 Quickstart: ONERA M6 wing

Run a 3D transonic RANS analysis with wall functions using a packaged example

This quickstart executes the packaged ONERA M6 half-wing verification case and reviews its 3D transonic-flow results. The example includes geometry and input files; no mesh generation or case authoring is required.

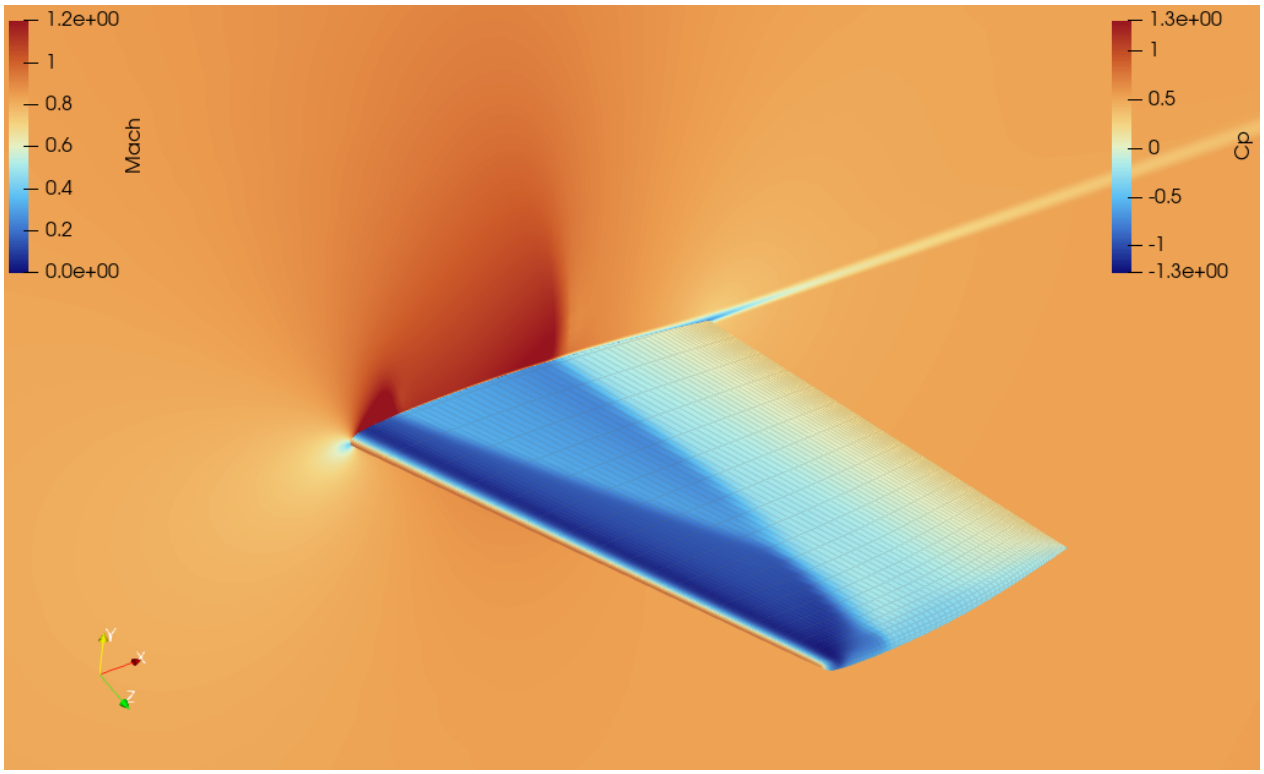


Figure 2.1: ONERA M6 wing surface pressure coefficient and farfield Mach number, showing the wing's characteristic lambda shock pattern

2.1 What this example demonstrates

Item	Value
Geometry	NASA GRC ONERA M6 half wing
Dimension	3D
Flow	Steady compressible RANS, Spalart-Allmaras turbulence model
Freestream	Mach 0.8395, angle of attack 3.06°
Wall treatment	**Wall function** on the wing surface (target y+ ≈ 30) — no boundary-layer mesh resolution required
Boundaries	wing no-slip wall (wall function), farfield, symmetry
Convective scheme	Roe (default); the example is also validated with AUSM+up2, HLLC, and SLAU2 — see Figure 2.4
Results to check	Residuals, force coefficients, Mach, pressure, `Cp`, `Cf`, and y+ output

> **Note**
>
> **Why a wall function here.** This case uses a wall function rather than a fully wall-resolved mesh (y+ ≈ 1). Wall functions let a coarser near-wall mesh reproduce the boundary-layer velocity profile at a fraction of the cell count and run time, which keeps this quickstart fast to reproduce. For cases that need wall-resolved accuracy (separation-sensitive flows, detailed skin-friction studies), see Recommended settings and troubleshooting.

2.2 1. Copy the example to a working directory

Copy the complete `examples/onera-m6-wing` directory from the distribution to a working location to avoid filling the installation directory with output files. Keep JSONC files, the SU2 mesh, and any inheritance files together.

```
my-wake-work/
└─ onera-m6-wing/
   ├─ case.jsonc
   ├─ m6wing.su2
   └─ (required auxiliary JSONC files)
```

2.3 2. Run the case

Run the 3D solver from the example directory.

```
# Linux
/path/to/wake/bin/wake_3d_mixed case.jsonc

# Windows PowerShell
& "C:\Program Files\Wake CFD\bin\wake_3d_mixed.exe" case.jsonc
```

> **Important**
>
> This example is a 3D case. Always use `wake_3d_mixed`.

2.4 3. Check the run

Review residuals and the wing-force monitor in the console.

- The calculation starts without input or mesh errors.
- Residuals decrease over iterations.
- Force coefficients stabilize toward the end of the run.

Use the history CSV to plot residuals and force coefficients. The filename is defined in the `monitoring` block of `case.jsonc`.

2.5 4. View the results

When the calculation finishes, open the volume or wing-surface results in ParaView.

1. Select `Mach` or `Pressure` as the color variable.
2. Set the view to show the wing upper surface.
3. Inspect abrupt changes in Mach and pressure around the shock wave — you should see the lambda-shaped shock pattern in Figure 2.1.
4. Select `Cp` in the surface results to view the pressure distribution along spanwise sections.
5. Select `Cf` and y+ to check the wall-function boundary layer (Figure 2.2, Figure 2.3). y+ should sit close to the target value (~30) across most of the wing, dipping locally only near the shock foot.

The conditions and result-review workflow for this example are based on NASA GRC ONERA M6 analysis assets. For output and monitor details, see File output and Monitoring.

> **Note**
>
> **Verification criterion**
> The installation check is complete when the solver finishes without an execution error and the Mach, pressure, or Cp field shows a flow pattern similar to the reference images

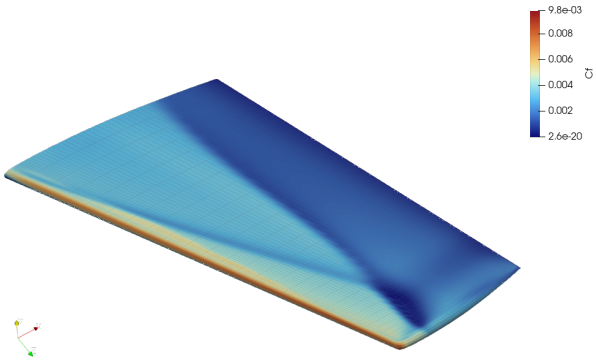


Figure 2.2: Skin-friction coefficient. The shock-foot band (where Cf drops locally) lines up with the shock pattern in Figure 2.1.

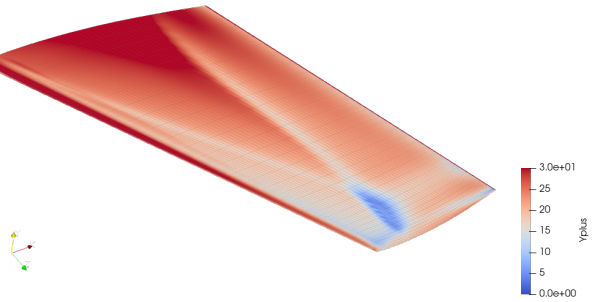


Figure 2.3: Wall y+. Mostly 25-30 as targeted by the wall function, with a local dip at the shock foot.

above. This first run verifies that the packaged solver and example execute correctly. For a quantitative check, Figure 2.4 compares sectional Cp from four convective schemes (Roe, AUSM+up2, HLLC, SLAU2) against the NASA GRC WIND-US published CFD solution and the original wind-tunnel experiment, at seven spanwise stations. The four schemes agree with each other and with both references across the span.

2.6 Related guides

- Input block meanings: Input-file structure
- Convergence and configuration changes: Recommended settings and troubleshooting
- Execution errors: CLI troubleshooting

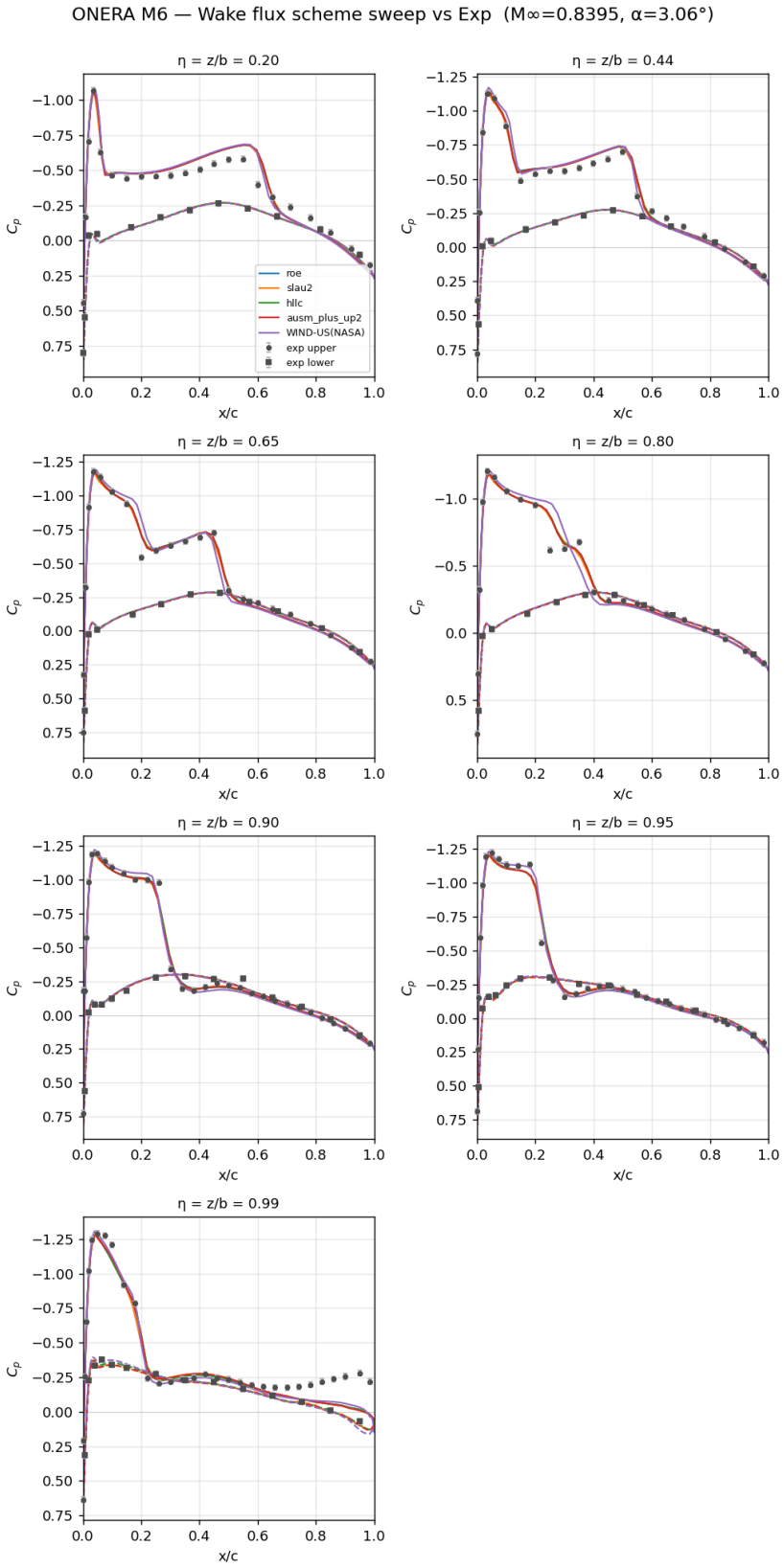


Figure 2.4: Sectional Cp at seven spanwise stations: Roe, AUSM+up2, HLLC, and SLAU2 versus the NASA WIND-US published solution and experiment

Part II

2. How to use Wake CFD

Wake CFD provides a desktop GUI, command-line interface (CLI), and Python API for case authoring and execution. All interfaces share the same JSONC input model, so a case prepared in one interface can be reused in another execution environment.

Each interface is designed for different stages and purposes of an analysis workflow rather than being interchangeable tools.

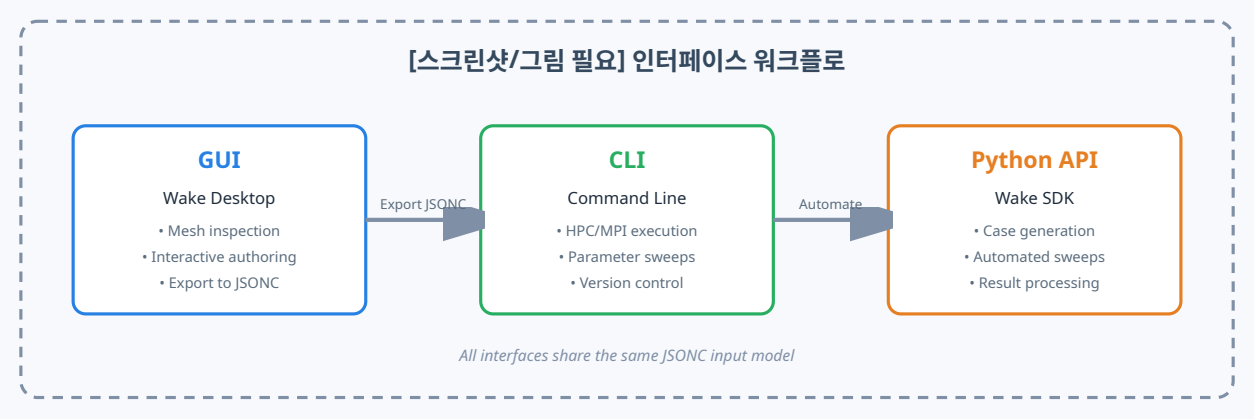


Figure 2.5: Placeholder for the user-interface workflow. Before public release, replace with a diagram showing GUI → CLI → Python API paths and their typical use cases.

Choosing an interface

> **Tip**
>
> **First-time users**
> If you have not yet run Wake CFD, complete the packaged example in Getting started to verify that Wake CFD executes correctly.

Interface	Best suited for	Notes
GUI	Mesh-marker inspection, interactive case authoring, and setting exploration	Visual case setup is available. Post-processing visualization is under development; currently use external tools such as ParaView.
CLI	Reproducible runs, parameter sweeps, HPC/MPI, and version control	You manage JSONC files, execution commands, and file paths directly. Post-process results in ParaView.

Interface	Best suited for	Notes
Python API	Case generation, repeated runs, parameter sweeps, and result-processing automation	Requires a Python environment and Wake CFD SDK setup.

Common workflows

Interactive case authoring Use Wake CFD Desktop to inspect mesh zones and boundary markers, then configure physics models, boundary conditions, numerical methods, and outputs. Run locally or export the case to JSONC.

GUI to HPC transition Author or review a case in Wake CFD Desktop, then export it to JSONC. Run the same case on a remote GPU server, MPI environment, or Slurm cluster using the CLI.

Repeated analysis and parameter sweeps Start from a packaged example or existing JSONC case. Use CLI overlays, patches, or the Python API to vary Mach number, angle of attack, mesh, and numerical settings across repeated runs.

External-system integration Submit cluster jobs with Slurm, monitor or control running solvers through gRPC, or use an MCP-based agent to author cases and attach to the live Desktop project. The agent and GUI share the same project model, so automated edits can be reviewed in the open interface. See Automation and integration.

Settings shared across all interfaces

The following items use the same case model in the GUI, CLI, and automation environments:

- Mesh files and zone/boundary markers
- Analysis method and physical models
- Materials and reference values
- Domains and boundary conditions
- Domain interfaces
- Numerical methods and run control
- Initial conditions
- Monitoring and file outputs

For the physical meaning and JSONC keys of each setting, see Input and configuration reference. The GUI edits these settings visually; the CLI and automation interfaces use the same content in JSONC.

Next steps

- First-time Wake CFD execution: Getting started
- Authoring cases with the GUI: GUI guide
- Running cases with the CLI: CLI guide
- Automating repeated work with Python: Python API
- Integrating Slurm, gRPC, or agents: Automation and integration
- Learning through worked examples: Tutorials

GUI guide

Wake CFD Desktop provides interactive case authoring and mesh inspection. The current Desktop targets the following workflows:

- Interactive mesh inspection and zone/marker identification
- Visual case assembly with boundary condition setup
- Project-based organization with multiple cases
- Local solver execution and convergence monitoring

The GUI and CLI share the same JSONC input model. Export projects to JSONC for CLI execution or HPC operation.

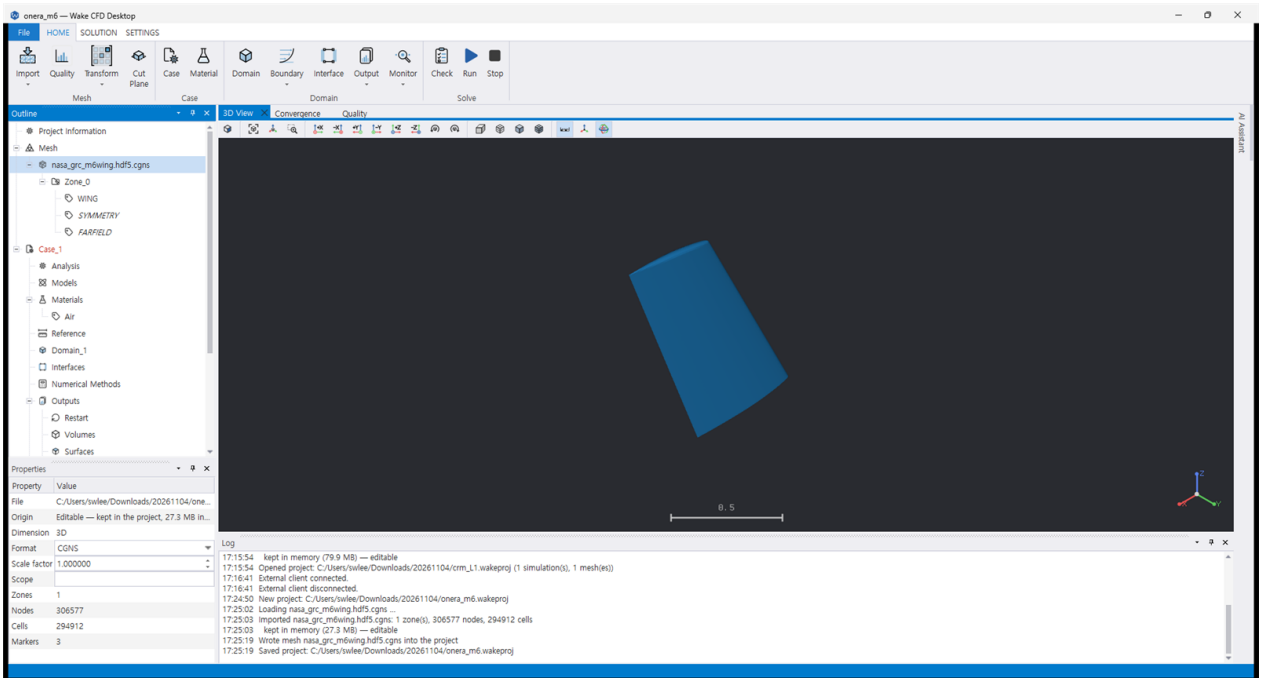


Figure 2.6: Wake CFD Desktop with an ONERA M6 mesh loaded. The project outline and properties are on the left, the 3D viewport is in the center, and the run log is at the bottom.

GUI chapters

ONERA M6 GUI quickstart Import, configure, and run the packaged example through the Desktop interface.

Create and configure a project Project creation, case setup, and configuration workflow.

Mesh and boundary conditions Mesh import, zone inspection, and boundary condition assignment.

Run and monitor Solver execution and convergence monitoring.

Review and export results Result inspection and JSONC export.

Find a topic

Task	Start here
Install Wake CFD (Solver + Desktop)	Installation
Run the first example	ONERA M6 GUI quickstart
Import a mesh	Create and configure a project
Inspect mesh zones and markers	Mesh and boundary conditions
Create a new case	Create and configure a project
Configure boundary conditions	Mesh and boundary conditions
Set physical models and numerics	Create and configure a project
Run the solver locally	Run and monitor
Monitor convergence	Run and monitor
Export to JSONC	Review and export results

GUI scope

The current Wake CFD Desktop addresses:

- **Visual case authoring** — Import meshes, configure physics, set boundary conditions
- **Project management** — Organize cases, track configurations
- **Local execution** — Run solvers, monitor convergence

For batch runs, parameter sweeps, and HPC operation, export to JSONC and use the CLI.

Related guides

- First run: Getting started
- Complete configuration reference: Input and configuration reference
- CLI workflows: CLI guide
- Python automation: Python API

3 ONERA M6 GUI quickstart

Draft workflow pending end-to-end validation

This page is reserved for a quickstart that imports the ONERA M6 wing example into Wake CFD Desktop, configures and runs it, then reviews the results. The workspace below is current, but the complete step sequence still requires end-to-end GUI validation and additional run/result captures.

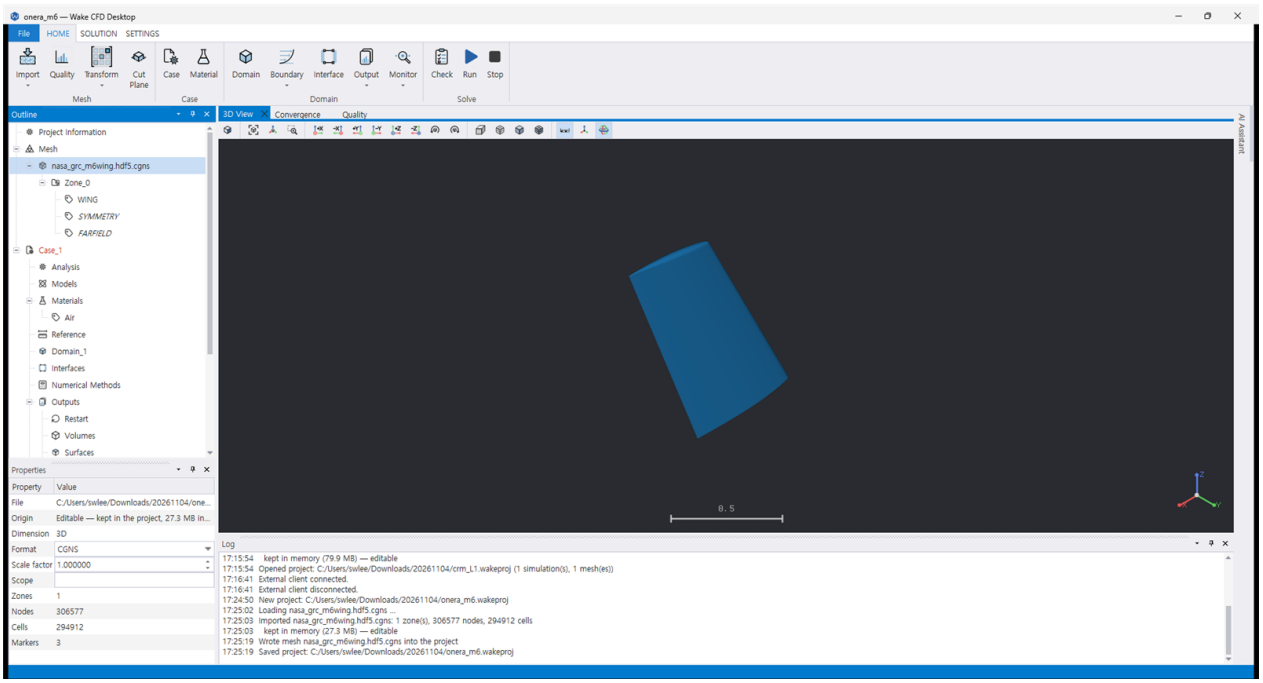


Figure 3.1: The ONERA M6 mesh during case authoring in Wake CFD Desktop. The project tree identifies the WING, SYMMETRY, and FARFIELD markers used in the workflow below; the latter two are hidden here, which is why only the wing appears in the viewport.

3.1 Planned workflow

1. Create a **New Project** and import the ONERA M6 mesh.
2. Create a 3D case and configure the Euler physical model and farfield condition.

3. Connect the `WING`, `FARFIELD`, and `SYMMETRY` markers to boundary conditions.
4. Configure Output and Monitoring, then select the 3D solver.
5. Run the case and review the history and ParaView results.

> **Note**
>
> **GUI validation pending**
> This sequence is a draft based on the current Desktop design and CLI example. Actual screens, commands, and save locations will be finalized after GUI end-to-end validation. For now, run the same example through the CLI quickstart.

4 Create and configure a project

Manage a shared mesh and one or more analysis cases

A project manages a mesh file once and places one or more analysis cases on it. Each case has independent physical, numerical, and output settings; on export, the project mesh information is connected to the case JSONC.

4.1 Create a new project

1. Select **File > New**.
2. Add a mesh with **Home > Import**.
3. Add an analysis case with **Home > Case**.
4. In the Project tree, select items under the case and configure `Analysis`, `Models`, `Materials`, `Domains`, `Numerical Method`, and `Run Control` in order.
5. Save to an empty project folder with **File > Save As**.

4.2 Save and export

- Saving a project stores project information and each case's settings in the working folder.
- **Solution > Export Deck** exports the selected case as a JSONC deck for CLI or HPC execution.
- If you edit JSONC manually, keep a separate copy from the original project before reopening it in the GUI.

> **Tip**
>
> In practice, save one baseline case, then create a new case for each condition to compare—such as Mach number, angle of attack, or mesh—to keep the change history separate.

Planned coverage: new projects, loading cases, saving settings, and compatibility with CLI configuration files.

5 Mesh and boundary conditions

Connect mesh markers to domains and boundary conditions

Desktop imports SU2, CGNS, and Gmsh meshes and displays their zones and markers in the Project tree and viewport. The imported names are connected to `mesh_zones` and `mesh_markers` in solver input.

5.1 Basic workflow

1. Import a mesh through **Home > Import**.
2. Expand Mesh in the Project tree to check zone and marker names and dimensions.
3. Create a Domain under the case and connect its zone.
4. Select the Domain, then add boundary-condition types through **Boundary**.
5. Connect mesh markers to each boundary condition and enter required physical values.

5.2 Required checks

- Confirm that wall, farfield, inlet, and outlet markers are neither omitted nor connected more than once.
- Connect 2D meshes to 2D cases and the 2D solver, and 3D meshes to 3D cases.
- Required values differ by boundary condition, such as `wall`, `farfield`, `pressure_inlet`, and `pressure_outlet`.
- Marker names must match their original names in the mesh file.

For shared setting definitions, first see Mesh and Domains and boundary conditions.

6 Run and monitor

Review local execution status and convergence history

Before running, the selected case must have a valid solver-binary path and input configuration. **Solution > Run** starts the solver with that case's settings, while **Stop** requests a safe termination.

6.1 Pre-run checks

- The dimension in `Analysis` matches the selected 2D or 3D solver.
- Mesh, domain, and boundary-marker connections are complete.
- `Run Control` defines an iteration or physical-time termination condition.
- `Monitoring` includes residuals, a history file, and monitors for quantities of interest.

6.2 Check convergence

For steady analyses, check that monitors of interest—such as force coefficients, mass flow, and pressure drop—stabilize as well as residuals. If a problem occurs, keep the run log and history CSV, then follow Troubleshooting.

> **Note**
>
> The current GUI execution controls focus on starting, checking status, and stopping. Use the CLI or API workflow for pause/resume and cluster submission.

For shared output and monitor concepts, see File output and Monitoring.

7 Review and export results

JSONC decks, CSV monitors, and ParaView result files

In the GUI, edit a case's `Outputs` and `Monitoring` to define files written by the solver. Configure `Volume`, `Surface`, `Slice`, and `Restart` output in the Project tree, and export the selected case's JSONC as a separate file with **Solution > Export Deck**.

7.1 Result-file roles

Result	Purpose	Recommended review tool
history CSV / residual CSV	Convergence and time history	Spreadsheet software, Python
volume / surface / slice	Spatial distributions and surface results	ParaView
restart	Resume after interruption	Wake CFD solver

The Desktop results post-processing screen is currently in preparation. Open volume and surface results in ParaView, and review CSV files in a separate plotting tool. File output is the source of truth for output formats, fields, and frequency.

For shared output formats, see File output.

CLI guide

The Wake CFD CLI executes cases from JSONC configuration files and supports batch runs, parameter sweeps, and HPC operation. Versioning JSONC files with execution commands ensures reproducible runs across workstations, GPU servers, and MPI clusters.

The CLI targets the following workflows:

- Parameter sweeps across Mach numbers, angles of attack, meshes, and numerical settings
- Remote GPU servers and MPI clusters with schedulers such as Slurm
- Unattended automation (CI, regression tests)
- Reproducible analysis records preserving input files, logs, and results

The GUI provides interactive case assembly and mesh inspection. Transition repeated calculations to the CLI for automation.

CLI chapters

Running Wake CFD Solver execution, parallel operation, configuration inheritance, and runtime overrides.

Environment variables Bootstrap configuration for logging, MPI, and runtime diagnostics.

CLI configuration-file reference Complete JSONC key reference for command-line workflows.

Troubleshooting Execution error diagnosis and resolution.

Find a topic

Task	Start here
Execute cases in parallel or on specific GPUs	Running Wake CFD
Override configuration without editing base files	Running Wake CFD
Reuse settings across multiple cases	Running Wake CFD
Adjust logging or MPI behavior	Environment variables
Set iteration limits or residual targets	Run control and convergence

Task	Start here
Look up JSONC key definitions	CLI configuration-file reference
Resolve execution errors	Troubleshooting

CLI scope

The CLI addresses two distinct aspects:

- **Execution** — Solver binaries, parallel operation, configuration inheritance, runtime overrides, and environment setup
- **Case definition** — Mesh, analysis mode, physical models, boundary conditions, numerical methods, and outputs

Case definitions are stored in JSON or JSONC files. The Input and configuration reference documents configuration settings; this chapter covers execution.

Related guides

- First run: Getting started
- JSONC structure and inheritance: Input-file structure
- Complete configuration reference: Input and configuration reference
- GUI workflows: GUI guide
- Python automation: Python API
- Cluster integration: Slurm integration

8 Running Wake CFD

Binaries, input-file composition, and parallel execution

8.1 Overview

Wake CFD executes cases from JSONC configuration files. Pass one input file to the solver binary; the solver reads mesh files, applies the configuration, and writes outputs to the working directory.

8.2 Basic execution

Wake CFD provides dimension-specific solver binaries. Use `wake_2d_mixed` for 2D cases and `wake_3d_mixed` for 3D cases. The binary dimension must match `analysis.dimension` in the input file.

Linux:

```
./wake_2d_mixed case.jsonc  # for analysis.dimension: 2
./wake_3d_mixed case.jsonc  # for analysis.dimension: 3
```

Windows PowerShell:

```
& "C:\Program Files\Wake CFD\bin\wake_2d_mixed.exe" case.jsonc
& "C:\Program Files\Wake CFD\bin\wake_3d_mixed.exe" case.jsonc
```

The executable also selects compile-time precision; it is not an input-deck key.

Suffix	State / residual / flux arithmetic	Linear system	Intended use
`_fp64`	Double precision	Double precision throughout	Numerical reference and CPU/GPU comparisons.
`_mixed`	Double precision	Preconditioner and the matrix copy it runs on in FP32; the outer Krylov iteration stays double	Production default.
`_fp32`	Float: the residual path and the solution and residual arrays alike	FP32	Speed-oriented runs with a higher residual floor.

Geometry coordinates, physical time, and the values written to output and restart files are double in every tier.

For example, `wake_3d_fp64` and `wake_3d_mixed` accept the same 3D deck. Available binaries depend on the installed build. Windows uses the same names with `.exe`. The removed `expert_parameters.krylov.precision` key cannot change executable precision. For DBNS FP32, nondimensional input is recommended; use monitored quantities to assess convergence when residuals reach their precision floor. The FP32 binary warns when `run_control.convergence.residual_absolute` is below `-6`, which is where the FP32 flux evaluation bottoms out. It also rejects any material that is not a calorically perfect gas — `eos.type: "peng_robinson"` and NASA-polynomial specific heat among them — because a real-gas law evaluated in float loses 1e-3 to 2e-2 relative accuracy. Run those cases with the mixed or FP64 binary.

Executing `wake_2d_mixed` with `analysis.dimension: 3` (or vice versa) triggers a dimension mismatch error.

8.3 Command-line options

Wake CFD CLI accepts options to override settings, validate input, and control execution behavior. The list below is the solver's own `--help` output.

Option	Description	Example
`--overlay <file>`	Apply JSON Merge Patch (RFC 7396)	`--overlay mach-085.json`
`--patch <file>`	Apply JSON Patch (RFC 6902)	`--patch changes.json`

Option	Description	Example
`--iter <N>`	Override iteration count	`--iter 500`
`--max-iter <N>`	Alias for `--iter`	`--max-iter 1000`
`--restart <dir|file>`	Continue a previous run; see below	`--restart out/`
`--device <N>`	Pin to GPU device N (single-rank only)	`--device 0`
`--license <file|port@host>`	Use this license instead of the usual lookup	`--license 27000@lic01`
`--validate`	Validate input and exit without execution	`--validate`
`--format json`	With `--validate`, emit one JSON result to stdout	`--validate --format json`
`--capabilities`	Print the binary's own profile as JSON and exit	`--capabilities`
`--export-schema [dir]`	Export JSON schemas and exit	`--export-schema docs/`
`--grpc <port|auto>`	Start gRPC runtime control server	`--grpc auto`
`-h, --help`	Show help and exit	`--help`

A CUDA build lists `--device`; a CPU build does not, so that a build never advertises a flag it would silently ignore. `--format json` is read only alongside `--validate`. `--export-schema` takes an optional directory and writes to `docs/input` when none is given. An unrecognized `--option` is a warning, not an error: the run continues without it, so check the log when a flag appears to have no effect.

Resuming a run: `--restart` takes either a directory, in which case the solver picks the latest `restart_flow_*.wsol` inside it, or one `.wsol` file. It injects `initialization.restart` and `run_control.resume: true` into the composed configuration, so no edit to the input file is needed.

Iteration override: `--iter` (or `--max-iter`) overrides `run_control.iteration` for one execution without modifying the input file. Use this to verify case setup before committing to the configured iteration count.

Linux:

```
./wake_3d_mixed case.jsonc --iter 100
```

Windows PowerShell:

```
& "C:\Program Files\Wake CFD\bin\wake_3d_mixed.exe" case.jsonc --iter 100
```

For environment-variable configuration (logging, MPI behavior), see Environment variables. For convergence criteria, CFL policy, and unsteady time stepping, see Run control and convergence.

8.4 Input-file composition

Wake CFD supports configuration inheritance and runtime overrides, allowing cases to share common settings and apply targeted changes without duplicating entire input files.

8.4.1 Starter template

Wake CFD provides a canonical JSONC template at `user-interfaces/cli/wake_template.jsonc`. This template includes all major configuration blocks with sensible defaults for a 3D compressible Euler case.

Template structure:

```
{
  "mesh": [{ "file_name": "mesh.cgns", "format": "cgns" }],
  "analysis": { "time": "steady", "solver": "dbns", "dimension": 3 },
  "models": { "type": "euler", "regime": "compressible" },
  "materials": [{ "name": "Air", "type": "pure", "eos": {...} }],
  "reference": { "pressure": 101325.0, "temperature": 273.15, ... },
  "domains": [{ "name": "fluid", "boundaries": [...] }],
  "numerical_method": { "dbns": {...}, "linear_solver": {...} },
  "initialization": { "type": "auto" },
  "run_control": { "iteration": 1000, "convergence": {...} },
  "outputs": { "restart": {...}, "volumes": [...] },
  "monitoring": { "history_files": [...], "console": {...} }
}
```

The outline above omits four optional top-level blocks the solver also accepts: `interfaces`, `expressions`, `expert_parameters`, and `wake_version` (informational, logged only). Unknown keys are rejected with the offending JSON path, so a typo stops the run instead of defaulting silently.

Start from this template and modify only the settings relevant to your case. For complete field definitions, see CLI configuration-file reference.

8.4.2 $extends (inheritance)

Concept: Inherit configuration from a base file and write only the differences. This avoids duplicating unchanged settings across parameter sweeps or configuration variants.

Scenario: Run the same wing geometry with different Mach numbers.

Step 1: Create base configuration (`case.jsonc`):

```
{
  "mesh": [{ "file_name": "wing.cgns", "format": "cgns" }],
  "analysis": { "time": "steady", "solver": "dbns", "dimension": 3 },
  "models": { "type": "euler" },
  "materials": [{ "name": "Air", "eos": { "type": "ideal_gas" } }],
  "domains": [{
    "name": "fluid",
    "material": "Air",
    "boundaries": [
      { "name": "wall", "type": "wall", "mesh_markers": ["WING"] },
      { "name": "farfield", "type": "farfield", "mesh_markers": ["FARFIELD"],
        "mach": 0.7, "pressure": 101325.0, "temperature": 273.15,
        "aoa": 2.0 }
    ]
  }],
  "numerical_method": { "dbns": { "convective": { "scheme": "roe" } } },
  "run_control": { "iteration": 3000 }
}
```

Step 2: Create variant with `$extends` (`mach-08.jsonc`):

```
{
  "$extends": "case.jsonc",
  "domains": [{
    "name": "fluid",
    "material": "Air",
    "boundaries": [
      { "name": "wall", "type": "wall", "mesh_markers": ["WING"] },
      { "name": "farfield", "type": "farfield", "mesh_markers": ["FARFIELD"],
        "mach": 0.8, "pressure": 101325.0, "temperature": 273.15,
        "aoa": 2.0 }
    ]
  }]
}
```

Step 3: Execute the variant:

Linux:

```
./wake_3d_mixed mach-08.jsonc
```

Windows PowerShell:

```
& "C:\Program Files\Wake CFD\bin\wake_3d_mixed.exe" mach-08.jsonc
```

A far field in this form carries four required inputs — `mach`, `pressure`, `temperature`, and a direction, given either as `aoa` (with optional `aos`) or as the `flow_direction` unit vector. An incomplete boundary is rejected before the run starts rather than defaulting silently.

The solver merges `mach-08.jsonc` onto `case.jsonc` before execution. Because arrays are replaced, the variant repeats the complete domain and boundary array while changing Mach. Settings outside that array inherit from the base. Use `--patch` below to change a single array element without repeating it.

Inheritance chain: `$extends` can reference another file that also uses `$extends`, creating a multi-level inheritance hierarchy. The solver resolves the chain from base to leaf, applying each layer in order.

Relative paths: Paths in `$extends` are relative to the file containing the `$extends` directive, not the working directory.

8.4.3 `--overlay` (Merge Patch, RFC 7396)

Concept: Apply a JSON Merge Patch (RFC 7396) to override settings without editing the input file. Objects are deep-merged; arrays are replaced entirely.

RFC 7396: IETF standard for partial JSON modification. Objects merge recursively; arrays replace completely; `null` deletes a key.

Scenario: Override Mach number and iteration count for a quick smoke test without modifying `case.jsonc`.

Overlay file (`mach-085.json`):

```
{
  "domains": [{
    "name": "fluid",
    "material": "Air",
    "boundaries": [
      { "name": "wall", "type": "wall", "mesh_markers": ["WING"] },
      { "name": "farfield", "type": "farfield", "mesh_markers": ["FARFIELD"],
        "mach": 0.85, "pressure": 101325.0, "temperature": 273.15,
        "aoa": 2.0 }
    ]
  }],
```

```
  "run_control": {
    "iteration": 500
  }
}
```

Execution:

Linux:

```
./wake_3d_mixed case.jsonc --overlay mach-085.json
```

Windows PowerShell:

```
& "C:\Program Files\Wake CFD\bin\wake_3d_mixed.exe" case.jsonc --overlay mach-085.j
```

The overlay deep-merges into the loaded configuration. In this example: - The farfield boundary's `mach` changes to 0.85 - `run_control.iteration` changes to 500 - All other settings remain unchanged

Array replacement: Merge Patch replaces entire arrays. If `case.jsonc` defines three boundaries and the overlay defines one boundary array with one item, the result contains only that one boundary. To modify individual array items without replacing the entire array, use `--patch` instead.

Use cases: - Parameter sweeps (vary Mach, angle of attack, CFL) - Quick convergence tests (reduce iteration count) - Execution-time configuration (change output frequency, disable restarts)

8.4.4 `--patch` (JSON Patch, RFC 6902)

Concept: Apply a JSON Patch (RFC 6902) to modify specific fields using path-based operations. This allows precise edits to array items and nested objects without replacing entire structures.

RFC 6902: IETF standard for expressing operations on JSON documents. Supports add, remove, replace, move, copy, and test.

Supported operations:

Operation	Description	Use case
`add`	Insert or append an item at a path	Add a boundary condition, append a material
`remove`	Delete the item at a path	Remove a domain, delete an output

Operation	Description	Use case
`replace`	Change the value at a path	Update Mach number, change iteration count
`move`	Relocate an item from one path to another	Reorder boundaries
`copy`	Duplicate an item to another path	Clone a material definition
`test`	Assert that a path has a specific value	Validate dimension before applying patch

Scenario: Change the Mach number of the second boundary in the first domain and update the iteration count.

Patch file (`changes.json`):

```
[
  { "op": "replace", "path": "/domains/0/boundaries/1/mach", "value": 0.85 },
  { "op": "replace", "path": "/run_control/iteration", "value": 1000 }
]
```

Execution:

Linux:

```
./wake_3d_mixed case.jsonc --patch changes.json
```

Windows PowerShell:

```
& "C:\Program Files\Wake CFD\bin\wake_3d_mixed.exe" case.jsonc --patch changes.json
```

Path syntax: JSON Pointer (RFC 6901) uses `/` as the separator. Array indices are zero-based. The path `/domains/0/boundaries/1/mach` refers to the `mach` field of the second boundary in the first domain.

Test operation: The `test` operation validates a value before applying subsequent operations. If the test fails, the entire patch is rejected.

```
[
  { "op": "test", "path": "/analysis/dimension", "value": 3 },
  { "op": "replace", "path": "/run_control/iteration", "value": 5000 }
]
```

This patch updates the iteration count only if `analysis.dimension` is 3. Use `test` to guard patches against applying to the wrong case type.

8.4.5 Composition order

The solver applies configuration layers in the following order:

```
1. Base file (the file passed to the solver)
   ↓
2. $extends chain (resolved recursively from base to leaf)
   ↓
3. --overlay (JSON Merge Patch)
   ↓
4. --patch (JSON Patch)
```

Each layer merges onto or modifies the result of the previous layer. The final composed configuration is validated before execution begins.

Example:

Linux:

```
./wake_3d_mixed case.jsonc --overlay mach-08.json --patch cfl-adjust.json
```

Windows PowerShell:

```
& "C:\Program Files\Wake CFD\bin\wake_3d_mixed.exe" case.jsonc --overlay mach-08.js
```

1. Load `case.jsonc`
2. Resolve `$extends` if present in `case.jsonc`
3. Apply `mach-08.json` via Merge Patch
4. Apply `cfl-adjust.json` via JSON Patch
5. Validate final configuration
6. Execute

8.5 MPI execution

Linux:

```
mpirun -n 4 ./wake_3d_mixed case.jsonc
mpiexec -n 4 ./wake_3d_mixed case.jsonc
```

Windows PowerShell:

```
mpirun -n 4 wake_3d_mixed.exe case.jsonc
mpiexec -n 4 wake_3d_mixed.exe case.jsonc
```

In a GPU-aware MPI environment, ranks and GPUs are generally mapped one to one.

Linux:

```
mpirun -n 4 --bind-to core --map-by ppr:1:numa ./wake_3d_mixed case.jsonc
```

The solver detects GPU-aware MPI automatically. If detection is incorrect for your stack, force it with `WAKE_GPU_AWARE_MPI`.

8.5.1 Slurm (Linux HPC clusters)

Slurm is a Linux-based HPC job scheduler. Under Slurm, use `srun` according to your cluster's policy.

Linux:

```
srun ./wake_3d_mixed case.jsonc
```

Slurm configurations vary by site. Adjust partition, node count, tasks per node, and GPU allocation in the job script. For complete Slurm integration, batch scripts, and working-directory considerations, see Slurm integration.

8.6 Pre-run verification

Verify the following before executing a case to avoid common setup errors. These checks are derived from the Best practices pre-run checklist.

- **Input file syntax**: Confirm that the input file uses valid JSONC syntax (`//` and `/* */` comments allowed; JSON5 trailing commas and unquoted keys rejected). Use `--validate` to check.
- **Dimension match**: Confirm that `analysis.dimension` matches the solver binary (`wake_2d_mixed` → 2, `wake_3d_mixed` → 3).
- **Mesh file paths**: Confirm that `mesh[].file_name` paths exist relative to the working directory.
- **Mesh zone and marker names**: Confirm that `domains[].mesh_zones` and `boundaries[].mesh_mar` match actual names in the mesh file.

- **FR time mode**: For `analysis.solver=fr`, confirm that `analysis.time=unsteady`. Steady FR is not supported.

For execution errors and resolution procedures, see Troubleshooting.

8.7 Related guides

- Environment variables — Logging, MPI configuration, runtime diagnostics
- Run control and convergence — Iteration limits, convergence criteria, and CFL policy
- CLI configuration-file reference — Complete JSONC key reference
- Troubleshooting — Execution error diagnosis and resolution
- Slurm integration — Cluster batch jobs and parameter sweeps (Linux)
- Best practices — Pre-run checklist and recommended settings
- Input and configuration reference — Configuration settings for all interfaces

9 Controlling a running job

Graceful stop, scheduled snapshots, and gRPC runtime control

A running solver accepts intervention through three channels. All of them end in the same graceful path: the current iteration is finished and the final output and restart file are written before the process exits — nothing is lost.

Channel	Intent	How
Signals	stop **now**	`Ctrl+C`, or `kill <pid>` / `kill -INT <pid>` from another shell
`wake.control.json`	stop or dump **at a chosen iteration**	drop a file into the run directory (below)
gRPC	remote low-latency control/monitoring	start the run with `--grpc`; used by the GUI, daemon, and SDK

9.1 Stopping with a signal

`Ctrl+C` (or `kill`/`kill -TERM`/`kill -INT` with the solver pid) requests a graceful stop; a **second** signal forces immediate exit. This works even when you forgot `--grpc`.

For MPI jobs, send the signal to **a rank process pid** (any rank works — the ranks agree on the stop collectively). Signalling the `mpirun` process itself also works but Open MPI escalates to a hard kill after a short timeout, so a heavy case may not finish writing its final output in time.

9.2 `wake.control.json` — scheduled stop and snapshots

For “stop at iteration N” or “give me a snapshot at iteration N without stopping”, write a `wake.control.json` file into the run output directory (the one that already contains `wake.status.json`):

```
// <case>_001/wake.control.json — comments are allowed
{
  "stop_at": 6000,                    // stop gracefully at iteration 6000
  "dump_solution_at": [3000, 4000],   // force a volume/surface output
  "dump_restart_at": 2500             // force a restart (.wsol) snapshot
                                      // (single value or array both work)
}
```

Rules:

- **Scheduled only.** There is intentionally no "stop immediately" key — GPU runs advance too fast for a file drop to mean a well-defined "now". Use a signal for that.
- **Past iterations are ignored**, not fired late. If the run is already at iteration 5200 when it reads `"stop_at": 5000`, the entry is dropped (and reported), so you never get a surprise stop.
- The file is consumed (deleted) as soon as it is read, and the solver answers with **`wake.control.ack.json`** in the same directory listing what was accepted, what was ignored, and any parse error. Check the ack file to confirm your command landed.
- Dropping a new file **replaces** any pending schedule from a previous one.
- A malformed file never disturbs the run — it is rejected with the reason in the ack file.

A stopped run pairs naturally with `run_control.resume`: restart from the `.wsol` it wrote with `run_control.resume: true` and the iteration counter, convergence-reduction origin, and adaptive-CFL state all continue where they left off.

9.3 gRPC

Starting a run with `--grpc <port|auto>` opens a low-latency control and monitoring channel used by the GUI, the daemon, and the Python SDK's `run.runtime()` client — see Python SDK: Results and runtime control. A standalone protocol reference for direct gRPC integration is planned but not yet published here.

10 Environment variables

Logging and GPU selection for Wake CFD v0.7.0

10.1 Overview

Wake CFD executes with default settings without environment variable configuration. Configure environment variables only when log files become excessively large, when selecting specific GPUs on multi-GPU servers, or when reducing log file count in MPI runs.

Case configuration belongs in JSONC input files. Environment variables control only settings that must be determined before the solver reads the input file—logging, for example, because the JSONC parser itself uses the logger.

10.2 Setting environment variables

10.2.1 Command-line (one-time use)

Windows PowerShell:

```
$env:WAKE_LOG_LEVEL = "info"
.\wake_3d_mixed.exe case.jsonc
```

Linux Bash:

```
WAKE_LOG_LEVEL=info ./wake_3d_mixed case.jsonc
```

This sets the variable for one execution only.

10.2.2 ~/.wake/env file (persistent defaults)

Create a ~/.wake/env file in the home directory to avoid re-exporting environment variables in every run script. The solver reads this file on startup and applies it to the process environment.

File location: - **Linux/Mac**: ~/.wake/env - **Windows**: %USERPROFILE%\.wake\env (e.g., C:\Users\YourName\.wake\env)

Example ~/.wake/env:

```
# Reduce log verbosity
WAKE_LOG_LEVEL=info

# Only rank 0 logs in MPI runs
WAKE_LOG_SCOPE=master
```

File syntax: - Lines beginning with # and blank lines are ignored - Use KEY=VALUE format - Surrounding whitespace is trimmed

Precedence:

```
Built-in defaults < ~/.wake/env < Shell environment variables
```

The file does not override variables already exported in the shell. To override a file setting for one run, export the variable on the command line:

```
WAKE_LOG_LEVEL=trace ./wake_3d_mixed case.jsonc  # overrides ~/.wake/env
```

When the file does not exist, the solver proceeds with built-in defaults.

10.3 Logging

10.3.1 WAKE_LOG_LEVEL

Controls the minimum severity recorded in .log files.

Value	Meaning	Application
trace	Most detailed	Debugging solver behavior
debug	Development diagnostics	Investigating unexpected results

Value	Meaning	Application
`info`	Execution flow	Reduce log size (recommended)
`warn`	Warnings and above	Minimal logging
`error`	Errors only	Production runs

Default (v0.7.0 technical preview): Current builds use debug when `WAKE_LOG_LEVEL` is unset. The default will be reduced as releases stabilize.

Scenario: Reduce log file size

Log files growing to several GB indicate excessive detail. Set `WAKE_LOG_LEVEL=info` to reduce verbosity.

Windows PowerShell:

```
$env:WAKE_LOG_LEVEL = "info"
.\wake_3d_mixed.exe case.jsonc
```

Linux Bash:

```
WAKE_LOG_LEVEL=info ./wake_3d_mixed case.jsonc
```

Or add to `~/.wake/env`:

```
WAKE_LOG_LEVEL=info
```

Note: Console convergence history is independent of this variable. Console output is controlled by `monitoring.console` in the JSONC input file.

10.3.2 WAKE_LOG_SCOPE

Reduces the number of log files in MPI runs.

Value	Behavior	Application
Unset	Every rank creates its own `.rank<N>.log` file	Investigating rank-specific problems
`master`	Only rank 0 creates a log file; other ranks do not	Normal MPI execution (recommended)

Scenario: Reduce log file count in MPI runs

Executing `mpirun -n 100` creates 100 separate log files by default. Set `WAKE_LOG_SCOPE=master` to create only one log file from rank 0.

Windows PowerShell:

```
$env:WAKE_LOG_SCOPE = "master"
mpirun -n 4 wake_3d_mixed.exe case.jsonc
```

Linux Bash:

```
WAKE_LOG_SCOPE=master mpirun -n 4 ./wake_3d_mixed case.jsonc
```

This reduces file count and I/O overhead in high-rank runs. Disable only when diagnosing a problem specific to one rank.

10.4 GPU selection

10.4.1 CUDA_VISIBLE_DEVICES

This standard CUDA environment variable restricts which GPUs Wake CFD can access. Use this on multi-GPU servers to reserve specific GPUs or avoid GPUs in use by other jobs.

Scenario: Select specific GPUs

On a server with 4 GPUs (0, 1, 2, 3), restrict Wake CFD to GPUs 2 and 3 by setting `CUDA_VISIBLE_DEVICES=2,3`.

Windows PowerShell:

```
$env:CUDA_VISIBLE_DEVICES = "2,3"
mpirun -n 2 wake_3d_mixed.exe case.jsonc
```

Linux Bash:

```
CUDA_VISIBLE_DEVICES=2,3 mpirun -n 2 ./wake_3d_mixed case.jsonc
```

Single GPU:

```
CUDA_VISIBLE_DEVICES=1 ./wake_3d_mixed case.jsonc  # Use only GPU 1
```

At startup, the solver logs the GPU list. When `CUDA_VISIBLE_DEVICES` is set, the log indicates that the GPU list was filtered.

10.5 Summary

Variable	Values	Application
`WAKE_LOG_LEVEL`	`trace`, `debug`, `info`, `warn`, `error`	Log files excessively large
`WAKE_LOG_SCOPE`	`master`	MPI runs create excessive log files
`CUDA_VISIBLE_DEVICES`	GPU index list (e.g., `0,2,3`)	Select specific GPUs on multi-GPU servers

Typical persistent defaults:

```
# ~/.wake/env
WAKE_LOG_LEVEL=info
WAKE_LOG_SCOPE=master
```

One-time override:

```
# Windows PowerShell
$env:WAKE_LOG_LEVEL = "trace"
.\wake_3d_mixed.exe case.jsonc

# Linux Bash
WAKE_LOG_LEVEL=trace ./wake_3d_mixed case.jsonc
```

10.6 Related guides

- Running Wake CFD — Execution and MPI
- Troubleshooting — Execution error diagnosis
- Monitoring — Console output and convergence history

11 CLI configuration-file reference

13 top-level input blocks and composition rules

11.1 Overview

Wake CFD CLI input files contain 13 top-level JSONC blocks. Each block maps to a specific configuration domain. This page provides a quick reference to each block's role and links to detailed documentation. For complete type definitions, enums, and defaults, consult the JSON schema or section-specific examples.

11.2 Input blocks

Block	Role	Details
`mesh`	One or more mesh files	Mesh
`analysis`	Solver, time, and dimension	Analysis
`models`	Euler/NS, compressibility, and turbulence	Physical models
`materials`	EOS, viscosity, thermal conductivity, and material properties	Materials
`reference`	Force, moment, and coefficient-normalization reference values	Reference values
`domains`	Domains and tagged-union boundary conditions	Domains and boundary conditions
`interfaces`	Connections between domains	Interfaces
`numerical_method`	Solver-specific numerical methods, CFL, and linear solvers	Numerical methods
`initialization`	Uniform, expression, restart, and FR presets	Initialization

Block	Role	Details
`run_control`	Iteration, convergence, and unsteady time stepping	Run control and convergence
`outputs`	Restart, volume/surface/slice, and diagnostics	File output
`monitoring`	Console, residual CSV, and named monitors	Monitoring
`expert_parameters`	Internal tuning for GPU, Krylov, AMG, and more	Expert parameters

11.3 JSONC syntax

Arrays: Use array syntax for all list-valued keys. Both object arrays and string arrays must use brackets.

```
"mesh": [{ "file_name": "wing.cgns" }]
"mesh_zones": ["fluid"]
"mesh_markers": ["WALL", "FARFIELD"]
```

Comments: Line (`//`) and block (`/* */`) comments are permitted. Trailing commas are not valid JSONC and trigger parse errors.

Enums: Some fields accept case-insensitive aliases. Use canonical names in new files for clarity.

11.4 Configuration composition

Wake CFD applies configuration layers in the following order:

```
1. Base file (the file passed to the solver)
2. $extends chain (resolved recursively)
3. --overlay (JSON Merge Patch, RFC 7396)
4. --patch (JSON Patch, RFC 6902)
```

Example:

Linux:

```
./wake_3d_mixed case.jsonc --overlay mach-sweep.json --patch cfl-adjust.json
```

Windows PowerShell:

```
& "C:\Program Files\Wake CFD\bin\wake_3d_mixed.exe" case.jsonc --overlay mach-sweep
```

For composition mechanics and examples, see Running Wake CFD. For input-file inheritance patterns, see Input-file structure.

12 Troubleshooting

Input validation, mesh connections, convergence, and divergence diagnosis

12.1 Input file rejected

- Confirm that the input file uses JSONC format (`.jsonc` extension with valid syntax).
- Remove trailing commas and unquoted keys (JSON5 syntax is not supported).
- Write `mesh`, `materials`, and `domains` as arrays, not wrapper objects.
- Write `mesh_zones`, `mesh_markers`, and `fields` as string arrays.
- Set `analysis.solver` to `dbns`, `pbns`, or `fr`.
- Do not mix `numerical_method` subblocks that do not apply to the selected solver.

12.2 Mesh names not found

Mesh zone and marker names in the configuration must match names in the mesh file:

- `domains[].mesh_zones` — zone names
- `domains[].boundaries[].mesh_markers` — boundary marker names
- `interfaces[].side1_mesh_markers`, `side2_mesh_markers` — interface marker names

Mesh format naming:

- CGNS: zone and BC names
- SU2: zone and marker tags
- Gmsh: physical-group names

For multiple meshes using `mesh[].scope`, verify the actual scoped names (e.g., `scope/zone`).

12.3 Convergence failure

1. Start with a low CFL using `run_control.cfl.type=constant`.
2. Reduce the convective `spatial_order` to 1 and simplify the limiter.
3. Check mesh quality and the physical consistency of boundary conditions.
4. Enable `monitoring.solver_state` and solver monitors to identify the equation that is stalling.
5. Use adaptive CFL for steady calculations only when needed. Adaptive CFL is unavailable for STANDARD unsteady calculations.

12.4 Solution divergence

- Lower the initial CFL, then increase it gradually with `run_control.cfl.type=schedule`.
- Check for strong shocks or low-Mach behavior and select an appropriate convective scheme.
- For NS/RANS cases, check viscous and turbulence preconditioners and wall thermal conditions.
- Use `outputs.diagnostics.on_diverge` to save the final state on divergence.

12.5 FR case errors

- `analysis.solver=fr` requires `analysis.time=unsteady`.
- Use the FR-specific `numerical_method.fr` block rather than copying DBNS keys. FR accepts `rk4`, `euler`, `tvd_rk3`, `rk34`, or `rk45` for `time_integration`; it does not accept `runge_kutta_explicit`.
- There is no `time_integration: "dual"`. Dual time is switched on by `run_control.time_stepping` — use `bdf1`, `sdirk3`, or `sdirk4`. FR does not implement `bdf2`.
- FR step-size and inner-loop keys live in `run_control`, not `numerical_method.fr`: `timestep_control` (physical Δt), `pseudo_time` (Δτ, including `pseudo_time.adaptive`), and `convergence.inner` (inner-loop bounds and residual gate).
- Entropy filtering cannot be combined with `run_control.timestep_control.controller=pi` or with dual time.
- FR output uses standard `outputs.volumes[]`, not a separate `fr_dump`.
- For cross-validation dumps, use `outputs.restart` or `fr_reference_compare`.

12.6 Related guides

- Running Wake CFD — Execution and validation
- CLI configuration-file reference — Complete JSONC key reference
- Environment variables — Logging and diagnostics

Part III

3. Tutorials

Resources for building practical problems step by step and learning reliable analysis procedures.

- Tutorial — A guided, step-by-step exercise
- Recommended settings and troubleshooting — Scenario-specific settings and diagnostics

Public validation results are available on the separate **Wake CFD Validation Reports** site.

First CLI analysis: 2D external airfoil flow

This tutorial is the first path for inspecting Wake CFD input files and run results. It does not begin with turbulence models or mesh generation. The goal is first to **run one validated example case and read its logs, force coefficients, and flow-field files**.

It is intended for everyone from new CFD users to practitioners evaluating a new solver. Follow it in order if you are new, or start at Review the results if you are already familiar with the workflow.

Prerequisites

- Wake CFD 2D executable (`wake_2d_mixed` or `wake_2d_mixed.exe`)
- The 2D airfoil example directory included with the distribution
- The mesh file specified by the example
- ParaView, if you want to visualize volume or surface results

> **!** Important
>
> **Keep the example directory and configuration files together.**
> Mesh paths and `$extends` in the input file are normally resolved from the example directory. Copying only configuration files elsewhere can cause execution to fail or inherited files to be unavailable.

1. What does it calculate?

It solves steady compressible flow around a 2D airfoil. The example demonstrates the following Wake CFD concepts together.

- `mesh` — An SU2, CGNS, or Gmsh mesh with named airfoil and farfield boundaries
- `models`, `materials`, `reference` — Fluid properties and reference values
- `domains.boundaries` — Airfoil wall and farfield conditions
- `numerical_method` — DBNS, convective scheme, CFL, and linear solver
- `monitoring` — Residual and force-coefficient records
- `outputs` — Volume and surface files for ParaView

The important point is not memorizing numbers, but understanding the connection between **mesh marker names → boundary conditions → monitor and output names**.

2. Copy the example to a separate working directory

Copy the entire example directory from the distribution to your working location. Below, `naca0012-2d` represents that example directory in the distribution.

```
my-wake-work/
└─ naca0012-2d/
   ├─ case.jsonc
   ├─ mesh/ or _mesh/
   └─ (required base/overlay JSONC files)
```

Preserve the original example. Running experiments that change analysis conditions in a copy makes results and configuration changes easier to track.

3. Run the case

Open a terminal in the directory containing the example's `case.jsonc` and invoke the 2D executable.

```
# Linux
/path/to/wake/wake_2d_mixed case.jsonc

# Windows PowerShell
& "C:\path\to\wake\wake_2d_mixed.exe" case.jsonc
```

First confirm that the example's `analysis.dimension` is 2 and that you are using the 2D binary. A 3D binary cannot run 2D input.

> **Tip**
>
> Do not change settings on the first run. You need a baseline result before you can distinguish problems caused by later changes.

4. What to check during the run

The console shows iteration number, residuals, and monitors requested by the example. For a steady analysis, check all three of the following.

1. **Residuals** — Whether they decrease sufficiently from their initial values
2. **Force coefficients** — Whether `CFx`, `CFy`, and `CMz` change very little late in the run

3. **Unsteady behavior** — If coefficients continue to oscillate substantially in a steady analysis, reconsider the settings, mesh, and physical model

Do not judge success from residuals alone. Even low residuals do not establish practical convergence if quantities of interest are still changing. Conversely, if residuals flatten but coefficients are stable and the cause of variation is understood, use convergence diagnostics to choose the next action.

Review the results

Example settings normally request the following two kinds of results.

Result	What to check	First review method
History CSV	Residuals, force coefficients, and monitor time history	Plot in a spreadsheet or Python
Volume/surface files	Spatial distributions such as Mach, pressure, Cp, and wall variables	Open in ParaView

In ParaView, first open volume results and color by `Mach` or `Pressure` to see whether the flow looks physically plausible. If surface results are available, inspect `Cp` on the airfoil surface. Filenames and field lists are defined in the input `outputs` block.

First settings to change

Preserve the baseline run, then change one item at a time and record the result differences.

Setting to change	Location	Expected change
Angle of attack	`aoa` in the farfield boundary condition	Changes in lift and pressure distribution
Mach number	Farfield and initialization	Possible changes in compressibility effects and shock location
Iteration limit	`run_control`	Changes in run time and convergence-review range
Output frequency	`outputs` / `monitoring`	Changes in disk use and temporal resolution

Mesh marker names, `boundary_names`, and `mesh_markers` must match the example mesh exactly. Do not rename them arbitrarily.

Next steps

- Input-block meanings: Input-file structure
- Execution and MPI: Running Wake CFD
- Convergence, mesh, and scheme selection: Recommended settings and troubleshooting
- Validated scope and comparison results: Wake CFD Validation Reports

Part IV

4. Best Practices

Recommended settings and troubleshooting

This chapter does not prescribe one set of values for every problem. Instead, it helps you use the physics, mesh, and convergence state to choose **a safe starting point and the next checks to make**. The input reference is authoritative for allowed values of each key.

Pre-run checklist

Check the following before committing to a long run.

- The analysis dimension matches the executable dimension (2D or 3D).
- Mesh zone and marker names match `domains.mesh_zones` and `boundaries.mesh_markers`.
- The physical model fits the flow problem (Euler or Navier–Stokes; laminar, SA, or SST).
- Reference length, area, and density agree with the definition of force coefficients.
- You have defined monitors and outputs for quantities of interest.
- The feature is within the validated scope.

> ⚠ Warning
>
> Some turbulence-model values other than `sa` and `sst` may pass input validation without producing a turbulence analysis. Always confirm the supported scope before evaluating turbulent-flow results.

Starting points by problem type

Problem	Physics/time	Numerical starting point	Result assessment
Low-speed, smooth external flow	DBNS, Euler or laminar NS, steady	Use second-order spatial accuracy and a limiter; start with conservative CFL	Stable force coefficients and pressure distribution
Compressible external flow with shocks	DBNS, steady	Shock-robust convective scheme and limiter; low initial CFL	Shock location, force coefficients, and nonphysical oscillations

Problem	Physics/time	Numerical starting point	Result assessment
Attached turbulent boundary layer	DBNS, RANS-SA or SST, steady	First match the wall first-cell height to model or wall-function assumptions	y+, skin-friction coefficient, and profiles
Unsteady flow, such as vortex shedding	DBNS or FR, unsteady	Decide time step and observation duration first, then configure BDF or dual time	Frequency, mean/amplitude, and time history
Internal flow	DBNS/PBNS appropriate for the boundary conditions	First check mass-conservation monitors and inlet/outlet consistency	Mass imbalance, pressure drop, and flow rate

This table provides starting points only. Before using results for design decisions, compare them with a Wake CFD Validation Reports case under similar conditions or an independent reference.

Schemes and CFL

Do not start with a high CFL or the most aggressive settings. This sequence improves reproducibility and root-cause analysis.

1. Produce a baseline result with a validated case or conservative settings.
2. Confirm that residuals and quantities of interest stabilize.
3. Raise CFL incrementally to see whether the same result can be reached faster.
4. Change schemes, preconditioners, and meshes one at a time.

Nonphysical oscillations near shocks, negative material properties, and early-iteration blowups usually first indicate **excessive CFL, poor mesh quality, or inconsistent boundary conditions**. Check those three before changing schemes alone.

Assessing convergence

For a steady analysis, “converged” means more than low residuals.

1. **Run completeness** — Are there no input errors, NaN/Inf values, or abnormal termination?
2. **Residuals** — Are their reduction from the initial values and downward trend sufficient?

3. **Quantities of interest** — Are design quantities such as force coefficients, mass flow, and pressure drop stable?
4. **Conservation** — For internal flow, is mass imbalance within tolerance?
5. **Spatial field** — Do Mach, pressure, and velocity fields agree with the boundary conditions and physics?

Use the following response order when residuals stall.

Symptom	Check first	Next action
Divergence or NaN from the start	Units, boundary conditions, negative mesh volumes, initial CFL	Lower CFL and simplify initial and boundary conditions
Residuals fall but force coefficients keep changing	Iteration count, monitor definition, actual flow unsteadiness	Extend the run or consider an unsteady analysis
Residuals stall at a high level	Mesh quality, scheme/limiter, linear solver	Change one item at a time and compare with the baseline
Unstable internal-flow results	Inlet/outlet conditions and mass balance	Recheck the boundary-condition combination and domain markers

Mesh and wall treatment

Good solver settings cannot compensate for a poor mesh. In particular, for turbulent wall analyses, first-cell height and y+ must fit the model and wall-function choice.

- Without wall functions: prepare a mesh that sufficiently resolves the near-wall region.
- With wall functions: confirm the target y+ range and record wall-function use with the results.
- Aspect ratio, skewness, and nonorthogonality can directly affect convergence and wall results.
- Do not use a single-mesh result as a design basis without checking how mesh refinement changes it.

Output and restart operation

Writing large volume files at every iteration rapidly increases I/O and disk use.

- Prioritize CSV monitors and residual output for convergence diagnostics.
- Set volume and surface frequencies separately for intermediate checks and final results.

- Enable restarts for long calculations and define a retention policy.
- Keep configuration files, solver version, mesh, and run logs in the same folder as results.

For detailed formats, see File output and Monitoring.

Part V

5. Input and configuration (GUI/CLI)

This section documents Wake CFD's configuration scope—mesh, physics, boundary conditions, and numerical methods. The GUI and CLI operate on the same project model; concepts are presented here once and apply to both interfaces.

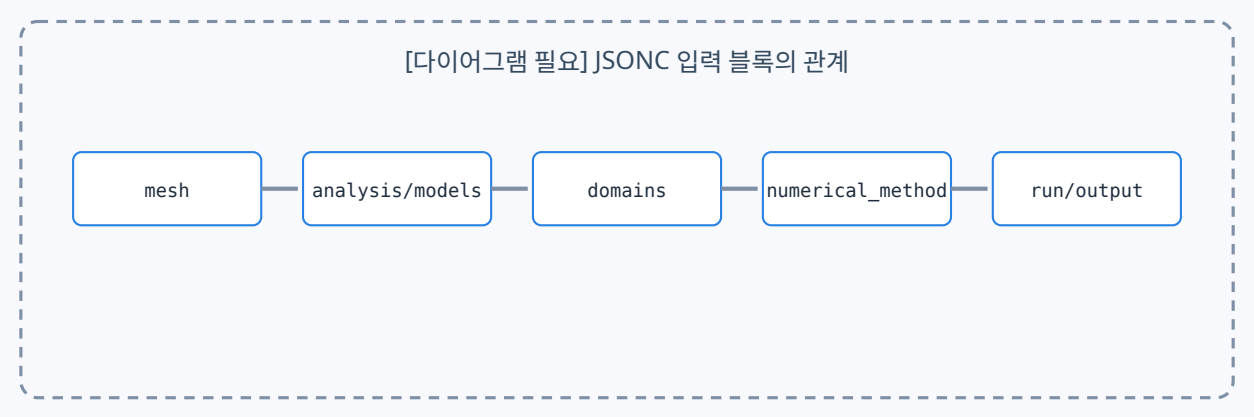


Figure 12.1: Placeholder for the input-file structure. Before public release, replace with a diagram showing relationships between configuration blocks.

How to use this reference

Each topic is structured in the following order.

1. **What it is** — The setting's role in an analysis
2. **Configure in the GUI** — Modal inputs and selection criteria
3. **Configure from the CLI** — JSONC examples for batch, HPC, and automation

The GUI provides visual configuration and writes to the same JSONC project model. Screenshots show the corresponding Desktop editor, while the JSONC examples are the same settings consumed by the CLI, AI agents, and automation workflows. Some advanced or solver-specific keys are not yet exposed by the GUI; those keys remain available through JSONC and are identified on the relevant page. Values visible in a screenshot describe one configured case; they are not necessarily the solver defaults or a recommendation for every case.

Find a topic

What to configure	Start here
Complete JSONC structure and required blocks	Input-file structure
Mesh files and zone/marker connections	Mesh, Domains and boundary conditions

What to configure	Start here
Solver, steady/unsteady analysis, and dimension selection	Analysis
Euler, Navier–Stokes, and turbulence models	Physical models
Fluid and solid properties and equations of state	Materials
Force, moment, and coefficient-normalization reference values	Reference values
Fluid/solid regions and boundary conditions	Domains and boundary conditions
Periodic, mixing plane, frozen rotor, sliding mesh, GGI, or CHT interfaces	Interfaces
Schemes, linear solvers, CFL, and time integration	Numerical methods, Run control and convergence
Initial fields, restart, or expression-based initial conditions	Initialization
Residual, force, mass flow, and probe monitoring	Monitoring
Restart, volume, surface, and slice file outputs	File output
Reusing cases with inheritance, overlays, and patches	Running Wake CFD

Case-authoring sequence

1. Check zone and marker names in `mesh`.
2. Define the physical problem through `analysis`, `models`, `materials`, and `reference`.
3. Connect `domains` and, if necessary, `interfaces`.
4. Write `numerical_method`, `initialization`, and `run_control` for the selected solver.
5. Add `monitoring` and `outputs` for quantities of interest.

> Tip
>
> **Start from an example**
> Copy a similar example and modify settings incrementally rather than authoring a complete input file directly. This approach reduces configuration errors.

Related guides

- To configure cases visually in the GUI: GUI guide
- To write and run JSONC files: CLI guide
- To automate case generation and sweeps with Python: Python API

- To integrate Slurm, gRPC, or agent workflows: Automation and integration
- To learn configuration through worked examples: Tutorials
- To choose which interface to start with: How to use Wake CFD

13 Mesh

Formats, multiple zones, scope namespaces, and scale_factor

13.1 What this setting is

A mesh is the computational model that divides the analysis space into finite cells. In Wake CFD, the mesh file provides names called **zones** and **markers**; domains use zones as physical regions and boundary conditions use markers as boundary faces. The required resolution and naming for geometry, boundary layers, and wakes are therefore the starting point for analysis setup.

13.2 Configure in the GUI

Use **Home → Import Mesh** or right-click the Meshes tree node to add a CGNS, SU2, or Gmsh file. Import **as link** to keep the source file read-only, or **as editable** to save a project-owned CGNS copy when the project is saved. Select an imported mesh in the tree to inspect its zones and boundary markers; select the mesh entry to edit its properties. Zone and marker names are then assigned in Domains, not in the `mesh` block.

GUI location / control	Input key	What it does
Import Mesh → selected file	`mesh[].file_name`	Adds a mesh file; the File property is read-only after import.
Mesh Properties → Format	`mesh[].format`	Sets the reader format for the imported file.
Mesh Properties → Scale factor	`mesh[].scale_factor`	Converts mesh coordinates to metres; use `0.001` for millimetres.
Mesh Properties → Scope	`mesh[].scope`	Names a mesh file when a case uses multiple files.

GUI location / control	Input key	What it does
Mesh tree → zones / markers	—	Shows imported names for later domain and boundary assignment; these names are not keys in `mesh[]`.

The GUI import picker lists CGNS, SU2, and Gmsh. UGRID can be supplied in JSONC; it is not listed in that picker. The `mesh` block is an array, even for one file.

13.3 Input keys

Key	Required	Meaning
`file_name`	✓	Path to the mesh file
`format`		`cgns` (default), `su2`, `gmsh`, or `ugrid`; set it explicitly for non-CGNS files
`scale_factor`		Coordinate scale factor (e.g. mm→m = `0.001`); see scale_factor below
`scope`		File-level namespace for zone paths across multiple files; see Multiple zones below

The array must contain at least one file, except when `initialization.type: "restart"` loads a full mesh from the restart file.

13.4 Supported formats

Format	`format` value	Notes
CGNS	`cgns`	Recognizes boundaries by BC_t node name
SU2	`su2`	Recognizes boundaries by MARKER_TAG

Format	`format` value	Notes
Gmsh	`gmsh`	Uses msh2 and physical-group names
AFLR UGRID	`ugrid`	Available to the solver through JSONC; Desktop import currently lists CGNS, SU2, and Gmsh

`format` accepts these four values. When it is omitted, the solver uses `cgns`; it does not select a format from the filename extension. Set it explicitly for SU2, Gmsh, or UGRID input. Desktop import recognizes supported extensions and writes the detected format into the case.

13.5 Multiple zones (multiple files)

With more than one mesh file, every entry needs a distinct, nonempty `scope`, and a scope cannot contain `/`. A zone is then addressed as `scope/zone` (or by the file's scope when selecting the whole file). Boundary markers keep their names and are resolved within the bound domain's zone; do not add the scope prefix to `mesh_markers`.

13.6 Boundary-name matching

`domains[].boundaries[].mesh_markers` in JSONC and mesh-file marker names are matched **without regard to ASCII letter case**, within the domain's bound zone. The spelling otherwise needs to match. To inspect zone, section, and marker names before authoring a case, run `wake_grid <mesh_file> --bc`. On mismatch:

```
Mesh location has no matching config boundary
```

Format	Boundary-name source	Notes
CGNS	**BC_t node name**	Not Family name (currently unsupported)
SU2	MARKER_TAG	In the `.su2` file
Gmsh	Physical-group name	`Physical Surface("inlet")` in `.geo`

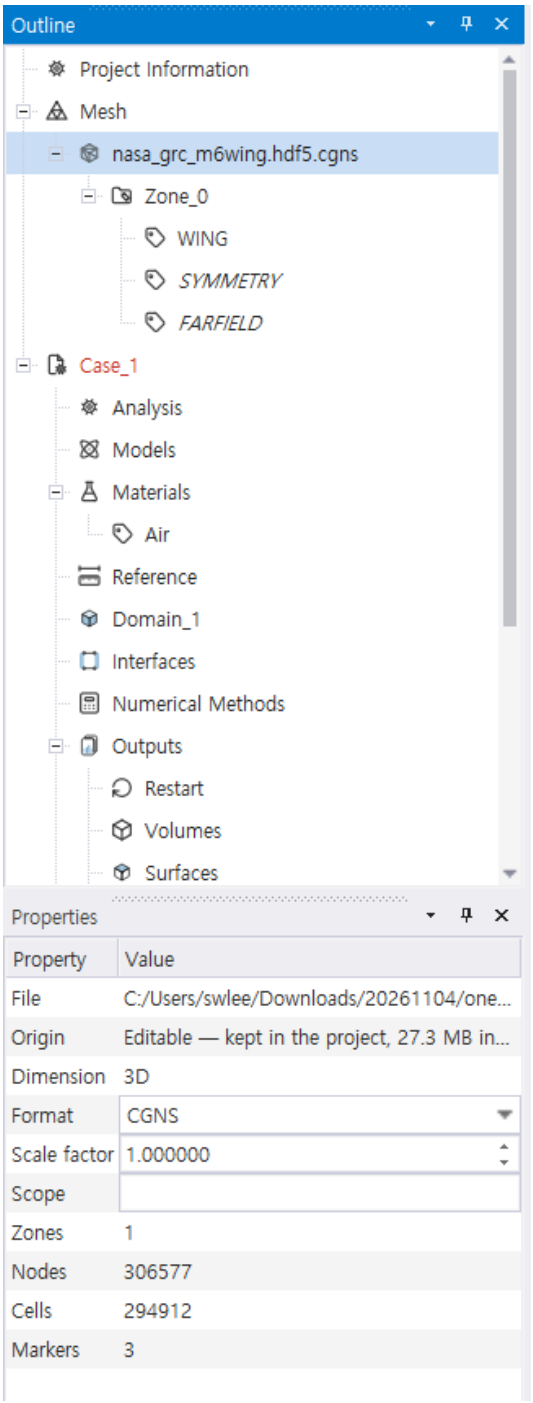


Figure 13.1: An imported mesh in the project outline, with its zone and boundary markers expanded, and the mesh properties below. Format, Scale factor, and Scope are editable; File, Dimension, and the element counts are read from the file.

> ⚠ Warning
>
> **CGNS notes**
>
> - Wake CFD reads ZoneBC_t/BC_t node names (`cg_boco_info`).
> - `FamilyName_t` is unsupported.
> - ICEM CFD: Part names are automatically converted to BC_t names.
> - Pointwise: Set BC names directly.
> - Names can be lost during Gmsh-to-CGNS conversion; use Gmsh directly when possible.

13.7 scale_factor

Use this for coordinate-unit conversion. All coordinates are internally converted to SI units (m) before processing.

13.8 JSONC examples

For a single CGNS mesh, keep the required array shape. The default format can be omitted:

```jsonc
{ "mesh": [{ "file_name": "naca0012.cgns" }] }
```

Convert millimetre coordinates to metres with `scale_factor`:

```jsonc
{
  "mesh": [
    { "file_name": "model_mm.cgns", "format": "cgns", "scale_factor": 0.001 }
  ]
}
```

Use distinct scopes for multiple files:

```jsonc
{
  "mesh": [
    { "file_name": "rotor.cgns", "format": "cgns", "scope": "rotor" },
    { "file_name": "stator.su2", "format": "su2", "scope": "stator" }
  ]
}
```

14 Analysis

Solver, time, and dimension

14.1 What this setting is

Analysis settings are the starting point for deciding which equations a case solves and how. Solver, steady or unsteady state, and 2D or 3D determine the available model and numerical-method options. They do not define fluid properties or boundary conditions; those are connected in separate settings later.

14.2 Configure in the GUI

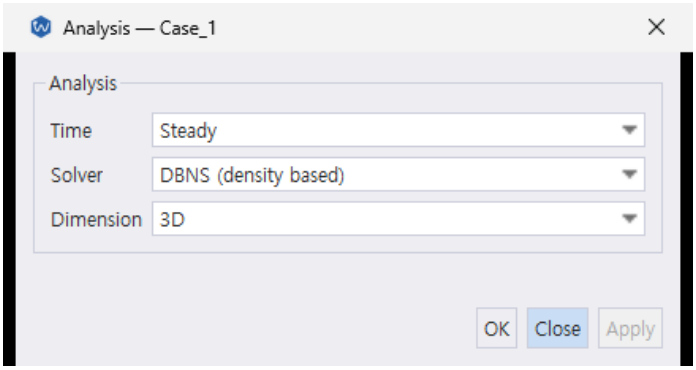


Figure 14.1: The Analysis dialog selects steady or unsteady time, the solver family, and the mesh dimension.

GUI control	Input key	What to choose
Solver	`analysis.solver`	DBNS is selectable in the Desktop. PBNS and FR are listed but disabled; configure those paths in JSONC.
Time	`analysis.time`	Steady or unsteady analysis.

GUI control	Input key	What to choose
Dimension	`analysis.dimension`	2D or 3D, matching the mesh and solver executable.

`analysis` selects the case's analysis axes. Numerical options belong in `numerical_method`; run-time settings belong in `run_control`. The first imported mesh sets the initial Dimension choice; you can change it in Analysis afterward. `analysis.nondimensional` is not exposed in this dialog, but remains available in JSONC.

Each key is covered in its own section below.

14.3 solver

Value	Meaning	Conditions
`dbns`	Density-based finite-volume solver	Default; selectable in the Desktop
`pbns`	Pressure-based finite-volume solver	3D only; JSONC configuration
`fr`	Flux-reconstruction solver	Unsteady path; JSONC configuration and an FR-enabled build

The `solver` value determines the active `numerical_method` subblock.

```
dbns → numerical_method.dbns
pbns → numerical_method.pbns
fr   → numerical_method.fr
```

`fvm` and `finite_volume` are accepted aliases for `dbns`; `flux_reconstruction` is an alias for `fr`. Use canonical values in new files.

> **!** Important
>
> `analysis.discretization` is not a current input key. Express FVM or FR selection with `analysis.solver`.

14.4 time

Value	Meaning
`steady`	Steady analysis
`unsteady`	Unsteady analysis; configure time stepping in `run_control`

DBNS and PBNS support both values. FR uses the unsteady path; selecting `steady` with FR produces a validation warning. Set `analysis.time` to `unsteady` for FR cases and configure their time stepping in `run_control`.

14.5 dimension

Value	Executable
2	`wake_2d_<tier>`
3	`wake_3d_<tier>`

The compiled `WAKE_DIM` must match the input value. `<tier>` is the precision build (`fp64`, `mixed`, or `fp32`); Windows executable names add `.exe`. PBNS currently requires 3D.

14.6 nondimensional

Value	Default / availability	Meaning
`true`	Default for DBNS; rejected for PBNS and FR.	Solve the DBNS equations in internally scaled variables.
`false`	Default for PBNS and FR; also accepted for DBNS.	Use dimensional solution variables.

This is an internal solver-variable scaling choice, not a change to the physical units requested for case inputs. It is normally best to omit the key and use the solver-specific default.

14.7 JSONC examples

The GUI writes these same keys for the DBNS choices it exposes. For the CLI, or for PBNS and FR, set them in JSONC. A steady 3D DBNS selection is:

```
{ "analysis": { "solver": "dbns", "time": "steady", "dimension": 3 } }
```

A 3D pressure-based selection is:

```
{ "analysis": { "solver": "pbns", "time": "steady", "dimension": 3 } }
```

An unsteady FR selection is:

```
{ "analysis": { "solver": "fr", "time": "unsteady", "dimension": 3 } }
```

These are analysis fragments, not complete runnable cases. PBNS and FR also need solver-appropriate models, numerical methods, and run-control settings.

15 Physical models

Flow equations, regime, and turbulence

15.1 What this setting is

Physical models declare which flow phenomena Wake CFD calculates. Select inviscid Euler or viscous Navier–Stokes, the compressibility regime, and the turbulence model here. These choices affect the required mesh resolution and how results are interpreted, so first assess the Reynolds number and the presence of shocks, boundary layers, and separation.

15.2 Configure in the GUI

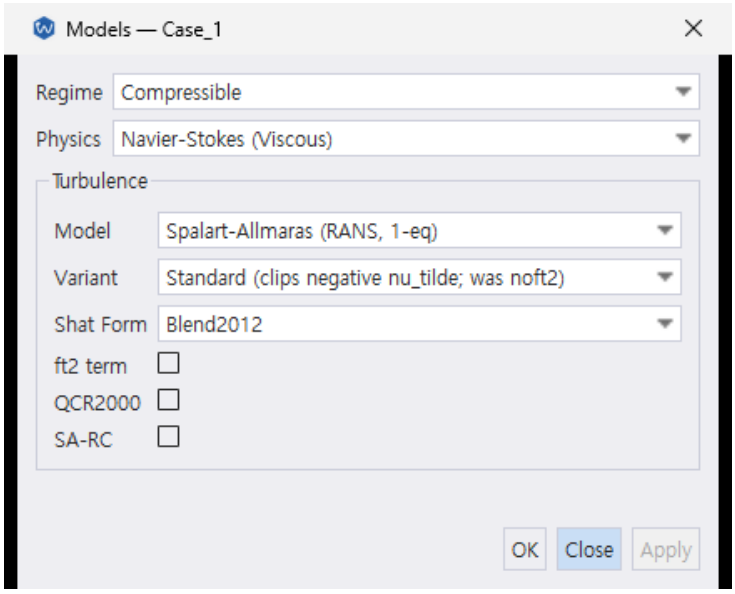


Figure 15.1: The Models dialog for a compressible Navier–Stokes case using the Spalart–Allmaras turbulence model. Model-specific controls appear after the turbulence model is selected.

The Desktop currently authors the DBNS model path. PBNS and FR can be configured through JSONC, though their solver choices are disabled in the Desktop Analysis dialog. The Models dialog writes its available choices to the `models` block.

GUI control	Input key	What to choose
Regime	`models.regime`	Compressible for DBNS. The dialog disables incompressible while DBNS is selected.
Physics	`models.type`	Euler for inviscid flow; Navier–Stokes for viscous flow.
Turbulence → Model	`models.turbulence.model`	With Navier–Stokes, choose Laminar (none), SA, or SST. Other listed models are disabled.
SA controls	`models.turbulence.sa.*`	Variant, S☒ form, ft2, QCR2000, and rotation correction appear when SA is selected.
SST controls	`models.turbulence.sst.*`	Variant and production form appear when SST is selected.

`models.axisymmetric`, wall-treatment constants, and closure-constant overrides are not exposed in this dialog. Set them in JSONC when needed; the accepted values and restrictions are described below. The JSONC examples come after the model choices.

15.3 type

Value	Meaning
`euler`	Inviscid flow.
`navier_stokes`	Viscous flow; `turbulence` distinguishes laminar from RANS.

15.4 regime

Value	Meaning
`compressible` (default)	Available with DBNS, PBNS, and FR; use a gas-law equation of state.
`incompressible`	PBNS constant-density flow or FR artificial compressibility (AC). DBNS rejects it; fluid materials must use `eos.type: "constant_density"`.

PBNS is **not** inherently incompressible: its default regime is compressible. For incompressible PBNS, use `velocity_inlet` and `pressure_outlet` rather than `farfield` or a total-conditions inlet. FR AC requires `numerical_method.fr.ac_zeta > 0` and dual-time stepping; see Numerical methods. The `freesurface` enum value parses but is rejected before the run: regime=freesurface is under development and not available yet.

The energy equation follows from the regime and is not a separate key. A compressible case carries the total-energy equation, including pressure work and, for viscous flow, viscous work. Incompressible PBNS and FR AC carry no energy equation: PBNS keeps temperature at its initial value and FR AC has no temperature state, so an isothermal or heat-flux wall is rejected there. Setting `numerical_method.pbns.coupling.energy.max_iter` to `0` is not a way to freeze temperature in compressible PBNS; it is rejected.

`models.gravity` is reserved and rejected if supplied, even as `false`.

15.5 axisymmetric

`axisymmetric` enables the 2D, no-swirl body-of-revolution formulation. The mesh uses `x` as the axis and `y` as the radius, so every mesh coordinate must have $y \geq 0$; define the axis itself as a `symmetry` boundary. It is not a post-processing revolution of a planar solution: Wake CFD adds the axisymmetric source terms and uses ring measures for applicable surface quantities.

Key	Values	Default	Use
`axisymmetric`	`false`, `true`	`false`	Enable the 2D axisymmetric, no-swirl formulation.

Wake CFD validates this mode before a run starts. It requires a 2D `wake_2d_<tier>` build, `analysis.dimension: 2`, `analysis.solver: "dbns"`, and `regime: "compressible"`. Euler, laminar Navier–Stokes, SA, and SST are available. Multigrid, PBNS, FR, rotating domains, interfaces, wall functions, and SA-QCR2000 are rejected rather than treated as planar cases.

For a body of revolution, force monitors must be axial (`direction: [1, 0, 0]` or `[-1, 0, 0]`), and moment monitors are not meaningful. These limits are intentional: radial and lateral components cancel over a full revolution.

15.6 turbulence

`turbulence.model` selects the turbulence model; model-specific subblocks (`sa`, `sst`) override closure constants. Omitted constants use the solver's baseline defaults.

15.6.1 model

Value	Meaning
`none`	Laminar Navier–Stokes or inviscid Euler.
`sa`	Spalart–Allmaras RANS.
`sst`	k-ω SST RANS.

Use `sa` or `sst` with viscous DBNS. PBNS has no turbulence equations and rejects these models; FR does not use these RANS models. Other model names visible in the schema or disabled in the GUI are not implemented and are rejected rather than silently falling back to laminar flow.

15.6.2 Model-constant overrides

15.6.2.1 SA (Spalart-Allmaras)

`variant`, `shat_form`, and `ft2` are covered below; `cfl_reduction` has its own section. All fields are optional.

Key	Default	Meaning
`variant`	`standard`	Negative $\tilde{\nu}$ handling; `neg` enables SA-neg.
`shat_form`	`blend2012`	$\hat{S}$ safeguard; `floor` is also available.
`ft2`	`false`	Classic SA laminar-suppression term.
`cv1`	`7.1`	Viscous sublayer correction.

Key	Default	Meaning
`cb1`, `cb2`	`0.1355`, `0.622`	Production and diffusion constants.
`sigma`, `kappa`	`2/3`, `0.41`	Diffusion denominator and von Kármán constant.
`cw2`, `cw3`	`0.3`, `2.0`	Destruction constants.
`cn1`	`16.0`	SA-neg negative-branch constant.
`ct3`, `ct4`	`1.2`, `0.5`	ft2 constants; `ct3` also enters the SA-neg negative branch.
`c2`, `c3`	`0.7`, `0.9`	$\bar{S}$ limiter constants.
`r_max`	`10.0`	Upper bound on `r`.
`prandtl_turb`	`0.9`	Turbulent Prandtl number of the energy closure.
`cfl_reduction`	`1.0`	Factor on the SA pseudo-time step.
`qcr2000`	`false`	Anisotropic QCR2000 stress correction; not with axisymmetry.
`rotation_correction`	`false`	SA rotation/curvature correction.

15.6.2.2 `variant`: treatment of negative $\tilde{\nu}$

`variant`	Meaning
`standard` (default)	Disallows negative $\tilde{\nu}$; clips with `max(ν̃, 0)` after update
`neg`	SA-neg (Allmaras 2012); permits negative values with a negative-source branch and `fn` diffusion correction

For solutions converging to $\tilde{\nu} \geq 0$, the two variants are theoretically equivalent. `neg` is a robustness extension.

> **Note**
>
> The `noft2` variant was renamed `standard`. The ft2 term is now a separate `ft2` boolean, independent of variant. `edwards` was removed from the enum to prevent silent fallback from a stub implementation.

15.6.2.3 `ft2`: laminar-suppression term

With `ft2: true`, the source term includes `ft2 = ct3·exp(−ct4·χ²)` (classic SA). The default `false` corresponds to SA-noft2.

Combinations of `variant` and `ft2` correspond to names used in the literature.

Combination	Literature name
`standard` + `ft2: false`	SA-noft2
`standard` + `ft2: true`	classic SA
`neg` + `ft2: true`	SA-neg (complete 2012 form)

15.6.2.4 `shat_form`: form preventing negative S̄

Value	Meaning
`blend2012` (default)	Allmaras 2012 smooth c2/c3 blend; standard for SA-neg
`floor`	Simple floor `max(Ω + S̄, ε)` for reproducing classic reference data

Using `floor` with neg is a nonstandard combination and emits a warning.

15.6.2.5 cfl_reduction

`dt_sa = dt_flow × cfl_reduction` reduces only the turbulence-equation pseudo-time step. The default is `1.0`; values of `0.5`–`0.8` can suppress turbulence-residual oscillations in stiff boundary layers or high-aspect-ratio meshes.

15.6.2.6 SA-QCR2000 (Quadratic Constitutive Relation)

Replaces Boussinesq isotropic stress with an **anisotropic QCR** form (Spalart 2000):

```
τ_QCR = τ_Bouss − 0.3·Σ_k (O_ik·τ_jk + O_jk·τ_ik)
O_ik = (∂u_i/∂x_k − ∂u_k/∂x_i) / |∇u|_F
```

The SA source term is unchanged; only mean-flow viscous stress is modified. Recommended for corner-separation and secondary-flow cases. It is variant-independent and combines freely with `standard` or neg.

15.6.2.7 SA-RC (Rotation Correction)

Suppresses runaway production in vortex cores (Dacles-Mariani et al.):

`Prod = Shat + CRot·min(0, ||S|| − ||ω||),  CRot = 2.0`

It has no effect in shear-dominated boundary layers where `||S|| ≈ ||ω||`, acting only in vortex cores. It is variant-independent and compatible with QCR2000. Recommended for rotor, propeller, and wing-tip vortex cases.

NASA TMR's recommended SA-RC-QCR2000 baseline is `rotation_correction: true`, `qcr2000: true`.

15.6.2.8 SST (k-ω SST)

Key	Default	Meaning
`variant`	`v2003`	See SST variant below.
`production`	`strain`	Production form: `strain` (P = μt·S²), `vorticity` (P = μt·Ω²), or `kato_launder` (P = μt·S·Ω).
`sigma_k1`, `sigma_w1`, `beta_1`	`0.85`, `0.5`, `0.075`	Inner-layer (k-ω) constants.
`a1`	`0.31`	Shear-stress limiter constant.
`sigma_k2`, `sigma_w2`, `beta_2`	`1.0`, `0.856`, `0.0828`	Outer-layer (k-ε) constants.
`beta_star`	`0.09`	Shared constant.
`kappa`	`0.41`	von Kármán constant.
`prandtl_turb`	`0.85`	Turbulent Prandtl number of the energy closure.
`cfl_reduction`	`1.0`	Factor on the SST pseudo-time step; lower values can help stiff cases.
`max_update_fraction`	`1.0`	Per-point relative k/ω update cap.

15.6.2.9 SST variant

Value	Meaning
`v2003` (default)	Menter 2003 SST with the selected production form.
`sust`	Adds sustaining production floors based on freestream k and ω; they prevent ambient turbulence from decaying below its prescribed level.

`sust` is implemented, not merely a schema placeholder. Its freestream values come from the turbulence inflow settings described below.

> **Note**
>
> `v1994` was removed on 2026-06-05 after a silent hybrid bug occurred when only the γ branch was implemented. There are no plans to restore it: NASA TMR SST-V changes only the production term to vorticity while retaining the 2003 limiter and constants, whereas V1994 changes the limiter and constants as well and therefore moves farther from the reference.

15.6.3 Wall treatment (global defaults)

Shared constants for boundaries using `wall_treatment: "automatic"` (see Domains). DBNS RANS uses the SA analytic wall law or SST Spalding/White–Christoph law. The treatment is continuous across y+; the former switching threshold has been removed.

Key	Default	Meaning
`kappa`	`0.41`	von Kármán constant.
`B`	`5.0`	Spalding log-law intercept.
`relaxation`	`0.5`	Wall-law Newton solve relaxation factor.
`relaxation_turb`	`0.7`	k/ω relaxation; SST wall function only.
`max_iter`	`200`	Wall-law Newton iteration cap.
`tol`	`1e-12`	Wall-law Newton convergence tolerance.

For **FR**, `wall_treatment: "automatic"` means a wall-modeled LES stress law, not RANS. It requires compressible Navier–Stokes and a no-slip wall; FR artificial-compressibility cases cannot use it. The following additional `models.wall_treatment` keys are for the FR path:

Key	Default	Meaning
`exchange.type`, `exchange.value`	`element_fraction`, `0.5`	Exchange point as a fraction of wall-element height; `exchange.value` must be in `(0, 2)`.
`energy_work`	`false`	Include wall shear work in the modeled energy flux.
`ldg_state`	`no_slip`	Auxiliary wall state: `no_slip`, `slip_mirror`, or `non_corrected`.
`inviscid_state`	`slip_mirror`	Inviscid ghost state: `slip_mirror` or `no_slip`.
`tau_jump`	`false`	Apply the LDG penalty on the modeled wall face.
`coarse_level`	`true`	Evaluate the wall model on p-multigrid coarse levels.
`wall_filter.alpha`, `wall_filter.order`, `wall_filter.kc`	`0`, `8`, `1`	Wall-normal modal filter; `alpha: 0` disables it.
`eddy_viscosity`, `eddy_viscosity_layers`	`false`, `1`	Optional wall-cell eddy viscosity and its number of element layers.

> **Note**
>
> **Location of `prandtl_turb` varies by model**
> `prandtl_turb` is a RANS closure constant, not a gas property. It belongs in the model-specific subblock (`models.turbulence.sa.prandtl_turb` or `models.turbulence.sst.prandtl_turb`), with different recommended values (SA: 0.9, SST: 0.85).

15.7 Four-axis solver identity

`models.type` and `models.turbulence` are two of the four axes. The other two are `analysis.solver` and `analysis.time`; their combination determines the solver mode.

```
euler         + none    → Euler (inviscid)
navier_stokes + none    → laminar NS
navier_stokes + sa      → RANS-SA
navier_stokes + sst     → RANS-SST
```

15.8 Location of freestream turbulence quantities

Set freestream or inlet turbulence in that boundary's `turbulence` subblock, not in `models`. The same fields are available for uniform initialization.

Key	Applies to	Meaning
`nu_factor`	SA	Inflow ν☒/ν ratio; default 3.
`intensity`	SST	Turbulence intensity as a fraction; default `0.05` (5%).
`eddy_viscosity_ratio`	SST	Inflow μt/μ; default `10` when no length scale is supplied.
`length_scale`	SST	Turbulence length scale in metres.
`hydraulic_diameter`	SST	Hydraulic diameter; the length scale becomes `0.07` times this value.

For SST, set at most one of `eddy_viscosity_ratio`, `length_scale`, and `hydraulic_diameter`. The former `turb2lam_ratio` is not a current input key. `intermittency` is reserved for a transition model and is rejected if supplied. See Domains for boundary-condition-specific usage.

15.9 JSONC examples

The GUI writes the same keys for the controls it exposes. For a CLI case or a setting not yet available in the GUI, use JSONC. `turbulence` must be an object: `"turbulence": "sa"` is not accepted.

Compressible DBNS with SA and an explicit total-energy setting:

```
{
  "analysis": { "solver": "dbns", "dimension": 3 },
  "models": {
    "type": "navier_stokes",
    "regime": "compressible",
    "turbulence": {
      "model": "sa",
      "sa": { "variant": "standard", "ft2": false, "qcr2000": true }
    }
  }
}
```

Incompressible PBNS with no energy equation (the fluid material must use `eos.type: "constant_density"`):

```
{
  "analysis": { "solver": "pbns", "dimension": 3 },
  "models": {
    "type": "navier_stokes",
    "regime": "incompressible",
    "turbulence": { "model": "none" }
  }
}
```

Compressible FR with total energy (FR also needs its numerical-method and time-stepping settings):

```
{
  "analysis": { "solver": "fr", "time": "unsteady", "dimension": 3 },
  "models": {
    "type": "navier_stokes",
    "regime": "compressible",
    "turbulence": { "model": "none" }
  }
}
```

SST with the sustaining variant and one valid set of inflow turbulence values:

```
{
  "models": {
    "type": "navier_stokes",
    "turbulence": {
      "model": "sst",
      "sst": { "variant": "sust", "production": "strain" }
    }
  },
  "domains": [{
    "boundaries": [{
      "name": "farfield", "type": "farfield", "mesh_markers": ["FARFIELD"],
      "mach": 0.5, "pressure": 101325.0, "temperature": 288.15,
      "turbulence": { "intensity": 0.05, "eddy_viscosity_ratio": 10.0 }
    }]
  }]
}
```

These are configuration fragments, not complete runnable cases; mesh, materials, and other required blocks are omitted. See Input-file structure for a full case layout.

16 Materials

Equation of state, specific heat, and transport properties

16.1 What this setting is

Materials define how a fluid or solid responds to changes in pressure and temperature. Each domain names one material through `domains[].material`. The equation of state and specific heat determine thermodynamic behavior; viscosity and thermal conductivity determine transport in viscous flow.

16.2 Configure in the GUI

Choose the material type first. For a pure fluid, the dialog has Equation of State, Specific Heat, Viscosity, and Thermal Conductivity groups; the last three can be collapsed. Changing a model replaces that group's parameters with editable starting values. Changing the material type replaces the entire material's model settings while preserving its name.

GUI control	Input key	What it configures
Name	`materials[].name`	Material name used by `domains[].material`.
Type	`materials[].type`	Pure fluid or solid. Mixture is listed but disabled.
Equation of State → Model	`materials[].eos.type`	Ideal gas, Peng–Robinson, or constant density, subject to solver and regime.
Equation of State → properties	`materials[].eos.*`	Gas constant/molar mass, critical properties, or constant density.
Specific Heat → Model and properties	`materials[].specific_heat.*`	Constant Cp or NASA polynomial with low/high temperature ranges.

GUI control	Input key	What it configures
Viscosity → Model and properties	`materials[].viscosity.*`	Constant, Sutherland, or power-law dynamic viscosity.
Thermal Conductivity → Model and properties	`materials[].conductivity.*`	Constant k, Sutherland k(T), or constant Prandtl number.
Solid properties	`materials[].solid.*`	Thermal conductivity k, density, and Cp when Type is Solid.

The GUI currently enables Peng–Robinson only for compressible DBNS and NASA polynomial only for DBNS with an ideal-gas EOS. The solver also accepts these models with compressible PBNS through JSONC. They are unavailable with FR; both require a `mixed` or `fp64` solver binary. Constant-density EOS requires `models.regime: "incompressible"`. See JSONC examples after the key descriptions.

16.3 Material entry and type

`materials` is an array with at least one entry. Each entry requires `name` and `type`; a domain references the name. Unknown keys are rejected.

`type`	Meaning
`pure`	Single-species fluid.
`solid`	Solid material for steady conjugate heat transfer (CHT).
`mixture`	Reserved; rejected by validation.

The shorter keys `eos` and `conductivity` are used below. `equation_of_state` and `thermal_conductivity` are accepted aliases. The former `eos.gamma` and top-level material `density` keys are not accepted: γ comes from specific heat, and constant density belongs at `eos.density`.

16.4 Equation of state (EOS)

Key	Unit	Meaning
`eos.type`	—	`ideal_gas` (default), `peng_robinson`, or `constant_density`.

Key	Unit	Meaning
`eos.gas_constant`	`J/(kg·K)`	Gas-law R; default `287.058` if neither R nor molar mass is given.
`eos.molar_mass`	`g/mol`	Gas-law molar mass; if R is omitted, R = `8314.46 / molar_mass`.
`eos.T_critical`	`K`	Required positive critical temperature for Peng–Robinson.
`eos.P_critical`	`Pa`	Required positive critical pressure for Peng–Robinson.
`eos.acentric_factor`	—	Peng–Robinson Pitzer factor; default `0`.
`eos.density`	`kg/m³`	Required positive ρ☒ for `constant_density`; rejected for gas-law EOS.

If both R and molar mass are set, R takes precedence; a disagreement greater than 0.1% produces a warning. With a constant specific heat, the heat-capacity ratio is derived as γ = Cp / (Cp − R); omitting Cp gives Cp = 3.5 R and γ = 1.4. A compressible gas-law EOS derives density from pressure and temperature.

`constant_density` must be paired with an incompressible PBNS or FR case and has no gas constant or critical properties. Peng–Robinson runs with compressible DBNS or PBNS, not FR or an `fp32` binary.

16.4.1 Property table

`eos.table` can be used with Peng–Robinson or an ideal gas with NASA-polynomial specific heat. Its optional keys are:

Key	Unit	Meaning
`eos.table.temperature`	`K`	`[T_min, T_max]`, with `0 < T_min < T_max`.
`eos.table.pressure`	`Pa`	`[P_min, P_max]`, with `0 < P_min < P_max`.
`eos.table.resolution`	—	`[n_density, n_energy]`; integers from 8 to 2048, default `[256, 256]`.

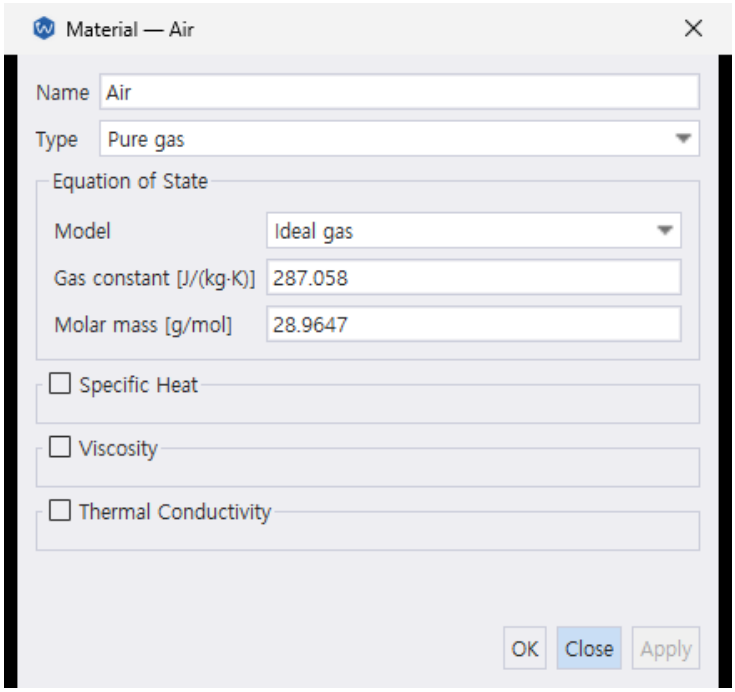


Figure 16.1: The Material dialog for a pure gas. The equation of state is always written; Specific Heat, Viscosity, and Thermal Conductivity are each enabled by their checkbox, and an unchecked group leaves the solver default in place.

Omitted ranges are derived from operating states in the case. If the case has no usable pressure or temperature range, specify the corresponding table range explicitly. A constant-Cp ideal gas uses a closed-form EOS and rejects `eos.table`.

16.5 Specific heat

Key	Unit	Meaning
`specific_heat.type`	—	`constant` (default) or `nasa_polynomial`.
`specific_heat.cp`	`J/(kg·K)`	Constant Cp; for a gas, omitting it gives `3.5 R` (about `1004.7` for default air).
`specific_heat.low.temperature`, `specific_heat.high.temperature`	K	Each range's `[T_min, T_max]`; low maximum must meet high minimum.
`specific_heat.low.coefficients`, `specific_heat.high.coefficients`		Seven NASA coefficients `[a1, …, a7]` for each range.

For a gas, explicit constant Cp must exceed R. NASA polynomial requires both ranges, `eos.type: "ideal_gas"`, and compressible DBNS or PBNS in a `mixed` or `fp64` build. Do not supply `specific_heat.cp` with it. The solver uses a1–a5 for Cp(T); a6 and a7 are stored but their enthalpy/entropy datum constants are not used. Outside the supplied ranges, endpoint Cp is held constant; a table range that exceeds the polynomial range produces a warning.

For constant-density PBNS with `conductivity.type: "constant_prandtl"`, give `specific_heat.cp` explicitly: there is no gas law from which to derive it.

16.6 Viscosity

The viscosity block defaults to `constant` with `value: 1.716e-5` Pa·s. It uses one model at a time; parameters for another model are rejected.

`viscosity.type`	Parameters and defaults	Meaning
`constant`	`value: 1.716e-5` Pa·s	Fixed dynamic viscosity μ.
`sutherland`	`mu_ref: 1.716e-5` Pa·s, `T_ref: 273.15` K, `S: 110.4` K	μ = μ_ref ·(T/T_ref)^1.5 · (T_ref+S)/(T+S).
`power_law`	`mu_ref: 1.716e-5` Pa·s, `T_ref: 273.15` K, `exponent: 1.5`	μ = μ_ref · (T/T_ref)^exponent.

Viscous FR artificial-compressibility cases have no temperature state and require `viscosity.type: "constant"`.

16.7 Thermal conductivity

The conductivity block defaults to `constant` with `value: 0.0242` W/(m·K). It uses one model at a time.

`conductivity.type`	Parameters and defaults	Meaning
`constant`	`value: 0.0242` W/(m·K)	Fixed thermal conductivity k.
`sutherland`	`k_ref: 0.0242` W/(m·K), `T_ref: 273.15` K, `S: 194` K	Temperature-dependent k(T).
`constant_prandtl`	`prandtl: 0.72`	Derives k(T) = μ(T) ·Cp(T) / Pr.

With direct `constant` or `sutherland` conductivity, Pr is derived from the properties; it is not an additional input. `prandtl` belongs inside the `conductivity` block, not at the material's top level. Turbulent Prandtl numbers belong under `models.turbulence.sa` or `models.turbulence.sst`, as described in Physical models.

16.8 Solid properties

For a solid material referenced by a solid domain, `solid.k` must be positive. Steady CHT uses k only; there `solid.rho` and `solid.Cp` set the pseudo-time scale alone. An unsteady run with a solid domain carries the thermal inertia $\rho C_p \, \partial T/\partial t$, so both must be given and positive, and `run_control.time_stepping` must be `bdf1` or `bdf2`.

Key	Unit	Meaning
`solid.k`	`W/(m·K)`	Solid thermal conductivity.
`solid.rho`	`kg/m³`	Solid density; reserved for unsteady conduction.
`solid.Cp`	`J/(kg·K)`	Solid specific heat; reserved for unsteady conduction.

16.9 Reserved inputs

`materials[].reference`, `materials[].table_gen`, and `materials[].radiation` have schema fields but no active material model. Supplying any of their fields is rejected by input validation, including boolean radiation fields set to `false`. Do not confuse material-local `reference` with the separate top-level Reference values.

16.10 JSONC examples

The GUI writes the same material keys for the controls it exposes. For a CLI case or a supported combination not enabled in the GUI, use JSONC. These are fragments; a runnable case also needs mesh, domains, models, and other settings.

An ideal gas with constant Cp and temperature-dependent viscosity:

```
{
  "materials": [{
    "name": "Air", "type": "pure",
    "eos": { "type": "ideal_gas", "gas_constant": 287.058 },
    "specific_heat": { "type": "constant", "cp": 1004.7 },
    "viscosity": { "type": "sutherland", "mu_ref": 1.716e-5, "T_ref": 273.15, "S":
    "conductivity": { "type": "constant_prandtl", "prandtl": 0.72 }
  }]
}
```

An incompressible PBNS material with constant density:

```
{
  "analysis": { "solver": "pbns", "dimension": 3 },
  "models": { "regime": "incompressible" },
  "materials": [{
    "name": "Water", "type": "pure",
    "eos": { "type": "constant_density", "density": 998.2 },
    "specific_heat": { "type": "constant", "cp": 4182.0 },
    "viscosity": { "type": "constant", "value": 1.0e-3 },
    "conductivity": { "type": "constant_prandtl", "prandtl": 7.0 }
  }]
}
```

Peng–Robinson for CO⊠ (compressible DBNS or PBNS, `mixed`/`fp64`):

```
{
  "materials": [{
    "name": "CO2", "type": "pure",
    "eos": {
      "type": "peng_robinson", "molar_mass": 44.01,
      "T_critical": 304.13, "P_critical": 7.3773e6,
      "acentric_factor": 0.225,
      "table": { "resolution": [256, 256] }
    },
    "specific_heat": { "type": "constant", "cp": 850.0 }
  }]
}
```

NASA-polynomial specific heat for air (also usable with compressible PBNS through JSONC):

```
{
  "materials": [{
    "name": "Air", "type": "pure",
    "eos": { "type": "ideal_gas", "gas_constant": 287.87 },
    "specific_heat": {
      "type": "nasa_polynomial",
      "low": {
        "temperature": [200.0, 1000.0],
        "coefficients": [3.56839620, -6.78729429e-4, 1.55371476e-6,
                         -3.29937060e-12, -4.66395387e-13, -1.06234659e3, 3.7158296
      },
      "high": {
        "temperature": [1000.0, 6000.0],
        "coefficients": [3.08792717, 1.24597184e-3, -4.23718945e-7,
                         6.74774789e-11, -3.97076972e-15, -9.95262755e2, 5.95960930
      }
```

```
    }
  }]
}
```

A steady CHT solid material:

```
{
  "materials": [{
    "name": "Steel", "type": "solid",
    "solid": { "k": 16.0, "rho": 7800.0, "Cp": 500.0 }
  }]
}
```

17 Reference values

Moment reference point, coefficient normalization, and Reynolds-number reporting

17.1 What this setting is

Reference values provide the geometric and velocity scales used to convert calculated forces and moments into coefficients designers can compare. Reference area, moment lengths, velocity, and moment origin must be defined consistently with the geometry and test conditions. They are not boundary conditions and do not define the operating state of the flow.

17.2 Configure in the GUI

In the dialog, “does not reach the solver” means that these values do not replace the farfield, inlet, or initialization state and are not used as the flow nondimensionalization state. The numerical-scale exception for PBNS is described below.

17.3 Configure in JSONC

```jsonc
{
  "reference": {
    "origin_moment": [0.25, 0.0, 0.0],
    "length_moment": [1.0, 1.0, 1.0],
    "area": 1.0,
    "pressure": 101325.0,
    "temperature": 288.15,
    "density": 1.225,
    "velocity": 50.0
  }
}
```

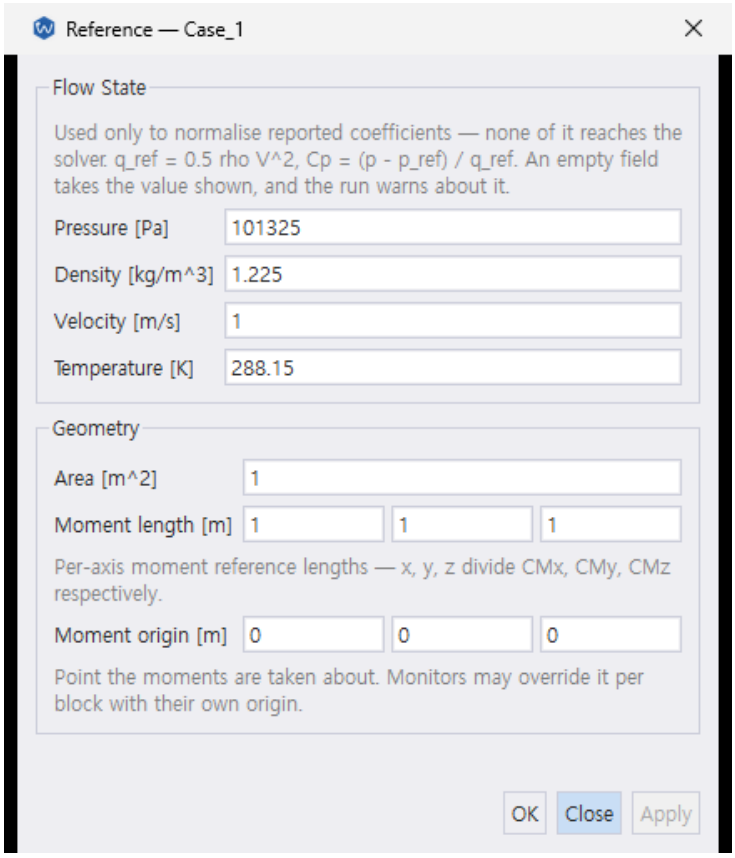


Figure 17.1: The Reference dialog separates coefficient flow-state values from geometric reference values. Untouched fields use the displayed defaults without writing those values to the deck.

> **Important**
>
> **`reference` does not define the operating condition.** The flow state used for boundary conditions and internal nondimensionalization is derived from the farfield, initialization, or inlet/outlet configuration, never from this block. If no operating condition can be established for a compressible, nondimensional run, the solver fails rather than falling back to `reference` with an implied `Mach = 0`.

`reference:` currently has these roles:

1. **Moment reference point and area** — `origin_moment`, `length_moment`, and `area` compute force and moment coefficients (CL, CD, CMx/CMy/CMz).
2. **Coefficient reference state** — density and velocity define `q_ref = 0.5 ρ_ref V_ref²`; pressure defines `Cp = (p − p_ref) / q_ref` and the pressure-force reference.
3. **Thermal reference temperature** — reserved for future Nu/St/HTC output.

Supply these values explicitly. Missing density, velocity, pressure, and temperature use fixed defaults of 1.225 kg/m^3, 1 m/s, 101325 Pa, and 288.15 K and emit `DefaultUsed` warnings. Zero/unset no longer means freestream auto. To copy operating values, run `wake_3d_mixed case.jsonc --validate --format json` (use your case dimension; add `.exe` on Windows) and inspect `operating_reference`. Older GUI screenshots may show the former defaults.

An explicitly supplied `reference.velocity` also contributes to the DBNS low-Mach reference speed. The coefficient default does not supply this numerical scale; hovering rotors should specify a representative tip speed.

> **Note**
>
> For steady PBNS with automatic pseudo-time stepping, `max(reference.length_moment)` and the maximum available flow or `reference.velocity` provide the numerical `L/U` time scale. This affects the convergence time scale, not the physical boundary condition.

17.4 Key definitions

Key	Unit	Description
`origin_moment`	`[x,y,z]` m	Moment reference point (coordinates for CMx/CMy/CMz)
`length_moment`	`[Lx,Ly,Lz]` m	Per-axis moment reference length, normalizing CMx/CMy/CMz respectively. Aircraft convention: `Lx = Lz = span`, `Ly = chord`.

Key	Unit	Description
`area`	m^2	Reference area (CL/CD denominator)
`pressure`	Pa	Reference pressure for Cp and pressure forces (≥ 0).
`temperature`	K	Reserved reference temperature for future Nu/St/HTC output.
`density`	kg/m^3	Positive coefficient reference density.
`velocity`	m/s	Positive coefficient reference velocity (the `q_ref` denominator).

17.5 NACA0012 example (chord = 1 m)

```
{
  "reference": {
    "origin_moment": [0.25, 0.0, 0.0],
    "length_moment": [1.0, 1.0, 1.0],
    "area": 1.0,
    "pressure": 101325.0,
    "temperature": 288.15,
    "density": 1.225,
    "velocity": 50.0
  }
}
```

`origin_moment` is the quarter-chord by convention; `length_moment`'s Ly slot is the chord, and `area` is chord × unit span for this 2D case. The example uses a 50 m/s coefficient reference; it does not prescribe the flow velocity. Match density, pressure, and velocity to the intended freestream or other normalization convention.

17.6 Reynolds-number reporting (not input)

Reynolds number is not an input (R-006). It's a post-processing value, computed fresh each time it's printed rather than stored in the config:

```
Re_derived = ρ_∞ · V_∞ · L / μ_∞(Sutherland)
L = reference.length_moment[1]   (the Ly / chord slot, by convention)
```

To match a Reynolds number from a paper, select `farfield.{P, T}` and Mach (→ V) accordingly.

18 Domains and boundary conditions

Domains, boundary conditions, MRF, and mesh motion

18.1 What this setting is

A domain is a computational region that assigns **physical meaning and material properties, such as fluid or solid**, to a mesh zone. Boundary conditions define how flow enters or leaves at a domain's outer faces (markers), meets a wall, or satisfies symmetry and farfield conditions. As in CFX, think of creating a region first and then attaching conditions to its boundaries; this safely connects mesh names to the analysis model.

18.2 Configure in the GUI

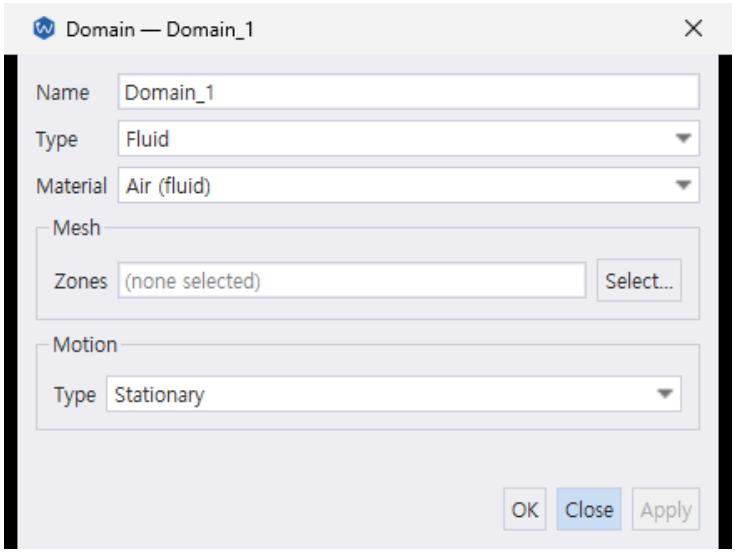


Figure 18.1: The Domain dialog connects a named physical domain to a material and to one or more imported mesh zones. The figure shows it before any zone is selected; use **Select…** to pick them.

Create the domain first, select its material and mesh zones, and then add boundary objects beneath it. The boundary editor changes its fields according to the selected boundary type. The resulting names are used by `domains[].mesh_zones` and `domains[].boundaries[].mesh_markers` in JSONC.

18.3 Configure in JSONC

```jsonc
{
  "domains": [
    {
      "name": "fluid",
      "type": "fluid",
      "material": "Air",
      "mesh_zones": ["fluid_zone"],
      "boundaries": [
        {
          "name": "airfoil", "type": "wall", "mesh_markers": ["AIRFOIL"],
          "momentum": "free_slip", "heat": "adiabatic"
        },
        {
          "name": "farfield", "type": "farfield", "mesh_markers": ["FARFIELD"],
          "mach": 0.8, "pressure": 101325.0, "temperature": 273.15,
          "flow_direction": [1.0, 0.0, 0.0]
        }
      ]
    }
  ]
}
```

Key	Meaning
`type`	`fluid` or `solid` (CHT) — see domain.type below
`material`	References a `materials[].name`
`mesh_zones`	Required array of mesh-zone names — see mesh_zones below
`boundaries`	Array of tagged-union boundary conditions — see Boundary conditions below

> **!** Important
>
> **v0.4.0 breaking changes**
>
> - `locations` → **`mesh_zones`** (domain) / **`mesh_markers`** (boundary, interface)
> - `mesh_zones` and `mesh_markers` are always string arrays (for example, `["Zone1"]`)

18.4 domain.type

Value	Status
`fluid`	production
`solid`	CHT (d0107)

18.5 mesh_zones

Accepts a zone name, scoped name (`scope/zone`), or volume-region name. It is required with MRF as well, for `grid_velocity` assignment.

```
{ "mesh_zones": ["Zone1"] }
{ "mesh_zones": ["ROTOR_ZONE", "ROTOR_TIP"] }
```

A single zone, or several for a domain spanning multiple mesh regions.

18.6 motion (MRF, Multiple Reference Frame)

Assign a rotating coordinate frame to the entire domain:

```
{
  "domains": [
    {
      "name": "rotor", "type": "fluid", "material": "Air",
      "mesh_zones": ["ROTOR_ZONE"],
      "motion": {
        "type": "rotating",
        "omega": -1340.5,
        "axis": [0, 0, 1],
        "center": [0.0, 0.0, 0.0],
```

```
        "mesh_motion": false
      }
    }
  ]
}
```

Key	Unit	Meaning
`type`	—	`stationary` (default) or `rotating`
`omega`	rad/s	Signed rotation rate (`rpm` × 2π/60)
`axis`	—	Rotation axis (unit vector, auto-normalized)
`center`	m	Rotation center
`mesh_motion`	—	`false` (default) = MRF frozen rotor — grid stays fixed, only the reference-frame transform is applied (steady, pairs with `interfaces[].type: "frozen_rotor"` or `ggi`). `true` = sliding mesh — the grid is actually rotated each step (unsteady dual-time only, pairs with `interfaces[].type: "sliding"`).

18.7 Boundary conditions (nine-type tagged union)

Category	`type`	Meaning
topology	`wall`	Wall surface
topology	`farfield`	Farfield boundary (characteristic-based)
topology	`symmetry`	Symmetry plane
topology	`internal`	Internal interface face (no BC treatment)
inlet	`pressure_inlet`	Total-pressure inlet (P0, T0, `flow_direction`)

Category	type	Meaning
inlet	velocity_inlet	Velocity-component inlet
inlet	mass_flow_inlet	Mass-flow inlet
outlet	pressure_outlet	Back-pressure outlet
outlet	mass_flow_outlet	Mass-flow outlet

> **Note**
>
> **Interface boundaries are separate**
> Define periodic, stage, frozen_rotor, sliding, and cht in the interfaces: section.

18.7.1 Wall

```
{
  "name": "airfoil", "type": "wall", "mesh_markers": ["AIRFOIL"],
  "momentum": "free_slip",
  "heat": "adiabatic",
  "wall_treatment": "resolved",
  "motion": { "type": "stationary" }
}
```

Key	Meaning
momentum	free_slip (Euler) or no_slip (NS/RANS)
heat	adiabatic, isothermal (add temperature [K]), or heat_flux (add heat_flux [W/m^2])
wall_treatment	resolved or automatic. DBNS RANS no-slip walls default to automatic except axisymmetric cases; other routes default to resolved.
motion	Tagged union, NS no-slip only — see Wall motion below

> **Note**
>
> **Wall function (per-marker)**
> Setting wall_treatment: "automatic" enables the model-specific wall law on that marker. Configure shared constants in models.wall_treatment. The old wall_function key and y_plus_min threshold were removed. Resolved treatment re-

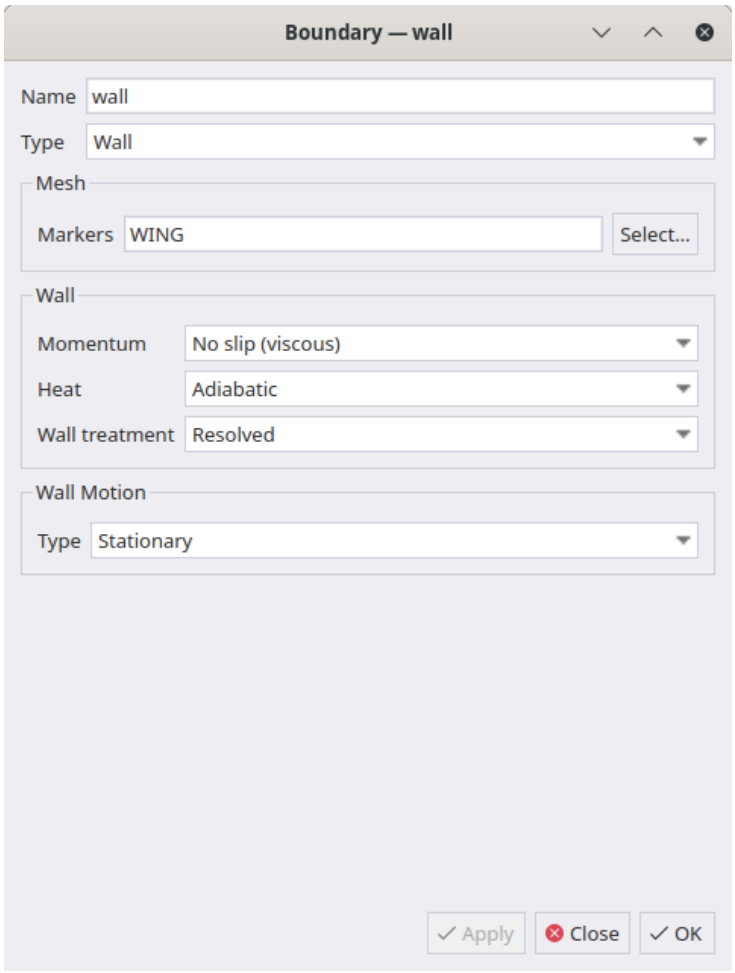


Figure 18.2: A viscous wall boundary connected to the WING mesh marker, with no-slip momentum, adiabatic heat transfer, resolved wall treatment, and stationary wall motion.

quires a mesh that resolves the viscous sublayer.

18.7.1.1 Wall motion (NS no-slip only, tagged union)

```
"motion": { "type": "stationary" }
```

```
"motion": { "type": "translating", "velocity": [10.0, 0.0, 0.0] }
```

```
"motion": { "type": "rotating", "omega": 157.08, "axis": [0, 0, 1], "center": [0, 0
```

type	Fields	Unit
`stationary` (default)	—	—
`translating`	`velocity` — `[x, y, z]` vector	m/s
`rotating`	`omega` (signed), `axis` (unit vector), `center`	rad/s, —, m

Wall velocity components are absolute-frame values. Omitting `motion` on the no-slip wall of a rotating domain means the wall turns with the domain; give `stationary` for a fixed casing.

18.7.2 Farfield

> **!** Important
>
> **`farfield` is for remote boundaries only.**
> Use `farfield` only on faces where a uniform freestream assumption is valid. Using it on a face crossed by a boundary layer or shear flow, such as a wind-tunnel outlet, creates a spurious pressure gradient. In one flat-plate validation, near-wall `dP/dx` reached −73 Pa/m and downstream `Cf` was six percentage points too high.
> For wind-tunnel-style cases, use `pressure_inlet` (total conditions) with `pressure_outlet` (static pressure).

```
{
  "name": "farfield", "type": "farfield", "mesh_markers": ["FARFIELD"],
  "mach": 0.8,
  "pressure": 101325.0,
  "temperature": 273.15,
  "aoa": 1.25,
```

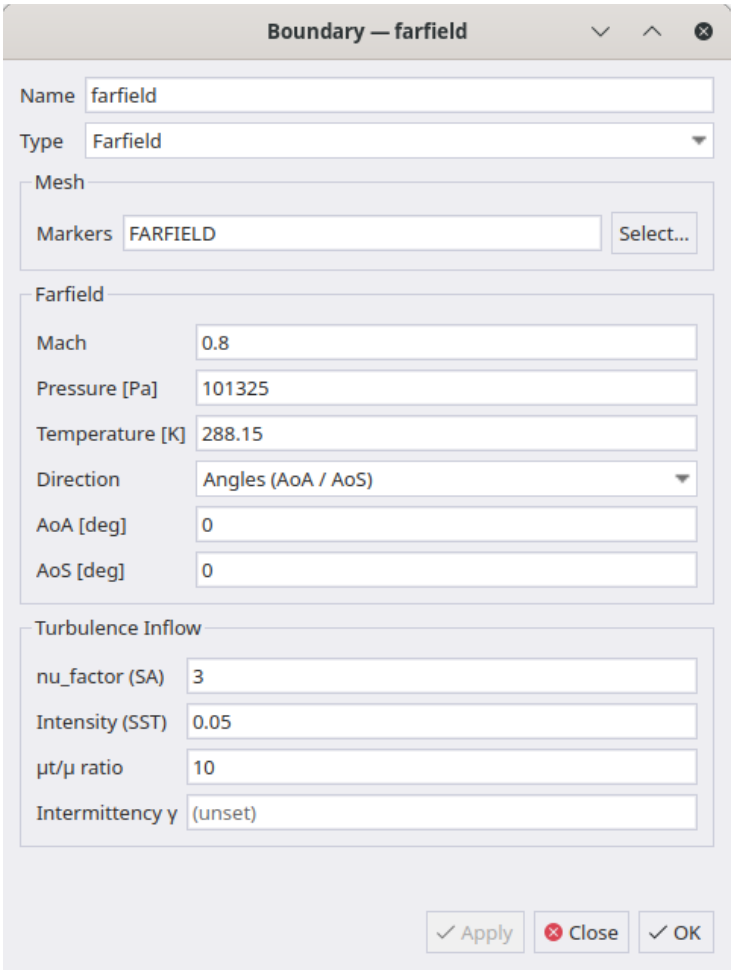


Figure 18.3: A farfield boundary connected to the FARFIELD mesh marker. The dialog writes freestream Mach, pressure, temperature, direction, and turbulence-inflow values.

```
  "aos": 0.0,
  "turbulence": {
    "nu_factor": 3.0,
    "intensity": 0.05,
    "eddy_viscosity_ratio": 10.0,
    "intermittency": 1.0
  }
}
```

Key	Unit	Meaning
`mach`	—	Freestream Mach number
`pressure`, `temperature`	Pa, K	Static reference state (density is derived through the EOS)
`aoa`	°	Angle of attack, −180 to 180: 2D rotates in x–y, 3D in x–z
`aos`	°	Sideslip angle, −90 to 90; omit or set to 0 in 2D
`flow_direction`	—	`[x, y, z]` unit vector, normalized on read; the alternative to `aoa`/`aos`
`turbulence.nu_factor`	—	SA only: `ν̃inf = nu_factor · ν_lam`
`turbulence.intensity`	—	SST/k-ω only: freestream turbulence intensity (`0.05` = 5%)
`turbulence.eddy_viscosity_ratio`		Freestream turbulent/laminar viscosity ratio `μ_t/μ`; `length_scale` [m] and `hydraulic_diameter` [m] are alternatives, and only one may be given
`turbulence.intermittency`	—	Transition intermittency γ (LM transition model)

Flow direction: Give `aoa` (with `aos` in 3D) when you have angles, or `flow_direction` when you already have the vector. One of the two is required in the Mach form — a far field without either is rejected. A non-zero `aoa`/`aos` overrides `flow_direction`, so supply one form, not both. Both are plain numbers here; expressions apply to the state values, not to the direction.

> **Note**
>
> **FR-only fields**
> Two further fields apply only under `analysis.solver=fr`: `characteristic` (compressible path — `true` switches this farfield to a low-reflection characteristic Riemann invariant BC instead of a fixed freestream ghost state; default `false`), and, only under `models.regime=incompressible` (the AC track), `velocity` (`|V|`, consumed by the AC characteristic BC), `ac_zeta`, and `ac_niters` (AC Newton-iteration count for the characteristic solve; default 4).

18.7.2.1 Direct primitive-variable input

When `rho`, `u`, `v`, `w`, and `p` are provided, these values are used **instead of** deriving the state from `(Mach, P, T)`; supplying both forms on one boundary is rejected. Each value may be a constant or an expression in `x`, `y`, `z`, and `t`. If any value is an expression, the boundary condition becomes space- and time-dependent and is reevaluated at every stage.

```
{
  "name": "inflow", "type": "farfield", "mesh_markers": ["INLET"],
  "rho": 1.0,
  "u":   { "type": "expression", "expr": "1.0 + 0.1*sin(2*pi*t)" },
  "v":   0.0,
  "w":   0.0,
  "p":   1.0
}
```

> **Warning**
>
> With this form, the farfield is excluded from nondimensional reference-state extraction. The `reference` block or another farfield must define the reference.

18.7.3 Symmetry

```
{ "name": "sym", "type": "symmetry", "mesh_markers": ["SYM"] }
```

18.7.4 Internal

```
{ "name": "rs_interface", "type": "internal", "mesh_markers": ["INTERFACE"] }
```

Internal mesh interface face: no BC treatment and no `vertex_normal` calculation.

18.7.5 `pressure_inlet`: total-pressure inlet

```
{
  "name": "inlet", "type": "pressure_inlet", "mesh_markers": ["INLET"],
  "formulation": "dirichlet",
  "total_pressure": 120000.0,
  "total_temperature": 300.0,
  "direction_type": "cartesian",
  "flow_direction": [1.0, 0.0, 0.0],
  "turbulence": { "intensity": 0.03, "eddy_viscosity_ratio": 10.0 }
}
```

Key	Unit	Meaning
`formulation`	—	`dirichlet` (default, algebraic), `riemann` (characteristic), or `giles` (1D NRBC)
`total_pressure`, `total_temperature`	Pa, K	Total (stagnation) conditions `P0`, `T0`
`direction_type`	—	`cartesian` (default, reads `flow_direction`), `normal` (inward face normal), or `cylindrical` (reads the `cyl_*` keys, see below). `aoa`/`aos` are rejected on an inlet
`flow_direction`	—	`[x, y, z]` unit vector; needed when `direction_type` is `cartesian`
`relaxation`	—	Giles NRBC relaxation factor, only used with `formulation: "giles"`

18.7.5.1 Cylindrical inlet (turbomachinery)

```
{
  "name": "inlet", "type": "pressure_inlet", "mesh_markers": ["INLET"],
  "total_pressure": 120000.0,
  "total_temperature": 300.0,
  "direction_type": "cylindrical",
```

```
  "cyl_axis": [0, 0, 1],
  "cyl_center": [0, 0, 0],
  "cyl_axial": 1.0,
  "cyl_tangential": 0.5,
  "cyl_radial": 0.0
}
```

`direction_type` is `cartesian` (reads `flow_direction`), `normal` (face-normal), or `cylindrical`. The cylindrical form takes `cyl_axis` through `cyl_center` plus the `cyl_axial`/`cyl_tangential`/`cyl_radial` **component ratios** — not m/s, as the magnitude comes from the inlet's own total conditions. Positive `cyl_tangential` is right-handed about `cyl_axis`.

18.7.6 `velocity_inlet`: velocity-magnitude inlet

```
{
  "name": "inlet", "type": "velocity_inlet", "mesh_markers": ["VEL_INLET"],
  "velocity": 50.0,
  "flow_direction": [1.0, 0.0, 0.0],
  "pressure": 101325.0,
  "temperature": 300.0
}
```

Key	Unit	Meaning
`velocity`	m/s	Inflow speed, 0 or greater. Required
`flow_direction`	—	`[x, y, z]` unit vector. Required
`pressure`, `temperature`	Pa, K	Optional static reference

Under DBNS `velocity` and `temperature` also accept expressions, in `x, y, z` for a per-vertex profile. PBNS takes numbers only.

18.7.7 `mass_flow_inlet`: mass-flow inlet

```
{
  "name": "inlet", "type": "mass_flow_inlet", "mesh_markers": ["MF_INLET"],
  "mass_flow": 2.5,
  "total_temperature": 300.0,
  "direction_type": "cartesian",
  "flow_direction": [1.0, 0.0, 0.0]
}
```

`mass_flow` [kg/s] and `total_temperature` [K] are both required; the direction uses the same `direction_type` and `flow_direction` / `cyl_*` keys as `pressure_inlet`. Total pressure is an outcome here, so the case still has a solution when the downstream chokes. DBNS, `ideal_gas`, and `formulation: "dirichlet"` only.

18.7.8 `pressure_outlet`: back-pressure outlet

```
{
  "name": "outlet", "type": "pressure_outlet", "mesh_markers": ["OUTLET"],
  "formulation": "dirichlet",
  "back_pressure": 101325.0,
  "supersonic": false
}
```

Key	Unit	Meaning
`formulation`	—	`dirichlet` (default), `riemann`, or `giles` — same meaning as `pressure_inlet` above
`back_pressure`	Pa	Static back pressure
`supersonic`	—	Supersonic-outlet treatment (extrapolate everything, ignore `back_pressure`)
`relaxation`	—	Giles NRBC relaxation factor, only used with `formulation: "giles"`

18.7.8.1 Outlet averaging (CFX-style)

```
{
  "name": "outlet", "type": "pressure_outlet", "mesh_markers": ["OUTLET"],
  "back_pressure": 101325.0,
  "averaging": {
    "type": "circumferential",
    "blend": 0.0,
    "n_span": 32,
    "axis": [0, 0, 1],
    "center": [0, 0, 0]
  }
}
```

Key	Meaning
`type`	`average` or `circumferential`
`blend`	$\beta \in [0,1]$: `0` = uniform, `1` = CFX-style
`n_span`	Number of circumferential bins (`circumferential` only)
`axis`, `center`	Rotation axis and center for the circumferential average

18.7.9 `mass_flow_outlet`: mass-flow outlet

```
{
  "name": "outlet", "type": "mass_flow_outlet", "mesh_markers": ["MF_OUTLET"],
  "mass_flow": 2.5
}
```

`mass_flow` [kg/s] is the target rate.

19 Interfaces

periodic, stage, frozen_rotor, sliding, GGI, CHT

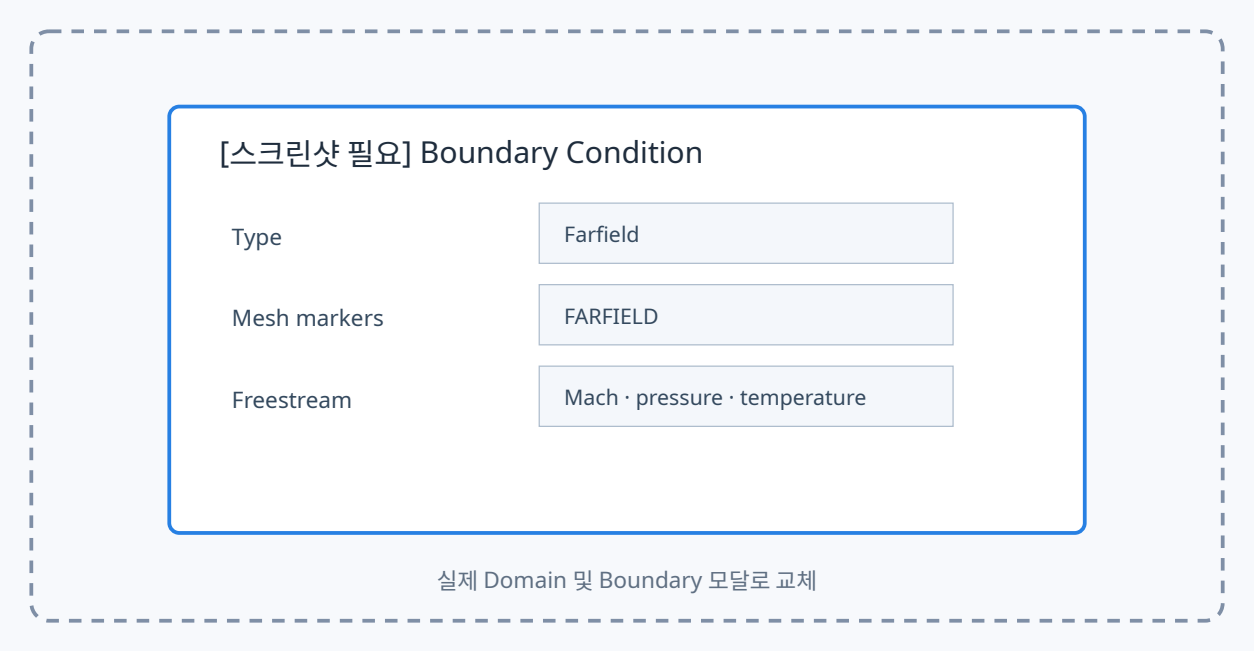


Figure 19.1: Placeholder for the GUI Interface panel. Replace with the actual screen for selecting the interface type and markers on both sides.

19.1 What this setting is

An interface defines faces from two regions as connected surfaces that exchange data rather than as ordinary boundaries. Use one when the solution on one face must continue onto another, such as for periodic flow, stationary and rotating regions, nonconformal meshes, or fluid–solid heat transfer. Unlike a simple inlet, outlet, or wall, an interface requires both domains and markers on both sides.

`interfaces:` defines a connection between two regions, or two faces of the same region. The parser automatically injects boundary conditions into the domains, so do not duplicate them in `domains[].boundaries`.

```
{
  "interfaces": [
    {
      "name": "passage_periodic", "type": "periodic",
      "side1_domain": "fluid",
      "side1_mesh_markers": ["PER1"],
      "side2_domain": "fluid",
      "side2_mesh_markers": ["PER2"],
      "periodicity": "rotational",
      "axis": [0, 0, 1],
      "angle": 30.0
    }
  ]
}
```

`side1_domain`/`side2_domain` plus their `mesh_markers` identify the two connected faces (same domain or different domains). `angle` [°] applies to `periodicity: "rotational"`; see below for the translational form.

19.2 type

Value	Meaning
`periodic`	Periodic boundary (rotational/translational)
`stage`	Mixing plane (steady turbomachinery)
`frozen_rotor`	Coordinate-frame transformation only (no averaging)
`sliding`	Sliding mesh (transient rotor–stator)
`ggi`	General nonconformal mesh connection between stationary domains
`cht`	Fluid–solid conjugate heat transfer

19.3 GGI

`ggi` connects nonconformal meshes in different domains without rotating-frame transformation. Set it as the interface `type`; there is no `mesh_connection` suboption. `ggi`, `frozen_rotor`, and `sliding` all use the same nonconformal supermesh engine. They differ only in how the two sides move: `ggi` joins stationary domains, `frozen_rotor` holds a fixed relative position, and `sliding` rotates the mesh with time.

```
{
  "interfaces": [{
    "name": "zone_junction",
    "type": "ggi",
    "side1_domain": "left",
    "side1_mesh_markers": ["INTERFACE"],
    "side2_domain": "right",
    "side2_mesh_markers": ["INTERFACE"]
  }]
}
```

19.4 Periodic (single passage)

```
{
  "interfaces": [
    {
      "name": "passage", "type": "periodic",
      "side1_domain": "fluid",
      "side1_mesh_markers": ["PER1_2D"],
      "side2_domain": "fluid",
      "side2_mesh_markers": ["PER2_2D"],
      "periodicity": "rotational",
      "axis": [0, 0, 1],
      "angle": 30.0
    }
  ]
}
```

`periodicity`	Fields	Unit
`rotational`	`axis` (unit vector), `angle`	—, °
`translational`	`translation` (displacement vector)	m

19.5 Stage / Mixing plane

Connects different fluid domains through spanwise averaging for steady turbomachinery.

```
{
  "interfaces": [
    {
```

```
      "name": "stator_rotor", "type": "stage",
      "side1_domain": "stator",
      "side1_mesh_markers": ["STATOR_OUTLET"],
      "side2_domain": "rotor",
      "side2_mesh_markers": ["ROTOR_INLET"],
      "axis": [0, 0, 1],
      "n_span": 20,
      "mixing_method": "area",
      "pitch_change": "automatic",
      "giles_relax": 0.1
    }
  ]
}
```

Key	Meaning
`axis`	Machine rotation axis (unit vector)
`n_span`	Number of span-wise averaging bands, hub → shroud
`mixing_method`	`algebraic`, `area`, `mass`, or `total` — the spanwise-averaging weighting
`pitch_change`	`automatic` (default) or `specified` (add `pitch_ratio`, e.g. `1.278` for a stator/rotor blade-count ratio of 46/36)
`giles_relax`	Giles NRBC under-relaxation applied to the band-averaged values (default `0.1`) — see below

`giles_relax` is the relaxation factor used to apply span-band averages to boundary conditions. A larger value may converge faster but can cause oscillation or divergence at the stage boundary.

19.6 Frozen rotor

Transforms only the coordinate frame, with no averaging. It preserves asymmetric results, making it useful for stall diagnostics and local-effect analysis.

```
{
  "interfaces": [
    {
      "name": "stator_rotor_frozen", "type": "frozen_rotor",
      "side1_domain":   "stator",
```

```
      "side1_mesh_markers": ["STATOR_OUTLET"],
      "side2_domain":   "rotor",
      "side2_mesh_markers": ["ROTOR_INLET"]
    }
  ]
}
```

19.7 Sliding mesh (transient)

The mesh physically rotates with time integration (`analysis.time: unsteady` is required). It is always nonconformal.

```
{
  "interfaces": [
    {
      "name": "stator_rotor_sliding", "type": "sliding",
      "side1_domain": "stator",
      "side1_mesh_markers": ["STATOR_OUTLET"],
      "side2_domain": "rotor",
      "side2_mesh_markers": ["ROTOR_INLET"]
    }
  ]
}
```

Sliding is inherently nonconformal, and it pairs with `domains[].motion.mesh_motion: true` on the moving domain. The two sides form one solve unit, and the intersection is regenerated every physical step.

19.8 CHT (Conjugate Heat Transfer)

Thermally couples a fluid domain and a solid domain. The solid-side domain uses `type: "solid"` and references the `solid` block in `materials`.

```
{
  "interfaces": [
    {
      "name": "blade_cht", "type": "cht",
      "side1_domain": "R1",
      "side1_mesh_markers": ["R1_BLADE_UPPER"],
      "side2_domain": "Blade_Solid",
      "side2_mesh_markers": ["BLADE_INNER"]
```

```
    }
  ]
}
```

A CHT interface takes **no extra keys**. The two walls are always paired through the supermesh, of which a conformal mesh is a special case. The solid side receives the fluid wall's conductance `k_eff / d_n` as a Robin coefficient; there is no relaxation factor to set. The fluid-side marker is an isothermal no-slip wall, and its `temperature` supplies only the value used before the first exchange.

The run writes `cht_balance.csv`, reporting `Q_side1`, `Q_side2`, the mapping integral, and `max|ΔT_w|`.

20 Numerical methods

DBNS, PBNS, FR, CFL, time integration, and linear solvers

20.1 What this setting is

Numerical methods determine how the selected equations are discretized and solved. Start with `analysis.solver`, then read the corresponding block below. Each table explains where a knob belongs, when it is active, and what changing it affects. Change one numerical setting at a time and compare residuals and the quantities you monitor.

`numerical_method` contains numerical options activated by `analysis.solver`. Do not mix solver-specific blocks with the linear solver shared by DBNS and PBNS.

20.2 Configure in the GUI

The Desktop editor covers the DBNS route through four tabs. The dialog says so in its header: PBNS and FR editors are not yet available, so those solvers are configured in JSONC. CFL belongs to **Run Control**, matching its `run_control` location in the deck, and therefore does not appear here.

Tab	GUI control	Input key
Time Integration	Time integration	`numerical_method.dbns.time_integration`
Time Integration	Update relaxation	`numerical_method.dbns.nonlinear.relaxation`
Convective	Scheme	`numerical_method.dbns.convective.scheme`
Convective	Spatial order	`numerical_method.dbns.convective.spatial_orde`
Convective	MUSCL basis	`numerical_method.dbns.convective.roe.reconstr`
Convective	Entropy fix	`numerical_method.dbns.convective.roe.entropy_`
Convective	Entropy fix (acoustic) / (shear)	`…roe.entropy_fix_acoustic`, `… roe.entropy_fix_shear`
Convective	Gradient	`numerical_method.dbns.gradient`
Limiter	Type	`numerical_method.dbns.convective.limiter.type`

Tab	GUI control	Input key
Linear Solver	Type / Error / Max iterations	`numerical_method.linear_solver.{type, error, max_iter}`
Linear Solver	Preconditioner Type / Sweeps / Omega	`numerical_method.linear_solver.preconditioner sweeps, omega}`

The scheme-specific box under **Convective** is refilled when the scheme changes, so the Roe options above appear only while `roe` is selected. The **Preconditioner** box edits the `flow` block; the `turb`, `energy`, and `solid` blocks are JSONC-only.

20.3 Configure in JSONC

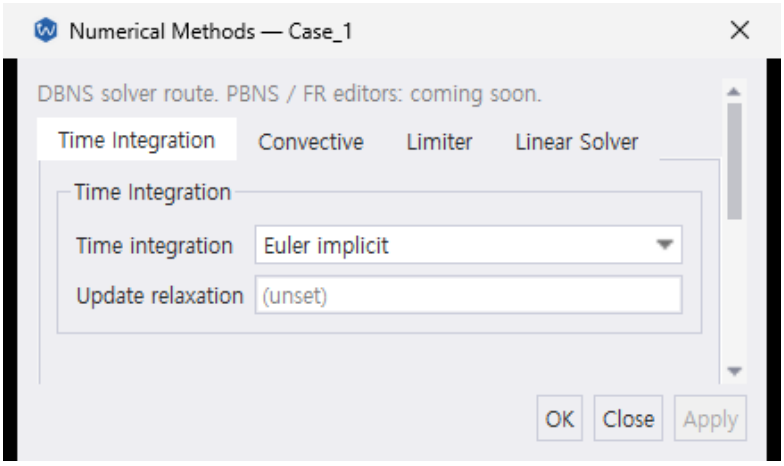
Solver	Active block	Purpose
`dbns`	`numerical_method.dbns`	Density-based FVM
`pbns`	`numerical_method.pbns`	Pressure-based 3D solver
`fr`	`numerical_method.fr`	Flux Reconstruction, unsteady only
`dbns`, `pbns`	`numerical_method.linear_solver`	Shared settings for implicit linear systems

Do not use legacy examples of generic `numerical_method.convective` based on `analysis.discretization` as a reference for the current input structure.

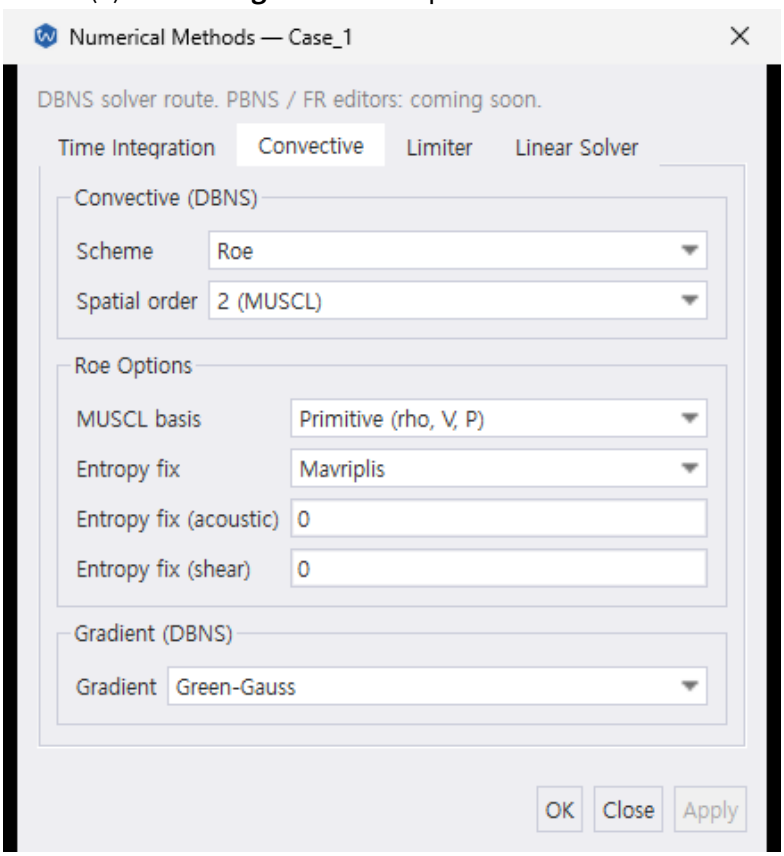
20.4 Where step-size knobs live

Physical-time schemes, stopping limits, and pseudo-time policies are explained in Run control and convergence. They are separate from the numerical-method blocks on this page:

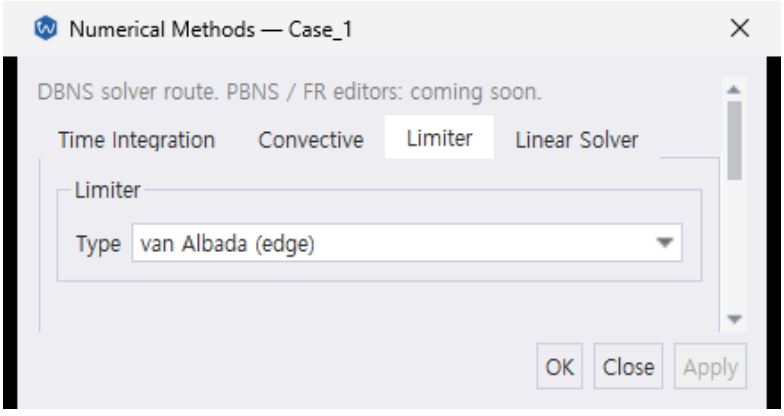
Solver	Step-size knob	When it applies
DBNS	`run_control.cfl`	Steady pseudo-time, unsteady dual-time inner iterations, or physical Δt with `standard` explicit stepping.
PBNS	`run_control.pseudo_time`	Steady pseudo-time only.
PBNS	`numerical_method.pbns.timestep.cfl_max`, `.cfl_flow`, `.cfl_diffusion`	Caps on the evolving physical Δt in unsteady analysis.
FR	`run_control.pseudo_time`	Dual-time inner iterations.
FR	`run_control.timestep_control`	Adaptive physical Δt for explicit `rk34`/`rk45`.



(a) **Time Integration** — the pseudo-time scheme.



(b) **Convective** — flux scheme, spatial order, scheme-specific options, and gradient.



(c) **Limiter** — the bound on second-order reconstruction.

> **Warning**
>
> For a PBNS case, putting CFL under `run_control.cfl` is **silently ignored** — that's the DBNS axis — and PBNS runs with its unset default. Use `run_control.pseudo_time` for steady analysis or `pbns.timestep.cfl_acoustic/cfl_flow/cfl_diffusion` for unsteady analysis.

20.5 DBNS

20.5.1 Convective terms

```
{
  "numerical_method": {
    "dbns": {
      "convective": {
        "scheme": "roe",
        "spatial_order": 2,
        "limiter": { "type": "venkatakrishnan", "coeff": 0.3 }
      }
    }
  }
}
```

The first decision is the convective flux. Put scheme-specific keys such as `jacobian`, Roe entropy fix, and JST `ad_coeff` in the subblock named for the selected scheme. The keys in this table live under `numerical_method.dbns.convective` unless a different path is shown.

Key	Values / default	What it controls
`scheme`	`jst`, `roe`, `ausm`, `ausm_plus_up`, `ausm_plus_up2`, `hllc`, `slau`, `slau2`	Numerical flux at cell faces. Change it when comparing shock capture, dissipation, or low-Mach behavior. `roe_turkel` is a scheme of its own, not an option of `roe`.
`spatial_order`	1 or 2	First-order reconstruction or second-order MUSCL reconstruction. Second order uses the reconstruction gradient and limiter.
`limiter.type`	`none`, `venkatakrishnan`, `barth`, `nishikawa_r3`, `venkatakrishnan_wang`, `van_albada_edge`	Bounds the second-order reconstruction near steep gradients. `none` removes this bound; `van_albada_edge` limits each edge separately.

Key	Values / default	What it controls
`limiter.coeff`	`0.05` in the input reference	Venkatakrishnan limiter strength: a larger value weakens limiting. Ignored by `van_albada_edge`.
`limiter.elem_length`	`0.1` m in the input reference	Characteristic mesh length used with `limiter.coeff`; choose a representative mesh scale when using the Venkatakrishnan limiter. Ignored by `van_albada_edge`.
`limiter.freeze_iter`	`-1` (disabled)	Positive outer iteration after which the nodal limiter stops updating. JFNK also freezes it when Newton activation begins. Ignored by `van_albada_edge`.
`umuscl_kappa`	`0.0`; allowed `[-1, 1)`	Mean-flow U-MUSCL face bias. Keep the default unless comparing reconstruction schemes; turbulence reconstruction is unaffected.

`numerical_method.dbns.gradient` is the physical gradient. A separate `convective.reconstruction_` changes only MUSCL reconstruction:

Key	Default	What it controls
`dbns.gradient`	`weighted_least_squares`	Physical gradient for viscous fluxes, turbulence, derived fields, and (unless overridden) MUSCL. Choose `weighted_least_squares` or `green_gauss`; unweighted `least_squares` is not accepted here.
`convective.reconstruction_gradient`	Reuses `dbns.gradient`	MUSCL gradient only: `weighted_least_squares`, `green_gauss`, or `least_squares`. The unweighted choice is rejected with JST and first-order reconstruction.

Omitting `reconstruction_gradient` reuses the physical gradient and avoids a second gradient field. For `umuscl_kappa`, `0.0` is the current upstream-biased second-order form; changing it alters the face state and should be evaluated against the same mesh and convergence target.

```
{
  "numerical_method": {
    "dbns": {
      "gradient": "weighted_least_squares",
```

```
      "convective": {
        "scheme": "roe",
        "roe": {
          "entropy_fix_type": "mavriplis",
          "entropy_fix_acoustic": 0.001,
          "entropy_fix_shear": 0.0,
          "jacobian": "roe_exact"
        }
      }
    }
  }
}
```

Key	Meaning
`entropy_fix_type`	`harten_hyman` or `mavriplis`
`entropy_fix_acoustic`	Acoustic-wave entropy fix; default `0.0` (off).
`entropy_fix_shear`	Contact/shear-wave fix; default `0.0`. Use for strong-shock carbuncle control, not as a general wall-flow adjustment.
`reconstruction`	`primitive` (default) or `eos` for Roe-family MUSCL face states. The EOS option derives face density from extrapolated pressure and temperature; validate shock convergence before changing it.
`jacobian`	`scalar_dissip`, `roe_exact`, `hllc_exact`, `ausm_ad`, `ausm_fd`, or `turkel_exact`. Default `roe_exact`; `convective.roe_turkel.jacobian` has the same values and defaults to `turkel_exact`

JFNK freezes the nodal limiter when Newton activation begins so its matrix-free linear operator stays fixed. `van_albada_edge` has no stored nodal limiter, so its `coeff`, `elem_length`, and `freeze_iter` keys are ignored.

20.5.2 Low-Mach dissipation and time stepping

Low-Mach treatment is one block. Select `scheme: "roe_turkel"` for the preconditioned dissipation, and set the single β that the dissipation, the preconditioned time step, and the Γ⁻¹ term all share in `numerical_method.dbns.low_mach`.

The reference Mach number `M_ref` is `max(freestream Mach, reference.velocity / freestream sound speed)`. Supply a representative speed in `reference.velocity` for a hovering rotor, such as tip speed.

All four keys below are inside `numerical_method.dbns.low_mach`.

Key	Default	Meaning
`beta_floor`	`1.0`	Lower β bound as a multiple of `M_ref`, not an absolute Mach. An absolute floor `z0` is `beta_floor = z0 / M_ref`. `0` follows the local Mach alone.
`beta_cap`	`0.2`	Upper β bound. The `beta_floor × M_ref` bound wins if it exceeds this cap; β reaches 1, meaning no preconditioning, only when `beta_floor × M_ref ≥ 1`.
`time_step`	`true` under `roe_turkel`, else `false`	Use the preconditioned spectral radius for the local time step.
`time_derivative`	`false`	Weiss–Smith Γ⁻¹ time-derivative term. Experimental: no measured benefit, and it diverges without `time_step`.

```
{
  "numerical_method": {
    "dbns": {
      "convective": {
        "scheme": "roe_turkel",
        "spatial_order": 2,
        "limiter": { "type": "venkatakrishnan", "coeff": 0.05 },
        "roe_turkel": { "jacobian": "turkel_exact" }
      },
      "low_mach": { "beta_floor": 1.0, "beta_cap": 0.2, "time_step": true }
    }
  }
}
```

Outside `roe_turkel`, a preconditioned time step remains experimental and raises a warning.

> **Note**
>
> The scheme-local `low_mach_dt`, `weiss_smith`, `low_mach_beta_max`, `low_mach_mach_lim`, and Roe–Turkel `beta_min`/`beta_max` keys are rejected.

20.5.3 Time integration

```
{
  "numerical_method": {
    "dbns": {
      "time_integration": "euler_implicit"
    }
  }
}
```

Canonical	Alias	Purpose
`euler_implicit`	`implicit`	Implicit pseudo-time
`runge_kutta_explicit`	`explicit`, `explicit_rk`	Explicit RK, used as a **steady-state smoother** — 1st-order accurate in time, not for time-accurate unsteady runs
`rk4_explicit`	`rk4`, `classical_rk4`, `classical_rk4_explicit`	Classical 4-stage explicit RK, **4th-order accurate in time** — use this one for a time-accurate `standard` unsteady run
`jfnk`	`nk`	Jacobian-free Newton–Krylov (matrix-free; the assembled Jacobian is used only as the preconditioner). Renamed from `newton_krylov` — the old name no longer parses

> **Warning**
>
> `runge_kutta_explicit` and `rk4_explicit` are **not aliases of each other** — they're different schemes with different time accuracy. Picking `runge_kutta_explicit` for an unsteady `standard`-time-stepping run gets you a 1st-order-in-time result even though the spatial discretization may be 2nd-order; use `rk4_explicit` when temporal accuracy matters.

20.5.4 Mean-flow viscous flux

The former `numerical_method.dbns.viscous_gradient` key was removed without an alias. Mean-flow viscous-flux selection now belongs to `expert_parameters.mean_flow_viscous_scheme` it is intentionally not covered in this user-facing reference yet. Do not retain the old key in a case—it is rejected as an unknown key.

20.5.5 CFL control

DBNS CFL policy belongs in `run_control.cfl`, not `numerical_method`. It was moved to run control because the schedule x-axis is outer iteration. See that page for the `constant`, `schedule`, and `adaptive` types and their fields.

> **!** Important
>
> `numerical_method.dbns` no longer accepts a `cfl` or `cfl_solid` key at all — the old scalar shorthand fallback was removed on 2026-08-21. Setting CFL is exclusively `run_control.cfl`'s job now; putting either key here fails with an unknown-key error rather than being silently accepted.

20.5.6 Nonlinear control

HANIM is a coupled flow–turbulence nonlinear workflow. Enable it only with `time_integration: "jfnk"` and `nonlinear_control.method: "hanim"`. `run_control.cfl.type` remains a CFL policy; it does not select HANIM.

Key	Default	Use
`method`	—	`hanim` enables the workflow; JFNK is required.
`quadratic_search`	`single`	`single` fits the quadratic once and then accepts with a frozen CFL or fails; `repeated` keeps the older repeated-reduction form.
`residual_weight`	`length`	Acceptance-norm weighting: `none`, `volume_capped`, `length`, or `wall_distance_turb`.
`weight_viscous_mf` / `_t`	`1.0`	Multipliers on the norm weight of no-slip wall points, mean flow and turbulence separately.
`weight_symmetry_mf` / `_t`	`1.0`	Same, for symmetry planes and free-slip walls.
`weight_farfield_mf` / `_t`	`0.0`	Same, for farfield points — `0` excludes them from the norm.
`rho_min_factor`, `p_min_factor`	`1e-2`	Trial-state density and pressure floors relative to the reference state.
`cfl_start`	`1.0`	Starting CFL. HANIM has no startup phase by default and owns the CFL from iteration 1.
`cfl_incr`, `cfl_decr`	`2.0`, `0.1`	CFL multipliers after a full-step success or failed trial.

Key	Default	Use
`cfl_min`, `cfl_max`	`1e-12`, `1e10`	Permitted HANIM CFL range. Reaching `cfl_min` on a failed attempt stops the solve.
`gcr_max_directions`, `gcr_target_reduction`	`5`, `0.92`	GCR search budget and residual target.
`max_scalings`	`20`	`quadratic_search: repeated` only — safety cap on trial-step reductions.
`precond_target_reduction`	`0.5` when HANIM omits it	Mean-flow target for a residual-controlled preconditioner; `0` disables it.
`precond_target_reduction_t`	`0`	Turbulence-block target; `0` follows the mean-flow value.
`precond_min_sweeps`, `precond_max_sweeps`	`1`, `300`	Sweeps performed before the first check, and the cap per solve.
`precond_divergence_factor`	`100`	Residual-growth factor that marks the preconditioner as failed.

The `precond_*` keys belong in `nonlinear_control`, not `linear_solver.preconditioner`. They replace fixed `preconditioner.flow.sweeps` when enabled and support Jacobi, DILU, and CPU-only SGS. Reaching the sweep cap bounds the work; it is not by itself a solver error.

Weighting and the two blocks. The GCR inner product and the acceptance norm share one set of point weights, built once as `node_weighting(cell) × boundary multiplier`. Mean flow and turbulence get their own weights throughout, which is how the reference implementation balances the two blocks. `residual_weight: wall_distance_turb` is the one mode where the two differ by formula rather than by multiplier — mean flow is weighted uniformly and turbulence interior points by $1/d^3$ with d the wall distance. It needs a viscous case with wall markers, and it is worth measuring per mesh: on meshes with degenerate cells the $1/d^3$ weight lets those cells dominate the norm.

Startup. By default HANIM starts on the first iteration at `cfl_start` and ignores `run_control.cfl`, matching the reference implementation, which initializes from freestream and lets the $\times 0.1$ failure path do the work a startup phase would otherwise do. Setting `jfnk.startup_iter` or `startup_residual_reduction` restores the older behaviour: the CFL policy drives the startup and HANIM inherits the CFL at activation. That is worth doing on meshes where a Newton step from freestream cannot stay physical — a case whose trial states hit the pressure floor in the first tens of iterations converges normally once a startup phase precedes HANIM.

> **!** Important
>
> Older `linear_verdict`, `step_control`, and `jacobian.lag` blocks are not current JSONC input. Remove them rather than migrating individual values.

20.5.7 Advanced blocks

Block / key	Conditions	Purpose
`jfnk`	`time_integration=jfnk`	`fin_diff_step` (default `1e-4`), startup gates, tolerance relaxation, and relaxation.
`multigrid`	Experimental	Geometric agglomeration multigrid.
`rk_alpha`, `n_rk_step`	`time_integration=runge_kutta`	Multi-stage Runge–Kutta coefficients and stage count.
`dualvol_power`	—	Dual-volume weighting exponent.

20.6 PBNS

PBNS is the 3D pressure-based path. Place PBNS-specific options in `numerical_method.pbns`.

20.6.1 Steady: `pseudo_time`

> **!** Important
>
> `pseudo_time` is **not** a `numerical_method.pbns` key — it moved to `run_control.pseudo_time`, a sibling of `run_control.cfl`, since PBNS and DBNS pseudo-time policy share that spot for the same reason CFL did. Placing it under `numerical_method.pbns` fails with an unknown-key error.

```
{
  "analysis": { "solver": "pbns", "time": "steady", "dimension": 3 },
  "numerical_method": {
    "pbns": {
      "coupling": { "max_iter": 20, "rho_tol": 1.0e-5 },
      "nonlinear": { "urf": 0.5 }
    },
    "linear_solver": {
      "type": "fgmres",
      "preconditioner": { "flow": { "type": "dilu" } }
    }
  },
  "run_control": {
    "pseudo_time": { "type": "auto", "factor": 1.0 }
  }
}
```

See run_control.pseudo_time for the auto/timestep/cfl types and their fields.

20.6.2 Unsteady: CFL limits

For unsteady PBNS, three CFL limits (still under numerical_method.pbns, unlike pseudo_time above) bound the evolving physical Δt. run_control.timestep is the initial value; PBNS adapts it during the run.

```
{
  "analysis": { "solver": "pbns", "time": "unsteady", "dimension": 3 },
  "numerical_method": {
    "pbns": {
      "coupling": { "max_iter": 20 },
      "timestep": {
        "cfl_acoustic":  50.0,
        "cfl_flow":      0.5,
        "cfl_diffusion": 0.5
      }
    }
  }
}
```

20.6.3 PBNS solver knobs

All keys below are inside numerical_method.pbns, in one of three subblocks: coupling (the outer pressure/energy loop), nonlinear (the Picard or Newton iteration), and timestep (the unsteady physical Δt). Steady pseudo-time still belongs to run_control.pseudo_time.

Key	Default	Applies to	What it controls
coupling.max_iter	20	Both	Maximum coupling iterations per time step. Raise it if the per-step solve reaches the cap before converging.
coupling.rho_tol	1e-5	Both	Coupling convergence: max|ρ − ρ₁| / max|ρ| over one coupling iteration. A constant-density run makes one coupling iteration.
coupling.max_retry	8	Unsteady	Maximum retries of a failed physical step.
coupling.energy.implicit	true	Both	Implicit bounded total-enthalpy solve; false uses Picard iteration.
coupling.energy.max_iter	50	Both	Cap on energy Picard iterations; 0 freezes temperature.

Key	Default	Applies to	What it controls
`coupling.energy.tol`	`1e-6`	Picard only	Relative energy fixed-point tolerance.
`coupling.energy.linear_tol`	`1e-3`	Implicit energy	Relative linear-solve tolerance.
`coupling.energy.urf`	`1.0`	Steady	Energy under-relaxation; 1 is none.
`timestep.cfl_acoustic`	`50`	Unsteady	Acoustic CFL cap on physical Δt; off for constant density.
`timestep.cfl_flow`	`0.5`	Unsteady	Convective CFL cap on physical Δt.
`timestep.cfl_diffusion`	`0.5`	Unsteady viscous	Diffusive CFL cap on physical Δt.
`timestep.effort.{min, max, grow}`	`0.3` / `0.5` / `1.2`	Unsteady	Effort band that grows or shrinks Δt, and the per-step growth cap.
`nonlinear.urf`	`0.5`	Steady	Pressure-update under-relaxation. Smaller values make the update more cautious.
`nonlinear.steady_inner`	`1`	Steady	Cap on Picard iterations per pseudo-step; 1 gives false-transient stepping.
`nonlinear.max_iter`	`40`	Unsteady	Cap on nonlinear iterations of one solve.
`nonlinear.max_update_fraction`	`0.2`	Steady	Per-point cap on the relative pressure and velocity update.
`nonlinear.tol.{fnorm, reduction, scstep}`	`1e-10` / `1e-3` / `3.67e-11`	Both	Absolute residual floor, relative reduction, and relative step floor of the nonlinear stop test.
`nonlinear.linear_operator`	`assembled`	Both	Operator of the nonlinear iteration.
`advection`	`linear_upwind`	Both	Momentum face reconstruction: second-order `linear_upwind` or first-order `upwind`.
`acoustic_dissipation`	`0.5`	Compressible	Rhie–Chow acoustic pressure dissipation; ignored in constant-density mode.
`lowmach_scaling`	`true`	Both	Two-sided diagonal scaling of the low-Mach linear system; turning it off removes that Krylov scaling.
`presf_lag`	`true`	Steady	Lag the Rhie–Chow pressure-force term.
`tied_theta`	`0.15`	Both	Neighbor-relative threshold for tied vertices; `0` disables it.

20.7 FR

FR uses `analysis.solver=fr` together with `analysis.time=unsteady`.

FR keys live in **two blocks**, split by what they answer:

- `numerical_method.fr` — the spatial discretization and the pseudo-step scheme.
- `run_control` — how large is the step and when does the inner loop stop. These are the same blocks DBNS and PBNS already use (`time_stepping`, `timestep_control`, `pseudo_time`, `convergence.inner`), so a deck that mixes solvers reads one vocabulary rather than three.

```
{
  "analysis": { "solver": "fr", "time": "unsteady", "dimension": 3 },
  "numerical_method": {
    "fr": {
      "order": 2,
      "riemann_solver": "rusanov",
      "time_integration": "rk4",
      "viscous": { "beta": 0.5, "tau": 0.1 }
    }
  }
}
```

Key	Meaning
`order`	Polynomial order p (1–5 in the current input reference); higher order increases spatial resolution and work per element.
`riemann_solver`	`rusanov`, `hll`, `hllc`, or `roe` — see below
`time_integration`	Explicit scheme; see time_integration below
`pseudo_scheme`	Inner-iteration scheme for dual time; see dual time
`viscous.beta`, `viscous.tau`	LDG viscous-interface upwind weight and jump penalty (`models.type=navier_stokes` only)
`ac_zeta`	Artificial-compressibility ζ (`models.regime=incompressible`)
`multigrid`	p-multigrid block; see dual time
`shock_capturing`	Entropy-filter block; see below
`anti_alias`	Flux over-integration block for under-resolved nonlinear flow; see below.
`source_terms`	Per-conserved-variable expression array

20.7.1 time_integration

FR `time_integration` values **differ** from DBNS. DBNS names such as `runge_kutta_explicit` are rejected.

Value	Meaning
`rk4` (default)	Four-stage explicit Runge–Kutta
`euler`	Explicit Euler
`tvd_rk3`	TVD RK3
`rk34`, `rk45`	Embedded 2R pairs; the only schemes that support physical-Δt PI control

There is no `dual` value. Dual time is not a scheme but a combination — set `run_control.time_stepping` to something other than `standard` and the run becomes dual-time, with `fr.pseudo_scheme` driving the inner iteration. This is how DBNS has always expressed it.

`riemann_solver` selects the interface flux shared by Euler and NS. The AC (incompressible) path is fixed to `rusanov` and ignores this key.

20.7.2 Physical-Δt PI control

`fr.time_integration`: `"rk34"` or `"rk45"` provides the embedded error estimate needed by `run_control.timestep_control.controller`: `"pi"`. Other FR integrators cannot use PI control. The physical-step tolerance and bound knobs are listed under `timestep_control`.

20.7.3 Entropy-filter shock capturing

This is for compressible flow only. It filters each element so the solution satisfies density, pressure, and entropy lower bounds.

```
{
  "numerical_method": {
    "fr": {
      "shock_capturing": {
        "type": "entropy_filter",
        "entropy_filter": {
          "formulation": "nonlinear",
          "density_min": 1.0e-6,
          "pressure_min": 1.0e-6,
          "entropy_tol": 1.0e-6,
          "filter_tol": 1.0e-4,
          "iterations": 2
        }
      }
    }
  }
}
```

Key	Meaning
`shock_capturing.type`	none (default) or `entropy_filter`
`entropy_filter.formulation`	`nonlinear` (default) or `linearised`
`entropy_filter.density_min`, `.pressure_min`	Density/pressure lower bounds the filter enforces
`entropy_filter.entropy_tol`, `.filter_tol`	Entropy and filter-fraction solve tolerances
`entropy_filter.iterations`	Filter solve iteration cap

> **Warning**
>
> Entropy filtering **cannot be used** with `run_control.timestep_control.controller=pi` or with dual time (`run_control.time_stepping` other than `standard`). The filter irreversibly modifies stage solutions, preventing reject and rollback; the combination stops with an error.

20.7.4 Flux anti-aliasing

`fr.anti_alias` over-integrates the nonlinear volume flux before projecting it back to the solution space. It is for compressible, under-resolved flow; leave it disabled for a normal starting configuration. It cannot be combined with the entropy filter.

Key	Default	What it controls
`anti_alias.mode`	none	none disables over-integration; `flux` enables it.
`anti_alias.quad_deg`	`0`	Quadrature strength. `0` chooses the solver's shape- and order-dependent rule; raise it only when comparing integration accuracy and cost.

20.7.5 dual time

FR uses dual time when `run_control.time_stepping` is `bdf1`, `bdf2`, `sdirk3`, or `sdirk4`. The physical scheme, pseudo-$\Delta\tau$, and inner stopping rule are described in Run control and convergence. The numerical-method knobs that affect the inner solve are:

Key inside numerical_method.fr	Default	What it controls
pseudo_scheme	tvd_rk3	Inner pseudo-time integrator: euler, tvd_rk3, rk34, or rk45. The latter two provide the embedded estimate required by run_control.pseudo_time.adaptive.controller: "local_pi".
multigrid.enabled	false	Enables the two-level p → p-1 FAS V-cycle during dual-time inner iterations.
multigrid.smooth_pre	2	Fine-level smoothing sweeps before the coarse correction.
multigrid.smooth_coarse	4	Coarse-level smoothing sweeps.
multigrid.smooth_post	2	Fine-level smoothing sweeps after the coarse correction.
multigrid.coarse_dtau_factor	1.75	Multiplier on the coarse-level pseudo-Δτ.

The SDIRK schemes are named by **order**, matching bdf1/bdf2 beside them — every time_stepping token reads as <family><order>. Both SDIRK schemes are three-stage; sdirk3 is stiffly accurate (L-stable) and sdirk4 is A-stable but not L-stable.

20.7.6 Artificial compressibility (AC)

The FR AC path with models.regime=incompressible requires fr.ac_zeta > 0. At boundaries, use farfield.{ac_zeta, ac_niters, velocity}; an omitted or zero boundary ac_zeta inherits numerical_method.fr.ac_zeta.

20.7.7 source_terms

This is an array of expression strings by conserved variable. Provide n_dim + 2 entries in the order [rho, rhou, rhov, (rhow), rhoE]. Omit it or provide an empty array for no source.

```
{ "numerical_method": { "fr": { "source_terms": ["0", "0", "-9.81*rho", "0"] } } }
```

20.7.8 Boundary conditions and output

FR reuses existing domains[].boundaries[]; it does not define separate FR boundary-condition types. Use outputs.volumes[] for volume output and monitoring.monitors[] for force monitors. outputs.diagnostics.fr_reference_compare is a development diagnostic; fr_snapshot was removed. Use outputs.restart for solution-point snapshots.

20.8 Shared linear solver

`numerical_method.linear_solver` is shared by DBNS and PBNS when an implicit linear system is solved. FR uses its own integration path. The following keys belong directly inside `linear_solver`:

Key	Reference value	What it controls
`type`	`fgmres`	Iterative method: `fgmres`, `bcgstab`, or `stationary`.
`error`	`1e-2`	Relative residual target of each linear solve. A smaller value asks for a tighter solve and usually more work.
`max_iter`	25	Maximum Krylov iterations; unused by `stationary`.
`relaxation`	`1.0`	Multiplier on the solved state update.
`max_update_fraction`	`0.2`	Per-point cap on the relative density/energy update; limits excessively large implicit steps.

The linear-algebra library and `row_scaling` are set in `expert_parameters.linalg`, not in `linear_solver`.

`linear_solver.preconditioner` has separate `flow`, `turb`, `energy` (PBNS), and `solid` blocks. The table below describes keys inside each system block, for example `preconditioner.flow.type`

```
{
  "numerical_method": {
    "linear_solver": {
      "type": "stationary",
      "preconditioner": {
        "flow": { "type": "jacobi", "sweeps": 25, "omega": 0.9 },
        "turb": { "type": "jacobi", "sweeps": 15, "omega": 0.9 }
      }
    }
  }
}
```

`stationary` performs fixed sweeps without Krylov iteration. It replaces the removed `richardson` name. Move an old Richardson `max_iter` count to the appropriate system's `sweeps`; remove `linear_solver.min_iter` and `linear_solver.smoother_relaxation`. The old flat `preconditioner.type` moves to `preconditioner.flow.type`; `expert_parameters.turb_` moves to `preconditioner.turb`.

Omitted sweeps use the following defaults; explicit values take precedence:

Method / preconditioner	Flow	Turbulence
Stationary / Jacobi	25	15
Stationary / DILU, ILU, SGS	5	3
Krylov / Jacobi	5	2
Krylov / DILU, ILU	1	1

Jacobi uses 2 sweeps for energy and 5 for solid. Solid now defaults to Jacobi, 5 sweeps. HANIM residual-controlled preconditioning has separate stopping controls.

Key	Reference value	What it controls
`type`	`dilu`	`jacobi`, `dilu`, `ilu`, `linelet`, `amg`, or `sgs`. `sgs` is CPU-only; a GPU build substitutes `dilu` with a warning.
`sweeps`	5 in the reference example	Number of fixed preconditioner applications. See the system- and method-specific defaults above. Ignored when HANIM residual-controlled preconditioning is active.
`omega`	`0.9` in the reference example	Preconditioner relaxation factor.
`sparsity_level`	`0`	ILU fill level: `0` for ILU(0), `1` for ILU(1).
`linelet_offline_sweeps`	2	Extra Jacobi sweeps on points outside linelets; only used with `linelet`.
`amg`	—	Nested AMG settings, only used when `type: "amg"`; its smoother, sweep counts, aggregation, and filtering knobs are nested keys that this guide does not cover.

20.9 Jacobian refresh interval

`numerical_method.dbns.jacobian_refresh_interval` defaults to `1`. For matrix-based `euler_implicit`, N > 1 rebuilds the flux Jacobian when `outer_iter % N == 0` and freezes it between those iterations. The pseudo-time diagonal is updated every iteration. JFNK and nonlinear control keep their own refresh policies.

20.10 Solid conduction

`numerical_method.solid.time_integration` selects `implicit` (default) or `explicit`; solid integration no longer follows `dbns.time_integration`. Use `run_control.cfl.solid` and `linear_solver.preconditioner.solid` for its CFL and preconditioner.

An unsteady run with a solid domain carries the thermal inertia ρC_p ∂T/∂t. It uses the dual-time stencil, so `run_control.time_stepping` must be `bdf1` or `bdf2`; an explicit physical step has no solid path. The solid material must define positive `solid.rho` and `solid.Cp`.

21 Run control and convergence

Iteration limits, convergence criteria, and steady/unsteady time progression

Choose `analysis.time` and `analysis.solver` first. Then use this page to set **when the run stops**, the **physical time step** (unsteady cases), and the **pseudo-time step** used to converge a steady solution or an implicit physical step. The GUI **Run Control** panel and the CLI `run_control` block write the same settings. Solver schemes and linear solvers are described in Numerical methods.

> **!** Important
>
> Physical time **Δt** advances the simulated clock. Pseudo-time **Δτ** only helps converge a steady solution or an implicit physical step; it does not advance that clock. `run_control.timestep` is Δt, while `run_control.pseudo_time.timestep` is Δτ. A CFL value can control either one, depending on the solver and time scheme below.

21.1 Configure in the GUI

The Desktop **Run Control** editor opens from the Run `Control` node of the project outline and writes the same `run_control` keys the CLI reads. Its three tabs separate the three questions this page answers: **Loop** sets the stopping limit, **Convergence** the criteria that may stop a run earlier, and **CFL** the pseudo-time step.

Tab	GUI control	Input key	What to choose
Loop	Iterations	`run_control.iteration`	The outer iteration limit. A steady run stops here unless a convergence criterion stops it first.
Loop	Resume	`run_control.resume`	Continue from a restart file instead of the initial field.
Convergence	Criteria	`run_control.convergence.residual` or `monitor`	Residual or monitor. The fields below belong to `residual`.

Tab	GUI control	Input key	What to choose
Convergence	Residual reduction [log10]	`run_control.convergence.residual_reduction`	Orders of magnitude the residual must fall from its starting value.
Convergence	Residual absolute [log10]	`run_control.convergence.residual_absolute`	An absolute floor. An empty field leaves the key unset.
Convergence	Minimum iterations	`run_control.convergence.min_iter`	Iterations to run before either criterion is tested.
CFL	Type	`run_control.cfl.type`	`constant`, `schedule`, or `adaptive`.
CFL	Fluid CFL	`run_control.cfl.fluid.value`	The fluid-zone CFL for `constant`.
CFL	Solid CFL	`run_control.cfl.solid.value`	The CHT solid-zone CFL, a fixed value only.

The CFL tab states that its values are consumed by the DBNS solver; a PBNS case takes its pseudo-time step from `pseudo_time` instead.

The unsteady keys — `timestep`, `total_time`, `pseudo_time` and `timestep_control` — are not in this dialog. Write them in JSONC.

21.2 Choose a setup

Analysis	Physical time and stopping rule	Convergence/step-size controls
DBNS steady	Set `iteration`; `convergence` may stop earlier.	Set `cfl` for the pseudo-time step.
PBNS steady	Set `iteration`; `convergence` may stop earlier.	Set `pseudo_time`; tune `numerical_method.pbns` outer/relaxation controls if needed. `cfl` does not control PBNS.
DBNS unsteady, `standard`	Set `total_time` and optionally `iteration`; whichever limit is reached first stops the run.	`cfl` sets the explicit physical Δt unless `timestep` is supplied. There is no inner loop.
DBNS unsteady, `bdf1`/`bdf2`	Set physical `timestep` and `total_time`; `iteration` is a second limit.	`cfl` controls inner pseudo-time progress; `convergence.inner` bounds each physical step's inner iterations.

Analysis	Physical time and stopping rule	Convergence/step-size controls
PBNS unsteady	Use `standard` (native adaptive-Δt BDF1), initial `timestep`, and `total_time`.	`numerical_method.pbns.timestep.cfl_ac` `.cfl_flow`, and `.cfl_diffusion` limit the evolving physical Δt; `pbns.coupling.max_iter`/`.rho_tol` govern the per-step solve.
FR unsteady, `standard`	Set `timestep` and `total_time`; optionally use `timestep_control` to adapt physical Δt.	Choose the explicit integrator in `numerical_method.fr.time_integration`. FR has no steady mode.
FR unsteady, `bdf1`/`bdf2`/`sdirk3`/`sdirk4`	Set physical `timestep` and `total_time`.	`pseudo_time` controls Δτ; `convergence.inner` controls the inner solve, with `numerical_method.fr.pseudo_scheme`.

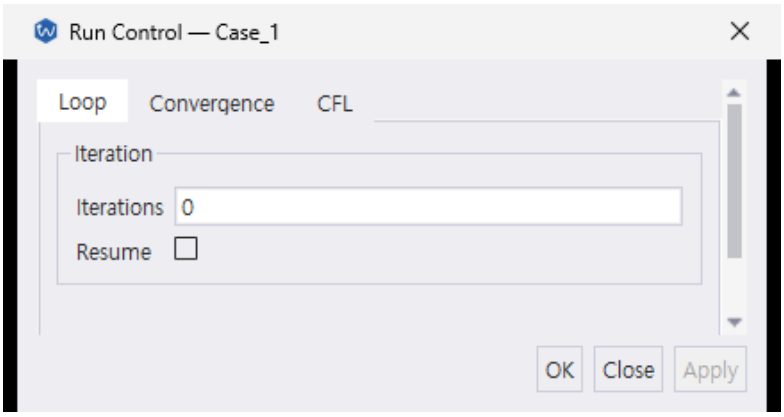
Start with the example matching your solver, then change the limits and step size for your case. These are `analysis`/`run_control` excerpts, not complete cases; mesh, models, and other required blocks are omitted.

21.2.1 DBNS steady

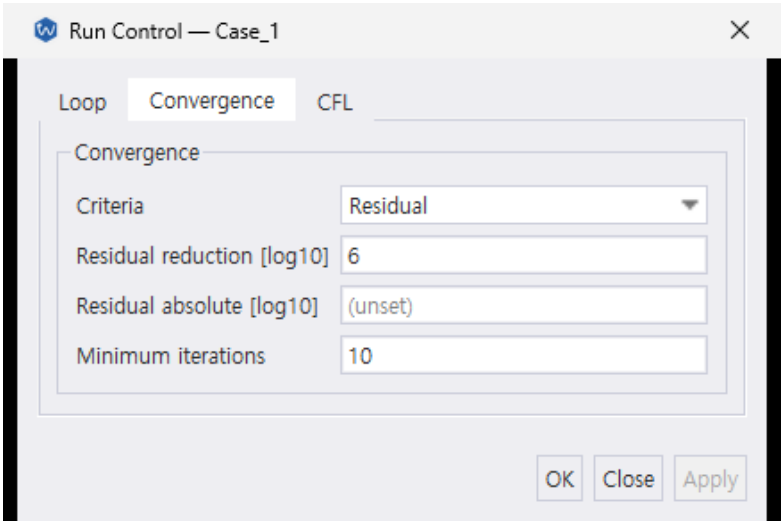
```
{
  "analysis": { "solver": "dbns", "time": "steady" },
  "run_control": {
    "iteration": 3000,
    "cfl": { "type": "constant", "fluid": { "value": 10 } },
    "convergence": {
      "criteria": "residual",
      "residual_reduction": 5,
      "min_iter": 10
    }
  }
}
```

21.2.2 PBNS steady

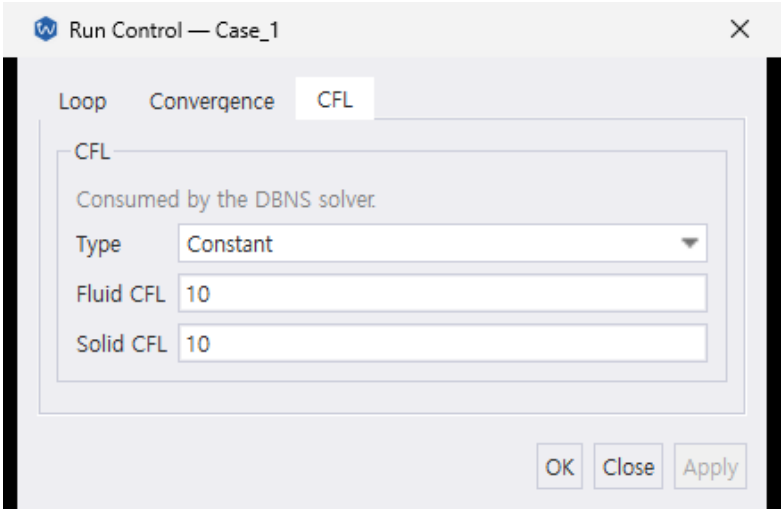
```
{
  "analysis": { "solver": "pbns", "time": "steady", "dimension": 3 },
  "run_control": {
```



(a) **Loop** — the iteration limit and restart behaviour.



(b) **Convergence** — the criteria that stop a run before its iteration limit.



(c) **CFL** — the pseudo-time step, consumed by the DBNS solver.

Figure 21.1: The three tabs of the Run Control editor, shown for a DBNS steady case.

```
    "iteration": 3000,
    "pseudo_time": { "type": "auto", "factor": 1.0 },
    "convergence": {
      "criteria": "residual",
      "residual_reduction": 5,
      "residual_absolute": -15
    }
  }
}
```

21.2.3 DBNS unsteady: explicit

```
{
  "analysis": { "solver": "dbns", "time": "unsteady" },
  "run_control": {
    "time_stepping": "standard",
    "cfl": { "type": "constant", "fluid": { "value": 1.0 } },
    "total_time": 0.1
  }
}
```

21.2.4 DBNS unsteady: dual time

```
{
  "analysis": { "solver": "dbns", "time": "unsteady" },
  "run_control": {
    "time_stepping": "bdf2",
    "timestep": 1.0e-3,
    "total_time": 0.1,
    "cfl": { "type": "constant", "fluid": { "value": 10 } },
    "convergence": { "inner": { "min_iter": 2, "max_iter": 200 } }
  }
}
```

21.2.5 PBNS unsteady

```
{
  "analysis": { "solver": "pbns", "time": "unsteady", "dimension": 3 },
```

```
  "numerical_method": {
    "pbns": {
      "timestep": { "cfl_acoustic": 50.0, "cfl_flow": 0.5, "cfl_diffusion": 0.5 },
      "coupling": { "max_iter": 20 }
    }
  },
  "run_control": {
    "time_stepping": "standard",
    "timestep": 1.0e-4,
    "total_time": 0.1
  }
}
```

21.2.6 FR unsteady: explicit

```
{
  "analysis": { "solver": "fr", "time": "unsteady" },
  "numerical_method": { "fr": { "time_integration": "rk4" } },
  "run_control": {
    "time_stepping": "standard",
    "timestep": 1.0e-4,
    "total_time": 0.1
  }
}
```

21.2.7 FR unsteady: dual time

```
{
  "analysis": { "solver": "fr", "time": "unsteady" },
  "numerical_method": { "fr": { "pseudo_scheme": "tvd_rk3" } },
  "run_control": {
    "time_stepping": "bdf2",
    "timestep": 1.0e-3,
    "total_time": 0.1,
    "pseudo_time": { "type": "cfl", "cfl": 1.0, "beta": 1.0 },
    "convergence": { "inner": { "min_iter": 2, "max_iter": 200 } }
  }
}
```

21.3 Limits and restart

21.3.1 `iteration`, `total_time`

`iteration` is the upper bound on **outer** iterations/physical steps, not the inner iterations of one dual-time step. `max_iter` is a compatibility alias; use `iteration` in new files. In steady analysis, convergence may stop the run before this limit. In unsteady analysis, `iteration` and `total_time` (physical seconds) are independent limits: the first reached stops the run. If you only care about elapsed physical time, set `total_time` explicitly; the unsteady default is 50 s and the iteration default is 999999.

For fixed Δt, estimate the required step count as `total_time / timestep`. With adaptive Δt (PBNS or FR PI control), inspect the physical-time column in the run output instead of assuming a fixed count.

21.3.2 `resume`

Controls whether `initialization.type=restart` also restores run state.

Value	Behavior
`false` (default)	Warm start: use the saved flow field as a new initial condition; reset the iteration counter, physical time, and time history. Use this when changing from steady to unsteady analysis.
`true`	Resume: restore the iteration counter, physical time, BDF history, and saved run state (including residual baseline and adaptive CFL state where applicable). Use this to continue an interrupted run.

21.4 CFL

`cfl` is **DBNS only**. It controls pseudo-time progress in steady and dual-time calculations; with `standard` unsteady stepping it controls the explicit **physical** Δt instead. Start with a constant value. If convergence is smooth but slow, try a schedule or adaptive control; if residuals oscillate or states become non-physical, reduce the CFL. For a CHT solid zone, set `cfl.solid.value` separately (fixed value only).

> **!** Important
>
> This block used to live at `numerical_method.dbns.cfl_control`. That path is no longer accepted and parsing fails if it is still present. Along with the move, `type: scheduled_ramp` was renamed to `schedule`.

type	Fields used
`constant`	`fluid.value`
`schedule` (alias `ramp`)	`fluid.schedule[]` — `iter`, `value`, `ramp_iters`
`adaptive`	`fluid.adaptive.*` — `start`, `min`, `max`, `factor_down`, `factor_up`, `acceptable_linear_residual`, `start_iter`, `per_point`

`adaptive` cannot be used with `standard` unsteady stepping.

21.4.1 CFL means something different for implicit and explicit schemes

The role of `cfl` depends entirely on `run_control.time_stepping` (see Unsteady physical time below), and this is the single most common point of confusion when setting up an unsteady DBNS case:

- **`bdf1` / `bdf2` (implicit dual-time)**: `run_control.timestep` fixes the physical Δt directly and dimensionally (e.g. `1.0e-3` seconds). `cfl` only controls how aggressively the inner pseudo-time iterations converge the nonlinear system at each physical step — it has **no effect on the physical time step**. Choose `timestep` for the temporal accuracy you need, then check that each physical step's inner residual converges within `convergence.inner.max_iter`. An implicit scheme does not guarantee a stable or accurate result for an arbitrarily large step.
- **`standard` (explicit)**: there is no pseudo-time loop, so **`cfl` directly determines the physical Δt** at every step: `dt = cfl · h / (|u| + a)`, evaluated per cell and reduced to a single global Δt applied to the whole domain (this is global time stepping — required for time accuracy; the per-cell local time stepping used to accelerate steady convergence would desynchronize the physical time across the domain and is not valid here). Because this Δt is recomputed from the current mesh and flow state at every step, it also drifts slightly as the solution develops.

  You can set `run_control.timestep` directly instead, but then you must check its stability limit against the smallest cells and local wave speeds. Starting with a modest CFL is usually easier.

  > **Tip**
  >
  > Small near-wall cells can force a very small global Δt. Check the reported Δt and estimated number of steps before committing to a long explicit run.

21.4.2 `schedule`

`iter` is the outer iteration at which that breakpoint takes effect, and `ramp_iters` is the transition length from the previous value. `null` or `0` means an immediate jump; the first entry's `ramp_iters` is ignored.

```
{
  "run_control": {
    "cfl": {
      "type": "schedule",
      "fluid": {
        "schedule": [
          { "iter": 0,   "value": 1,  "ramp_iters": null },
          { "iter": 100, "value": 5,  "ramp_iters": 20 },
          { "iter": 500, "value": 20, "ramp_iters": 10 }
        ]
      }
    }
  }
}
```

21.4.3 adaptive

adaptive adjusts the CFL from what the previous iteration already told you: the linear solver's relative residual, each point's implicit under-relaxation, and the trend of the nonlinear residual. It is a CFL policy — unlike HANIM it never rejects a trial state, restores U^n, or retries an iteration.

Set run_control.cfl.type to adaptive. The keys below are inside run_control.cfl.fluid.adaptive

Key	Default	Meaning
start	5.0	Initial CFL; must lie in [min, max]. The per-point field is seeded uniformly with it.
min	5.0	Band floor, and the target of a reset.
max	100.0	Band ceiling.
factor_down	0.1	Multiplier applied when a point is strongly under-relaxed or the linear solve missed its target.
factor_up	1.2	Multiplier applied when the point took the full update and the linear solve met its target.
acceptable_linear_residual	0.1	Linear-residual target for the controller (see below).
start_iter	0	No raise or cut before this outer iteration.
per_point	true	Per-point CFL field; false drives one scalar CFL instead.

```
{
  "run_control": {
    "cfl": {
      "type": "adaptive",
```

```
      "fluid": { "adaptive": { "start": 5.0, "min": 5.0, "max": 100.0 } }
    }
  }
}
```

How a step is decided. The controller first forms one target from your two tolerances, `linTol = max(acceptable_linear_residual, linear_solver.error)`, and compares it against the worst relative linear residual of the previous iteration (mean flow and turbulence):

- below `linTol` → the point may be raised,
- above `1.2 · linTol` → the point is cut,
- in between → held. The 20 % dead band exists to stop the CFL flip-flopping.

A linear residual above `0.99`, or a nonlinear residual that has risen more than two orders across the last 100 iterations, resets every point to `min`.

Each point then applies its own under-relaxation `UR = min(UR_flow, UR_turb)`, the factor the implicit update had to use to keep the state physical: `UR < 0.1` multiplies by `factor_down`, a full `UR = 1` with a healthy linear solve multiplies by `factor_up`, and anything between holds. With `per_point: true` this runs on the CFL field, so stiff cells (a shock, a degenerate wall cell) hold or drop while the rest of the mesh climbs; the scalar CFL you see in the console and in restarts is the **minimum over points**. With `per_point: false` the same three-way rule is driven by the global minimum UR instead.

> **Tip**
>
> `max` is part of the design, not a safety net you should remove. The controller only sees the linear solver, so if the linear solve keeps succeeding it will keep raising the CFL until the state itself goes non-physical. Keeping the default ceiling of 100 is what makes the policy stable; raising it to `1e4` on a 2-D airfoil case produced a CFL sawtooth and non-physical states that had to be rolled back, with a worse residual than the capped run.

HANIM is selected separately through `numerical_method.dbns.nonlinear_control.method: "hanim"` with `time_integration: "jfnk"`. By default HANIM has no startup phase and does not use `run_control.cfl` at all — it starts from `nonlinear_control.cfl_start` on the first iteration. If the deck sets `jfnk.startup_iter` or `startup_residual_reduction`, `adaptive` (or any other policy) drives that startup and HANIM inherits the CFL when it activates, taking the minimum over points but never below `cfl_start`.

> **Warning**
>
> Do not use `adaptive` with `time_stepping: "standard"` unsteady analysis. The validator rejects that combination. `run_control.cfl` is DBNS-only; PBNS and FR use their solver-specific pseudo-time controls.

> **!** Important
>
> `numerical_method.dbns.cfl` and `cfl_solid` were removed. Set DBNS CFL only under `run_control.cfl`.

21.5 pseudo_time

The pseudo-Δτ policy. Two solvers consume it:

- **PBNS steady** — the false-transient step used to approach the steady solution (`analysis.solver=pbns` with `analysis.time=steady`).
- **FR dual time** — the inner-iteration Δτ, whenever `time_stepping` is something other than `standard`.

PBNS unsteady does **not** consume this block. It uses `numerical_method.pbns.{cfl_acoustic,cfl_flo` to limit an adaptive physical Δt. Start PBNS steady with `auto`; if it is too aggressive, lower `factor`. Use `timestep` when a measured pseudo-time scale is known, or `cfl` when the mesh and local flow should determine Δτ.

```
{
  "run_control": {
    "pseudo_time": {
      "type": "auto",
      "factor": 1.0,
      "timestep": 1.0e-3,
      "cfl": 5.0,
      "beta": 1.0,
      "adaptive": {
        "controller": "local_pi",
        "atol": 1.0e-5,
        "pi_alpha": 0.7,
        "pi_beta": 0.4,
        "safety_factor": 0.8,
        "factor_down": 0.98,
        "factor_up": 1.01,
        "max_factor": 2.0
      }
    }
  }
}
```

type	Active fields	Meaning
auto (default)	factor	Fixed Δτ = factor · L/U, determined at run start. L comes from reference.length_moment; U uses the initial/farfield/inlet speed and reference.velocity, with sound speed as the compressible zero-speed fallback. Constant-density cases need a nonzero velocity reference. Lower factor if the steady solve is unstable.
timestep	timestep	Uniform pseudo-time Δτ in seconds; this is **not** the physical run_control.timestep.
cfl	cfl, beta	Local Δτ = cfl · h / (|u| + β·a); β = 1 includes the acoustic speed, while a smaller positive β moves toward a convective scale. FR additionally accounts for polynomial order.

adaptive is an FR-only embedded-error controller for the pseudo step; it needs numerical_method.fr.ps to be rk34 or rk45. It spells its gains exactly like timestep_control below, because they are the same kind of controller applied to a different step.

pseudo_time.adaptive key	Effect (FR dual time only)
controller	none (default) or local_pi; enable the latter only with an embedded rk34/rk45 pseudo scheme.
atol	Absolute error scale for the embedded estimate.
pi_alpha, pi_beta	PI response to current and previous error. Start with defaults (0.7, 0.4).
safety_factor	Multiplies the proposed next Δτ; default 0.8.
factor_down, factor_up	Bound how much Δτ can shrink or grow per step; defaults 0.98, 1.01.
max_factor	Caps the initial Δτ multiplier; default 3.0.

> **Note**
>
> This key lives at run_control.pseudo_time, not inside a solver's numerical_method block — it moved here for the same reason cfl did. The three are not exclusive but consumed by different solvers: cfl drives DBNS, pseudo_time drives PBNS steady and FR dual time, and timestep_control drives FR explicit.

21.6 timestep_control

The physical-Δt policy: an embedded-error PI controller that resizes Δt every step. **FR explicit only**, and only with numerical_method.fr.time_integration set to rk34 or rk45. Leave

`controller: "none"` (the default) for a fixed `run_control.timestep`. Do not combine PI control with FR entropy filtering or dual time; the solver rejects those combinations.

```
{
  "run_control": {
    "timestep_control": {
      "controller": "pi",
      "atol": 1.0e-6,
      "rtol": 1.0e-6,
      "error_norm": "l2",
      "pi_alpha": 0.58,
      "pi_beta": 0.42,
      "safety_factor": 0.8,
      "factor_down": 0.3,
      "factor_up": 2.5,
      "min": 1.0e-12,
      "max": 1.0e2
    }
  }
}
```

Key	Meaning
`controller`	`none` (default) or `pi`
`atol`, `rtol`	Absolute/relative error tolerances driving the step-size estimate; tighten them for a more accurate, usually slower run. Defaults are `1e-6` each.
`error_norm`	`l2` or `uniform`
`pi_alpha`, `pi_beta`	PI-controller gains; defaults `0.58`, `0.42`
`safety_factor`	Multiplier on the proposed step; default `0.8`
`factor_down`, `factor_up`	Limits on the change per step; defaults `0.3`, `2.5`
`min`, `max`	Absolute bounds on physical Δt; below `min` the run fails. Defaults `1e-12`, `1e2` s.

21.7 Convergence criteria

For a steady run, convergence can end the outer loop before `iteration`. For DBNS/FR dual time, it can end each inner solve before `convergence.inner.max_iter`. Standard unsteady runs advance until `total_time` or `iteration` instead. In **DBNS** residual convergence, the relative and absolute tests form an **OR**: whichever threshold is met first ends that loop. **FR dual time** uses `inner.residual_tol` (or the iteration cap when that tolerance is zero). `criteria: "monitor"` selects a monitor-based steady stopping rule instead.

21.7.1 residual

The default `criteria`. For DBNS, `residual_reduction` is the log10 order-of-magnitude drop relative to a baseline. **PBNS steady is different**: it compares its already-scaled pressure-based residual directly with `10^-residual_reduction`, without dividing by an initial residual. Use the pressure-based residual to judge PBNS steady convergence; the absolute threshold is usually a deep safety net (`-15` in the reference deck). `min_iter` delays the steady outer-loop test.

```
{
  "run_control": {
    "convergence": {
      "criteria": "residual",
      "residual_reduction": 5,
      "min_iter": 10,
      "inner": { "min_iter": 2, "max_iter": 200 }
    }
  }
}
```

Key	Meaning
`criteria`	`residual` (default) or `monitor`; picks which steady stopping test to use.
`residual_reduction`	Default 5. DBNS: stop after this many orders of reduction from the loop baseline. PBNS steady: require its scaled pressure-based residual to be at most `10^-N`.
`residual_absolute`	Stop when `log10|R|` reaches this value; default `-8` for steady and `-14` for dual-time inner solves. DBNS uses L2 RMS; PBNS uses a weighted RMS, so do not compare the two values directly.
`min_iter`	Outer loop only: don't evaluate convergence before this many iterations. Not inherited by `inner`.
`inner.min_iter` / `inner.max_iter`	Minimum/maximum pseudo-time iterations per physical step for DBNS/FR dual time; defaults 2/`200`. Replace the old top-level `inner_iter` with `inner.max_iter`. If the cap is reached while the inner residual is still high, increase it or ease the physical `timestep`.
`inner.residual_tol` / `inner.residual_norm`	FR inner residual gate. `0` (default) uses the iteration cap without this tolerance test; `residual_norm` is `l2` or `uniform`.

The DBNS relative-residual baseline depends on the loop:

Loop	Baseline for `residual_reduction`
Steady outer	The maximum residual over the first 5 samples (a warm-up peak, more representative than iteration 1)
Dual-time inner	The residual at the start of that physical step

`residual_absolute`'s default follows the same DBNS split: for an inner solve the absolute check is only an escape hatch for a step that enters already near machine-zero residual — the steady default (`-8`) would fire on nearly every step there, so the inner default is a much deeper `-14`.

> **Note**
>
> PBNS unsteady has a native BDF1 outer solve per physical step. Its per-step controls are `numerical_method.pbns.max_outer` and `outer_tol`, not the DBNS/FR dual-time `convergence.inner` settings.

21.7.2 monitor

Stop once a named monitor's value has settled, instead of waiting on the residual — useful when the residual sits on a floor without dropping further while the quantity you actually care about (a force coefficient, say) has already stopped changing.

```
{
  "run_control": {
    "convergence": {
      "criteria": "monitor",
      "monitor": {
        "fields": ["CFx", "CFy"],
        "eps": 1.0e-5,
        "window": 100
      }
    }
  }
}
```

`monitor.fields` lists the monitor names to test, `window` is the number of recent outer iterations (default `100`), and `eps` is the tolerance (default `1e-5`). For each listed field:

```
(max − min) / max(|max|, 1.0) < eps
```

so a coefficient with `|value|` ≤ 1 is judged on an absolute basis (the default `eps` = `1e-5` is roughly 0.1 drag/lift count), while a larger physical quantity is judged relatively.

Every listed field has to settle — it's an AND, not an OR. `fields` entries must match a `monitoring.monitors[]` block's name (or <name>.<boundary> for a per-boundary scalar); an unknown name is rejected at validation time, not silently ignored at run time. The window also has to fill up within this run before it can trigger, so resuming a run doesn't stop it immediately just because the first few post-restart samples happen to look flat.

21.8 Unsteady physical time

Set `analysis.time: "unsteady"` before choosing `time_stepping`. The physical clock advances by Δt until `total_time` or `iteration` is reached.

```
{
  "analysis": { "time": "unsteady" },
  "run_control": {
    "time_stepping": "bdf2",
    "timestep": 1.0e-3,
    "total_time": 3.0
  }
}
```

time_stepping	DBNS	PBNS	FR
standard (default)	Explicit; CFL-derived physical Δt, or user-supplied timestep.	Native first-order BDF with adaptive Δt and per-step outer solve.	Explicit integrator selected by numerical_method.fr.time_integrati fixed timestep or adaptive timestep_control.
bdf1	First-order dual time.	Alias of standard; the same native BDF1 path.	First-order dual time.
bdf2	Second-order dual time.	Rejected: not implemented.	Second-order dual time, supported since September 2026. First step and steps after Δt changes start with BDF1; a resumed run can restore BDF2 history from its restart file.
sdirk3, sdirk4	Not supported.	Not supported.	Three-stage, third-/fourth-order dual-time schemes respectively.

21.8.1 timestep and total_time

timestep is the physical initial or fixed Δt in seconds, depending on solver:

- **DBNS `standard`**: omit it to derive Δt from `run_control.cfl`; if supplied, verify explicit stability yourself. **DBNS dual time**: it is the fixed Δt; choose it for time accuracy, then ensure the inner solve converges.
- **PBNS unsteady**: it is the **initial** Δt, not a fixed step. The solver changes it according to its acoustic, flow, and diffusion CFL limits and per-step solve behavior. Check reported physical time and actual Δt when comparing results. PBNS has no fixed-Δt mode.
- **FR `standard`**: it is the fixed starting Δt. With `timestep_control.controller: "pi"`, the embedded error estimate changes it each step. **FR dual time**: it is the physical Δt; `pseudo_time` sets the separate inner Δτ.

`total_time` sets the stopping time in seconds. State it explicitly in an unsteady deck; if omitted, the default is 50 s. `iteration` is also active, and whichever limit is reached first ends the run. For DBNS/FR dual time, the per-step inner limit belongs to `convergence.inner.max_iter`.

These time-progression keys are not used when `analysis.time=steady`.

> **Note**
>
> `run_control.convergence` (this page) and `outputs.restart` are independent axes — one governs when the run stops, the other governs what gets saved along the way and whether you can resume a crashed run. See File output for restart frequency and retention policy.

21.9 Controlling a running job

For graceful stop signals, scheduled stop/dump via `wake.control.json`, and gRPC runtime control, see Controlling a running job.

21.10 Restart numbering and FP32 convergence

Iteration-number controls (`iteration`, `min_iter`, `start_iter`, `startup_iter`, freeze iterations, and CFL schedule `iter`) use absolute outer-iteration numbers. Resuming does not restart these counters.

The `wake_<2|3>d_fp32` build uses float arithmetic on the residual path and may reach a relative residual floor around 1e-7. This is not a universal stopping threshold: check the monitored physical quantities and choose `criteria: "monitor"` when appropriate. Use `_fp64` for precision comparisons and `_mixed` for double-precision residual arithmetic with FP32 linear solves.

22 Initialization

auto, specified, restart, FR preset

22.1 What this setting is

Initial conditions define the flow field from which iterative calculation starts. Steady analyses often begin from a uniform freestream; for complex or unsteady flows, a previous solution (restart) can greatly reduce convergence time. The initial field does not replace final physical conditions, but a field that differs too much from boundary conditions can cause early divergence or unnecessary iterations.

22.2 Configure in the GUI

GUI control	Input key	What to choose
Type	`initialization.type`	`auto`, `specified`, or `restart`; `fr_preset` is listed but disabled. The rest of the dialog follows this choice.
Mach	`initialization.mach`	DBNS only. An empty field takes the operating state's value.
Pressure [Pa]	`initialization.pressure`	Required closure on both DBNS and FR.
Temperature [K]	`initialization.temperature`	DBNS closure. FR ignores it.
Direction type	`initialization.direction.type`	`cartesian` (Cartesian components) or `angles` (AoA / AoS). Leave the direction unset to take the farfield's.
Direction X / Y / Z	`initialization.direction.cartesian.[x,y,z]`	The flow direction, normalised by the solver.
AoA / AoS [deg]	`initialization.direction.angles.[aoa,aos]`	Shown in place of the components when Direction type is Angles.

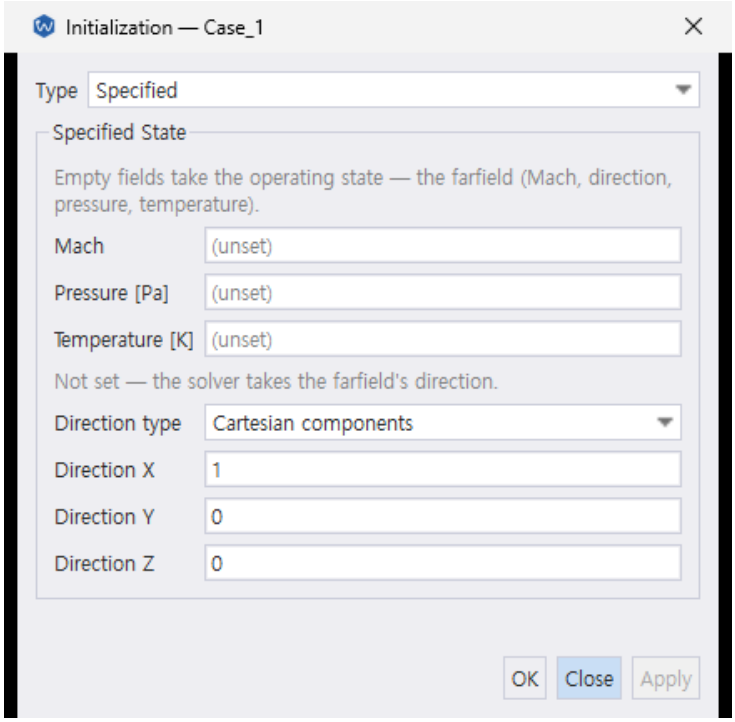


Figure 22.1: The Initialization dialog with `specified` selected. The fields shown belong to that type; choosing another type replaces them.

An empty field is not zero: the dialog greys it as `(unset)` and the key is left out of the deck, so the operating state supplies the value. The dialog says so above the fields it applies to.

> **Note**
>
> **The Desktop writes a newer `initialization` than the verified solver reads.**
> The Desktop has no v0.23.0 release, so this section describes the current build. Its deck vocabulary is ahead of the command-line solver documented in the rest of this page: it writes `type: "specified"` and a `direction` object where v0.23.0 reads `type: "uniform"` and `flow_direction`. Author a deck for the command-line solver from the JSONC sections below, not from the key column above. The two converge in a later release.

`initialization` selects how to create the solution at calculation start. It is a tagged union whose active fields depend on `type`.

```
{
  "initialization": {
    "type": "uniform",
    "mach": 0.8,
    "pressure": 101325.0,
    "temperature": 273.15,
```

```
    "flow_direction": [1.0, 0.0, 0.0]
  }
}
```

22.3 type

Value	Meaning
`auto`	Extract the initial state automatically — **DBNS-only behavior**; see the callout below
`uniform`	One uniform state from the constant fields of this block. An omitted field takes the operating state's value
`expression`	A per-point field: each value is an expression in x, y, z
`fr_preset`	Preset for FR validation
`restart`	Read a `.wsol` restart file

> ⚠ Warning
>
> **`type: "auto"` means something different on each solver.** DBNS extracts the state from a boundary condition (`farfield` first, then an isentropic inlet/outlet, else it's an error). FR has no such BC-extraction path at all — `auto` on FR is always **quiescent**: `ρ = 1`, `u = v = 0`, `p = 1`, regardless of what boundary conditions the case defines.

22.4 uniform

The same key means something different — or nothing at all — depending on the solver's thermodynamic closure.

Key	DBNS (FVM)	FR
`mach`	Consumed	**Not consumed — silently has no effect**
`flow_direction`	Consumed	**Not consumed — silently has no effect**
`temperature`	Required closure (`EOS(P,T)` derives ρ)	**Not consumed — silently has no effect**
`density`	Not consumed (derived from EOS)	Required closure

Key	DBNS (FVM)	FR
`pressure`	Required closure	Required closure

> **!** Important
>
> Setting `mach`/`flow_direction`/`temperature` on an FR case does **nothing** — no warning, no error, just silently ignored — because FR's closure is `(ρ, u, v, w, p)` only. To give an FR case a freestream, write `density` + `pressure` + `velocity_x`/`velocity_y`/`velocity_z` instead.

```
{
  "initialization": {
    "type": "uniform",
    "mach": 0.8,
    "pressure": 101325.0,
    "temperature": 273.15,
    "flow_direction": [1.0, 0.0, 0.0]
  }
}
```

```
{
  "initialization": {
    "type": "uniform",
    "density": 1.225,
    "pressure": 101325.0,
    "velocity_x": 100.0,
    "velocity_y": 0.0,
    "velocity_z": 0.0
  }
}
```

`flow_direction` is a `[x, y, z]` unit vector, normalised on read; leaving it out takes the farfield's direction. Angle of attack is not set here — it is a flight condition and belongs to the far-field boundary (`domains[].boundaries[].aoa`). Giving any `velocity_x/y/z` instead replaces both `mach` and `flow_direction`.

22.4.1 Turbulence seeding

Optionally seed the internal turbulence field directly, independent of the farfield/inlet boundary condition's own `turbulence` block:

```
{
  "initialization": {
    "type": "uniform",
    "mach": 0.8, "pressure": 101325.0, "temperature": 273.15,
    "flow_direction": [1.0, 0.0, 0.0],
    "turbulence": {
      "intensity": 0.05,
      "intermittency": 1.0,
      "nu_factor": 3.0,
      "eddy_viscosity_ratio": 10.0
    }
  }
}
```

Key	Meaning
`intensity`	SST/k-ω turbulence intensity (`0.05` = 5%)
`intermittency`	Transition intermittency γ (transition models)
`nu_factor`	SA freestream ν̃/ν multiplier
`eddy_viscosity_ratio`	Inflow μ_t/μ ratio; `length_scale` [m] and `hydraulic_diameter` [m] are alternatives, and only one may be given

Setting **any one** of these switches the solver to seeding the whole field from this block instead of scanning boundary-condition `turbulence` values (the default behavior when the block is omitted entirely). The block is ignored with `type: "restart"`, with a warning.

22.4.2 Per-point fields

Per-point fields have a branch of their own: set `type: "expression"` and write each scalar as an expression in `x`, `y`, and `z`. A value is a bare string, or the object form `{ "type": "expression", "expr": "…" }`; a bare number, or `{ "type": "constant", "value": … }`, is still a constant.

```
{
  "initialization": {
    "type": "expression",
    "pressure": { "type": "expression", "expr": "101325 + 1000*exp(-(x^2 + y^2))" }
    "temperature": 300.0,
    "velocity_x": "sin(2*pi*x)",
    "velocity_y": 0.0,
    "velocity_z": 0.0
```

```
  }
}
```

`pressure` and `temperature` are required in this branch on the finite-volume solvers. `mach` is a plain number and takes no expression. PBNS accepts `auto` and `uniform` only.

22.5 FR preset

Use the following preset for an FR validation case.

```
{
  "initialization": {
    "type": "fr_preset",
    "fr_preset": {
      "name": "taylor_green",
      "taylor_green": { "Ps": 111.607 }
    }
  }
}
```

Supported presets are `isentropic_vortex`, `yee_shu`, `taylor_green`, and `density_wave_diag`. Place each preset's parameters in a subblock with the same name: `isentropic_vortex.{M, R, S}`, `yee_shu.{eps, u_inf, v_inf}`, `taylor_green.Ps`, or `density_wave_diag.{rho0, u0, v0, p0, A}`.

22.6 restart

```
{
  "initialization": {
    "type": "restart",
    "restart_file": "restart_flow_001.wsol"
  },
  "run_control": { "resume": true }
}
```

These are two different intents, distinguished by `run_control.resume`:

`run_control.resume`	Intent	Behavior
`false` (default)	**Warm-start**	Only the flow field is taken as the initial value; the iteration counter, physical time, and BDF history all reset. Use this to start a new case (an AoA/CFL sweep, say) from an existing solution.
`true`	**Continuation**	The iteration counter, physical time, residual-reduction origin, and adaptive-CFL state are all restored, so the run continues as if uninterrupted — see `resume`.

> **Note**
>
> **Solver support.** All three solvers read a restart file. PBNS accepts `auto`, `uniform`, and `restart`, and rejects `fr_preset` before the run starts; a PBNS restart file must carry u, v, w, p, and T. `fr_preset` is FR-only — using it on a DBNS or PBNS case is an input error. If the restart file carries no turbulence fields, they are filled from the first farfield, then the first inlet, then the defaults (SST I 0.05 and μ_t/μ 10, SA 3).

22.7 Configure from the CLI

```
{
  "initialization": {
    "type": "uniform",
    "mach": 0.8395,
    "pressure": 101325.0,
    "temperature": 273.15,
    // 3.06 deg angle of attack as a unit vector; the angle itself is set on
    // the farfield boundary, which this only has to agree with
    "flow_direction": [0.998574, 0.0, 0.053387]
  }
}
```

23 File output

restart, volume/surface/slice, diagnostics

23.1 What this setting is

File output writes the flow field and restart data you inspect during and after a run. The full volume, wall/boundary surfaces, and probe slices serve different purposes and carry very different storage costs, so pick only what you need. In unsteady analyses in particular, output frequency governs both temporal resolution and disk usage at once — decide the time interval your post-processing actually needs before picking a frequency, not after.

23.2 GUI configuration

Field	Description
Restart	Save frequency and retention policy for restart files.
Volume / Surface / Slice	Add the domain, boundary, or cut-plane results to save.
Enabled	Include or temporarily disable an output entry without deleting it.
Fields	Physical quantities to inspect in ParaView.
Frequency	The iteration or time interval at which to write a file.
Start iteration	Skip periodic writes before this iteration; the final output is still kept.

`outputs` writes computed results to disk. Each output block can hold several independent array entries.

23.3 restart

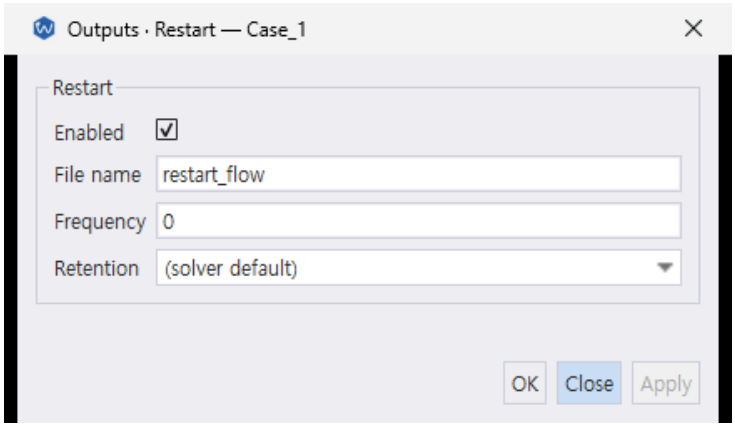


Figure 23.1: The Restart output dialog enables checkpoint writing and configures its file name, frequency, and retention policy.

The Desktop exposes the operational restart controls shown above. The `contents` level in the JSONC example is an advanced CLI setting and is not currently shown in this dialog.

```
{
  "outputs": {
    "restart": {
      "enabled": true,
      "file_name": "restart_flow",
      "frequency": 100,
      "contents": "full",
      "retention": { "policy": "keep_last", "count": 2 }
    }
  }
}
```

`contents` controls how much mesh data is embedded in the `.wsol` restart. `frequency: 0` writes only at the final iteration. Choose the content level and retention policy from the restart workflow you actually need.

`contents`	What the restart contains	Use when
`fields`	Solution fields only	The original mesh is always available and you want the smallest restart files.
`geometry`	Fields, coordinates, and boundary-node sets	You need to remap onto a mesh without reopening the original mesh file.

`contents`	What the restart contains	Use when
`full` (default)	Fields plus connectivity	You need a mesh-file-free restart; this is the safest operational default.

`retention.policy` is `keep_last`, `keep_all`, or `milestone`:

Policy	What it keeps	When to use it
(no periodic writes; *`frequency: 0`*)	Final iteration only	Short or cheap-to-rerun cases where losing all progress on a crash is an acceptable risk. No mid-run write overhead at all.
`keep_last` + `count`	The `count` most recent writes; older ones are pruned as new ones arrive	The default choice for a long run: crash recovery within one `frequency` interval of progress, bounded disk use.
`milestone` + `milestones`	Only the listed iterations, plus the final iteration	You want both crash recovery and permanent snapshots at specific points you'll want to inspect later (e.g. "did the flow reach its expected transient-to-periodic transition by iteration X?").

> **!** Important
>
> `milestone` **ignores `frequency` entirely**. The solver writes only at the listed `milestones` (which do **not** need to be multiples of `frequency` — any iteration number is allowed) plus the final iteration; no other restart file is ever created, so there is nothing to prune. If you want, say, one checkpoint roughly every second of physical time on a run with a CFL-limited, slowly-drifting Δt, compute approximate target iterations from your expected Δt and list them explicitly — exact alignment with the true Δt at that point in the run isn't necessary, since these are just recovery points.

```
{
  "outputs": {
    "restart": {
      "enabled": true,
      "file_name": "restart_flow",
      "retention": {
        "policy": "milestone",
```

```
        "milestones": [928505, 1857010, 2785515, 3714020]
      }
    }
  }
}
```

23.4 volumes

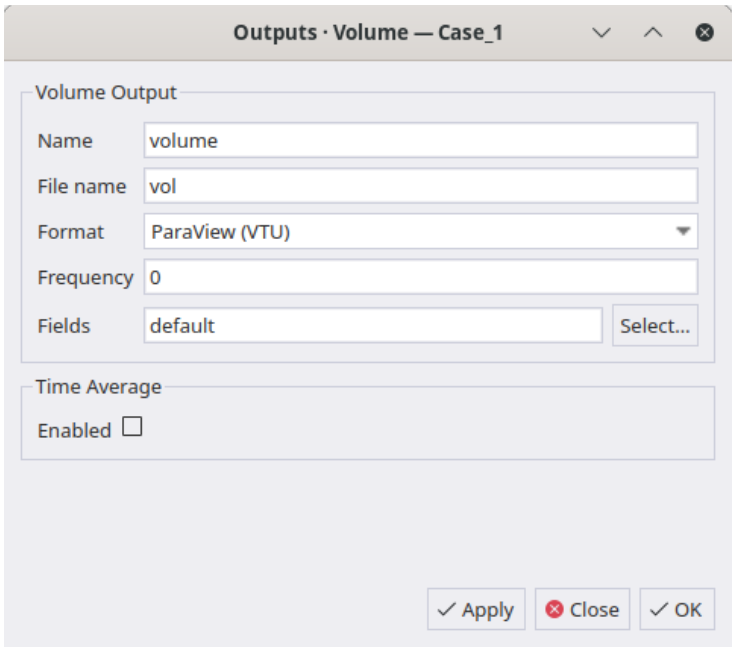


Figure 23.2: A volume-output entry writing the default flow fields in ParaView VTU format. A frequency of zero means final iteration only.

> **Note**
>
> The current source additionally exposes **Enabled** and **Start iteration** above the fields shown in this capture. They map to `enabled` and `start_iter` in the same output entry.

```
{
  "outputs": {
    "volumes": [
      {
        "name": "flowfield",
        "file_name": "volume_output",
        "format": "vtu",
        "enabled": true,
```

```
        "start_iter": 0,
        "frequency": 100,
        "fields": ["default", "Entropy"],
        "enable_vtm": false,
        "subdivision": -1
      }
    ]
  }
}
```

`format` is one of `vtu`, `vtkhdf`, `tecplot`, `cgns`, or `tecplot_binary`. These are container names and take no aliases: `paraview` is not a value, though both `vtu` and `vtkhdf` open in ParaView. `vtkhdf` is the one to reach for on an MPI run: it writes a **single** HDF5 container across ranks instead of the `.pvtu` + one `.vtu`-per-rank split that plain `vtu` produces.

`fields` accepts `"default"` (or omit `fields` entirely — same thing) plus any individual field names to add on top: `["default", "Entropy", "TotalPressure"]`. `"default"` itself adapts to the physics model you declared in `models` — Euler gives density/velocity/pressure/temperature/ Mach/Cp, Navier–Stokes adds laminar viscosity and wall distance, RANS adds eddy viscosity and the turbulence variable(s). The old per-model presets (`"euler"`, `"ns"`, `"rans"`) are gone — since the physics model is already declared in `models`, repeating it in `outputs` was redundant, and today it's rejected outright rather than silently accepted.

`subdivision` renders high-order FR results as subdivided VTK cells; FVM ignores it.

23.5 surfaces and slices

Surface selections use the names of configured boundary objects, not raw mesh marker names. Slice entries use the same common output controls and add plane origin and normal fields; no slice screenshot is included in this set.

```
{
  "outputs": {
    "surfaces": [{
      "name": "airfoil_surface",
      "file_name": "surface_airfoil",
      "format": "vtu",
      "enabled": true,
      "start_iter": 0,
      "frequency": 100,
      "fields": ["Pressure", "Cp", "Cf", "Yplus"],
      "boundary_names": ["AIRFOIL"]
    }],
```

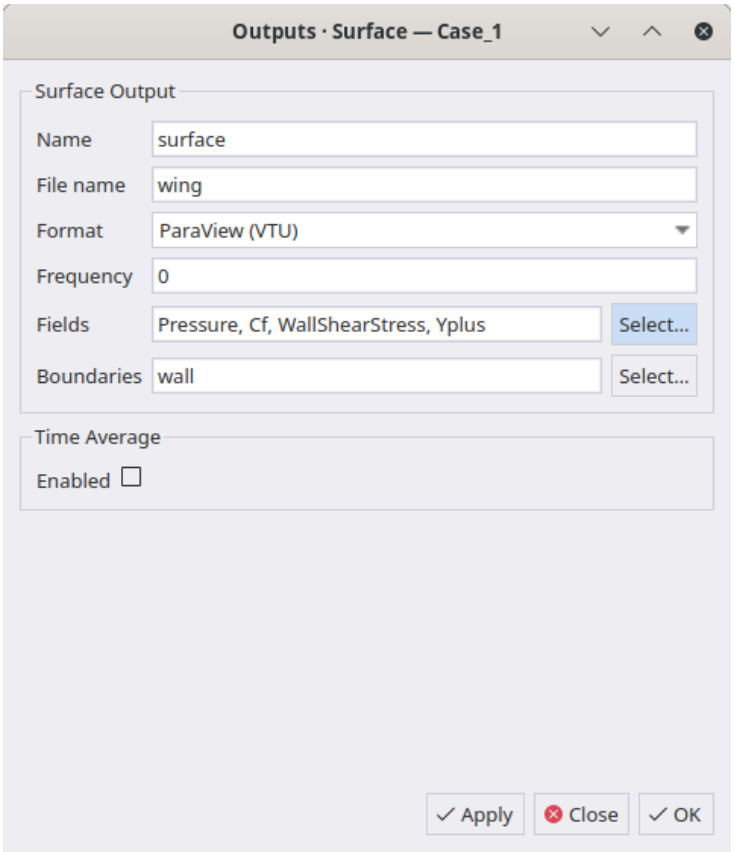


Figure 23.3: A surface-output entry writing pressure, skin friction, wall shear stress, and y+ on the named wall boundary.

```
    "slices": [{
      "name": "mid_plane",
      "file_name": "slice_mid",
      "format": "vtu",
      "enabled": true,
      "start_iter": 0,
      "frequency": 100,
      "fields": ["Pressure", "Mach"],
      "normal": [0, 0, 1],
      "origin": [0, 0, 0.5]
    }]
  }
}
```

`surfaces` carries wall/boundary quantities such as `Cf`, `Yplus`, and wall pressure — force-and-moment post-processing territory. It is **not** what shows flow structures like a vortex street or a shock: for that you need `volumes` (the full-domain field). In a 2D case in particular, a `surfaces` entry is just the wall curve — useful for `Cf`/`WSS`/wall-pressure plots (which are already covered by a force/pressure `monitoring` history CSV in most cases), but pointless for animating the flow field itself. Don't enable `surfaces` output purely to build a wake/shock animation; enable `volumes` instead and leave `surfaces` at `frequency: 0` unless you specifically need per-frame wall data.

23.6 Choosing a `frequency` for volume/surface output

`frequency` is in solver iterations, not physical time, so the right value depends on what the output is for:

- **Quantitative comparison only** (final `CD`/`CL`, a converged wall-pressure distribution, and so on): `frequency: 0` (final iteration only) is enough — there is no reason to pay for intermediate files.
- **Animation of an unsteady phenomenon** (vortex shedding, a moving shock, transient startup): work backward from how many frames per cycle you want, then convert to an iteration interval:

  `frequency ≈ (period of the phenomenon) / (target frames per period) / Δt`

  For example, a shedding period `T_shed ≈ 0.358 s` and a fixed BDF Δt of `1e-3 s` gives `T_shed / Δt ≈ 358` steps per period; a `frequency: 25` yields about 14 frames per period — light enough to keep file counts low over a long run (tens of thousands of steps) while still showing the motion. Wanting a smoother ~25–30 frames per period instead means using a smaller `frequency` (or, for an explicit run whose Δt is itself CFL-driven and not fixed, recomputing the target iteration interval from the current Δt rather than assuming it stays constant across the run).

Every volume/surface write costs real disk space (a fine 2D mesh's flow volume file can run a few megabytes each), so before committing to a frequency for a run that will take many hours, multiply out `(iterations to run) / frequency × (size of one file)` and compare against available disk — it's a cheap sanity check that avoids discovering you picked too dense an interval after the run is already hours in.

23.7 time_average

Instead of the old `statistics` block, attach `time_average` to each output entry individually.

```
{
  "outputs": {
    "volumes": [{
      "name": "mean_flow",
      "format": "vtu",
      "fields": ["Pressure", "Velocity_x"],
      "time_average": {
        "enabled": true,
        "start_iter": 500,
        "rms": false,
        "fields": ["Pressure", "Velocity_x"]
      }
    }]
  }
}
```

Time averaging is supported for volumes only. Enabling it on a surface or slice is rejected.

23.8 diagnostics

```
{
  "outputs": {
    "diagnostics": {
      "state_extrema": { "enabled": true, "frequency": 1 },
      "on_diverge": { "volume_dump": true },
      "fr_reference_compare": { "path": "", "centred": true },
      "dump_linsys": { "enabled": false, "iter": [100, 200], "path": "linsys" }
    }
  }
}
```

`state_extrema` records the min/max of every conserved variable (and where it occurred) to `solver_state_extrema.csv`. Its `frequency` controls the diagnostic interval. Disable it to avoid the associated GPU reduction and MPI merge.

Diagnostic	Use	Operational note
`state_extrema`	Locate a nonphysical state before or during divergence.	Costs a GPU reduction and MPI merge at each configured frequency.
`on_diverge`	Write the volume field at divergence.	`volume_dump` is the only key: the dump follows the configured volume output rather than taking its own field list.
`dump_linsys`	Inspect the active linear system at selected outer iterations.	Development or solver-diagnosis tool; do not leave it enabled in normal production runs.
`fr_reference_compare`	Compare an FR solution against a reference implementation.	Cross-validation only, not ordinary result output.

`dump_linsys` writes the active solver's Jacobian and RHS in Matrix Market format. Its `iter` list accepts at most 16 outer iterations; an empty `path` uses the run output directory.

`dump_linsys` is a diagnostic option that dumps the linear system to a file at the specified iterations.

`print_inner_iter` (dual-time inner-vs-outer row selection, shared between the console and the residual CSVs) lives under `monitoring`, not here.

FR also doesn't use a separate `fr_dump` or `fr_force_monitor`. It uses the standard `volumes[]` and `monitoring.monitors[]`. Use `outputs.restart` for solution-point snapshots; the former `diagnostics.fr_snapshot` key was removed. `fr_reference_compare` is for cross-checking against a reference implementation.

23.9 Setting this from the CLI

```
{
  "outputs": {
    "surfaces": [{
      "name": "wing_surface",
      "boundary_names": ["wing"],
      "file_name": "wing_surface",
```

```
      "format": "vtu",
      "frequency": 1000,
      "fields": ["Pressure", "Cp", "Mach"]
    }]
  }
}
```

23.10 Rotating-frame energy field

`Rothalpy` is `I = H − U·v` in J/kg, where U is grid/frame velocity. It is available in volume, surface, and slice outputs. In rotating DBNS cases it helps compare the relative-frame energy balance.

Surface output requires explicit `boundary_names` and `fields`; field bundles such as `default` are not accepted there. The removed `wallvars` bundle must be replaced with individual fields such as `Cf`, `WallShearStress`, `HeatFlux`, and `Yplus`. Wall quantities require DBNS viscous no-slip walls for every selected boundary. Surface CGNS and surface/slice time averaging are rejected. Explicit fields that the physics cannot produce are rejected rather than silently omitted; 2D `Helicity` is also rejected.

24 Monitoring

history CSV, console, residual, named monitors

24.1 What this setting is

Monitoring is the instrumentation that lets you judge, while the run is still going, whether it's actually converging. Don't rely on the residual alone — record whatever physical quantity actually matters for the problem alongside it: lift/drag, mass flow, a probe value. For steady analyses, judge convergence by that quantity settling, not just the residual. For unsteady analyses, judge it by the period, mean, and amplitude of that quantity settling into a repeatable pattern.

24.2 GUI configuration

Field	Description
Console	How often to print progress.
Residual	Whether to record the equation-residual CSV, and how often.
History file	A CSV file recording quantities of interest.
Monitor	Force coefficients, mass flow, probes, and other quantities to track.

`monitoring` records solver state and quantities of interest to CSV and the console at each iteration.

```
{
  "monitoring": {
    "history_files": [{
      "id": "main",
      "file_name": "history",
      "frequency": 1,
```

```
      "enabled": true
    }],
    "console": { "frequency": 10, "residual_basis": "scaled" },
    "solver_state": { "frequency": 1 }
  }
}
```

24.3 history_files

You can define several CSV streams. A monitor's `history_file` refers to one by its `id`.

24.3.1 Choosing `frequency` for post-processing (e.g. FFT)

If a history CSV feeds a frequency-domain analysis later — extracting a vortex-shedding frequency from a force-coefficient trace, for example — the effective sampling rate is `1 / (frequency · Δt)`. For a valid FFT read this needs to clear the Nyquist limit (at least twice the frequency you're trying to resolve) with real margin, not just barely. In practice this is rarely a real constraint: at a physical `timestep` chosen to already resolve the phenomenon well (say, several hundred steps per expected period, as is typical for an implicit BDF run), recording every iteration (`frequency: 1`) oversamples by two to three orders of magnitude relative to Nyquist — harmless for FFT quality, and cheap in disk terms for a scalar CSV row. Recording less often only matters if the file size itself becomes a problem (a very long explicit run with a tiny, CFL-driven Δt can otherwise produce a multi-hundred-thousand-row CSV) — in that case, dropping `frequency` to, say, every 100 or 1000 iterations still leaves enormous Nyquist margin for anything in the audible-vibration-to-shedding frequency range.

24.4 console

Prints iteration status to the console every `frequency` iterations.

```
{
  "monitoring": {
    "console": {
      "frequency": 10,
      "residual_basis": "scaled"
    }
  }
}
```

`residual_basis` selects which basis the console residual column is displayed on.

Value	Display
`scaled` (default)	`R / R₀` — orders of magnitude dropped from the start. Mean-flow `R₀` is the max over the first 5 iterations; turbulence equations use `‖a_p·φ_p‖`.
`absolute`	Raw `log10(L2 RMS)`.

`scaled` uses the same basis as the `run_control.convergence.residual_reduction` criterion. Which mode is active is announced once in the console right after the solver starts.

This key only affects the console column. The CSV always writes both bases to separate files — `solver_residual.csv` is absolute, and `solver_residual_reduction.csv` is scaled.

24.5 print_inner_iter

A sibling of `console` and `solver_state`, not nested under either — it applies symmetrically to both. In dual-time (`bdf1`/`bdf2`) unsteady analyses, setting it `true` prints/records the **inner** (pseudo-time) row instead of the outer physical-step row, for both the console and the `solver_state` CSVs.

```
{ "monitoring": { "print_inner_iter": false } }
```

24.6 solver_state

A separate stream that records `solver_residual.csv` (and its companions, `solver_residual_reductio` and the time-integration log). The default frequency is `1` for DBNS/PBNS and 20 for FR; an explicit value overrides that. `0` records only the final iteration.

```
{
  "monitoring": {
    "solver_state": {
      "frequency": 1,
      "console_fields": ["Flow_Res", "Flow_Rho", "CFL_Max", "Wall_Time"]
    }
  }
}
```

There's no `enabled` flag here — this stream is the run's convergence record, not a display you toggle off; to write less of it, raise `frequency` (even `0`, final-iteration-only, still gives you the minimum record).

`console_fields` picks which residual/CFL columns appear on the console, replacing the default expansion (one column per residual variable, plus CFL). It takes solver-exposed scalar names (`Flow_Res`, `CFL_Max`, `Wall_Time`, and so on) — **not** the names of your own `monitors[]` blocks. A monitor's console column is controlled by that monitor's own `console: true`, independently of this key.

24.7 monitors

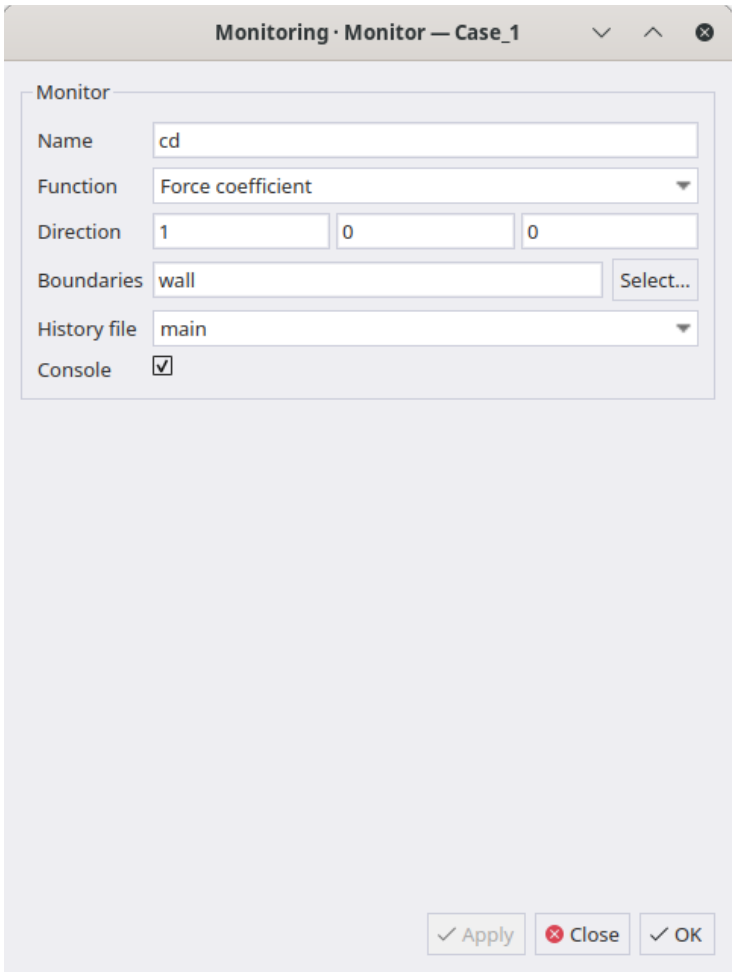


Figure 24.1: A force-coefficient monitor projected in the x direction, integrated over the named wall boundary, routed to the main history file, and shown on the console.

The GUI changes the available rows when **Function** changes. The selected values are written to one entry in `monitoring.monitors`; the JSONC form below shows the same contract and

also documents functions not visible in this particular capture.

A monitor is defined by three axes — **where** (`target`), **how** (`function`), and, for functions that need it, **what** (`variable`). Every function's primary output is a single scalar; the block's name is both the CSV column name and the console column name (when `console: true`).

```
{
  "monitoring": {
    "monitors": [
      {
        "name": "CFy",
        "target": { "boundaries": ["AIRFOIL"] },
        "function": "force_coefficient",
        "direction": [0.0, 1.0, 0.0],
        "history_file": "main",
        "console": true
      },
      {
        "name": "CFx",
        "target": { "boundaries": ["AIRFOIL"] },
        "function": "force_coefficient",
        "direction": [1.0, 0.0, 0.0],
        "history_file": "main",
        "console": true
      },
      {
        "name": "CMz",
        "target": { "boundaries": ["AIRFOIL"] },
        "function": "moment_coefficient",
        "axis": [0.0, 0.0, 1.0],
        "origin": [0.25, 0.0, 0.0],
        "history_file": "main",
        "console": true
      },
      {
        "name": "wake_pressure",
        "target": { "point": [2.0, 0.5, 0.0] },
        "function": "probe",
        "variable": "Pressure",
        "history_file": "main"
      },
      {
        "name": "lift_to_drag",
        "function": "expression",
        "expression": "CFy / CFx",
        "history_file": "main"
      }
```

```
    ]
  }
}
```

24.7.1 target

Form	Meaning
`{ "boundaries": [...] }`	A surface function integrated/evaluated over one or more named boundaries.
`{ "point": [x, y, z] }`	A probe: sample the flow field at a single point.
(omitted)	Only valid for `function: "expression"`, which references other monitors instead of the flow field directly.

24.7.2 function

Function	What it computes	Needs `variable`?
`force`, `force_coefficient`	∫(p − p∞)n̂ dA + ∫τ⃡·n̂ dA, projected onto `direction` (`force_coefficient` also divides by q∞·S_ref)	No
`moment`, `moment_coefficient`	∫r⃗ × (stress) dA, projected onto `axis` about `origin` (default: `reference.origin_moment`)	No
`area`	∫dA	No
`mass_flow`	∫ρV·n̂ dA	No
`momentum_flux`	∫ρ(V·n̂)² dA	No
`integral`	∫φ dA	Yes
`area_average`	∫φ dA ÷ A	Yes
`flow_rate`	∫(ρV·n̂)φ dA — signed physical flux of φ	Yes
`mass_average`	∫φ|ρV·n̂| dA ÷ ∫|ρV·n̂| dA (backflow-safe)	Yes
`probe`	Point sample of φ	Yes
`expression`	Evaluates `expression` against other monitors' values (no `target`)	No

`variable` takes a FlowVariable name and is **only** accepted by the five functions above that use it — supplying it elsewhere (or omitting it where required) is rejected at validation time.

> **Tip**
>
> Force and moment monitors on the same `target` share a single GPU reduction — defining all six body-axis components (`CFx`/`CFy`/`CFz`/`CMx`/`CMy`/`CMz`) on the same boundary set costs one surface integral pass, not six. There is no composite "all six at once" monitor anymore; write one block per component and let the shared-target reduction absorb the cost.

`direction` (force) and `axis` (moment) don't have to be a coordinate axis — the old body-axis `CFx`/`CFy` is just the special case `direction = [1,0,0]` / `[0,1,0]`. For wind-axis CL/CD at angle of attack α, write `direction: [-sin α, cos α, 0]` (lift) / `[cos α, sin α, 0]` (drag) explicitly — the solver normalizes whatever vector you give it.

If a viscous run's `target` includes a no-slip wall, force/moment monitors automatically get two extra CSV columns, `<name>.pressure` and `<name>.friction`, decomposing the total (`CFy ≈ CFy.pressure + CFy.friction`). An inviscid run only ever has the one total column.

> **Warning**
>
> A scalar function (`area`, `integral`, `area_average`, …) applied to **several boundaries at once** does not sum across them — it produces one column per boundary, named `<name>.<boundary>`. There is no plain `<name>` column to put on the console for a multi-boundary scalar monitor; reference the individual `<name>.<boundary>` columns instead (including in `run_control.convergence.monitor.fields`).

24.8 Time-averaging a monitor

A monitor doesn't take the `time_average.fields` used by output blocks — averaging applies to the monitor's entire scalar output instead.

```
{
  "monitoring": {
    "monitors": [{
      "name": "cd_mean",
      "target": { "boundaries": ["AIRFOIL"] },
      "function": "force_coefficient",
      "direction": [1.0, 0.0, 0.0],
      "history_file": "main",
      "console": true,
      "time_average": {
        "enabled": true,
```

```
        "start_iter": 500,
        "rms": true
      }
    }]
  }
}
```

The old flat `statistics_*` keys are no longer used.

24.9 Setting this from the CLI

```
{
  "monitoring": {
    "console": { "frequency": 10 },
    "solver_state": { "frequency": 1 },
    "history_files": [{
      "id": "main",
      "file_name": "history",
      "frequency": 1,
      "enabled": true
    }]
  }
}
```

25 Expert parameters

GPU accumulation, linear algebra, adaptive CFL, and turbulence preconditioners

> **Warning**
>
> Use defaults for ordinary cases first. Adjust this block only for performance tuning, algorithm comparisons, or diagnostics.

```
{
  "expert_parameters": {
    "gpu_edge_accum": "atomic",
    "spectral_radius_factor": 0.25,
    "partitioner": { "type": "auto", "imbalance": 0.03 },
    "krylov": { "orthogonalization": "cgs2" },
    "vtu_encoding": "binary",
    "vtkhdf_compression": 1
  }
}
```

25.1 GPU edge accumulation

`gpu_edge_accum`	Meaning
`atomic` (default)	Atomic accumulation path
`coloring`	Coloring-based race-free accumulation

25.2 Linear algebra

The choice of linear-algebra library belongs to `numerical_method.linear_solver`; only the Krylov knobs sit in `expert_parameters`. Omit both to keep the defaults.

```
{
  "numerical_method": {
    "linear_solver": {
      "backend": "cw_linalg",
      "row_scaling": false
    }
  },
  "expert_parameters": {
    "krylov": { "orthogonalization": "cgs2" }
  }
}
```

Key	Default	Meaning
`linear_solver.backend`	`cw_linalg`	`cw_linalg` (all builds) or `amgx` (NVIDIA AMGX; Linux CUDA builds with `WAKE_WITH_AMGX`). `amgx` covers the DBNS flow system on a single rank with `euler_implicit`, `linear_solver.type: "fgmres"`, and a `dilu`, `jacobi`, or `amg` flow preconditioner; anything else is a validation error.
`linear_solver.amgx.config` / `.config_file`	`""`	An AMGX configuration string, or a file path relative to the deck. Give at most one; either one replaces the automatic translation. Only read with `backend: "amgx"`.
`linear_solver.row_scaling`	`false`	Experimental row equilibration ($1/\max\|A_{ij}\|$) before the solve. Cannot be combined with `linelet` or `backend: "amgx"`.
`expert_parameters.krylov.orthogonalization`	`cgs2`	`cgs2` is Classical Gram-Schmidt with reorthogonalization; `mgs` is Modified Gram-Schmidt.

25.3 Viscous spectral-radius factor

This is the viscous pseudo-time stability factor for `models.type=navier_stokes`. Its default is `0.25`, and it is not used for Euler cases.

```
{ "expert_parameters": { "spectral_radius_factor": 0.25 } }
```

> **Note**
>
> Fine-tuning the `run_control.cfl.type=adaptive` (HANIM/ANK) workflow lives at `numerical_method.dbns.nonlinear_control`, not here — there's no `expert_parameters` knob for it.

25.4 Turbulence preconditioner

Configure `numerical_method.linear_solver.preconditioner.turb`; the older `expert_parameters.turb_preconditioner` key has been removed. See Shared linear solver for system-specific defaults and stationary sweep counts.

25.5 MPI partitioner

`partitioner` is an object, not a bare string.

```
{
  "expert_parameters": {
    "partitioner": {
      "type": "auto",
      "imbalance": 0.03,
      "kahip": { "quality": "eco" }
    }
  }
}
```

Key	Meaning
`type`	`auto` (default), `metis`, `kahip`, `linear`, `parhip`, or `parmetis` (simple linear partitioning, for diagnosis and reproduction)

Key	Meaning
`imbalance`	Maximum partition imbalance fraction (`0.03` = 3%); the concept is backend-neutral (KaHIP's `imbalance` and METIS's `ufactor` mean the same thing here)
`kahip.quality`	`fast`, `eco`, or `strong` — KaHIP-specific, only meaningful with `type: "kahip"`

`auto` falls back through `metis` → `kahip` → `linear` in that order, depending on which partitioner libraries the build was linked against.

> **Note**
>
> State-extrema diagnostics (conserved-variable min/max + location, useful for locating a divergence) moved out of `expert_parameters` on 2026-08-27 — it's `outputs.diagnostics.state_extrema` now.

25.6 Other settings

- `vtu_encoding`: `ascii` or `binary`, default `binary` — applies to `format: "vtu"` output only (`vtkhdf` is always binary).
- `vtkhdf_compression`: compression level for `format: "vtkhdf"` output.
- `jacobian_type` and `jacobian_fix_factor` no longer belong in `expert_parameters`. Configure them in each scheme's `convective` subblock.

25.7 Precision and Krylov storage

Precision is selected by the executable: `wake_<2|3>d_fp64`, `wake_<2|3>d_mixed`, or `wake_<2|3>d_fp32`. The former `krylov.precision` input has been removed. FP64 uses double-precision Krylov arithmetic; MIXED and FP32 use fp32.

```
{
  "expert_parameters": {
    "krylov": {
      "orthogonalization": "cgs2",
      "restart": 0,
      "offdiag_fp16": false
    }
  }
}
```

`restart: 0` (or a value at least `linear_solver.max_iter`) disables FGMRES restart. A smaller value reduces basis memory but may slow convergence; the total iteration cap remains `max_iter`. `offdiag_fp16` stores off-diagonal Jacobian blocks in fp16 with a per-row power-of-two scale, retaining fp32 diagonals and arithmetic. It is available only in MIXED/FP32 builds, is ignored with a warning in FP64 builds, and is ignored for linelet. It is independent of `linalg.row_scaling`.

25.8 Mesh and material diagnostics

- `mesh_check: "warn" | "fatal"`: warn (default) or stop on mesh metric violations.
- `eos_table.disable: true`: use the analytic Peng–Robinson path for CPU verification; unsupported in GPU builds.
- `mean_flow_viscous_scheme: "avg_grad_corrected" | "avg_grad"`: corrected face gradients (default) or simple averaging for mean-flow viscous fluxes.
- `wall_velocity_law: "auto" | "spalding" | "sa" | "sst"`: keep `auto` for supported wall treatment; explicit alternatives are experimental.
- `fr_vtu_high_order: true`: high-order VTK Lagrange cells for FR hex/quad output; use `false` for viewers requiring linear subcells.

25.8.1 Parallel mesh reading

`partitioner.type: "auto"` uses serial reading and METIS, falling back to KaHIP and then linear partitioning when unavailable. Explicit `parhip` or `parmetis` selects parallel reading and requires compatible HDF5 CGNS meshes and build support; unsupported combinations fail instead of falling back. ParMETIS is restricted to internal evaluation builds, not distributed binaries. `metis.method` accepts `auto`, `recursive`, or `kway`; `parhip.quality` accepts `ultrafast`, `fast` (default), or `eco`. If imbalance is omitted, METIS recursive uses 0.001 and kway uses 0.03; KaHIP/ParHIP/ParMETIS use 0.03.

26 Expressions

Named constants and reusable physical-quantity expressions

26.1 What this setting is

`expressions` is an optional case-wide dictionary of named constants and reusable expressions. Use it for repeated physical values and coordinate-based profiles. It does not apply to numerical controls such as CFL or tolerances.

26.2 Parameter table

Key or form	Type	Meaning	Valid use
`expressions.<name>: <number>`	number	Named constant.	Physical-quantity expression.
`expressions.<name>: <string>`	string	Reusable expression, expanded before evaluation.	Physical-quantity expression.
`"$name"`	string	Complete-field reference shorthand.	Physical-quantity field only.
`"name + 1"`	string	Inline expression; bare names are expanded in larger expressions.	Physical-quantity field only.
`{ "type": "expression", "expr": "…" }`	object	Explicit expression form; equivalent to the bare string.	Physical-quantity field only.
`{ "type": "constant", "value": … }`	object	Explicit constant form; a bare number is auto-converted to it.	Physical-quantity field only.

26.3 Constants and reusable expressions

```
{
  "expressions": {
    "V0": 10.0,
    "H": 0.02,
    "inlet_profile": "V0*(1-(y/H)^2)"
  },
  "domains": [{
    "boundaries": [{
      "name": "INLET",
      "type": "velocity_inlet",
      "mesh_markers": ["INLET"],
      "velocity": { "type": "expression", "expr": "$inlet_profile" },
      "flow_direction": [1.0, 0.0, 0.0]
    }]
  }]
}
```

A number defines a constant. A string defines a reusable expression. Inside a field, write the expression text either as a bare string or in the `expr` key of an explicit expression object; the two are equivalent. Use the `$` prefix only when the whole text is one reference, such as `{ "type": "expression", "expr": "$p_inlet" }`; inside a larger expression, use the bare name, as in `{ "type": "expression", "expr": "p_inlet + dp" }`.

26.4 Reserved names and validation

`x`, `y`, `z`, and `t` are reserved in boundary, initialization, and source expressions. `T`, `P`, and `rho` are additionally reserved in material and source expressions; `pi` is always reserved. Do not redefine them.

Situation	Wake CFD behavior	Action
Expression refers to another named expression	Dependencies resolve before evaluation.	Keep definitions small and readable.
A reference cycle exists	Input validation fails.	Break the cycle.
A name is unknown	Input validation fails.	Correct it or define it.
Expression used for CFL, tolerance, or relaxation	Not supported.	Use a numeric value.

Start with a constant baseline, then introduce one expression at a time.

27 Complete input structure

Strict JSONC and 14 top-level blocks

A Wake CFD case combines mesh, physical models, domains and boundary conditions, numerical methods, and run and output settings in one input structure. Desktop enters the same information through panels; CLI, batch, and automation workflows pass it as a JSONC file. Read this page after the individual topics to review their relationships and the reproducible run unit.

27.1 Format rules

- `//` line comments and `/* ... */` block comments are allowed.
- JSON5 syntax such as trailing commas and unquoted keys is not allowed.
- Write arrays as arrays; for example, `"mesh": [{ ... }]` and `"domains": [{ ... }]`.
- Write string lists as arrays too; for example, `"fields": ["default"]`.

27.2 The 14 top-level blocks

27.3 The 13 top-level blocks

Block	Required	Role
`mesh`	✓	Mesh files and scope
`analysis`	✓	Solver, time, and dimension
`models`	✓	Euler/NS, regime, and turbulence
`materials`	✓	EOS and material properties
`reference`	✓	Reference pressure, temperature, density, velocity, and moment reference

Block	Required	Role
`domains`	✓	Domains, zones, and boundary conditions
`interfaces`	△	Periodic, stage, GGI, and other domain connections
`numerical_method`	✓	Solver-specific numerical methods and linear solvers
`initialization`	✓	Initialization or restart
`run_control`	✓	Iteration, convergence, and time stepping
`outputs`	△	Restart and file output
`monitoring`	△	Console, residual CSV, and named monitors
`expert_parameters`	△	Internal algorithm tuning
`expressions`	△	Named constants and reusable physical-quantity expressions

`wake_version` is an additional optional file-level metadata key. `expressions` is a regular optional top-level configuration block.

Key	Required	Role
`wake_version`	△	Solver version when the file was authored (for compatibility checks)

27.4 Minimal structure

```
{
  "mesh": [{ "file_name": "mesh.cgns", "format": "cgns" }],
  "analysis": { "solver": "dbns", "time": "steady", "dimension": 2 },
  "models": { "type": "euler" },
  "materials": [{
    "name": "Air",
    "type": "pure",
    "eos": { "type": "ideal_gas", "gas_constant": 287.058 }
```

```
  }],
  "reference": {
    "pressure": 101325.0,
    "temperature": 273.15,
    "density": 1.225
  },
  "domains": [{
    "name": "fluid",
    "material": "Air",
    "boundaries": []
  }],
  "initialization": { "type": "auto" },
  "numerical_method": {
    "dbns": {
      "convective": { "scheme": "roe", "spatial_order": 2 },
      "time_integration": "euler_implicit"
    }
  },
  "run_control": {
    "iteration": 1000,
    "cfl": { "type": "constant", "fluid": { "value": 5.0 } }
  }
}
```

27.5 Four analysis axes

Axis	Key	Values
Physics	`models.type`	`euler`, `navier_stokes`
Solver	`analysis.solver`	`dbns`, `pbns`, `fr`
Turbulence	`models.turbulence.model`	`none`, `sa`, `sst`, LES/DES models
Time	`analysis.time`	`steady`, `unsteady`

For example, the combination of `models.type=navier_stokes` and `models.turbulence.model=sa` represents RANS-SA. There is no separate `rans` solver enum.

27.6 Constants and expressions

For the full reference, including reusable-expression expansion, reserved names, and validation behavior, see Expressions.

Physical-quantity fields—such as boundary-condition velocity, pressure, and temperature, and initialization fields—can accept expressions instead of constants. The following forms are supported.

```
{
  "expressions": {
    "V0": 10.0,
    "H": 1.0,
    "inlet_profile": "V0*(1-(y/H)**2)"
  },

  "domains": [{
    "boundaries": [
      { "velocity": 10.0 },
      { "velocity": { "type": "constant", "value": 10.0 } },
      { "velocity": { "type": "expression", "expr": "$inlet_profile" } },
      { "velocity": "V0*(1-(y/H)**2)" }
    ]
  }]
}
```

The boundary objects above illustrate field syntax only; they omit required name, type, and mesh-marker bindings and are not a runnable case.

The four `velocity` forms above, in order: a bare number; the same thing written explicitly (the bare number is auto-converted to this form during parsing); a reference to a named expression declared in `expressions` (the whole text is the reference, so it takes the `$name` shorthand); and an inline expression written as a bare string, mixing a named constant (`V0`) into a larger formula (no `$` needed when it is only part of the expression). The object form carries the text in its `expr` key; the bare string is the same thing written short.

Numerical parameters such as CFL, tolerance, and relaxation do not use this form; it applies only to physical-quantity fields. A handful of names are reserved by context and can't be redefined: `x`, `y`, `z`, `t` (boundary/initialization/source expressions), `T`, `P`, `rho` (material/source expressions), and `pi`.

27.7 Inheritance and composition

```
{ "$extends": "base.jsonc", "run_control": { "iteration": 3000 } }
```

`$extends` and `--overlay` use RFC 7396 Merge Patch: objects merge while arrays are replaced. To modify only a specific array element, use `--patch` with RFC 6902 JSON Pointer.

```
$extends chain → --overlay → --patch
```

For execution and case reuse, see Running Wake CFD; for each block's meaning, see its individual topic in this reference.

Part VI

6. Python API

pyWakeCFD is a Python SDK for authoring Wake CFD cases programmatically, controlling solver execution, and automating post-processing workflows.

- Python SDK — Case authoring, validation, execution control, and parameter-sweep automation

Related guides

- CLI guide — Batch and HPC execution
- AI agent workflows — Conversational automation

28 Python SDK

Author and run Wake CFD cases in Python, and automate results and sweeps

pyWakeCFD is an SDK for assembling Wake CFD cases, validating input, and running and observing solvers from Python workflows. It does not connect CUDA or MPI directly to the Python process. Python orchestrates work and manages result files while the solver runs as a separate process.

28.1 When to use it

- To create design conditions and compare results in a Jupyter notebook
- To automatically vary repeated conditions such as Mach number, angle of attack, and CFL
- To connect Wake CFD to internal optimization, post-processing, and data pipelines
- To create, validate, and track CLI batch jobs from Python code

While the CLI provides run commands and scheduler jobs, the Python SDK is an automation layer for managing cases and repeated work on top of it.

28.2 Installation

Python 3.10 or later is required. Package name and installation path will be finalized here when public distribution is confirmed.

```
python -m pip install pywakecfd

# To read history CSV files as pandas DataFrames
python -m pip install "pywakecfd[results]"
```

In the development repository, install from the SDK directory in editable mode.

```
cd launcher/python
python -m pip install -e .
```

28.3 Shortest workflow

Create a case with Case, validate it against the solver, then run it with Run.

```
from pywakecfd import Case, Mesh, Run

case = (
    Case("naca")
    .mesh(Mesh.from_file("naca.su2"))
    .analysis(dimension=2)
    .fluid("Air")
    .freestream(mach=0.5, aoa_deg=2.0)
    .dbns(convective_scheme="roe", order=2)
    .run_control(max_iter=10_000)
)

case.validate("wake_2d_mixed")
run = Run(case, executable="wake_2d_mixed", workdir="runs/naca").start()
returncode = run.wait()
print(run.status())
```

The SDK writes runnable JSONC and a manifest to the selected working directory and manages result directories created by the solver.

```
runs/naca/
├─ case.jsonc
├─ pywakecfd.manifest.json
└─ case_001/
   ├─ wake.status.json
   ├─ history CSV
   ├─ VTK results
   └─ restart files
```

28.4 Start from an existing JSONC case

Read JSONC exported from the GUI or included in the distribution, then change only the conditions in Python.

```
from pywakecfd import Case

base = Case.from_jsonc("examples/onera-m6-wing/case.jsonc")
mach_085 = base.set("domains.0.boundaries.1.mach", 0.85)
mach_085.emit("runs/onera-m6-mach-085")
```

`set()` returns a changed new Case without modifying the original. This is useful for safely creating comparison cases and retaining their history.

28.5 Parameter sweeps

Sweep creates multiple cases from a baseline Case and paths and values to change.

```
from pywakecfd import Case, Sweep

base = Case.from_jsonc("examples/onera-m6-wing/case.jsonc")
sweep = Sweep(base, [
    {"numerical_method.dbns.cfl": 5.0},
    {"numerical_method.dbns.cfl": 20.0},
    {"numerical_method.dbns.cfl": 50.0},
])

runs = sweep.run_local(
    executable="wake_3d_mixed",
    root="runs/cfl-sweep",
    max_workers=2,
)
```

`run_local()` manages the number of local subprocesses. The SDK does not submit to cluster schedulers such as SLURM, PBS, or MPI, so run generated cases in HPC environments through the CLI and scheduler scripts.

28.6 Results and runtime control

Run.`results` reads files from the run directory rather than solver memory.

```
results = run.results
print(results.history_files())
print(results.volume_files())
print(results.restart_files())
```

When Run starts a gRPC-enabled solver, you can read its state and control CFL, stopping, and restart saving.

```python
with run.runtime(timeout=10) as client:
    for state in client.stream(interval=10):
        print(state.iter, state.res_l2, state.cfl)
        if state.res_l2 < 1e-8:
            client.stop()
            break
```

Available commands are `set_cfl`, `pause`, `resume`, `dump`, `dump_restart`, and `stop`. Runtime control requires a gRPC-enabled solver build.

28.7 Related documentation

- Batch, MPI, and HPC execution: CLI guide
- Project automation with an AI agent: Connect an AI agent

Part VII

7. Automation and integration

Wake CFD integrates with HPC schedulers, external orchestration systems, and AI agents. These interfaces support cluster job submission, real-time solver control, conversational case authoring, and live manipulation of the Desktop project model.

Integration paths

Cluster execution Submit Wake CFD jobs to Slurm clusters with resource management, array jobs, and dependency chains. See Slurm integration.

AI agent workflows Connect Claude or compatible MCP clients to author cases conversationally, validate input, orchestrate runs, or attach to an open Wake CFD Desktop session. The agent and GUI can share the same project model, so machine-driven edits appear in the interface for review. See Connect an AI agent.

When to use each interface

Interface	Use when
Slurm	Running parameter sweeps or production jobs on shared GPU clusters
AI agent	Exploring case configurations, automating repeatable edits, or controlling a live Desktop project

For programmatic case authoring without conversational interaction, use the Python API instead.

29 Slurm integration

Run Wake CFD as a cluster batch job

Wake CFD CLI passes case input to an executable, so under Slurm it runs as a batch job like any MPI program. The job script, mesh, input files, and output paths must be accessible from compute nodes. On a typical cluster, the submission directory is on shared storage and can be used directly.

29.1 Basic job script

The example below is a minimal template for a 3D case using four MPI ranks. Adjust partition, time limit, account, and module names for your cluster policy.

```
#!/bin/bash
#SBATCH --job-name=wake-wing
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --time=02:00:00
#SBATCH --output=logs/%x-%j.out
#SBATCH --error=logs/%x-%j.err

set -euo pipefail

cd "$SLURM_SUBMIT_DIR"
mkdir -p logs results

# Load cluster MPI and compiler modules if needed.
# module load mpi

srun ./wake_3d_mixed.exe case.jsonc
```

Save the file as `wake.slurm`, then submit it.

```
sbatch wake.slurm
squeue --me
```

29.2 Working directory

Meshes, `case.jsonc`, base input files referenced by `$extends`, and restart files must all be readable at the same paths on compute nodes. Writing output to the submission directory or a shared results path lets you post-process it directly from a login node. For clusters using node-local scratch, add steps to copy input at job start and return results to shared storage before it ends.

29.3 Next steps

- For executable and input-file basics, see Running Wake CFD.
- Add GPU, MPI-launcher, and CPU-binding settings to the job script after reviewing cluster policy and the supported scope of the Wake CFD distribution.
- Automate parameter sweeps by combining Slurm job arrays with the Python API.

30 Connect an AI agent

Build and run Wake CFD projects together through MCP

Wake CFD is designed for AI agents to do more than generate JSONC text. Through the shared **WakeProjectModel**, an agent can assemble, validate, save, run, and control cases using the same project state that Wake CFD Desktop displays. Engineers provide analysis goals and constraints; agents perform repeatable edits while the engineer reviews the resulting model and solution.

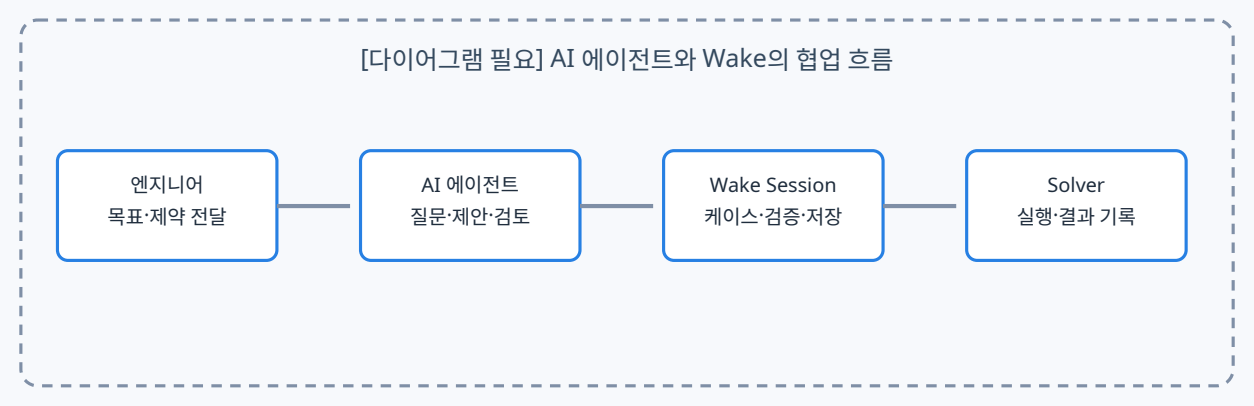


Figure 30.1: Placeholder for the AI-agent workflow. Before public release, replace with a diagram showing an AI client, Wake CFD Desktop or CLI, and result review.

30.1 What agents can do

Through the Model Context Protocol (MCP), an agent can perform the following Wake CFD tasks.

- Create projects and analysis cases, including derived cases from existing ones
- Register meshes and change physical models, boundary conditions, and numerical settings
- Inspect the input schema and packaged examples
- Validate JSONC, apply patches and overlays, and configure parameter and mesh sweeps
- Save projects, start local solvers, and check run status
- Attach to an open Wake CFD Desktop session and edit the live project shown in the GUI

- Control an active run by checking status, pausing, resuming, stopping, changing supported runtime parameters, or requesting a one-shot output

For example, you can request:

> Starting from the ONERA M6 example, create cases for Mach 0.82 and 0.85, enable Cp output for each, then validate the input.

Rather than merely copying an example, the agent can retain derived cases and their changes, keeping comparison conditions and work history together.

30.2 Connection method

Wake CFD provides `wake_session_mcp`, a standard-input/output MCP server. It exposes the same session operations used by the Desktop control plane: project and case edits, schema and example lookup, validation, run start, and run control. The server can use a local model for headless authoring or proxy requests to a live Desktop session.

In a release distribution, connect by registering the Wake CFD server in the AI client's MCP configuration. Configuration-file locations and UI names differ by client, but the required information is the same.

```
{
  "mcpServers": {
    "wake": {
      "command": "wake_session_mcp",
      "args": [
        "--schema-dir", "<Wake CFD installation path>/share/input-reference",
        "--examples-dir", "<Wake CFD installation path>/examples",
        "--solver", "<Wake CFD installation path>/bin/wake_3d_mixed"
      ]
    }
  }
}
```

On Windows, add `.exe` to executable names. Linux installations may add `wake_session_mcp` and `wake_3d_mixed` to PATH. Follow release notes and installation instructions for actual distribution paths.

> **Note**
>
> **Two connection modes**
> The default configuration starts a private, headless session owned by the MCP process. To operate the project currently open in Wake CFD Desktop, register an attached server instead:

```
{
  "mcpServers": {
    "wake-desktop": {
      "command": "wake_session_mcp",
      "args": ["--attach"]
    }
  }
}
```

`--attach` discovers the Desktop loopback control endpoint from its local `control.json`. An explicit `host:port` can be supplied when required. If no Desktop session is available, the server can fall back to a local model, so the same MCP client remains useful for headless authoring.

30.3 Recommended workflow

1. Open or create a project, then decide whether the agent should use a headless session or attach to the open Desktop project.
2. Give the agent specific goals, conditions to change, and constraints to preserve.
3. Review the agent's changes and input-validation results in the GUI; attached edits are reconciled into the live project automatically.
4. Ask the agent to save and run the case, then inspect residuals, force coefficients, and spatial results in the GUI or result tools.
5. If results are sound, promote a derived case to the baseline or extend the sweep.

30.4 Use with Wake CFD Desktop

When Wake CFD Desktop is running, use the `--attach` MCP configuration to connect to its loopback ControlServer. The MCP client and Desktop then share one live `WakeProjectModel`: an agent can change cases, meshes, and settings, while the Desktop reconciles those changes into its project tree and editors. Conversely, edits made in the GUI are available to the agent the next time it reads the session.

Only one control surface drives the live session at a time. When the Desktop's in-process AI Assistant is enabled, its own agent owns the control path; external `--attach` access is available when that assistant is off. This prevents two agent loops from editing the same case concurrently.

30.5 Engineer review points

AI agents speed up repeated case setup and comparison, but responsible analysis judgment remains with the engineer. Before running, check the following.

- Mesh-marker and boundary-condition connections
- Consistency of physical models, reference values, and units
- Convergence monitors and required output settings
- Whether results match expected physical behavior
- Whether the agent is connected to the intended project (headless session or live Desktop session)

For detailed input meanings, see the Input and configuration reference; for execution and convergence diagnostics, see How to use Wake CFD.