NASA/TM–20230004211

FUN3D Manual: 14.0.1

William K. Anderson, Robert T. Biedron, Jan-Reneé Carlson, Joseph M. Derlaga, Cameron T. Druyor Jr., Peter A. Gnoffo, Dana P. Hammond, Kevin E. Jacobson, William T. Jones, Bil Kleb, Elizabeth M. Lee-Rausch, Gabriel C. Nastac, Eric J. Nielsen, Michael A. Park, Christopher L. Rumsey, James L. Thomas, Kyle B. Thompson, Aaron C. Walden, Li Wang, Stephen L. Wood, and William A. Wood

Langley Research Center, Hampton, Virginia

Boris Diskin and Yi Liu
National Institute of Aerospace, Hampton, Virginia

Xinyu Zhang
Analytical Mechanics Associates, Hampton, Virginia

April 2023

NASA/TM–20230004211

FUN3D Manual: 14.0.1

William K. Anderson, Robert T. Biedron, Jan-Reneé Carlson, Joseph M. Derlaga, Cameron T. Druyor Jr., Peter A. Gnoffo, Dana P. Hammond, Kevin E. Jacobson, William T. Jones, Bil Kleb, Elizabeth M. Lee-Rausch, Gabriel C. Nastac, Eric J. Nielsen, Michael A. Park, Christopher L. Rumsey, James L. Thomas, Kyle B. Thompson, Aaron C. Walden, Li Wang, Stephen L. Wood, and William A. Wood

Langley Research Center, Hampton, Virginia

Boris Diskin and Yi Liu
National Institute of Aerospace, Hampton, Virginia

Xinyu Zhang
Analytical Mechanics Associates, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

April 2023

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

This manual describes the installation and execution of FUN3D version 14.0.1, including optional dependent packages. FUN3D is a suite of computational fluid dynamics simulation and design tools that uses mixed-element unstructured grids in a large number of formats, including structured multiblock and overset grid systems. A discretely-exact adjoint solver may be used for formal design optimization, error estimation, and mesh adaptation. FUN3D also offers a reacting, real-gas capability and provides GPU acceleration of many common simulation options.

Contents

About this Document 11

Acknowledgments 12

Quick Start 13

1 Introduction 17
- 1.1 Primary Capabilities and Features . . . 17
- 1.2 Requirements . . . 18
- 1.3 Grid Generation . . . 18
- 1.4 Obtaining Fun3D . . . 19
- 1.5 Prebuilt Modules . . . 19

2 Conventions 20
- 2.1 Compressible Equations . . . 21
- 2.2 Incompressible Equations . . . 23
- 2.3 Generic Gas Equations . . . 24
- 2.4 Unsteady Flows . . . 25
- 2.5 Turbulent Flows . . . 26

3 Boundary Conditions 27
- 3.1 X-Symmetry, Y-Symmetry, and Z-Symmetry . . . 29

4 Grids 30
- 4.1 File Endianness . . . 30
- 4.2 Supported Grid Formats . . . 30
  - 4.2.1 AFLR3 Grids . . . 31
  - 4.2.2 FAST Grids . . . 31
  - 4.2.3 VGRID Grids . . . 32
  - 4.2.4 FieldView Grids . . . 32
  - 4.2.5 FELISA Grids . . . 33
  - 4.2.6 Fun2D Grids . . . 33
- 4.3 Translation of Additional Grid Formats . . . 34
  - 4.3.1 PLOT3D Grids . . . 34
  - 4.3.2 CGNS Grids . . . 35
- 4.4 Implicit Lines . . . 36
- 4.5 Grid Mirroring . . . 36

5 Flow Solver, nodet 38
- 5.1 Flow Solver Execution . . . 38
- 5.2 Command Line Options . . . 38
- 5.3 Output Files . . . 39

5.3.1 Flow Visualization 39
5.3.2 Flow Visualization Output From Existing Solution . . 40

6 Adjoint Solver, DUAL 41
6.1 Convergence of the Linear Adjoint Equations 41
6.2 Required Directory Hierarchy and Executing DUAL 41
6.3 `rubber.data` 42
6.4 Output Files 42

7 Grid Motion 43
7.1 Overview of `moving_body.input` 44
7.2 Choosing the Type of Mesh Movement 44
7.3 4×4 Transform Matrices 45
7.4 Verifying Grid Motion Inputs 47
7.5 Static Grid Manipulations 48
7.5.1 Grid Transform 48
7.5.2 Body Transform 48
7.5.3 Surface From File 48
7.6 Aeroelastic Modal Analysis 49
7.6.1 Mode Shape Input Files 50
7.6.2 Mode History Files 51
7.6.3 Modal Mesh Deformation 51

8 Grid Adaptation 52
8.1 Grid Adaptation General Solver Guidance 53
8.2 Scripting Grid Adaptation 54
8.2.1 Input File `case_specifics` for `f3d` Script 55
8.3 External Grid Adaptation with Geometry Evaluation 57
8.3.1 REFINE Execution 57
8.3.2 External Grid Adaptation Geometry Preparation . . . 58
8.3.3 External Grid Adaptation Bootstrap 61
8.3.4 External Grid Adaptation Required Namelists 64
8.3.5 External Grid Adaptation Workflow via `f3d` and `case_specifics` 66
8.3.6 ONERA M6 Grid Adaptation Tutorial 68
8.4 External Grid Adaptation without Geometry 71
8.5 Deprecated Grid Adaptation without Geometry 71
8.5.1 Built-in Grid Adaptation Plane Specification and Grid Freezing 72
8.5.2 Built-in Feature-Based and Interpolation-Error Adaptation 73
8.5.3 Built-in Adjoint-Based Adaptation 74
8.5.4 Built-in 2D Grid Adaptation 75

9 Design Optimization 76
9.1 Objective/Constraint Functions . . . 77
9.1.1 Terminology . . . 77
9.1.2 Functional Form . . . 78
9.2 Some Details on Specific Objective/Constraint Functions . . . 80
9.2.1 Lift-to-Drag Ratio (Keyword: `clcd`) . . . 80
9.2.2 Rotorcraft Figure of Merit (Keyword: `fom`) . . . 80
9.2.3 Rotorcraft Propulsive Efficiency (Keyword: `propeff`) . 81
9.2.4 Rotorcraft Thrust (Keyword: `rtr_thrust`) . . . 81
9.2.5 RMS of Stagnation Pressure (Keyword: `pstag`) . . . 81
9.2.6 Engine Inflow Distortion (Keyword: `distort`) . . . 81
9.2.7 Near-field p/p_∞ Pressure Target (Keyword: `boom_targ`) 83
9.2.8 Coupled sBOOM Ground-Based Signatures, Noise Metrics, and Equivalent Areas (Keyword: `sboom`) . . . 84
9.2.9 Supersonic Mach Cut Equivalent Area Distribution (Keyword: `ae`) . . . 85
9.2.10 RMS of Pressure in User-Defined Box, Pointwise dp/dt, $d\rho/dt$ (Keyword: `press_box`) . . . 87
9.2.11 Target Pressure Distributions (Keyword: `cpstar`) . . . 87
9.3 Geometry Parameterizations . . . 87
9.3.1 Surface Grid Extraction . . . 88
9.3.2 Access to Executables . . . 88
9.3.3 Notes on Using Sculptor™ . . . 88
9.3.4 Using Other Parameterization Packages . . . 89
9.4 Design Optimization Directory Structure . . . 91
9.5 Contents of the `ammo` Directory . . . 92
9.5.1 &design namelist in `design.nml` . . . 92
9.6 Contents of the `description.i` Directory . . . 95
9.6.1 Geometry Parameterization Files . . . 96
9.6.2 `rubber.data` . . . 98
9.6.3 `ae_target.dat` (optional) . . . 105
9.6.4 `body_grouping.data` (optional) . . . 105
9.6.5 `command_line.options` (optional) . . . 106
9.6.6 `cpstar.data.j` (optional) . . . 106
9.6.7 `files_to_save.data` (optional) . . . 106
9.6.8 `machinefile` (optional) . . . 107
9.6.9 `pressure_target.dat` (optional) . . . 107
9.6.10 `transforms.i` (optional) . . . 107
9.7 Contents of the `model.i` Directory . . . 108
9.8 Running the Optimization . . . 108
9.8.1 Filesystem Latency Problems . . . 110
9.9 Multiobjective Design . . . 110

9.9.1 KSOPT 110
9.9.2 PORT, SNOPT™ 110
9.10 Multipoint Design 110
9.10.1 Linear Combination of Objective Functions 112
9.10.2 Combination of Objective Functions using the Kreisselmeier-Steinhauser Function 112
9.10.3 Single-Point Objective Function with Off-Design Constraint Functions 112
9.11 Optimization of Two-Dimensional Geometries 113
9.12 Rotor Optimization with Actuator Disk Modeling 113
9.13 Using a Different Optimization Package 118
9.14 Implementing New Cost Functions/Constraints 119
9.15 Forward Mode Differentiation Using Complex Variables 120

10 Accelerating Solutions Using Graphics Processing Units 122
10.1 GPU Quick Start Guide 122
10.2 Background 122
10.3 Requirements 123
10.4 Supported Options 123
10.4.1 Perfect Gas Path Options 124
10.4.2 Incompressible Gas Path Options 125
10.4.3 Generic Gas Path Options 125
10.5 GPU Execution Strategy and Concerns 126
10.5.1 Running a Case 127
10.6 Device Information 127
10.7 Device Selection 128
10.8 Expected Performance 129
10.9 Using CUDA-Aware MPI and GPUDirect™ 129
10.10 Running Multiple MPI Ranks per GPU 130
10.11 Implications of Asynchronous Execution 132
10.12 Troubleshooting 133
10.13 Half-Precision Solver 134

11 Stabilized Finite Elements (SFE) 135
11.1 Supported Options 136
11.2 SFE Execution 137
11.3 SFE Output Files 137
11.4 Adjoint 137
11.5 Static Aeroelastic Analysis with SFE 138
11.6 Linearized Frequency Domain Analysis 138
11.6.1 LFD Inputs 139
11.6.2 `[project_rootname]_gafs.dat` 140

11.6.3 Visualization of the Linearized Flow State 140

12 Yoga **142**

References **143**

A Installation **156**

A.1 Extracting Files 156
A.2 Configure Introduction 156
A.3 Alternative Installation Path 157
A.4 Fortran Compiler Option Tuning (FTune) 158
A.5 C++ Compiler Requirements 158
A.6 C++ Compiler Option Tuning (CXXTune) 158
A.7 Support for Multiple Simultaneous Configurations 159
A.8 Specialized AVX-512 Support 160
A.9 GPU Support 160
A.10 Complex Variable Version 160
A.11 SLAT 161
A.12 Yoga 161
A.13 External Libraries 161
A.13.1 MPI 161
A.13.2 ParMETIS 163
A.13.3 Zoltan 165
A.13.4 SUGGAR++-1.0.10 or Higher 166
A.13.5 DiRTlib v1.40 or higher 166
A.13.6 6-DOF 167
A.13.7 KSOPT 167
A.13.8 PORT 167
A.13.9 SNOPT™ 168
A.13.10NPSOL™ 168
A.13.11DOT/BIGDOT™ 168
A.13.12Tecplot™ 168
A.13.13CGNS 169
A.13.14sBOOM 169
A.13.15SPARSKIT 170
A.13.16ESP 170

B Fun3D Input Files **171**

B.1 `stop.dat` 172
B.2 `[project_rootname].flow` 172
B.3 `remove_boundaries_from_force_totals` 172
B.4 `fun3d.nml` 173
B.4.1 &project 176

B.4.2 &raw_grid . . . 177
B.4.3 &force_moment_integ_properties . . . 180
B.4.4 &grid_transform . . . 183
B.4.5 &body_transform . . . 187
B.4.6 &governing_equations . . . 191
B.4.7 &reference_physical_properties . . . 194
B.4.8 &noninertial_reference_frame . . . 197
B.4.9 &inviscid_flux_method . . . 199
B.4.10 &turbulent_diffusion_models . . . 206
B.4.11 &turbulence . . . 210
B.4.12 &spalart . . . 211
B.4.13 &gammaretsst . . . 213
B.4.14 &code_run_control . . . 214
B.4.15 &nonlinear_solver_parameters . . . 219
B.4.16 &hanim . . . 223
B.4.17 &update_limits . . . 232
B.4.18 &shuffle_generic . . . 235
B.4.19 &linear_solver_parameters . . . 237
B.4.20 &elasticity_gmres . . . 239
B.4.21 &boundary_conditions . . . 243
B.4.22 &periodicity . . . 265
B.4.23 &two_d_trans . . . 266
B.4.24 &three_d_trans . . . 268
B.4.25 &filters . . . 270
B.4.26 &flow_initialization . . . 272
B.4.27 &component_parameters . . . 275
B.4.28 &time_avg_params . . . 285
B.4.29 &global . . . 286
B.4.30 &volume_output_variables . . . 289
B.4.31 &boundary_output_variables . . . 302
B.4.32 &sampling_output_variables . . . 315
B.4.33 &sampling_parameters . . . 332
B.4.34 &spatial_avg . . . 346
B.4.35 &slice_data . . . 349
B.4.36 &overset_data . . . 362
B.4.37 &rotor_data . . . 366
B.4.38 &adapt_metric_construction . . . 368
B.4.39 &adapt_mechanics . . . 376
B.4.40 &massoud_output . . . 382
B.4.41 &sonic_boom . . . 385
B.4.42 &sboom . . . 387
B.4.43 &equivalent_area . . . 395
B.4.44 &press_box_function . . . 396

B.4.45 &pstag_function . . . 398
B.4.46 &fan_distortion . . . 399
B.4.47 &special_parameters . . . 400
B.4.48 &partitioning . . . 402
B.4.49 &fwh_acoustic_data . . . 406
B.4.50 &vortex_generator . . . 409
B.4.51 &gust_data . . . 413
B.4.52 &gpu_support . . . 417
B.5 `moving_body.input` . . . 420
B.5.1 &body_definitions . . . 422
B.5.2 &forced_motion . . . 430
B.5.3 &observer_motion . . . 434
B.5.4 &motion_from_file . . . 437
B.5.5 &surface_motion_from_file . . . 440
B.5.6 &sixdof_motion . . . 441
B.5.7 &aeroelastic_modal_data . . . 445
B.5.8 &composite_overset_mesh . . . 452
B.6 `rotor.input` . . . 453
B.6.1 Header . . . 454
B.6.2 Actuator Surface Model . . . 455
B.6.3 Rotor Reference System . . . 457
B.6.4 Rotor Loading . . . 458
B.6.5 Blade Parameters . . . 459
B.6.6 Blade Element Parameters for `Load Type`=3 . . . 460
B.6.7 Pitch Control Parameters for `Load Type`=3 . . . 460
B.6.8 Prescribed Flap Parameters . . . 461
B.6.9 Prescribed Lag Parameters . . . 461
B.6.10 `bladegeom_rotor.dat` . . . 462
B.7 `tdata` . . . 464
B.7.1 `perfect_gas` Keyword . . . 464
B.7.2 `equilibrium_air_t` Keyword . . . 464
B.7.3 `equilibrium_air_r` Keyword . . . 465
B.7.4 `one` Keyword . . . 465
B.7.5 `two` Keyword . . . 465
B.7.6 `FEM` Keyword . . . 466
B.8 `species_thermo_data` . . . 466
B.9 `kinetic_data` . . . 468
B.10 `species_transp_data` . . . 470
B.11 `species_transp_data_0` . . . 471
B.12 `hara_namelist_data` . . . 471
B.13 `controller.nml` . . . 478
B.13.1 &tunnel_control . . . 479
B.14 `fun3d.nml` inputs for SFE . . . 482

B.15 `sfe.cfg` . . . 482
B.15.1 Nonlinear Controller . . . 482
B.15.2 Linear algebra . . . 484
B.15.3 Dynamic reordering . . . 487
B.15.4 Thermodynamics . . . 488
B.15.5 Boundary conditions . . . 488
B.15.6 Turbulence model parameters . . . 489
B.15.7 Smoothing . . . 489
B.15.8 Residual smoothing . . . 491
B.15.9 Code Run Control . . . 491
B.15.10Adjoint . . . 492
B.16 Yoga Input Files . . . 493

C Troubleshooting 494
C.1 What if I cannot run the solver in parallel? . . . 494
C.2 What if the solver has trouble starting or reports NaNs or infinities? . . . 494
C.3 What if the forces and moments aren't steady or residuals don't converge to steady-state? . . . 495
C.4 What if the solver dies unexpectedly? . . . 495
C.5 What if the compiler complains, "This module file was not generated by any release of this compiler"? . . . 496
C.6 What if the solver dies with an error like "Probable incomplete read of namelist: &some_namelist"? . . . 496
C.7 What if a segmentation fault occurs after "Calling ParMETIS"?496
C.8 What if the solver dies with an error like "input statement requires too much data" after echoing the number of tetrahedra and nodes for a VGRID mesh? . . . 496
C.9 What if the solver reports that the Euler numbers differ? . . 496
C.9.1 Euler Number Description . . . 497
C.9.2 Determining the Impact of an Euler Number Mismatch 498
C.10 What if the initial residuals reported by the elasticity solver are unusually large when the input surface mesh is intended to recover the existing mesh? . . . 499

About this Document

This manual is intended to guide an application engineer through configuration, compiling, installing, and executing the FUN3D simulation package. The focus is on the most commonly exercised capabilities. Therefore, some of the immature or rarely exercised capabilities are intentionally omitted in the interest of clarity.

This document is updated and released with each subsequent version of FUN3D. In fact, a significant portion is automatically extracted from the FUN3D source code. If you have difficulties, find any errors, or have any suggestions for improvement please contact the authors at

`Fun3D-Support@lists.nasa.gov`

We would like to hear from you.

Acknowledgments

A large portion of this content originated as a web-based manual, which contained contributions from Jan-Renee Carlson, Karen Bibb, Bill Jones, Jeff White, Eric Lynch, Dave O'Brien, Marilyn Smith, and Clara Helm. Sriram Rallabhandi provided the description of sBOOM. The namelist documentation is parsed from Fun3D source code, which is written by the developers. Jaime Akkala suggested the use of modal decomposition of mesh deformation for internal aeroelastic cases. Contributors to Fun3D include Ponnampalam (Bala) Balakumar, Karen Bibb, Bob Biedron, Jan Carlson, Mark Carpenter, Peter Gnoffo, Dana Hammond, Bill Jones, Bil Kleb, Beth Lee-Rausch, Eric Nielsen, Hiroaki Nishikawa, Mike Park, Chris Rumsey, Jim Thomas, Veer Vatsa, Jeff White, Alejandro Campos, Rajiv Shenoy, Marilyn Smith, Joe Derlaga, Natalia Alexandrov, W. Kyle Anderson, Harold Atkins, Bill Wood, Austen Duffy, Clara Helm, Chris Cordell, Kyle Thompson, Hicham Alkandry, Shelly Jiang, Eric Lynch, Jennifer Abras, Nicholas Burgess, Dave O'Brien, Tommy Lambert, Yi Liu, Ved Vyas, Shatra Reehal, Kan Yang, Andrew Sweeney, Brad Neff, Genny Pang, Gregory Bluvshteyn, Dan Gerstenhaber, Geoff Parsons, Rena Rudavsky, Aaron Walden, Li Wang, Xinyu Zhang, Sreekanth Ravindran, and Kevin Jacobson. Since the release of the NASA TM version of the manual (v12.4), the following people have reported bugs, usability issues, and/or suggested documentation improvements: Bill Sellers, Tim Conners, Karen Deere, Boris Diskin, Ryan Durscher, Troy Lake, Kelly Thomas, Ken Waithe, Joel Malissa, Jacob Cooper, Jia Feilin, Paresh Parikh, Ray Gomez, and Eli Shellabarger.

Quick Start

This section takes you from source code tarball to a rudimentary flow solution using single processor execution on a typical Unix-style environment (e.g. Linux, Mac® OS) with a Fortran compiler and the GNU Make utility. FUN3D is most commonly executed in parallel, but the intent here is to provide the most basic installation, setup, and execution of the FUN3D flow solver without the complexity of any third-party libraries or packages.

See section 1.4 for instructions on obtaining the FUN3D source code tarball. Once you have it, unpack the source code tarball, configure it for your system (section A), compile it, and add the executables directory to your search path. For C Shell, e.g.,

```
tar zxf fun3d_intg-14.0.1-*.tar.gz
cd fun3d_intg-14.0.1-*
mkdir _seq
cd _seq
  ../configure --prefix=${PWD}
  make install
  setenv PATH ${PWD}/bin:${PATH}
cd ..
```

For Bourne Shell, the `setenv` command is `export PATH=${PWD}/bin:${PATH}`. The change to the `PATH` environment variable can be made permanently by adding the `setenv` or `export` command to your shell start up file. Next, move to the `doc/quick_start` directory,

```
cd doc/quick_start
```

where you will find a very coarse 3D wing grid (`inv_wing.fgrid`) intended for inviscid flow simulation (section 4). Also in this directory are the associated boundary conditions file `inv_wing.mapbc` (section 3) and a FUN3D input file `fun3d.nml` in Fortran namelist format (section B.4).

Execute the flow solver (section 5.1) by running the command

```
nodet
```

This should produce screen output similar to

```
FUN3D 14.0.1-38ffd23 Flow started 04/17/2023 at 10:26:27 with 1 processes
Contents of fun3d.nml file below------------------------
&project
  project_rootname = 'inv_wing'
/
&raw_grid
  grid_format = 'fast'
  data_format = 'ascii'
/
&governing_equations
  viscous_terms = 'inviscid'
```

```
/
&reference_physical_properties
  mach_number     = 0.7
  angle_of_attack = 2.0
/
&code_run_control
  restart_read       = 'off'
  steps              = 150
  stopping_tolerance = 1.0e-12
/
&global
  boundary_animation_freq = -1
/
&boundary_output_variables
  number_of_boundaries = -1
  boundary_list        = '2,7-8'
/
Contents of fun3d.nml file above------------------------
  ... Use the reference Mach number for the freestream:  T
    ... opening inv_wing.fgrid
    ... nnodesg: 6309 ntet: 35880 ntface: 1392

cell statistics: type,      min volume,      max volume, max face angle
cell statistics:  tet,  0.38305628E-05,  0.14174467E+02,  143.526944837
cell statistics:  all,  0.38305628E-05,  0.14174467E+02,  143.526944837

     ... Constructing partition node sets for level-0...                35880 T
     ... Edge Partitioning ....
     ... Boundary partitioning....
     ... Reordering for cache efficiency....
     ... Write global grid information to inv_wing.grid_info
     ... Time after preprocess TIME/Mem(MB):     175477.52    182.31    182.19
  NOTE: kappa_umuscl set by grid: .00

 Grid read complete
  y-symmetry metrics modified/examined: 1102/1102
 Iter            density_RMS  density_MAX   X-location   Y-location   Z-location
    1  0.611767950151108E-04  0.68431E-03  0.25001E-01  0.43479E-01  0.00000E+00
       Lift  0.775285036093966E-01         Drag  0.352117247285817E-01
    2  0.342644121204683E-04  0.11389E-02  0.41388E+01  0.24706E+01 -0.21321E+01
```

⋮

```

   61  0.217555843541799E-12  0.83222E-11  0.65000E+01  0.00000E+00  0.65000E+01
       Lift  0.809619929262272E-01         Drag  0.108249850731833E-01

 Writing inv_wing.flow (version 12.2)
  inserting current history iterations 61
 Time for write: .0 s


 Writing boundary output: inv_wing_tec_boundary.dat
  Time step: 61, ntt: 61, Prior iterations: 0
 Done.
```

If FUN3D completed successfully, a Mach 0.7 inviscid flow over a very coarse representation of an ONERA M6 semi-span wing [1] at two degrees angle of attack is available. If not, please refer to Troubleshooting on page 494.

With visualization software capable of reading Tecplot™ files, you can visualize various surface quantities with `inv_wing_tec_boundary.dat` as shown

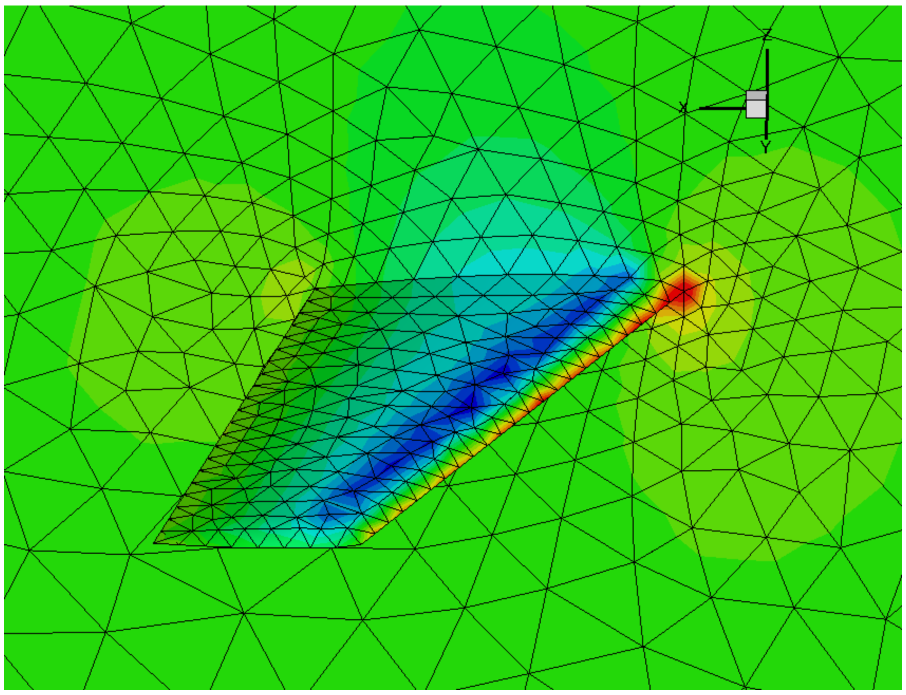


Figure 1: Mach 0.7 flow about a coarse ONERA M6 semi-span wing at 2 degrees angle of attack.

by the pressure contours in Fig. 1. Iterative convergence history can be plotted from `inv_wing_hist.dat` as shown in Fig. 2. Histories of all five conservation equation residual norms are denoted `R_1`–`R_5`, and the lift coefficient convergence history is denoted `C_L`.

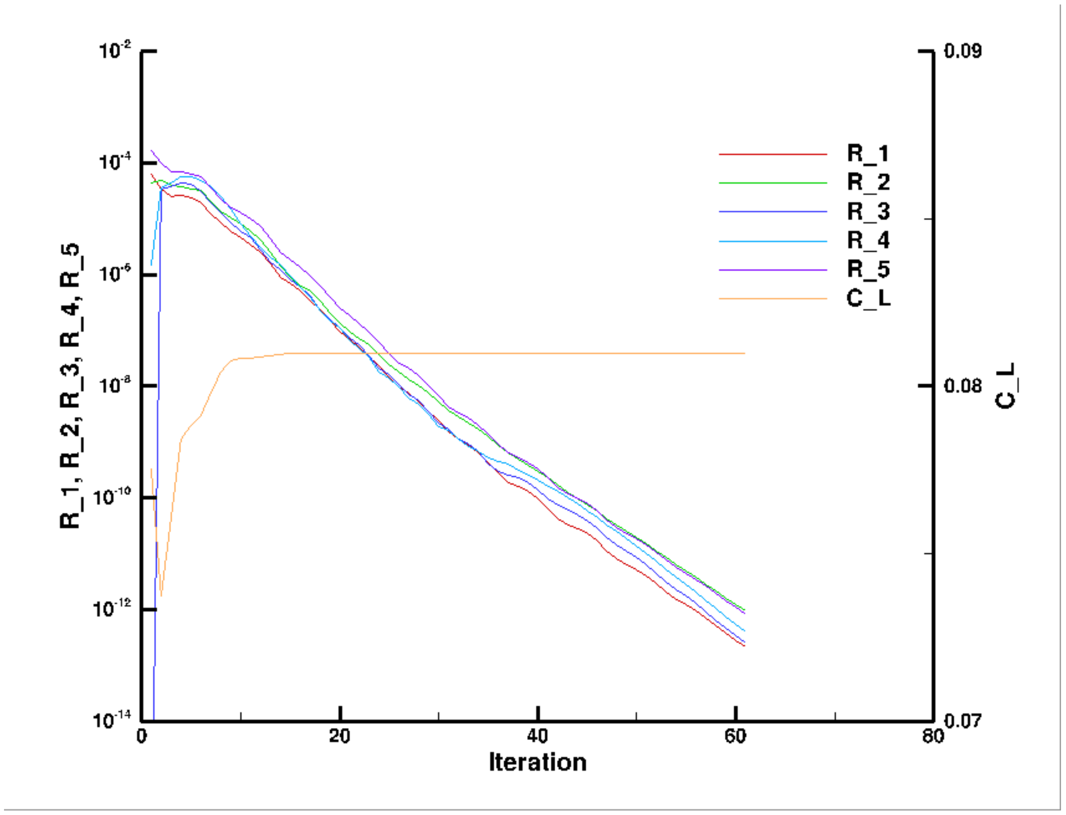


Figure 2: Iterative convergence history for coarse ONERA M6 wing.

1 Introduction

Fun3D began as a research code in the late 1980s. [2] The code was created to develop new algorithms for unstructured-grid fluid dynamic simulations of incompressible and compressible transonic flows. The project has since grown into a suite of codes that cover not only flow analysis, but adjoint-based error estimation, mesh adaptation, and design optimization of fluid dynamic problems extending into the hypersonic regime. [3]

Fun3D is currently used as a production flow analysis and design tool to support NASA programs. Continued research efforts have also benefited by the improvements to stability, ease of use, portability, and performance that this shift to simultaneous support of development and production environments has required. These benefits also include the rapid evaluation of new techniques on realistic simulations and a rapid maturation of experimental techniques to production-level capabilities.

1.1 Primary Capabilities and Features

The primary capabilities of Fun3D are:

- Parallel domain decomposition with Message Passing Interface (MPI) communication for distributed computing
- Two-dimensional (2D) and Three-dimensional (3D) node-based, finite-volume discretization
- Thermodynamic models: perfect gas (compressible and incompressible) and thermochemical equilibrium, and nonequilibrium
- Time-accurate options from first- to fourth-order with temporal error controllers
- Upwind flux functions: flux difference splitting, flux vector splitting, artificially upstream flux vector splitting, Harten-Lax-van Leer contact, low dissipation flux splitting scheme, and others
- Turbulence models: Spalart-Allmaras, Menter k-omega SST, Wilcox k-omega, detached eddy simulation, and others, including specified or predicted transition
- Implicit time stepping where the linear system is solved using either point-implicit, line-implicit, or Newton-Krylov (multigrid is also under active development)
- Boundary conditions for internal flows and propulsion simulation including inlets, nozzles, and system performance
- Grid motion: time-varying translation, rotation, and deformation including overset meshes and six degrees of freedom trajectory computations

- Adjoint- and feature-based grid adaptation
- Gradient based sensitivity analysis and design optimization via hand-coded discrete adjoint for reverse mode differentiation and automated complex variables for forward mode differentiation

Before exploring more advanced applications (e.g., grid adaptation, moving grids, overset grids, design optimization), the user should become familiar with FUN3D's basic flow solving capabilities and have appropriate computational capability available as indicated in the next section.

1.2 Requirements

The FUN3D development team's typical computing platform is Linux clusters; so this is the most thoroughly tested environment for the software. A number of users also run on other UNIX-like environments including Mac OS X™; these platforms are supported as well. Users have also run on other architectures such as Microsoft Windows™-based PC's; however, the team cannot provide explicit support for these environments.

The user will need GNU Make and a Fortran compiler that supports at least the Fortran 2003 standard. During configuration, the Fortran compiler is tested. Any newer Fortran features or extensions are detected are used to the greatest extent possible. Intel®, Portland Group®, and GFortran are tested by an automated build framework.

While the code can be compiled to run on only a single processor, as demonstrated in the Quick Start section, most applications will require compiling against an MPI implementation and one of the supported domain decomposition libraries to allow parallel execution.

The flow solver uses approximately 2.4 kilobytes of memory per grid point for a perfect gas RANS simulation with a loosely-coupled turbulence model. For example, a grid with one million mesh points would require approximately 2.4 gigabytes of memory. Memory usage will increase slightly with the increase in the number of processors because of the increasing boundary data exchanged. Different solution algorithms and co-visualization options will also require additional memory. Typically, one CPU core per 50,000 grid points is suggested, where a 3D mesh of 20 million grid points would require 400 cores.

1.3 Grid Generation

FUN3D has no grid generation capability. For internal development at NASA, the most common sources of 3D grids are VGRID (ViGYAN, Inc. and NASA Langley), SolidMesh/AFLR3 (Mississippi State), Pointwise (Pointwise, Inc.), and GridEx (NASA Langley).

For 2D grids, the development team normally uses the AFLR2 software written by Prof. Marcum et al. at Mississippi State University Center for Advanced Vehicular Systems (CAVS) SimCenter. Scripts are available to facilitate the use of this grid generator, but the generator itself must be obtained from Prof. Marcum. BAMG [4] is also used for 2D grid generation and adaptation.

1.4 Obtaining Fun3D

Fun3D is export restricted and can only be given to a "U.S. Person," which is a citizen of the United States, a lawful permanent resident alien of the U.S., or someone in the U.S. as a protected political asylee or under amnesty. The word "person" includes U.S. organizations and entities, such as companies or universities, see 22 CFR §120.15 for the full legal definition.

To obtain the Fun3D software suite please visit NASA Software Catalog.

1.5 Prebuilt Modules

Prebuilt FUN3D modules are available in the NASA Advanced Supercomputing (NAS) and NASA LaRC K-cluster environments. Please visit the FUN3D website `https://fun3d.larc.nasa.gov` to get the instructions on how to use these modules.

2 Conventions

This chapter discusses the coordinate system orientation and nondimensionalization used by FUN3D. The nomenclature for this section is

a	=	Speed of sound
C	=	Sutherland constant
e	=	Energy per unit volume; generic-gas: per unit mass
f	=	Frequency
h	=	Enthalpy per unit volume; generic-gas: per unit mass
k	=	Thermal conductivity
L	=	Length
M	=	Mach number
p	=	Pressure
R	=	Gas constant
Re	=	Reynolds number
t	=	Time
T	=	Temperature
u, v, w	=	Cartesian components of velocity
x, y, z	=	Cartesian directions
α	=	Angle of attack
β	=	Angle of sideslip
γ	=	Heat capacity ratio
μ	=	Viscosity
ρ	=	Density

where an asterisk (*) denotes a dimensional quantity. A subscript *ref* denotes a reference quantity. For fluid variables, such as pressure, *ref* usually corresponds to the value 'at ∞' for external flows or another condition for internal flows. The units of various reference quantities must be consistent. For example, if the reference speed of sound is defined in feet/sec, then the dimensional reference length, $L^*{}_{ref}$, must be in feet. In what follows, L_{ref} is the length in the grid that corresponds to the dimensional reference length, $L^*{}_{ref}$; L_{ref} is considered dimensionless.

FUN3D's angle of attack, sideslip angle, and associated force coefficients are based on a body-fixed coordinate system:

- positive x is toward the back of the vehicle;
- positive y is toward the right of the vehicle; and

- positive z is upward

as shown in Fig. 3. This differs from the standard wind coordinate system by a 180 degree rotation about the y axis. The α and β flow angle conventions are shown in Fig. 4.

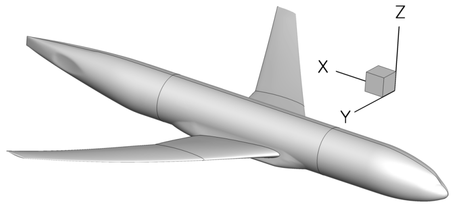


Figure 3: FUN3D body coordinate system.

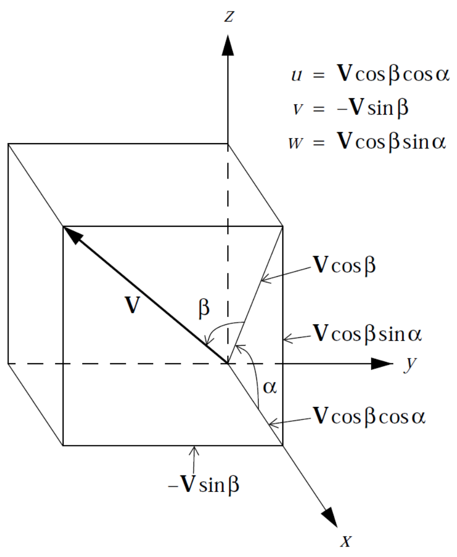

z
$u = \mathbf{V}\cos\beta\cos\alpha$
$v = -\mathbf{V}\sin\beta$
$w = \mathbf{V}\cos\beta\sin\alpha$
$\mathbf{V}\cos\beta$
V
β
$\mathbf{V}\cos\beta\sin\alpha$
y
α
$\mathbf{V}\cos\beta\cos\alpha$
$-\mathbf{V}\sin\beta$
x

Figure 4: FUN3D freestream flow angle definition.

2.1 Compressible Equations

$$\begin{aligned}
x &= x^*/(L^*{}_{ref}/L_{ref}) \\
y &= y^*/(L^*{}_{ref}/L_{ref}) \\
z &= z^*/(L^*{}_{ref}/L_{ref}) \\
t &= t^* a^*{}_{ref}/(L^*{}_{ref}/L_{ref})
\end{aligned}$$

$$\begin{aligned}
\rho &= \rho^*/\rho^*{}_{ref} & \rho_{ref} &= 1 \\
|V| &= |V^*|/a^*{}_{ref} & |V|_{ref} &= M_{ref}
\end{aligned}$$

$$
\begin{array}{llll}
u & = u^*/{a^*}_{ref} & u_{ref} & = M_{ref}\cos\alpha\cos\beta \\
v & = v^*/{a^*}_{ref} & v_{ref} & = -M_{ref}\sin\beta \\
w & = w^*/{a^*}_{ref} & w_{ref} & = M_{ref}\sin\alpha\cos\beta \\
p & = p^*/({\rho^*}_{ref}{a^*}^2_{ref}) & p_{ref} & = 1/\gamma \\
a & = a^*/{a^*}_{ref} & a_{ref} & = 1 \\
T & = T^*/{T^*}_{ref} & T_{ref} & = 1 \\
e & = e^*/({\rho^*}_{ref}{a^*}^2_{ref}) & e_{ref} & = 1/(\gamma(\gamma-1)) + M^2_{ref}/2
\end{array}
$$

To see how the nondimensional Navier-Stokes equations that are solved in FUN3D are obtained from their dimensional counterparts, it is sufficient to look at the unsteady, one-dimensional equations for conservation of mass, momentum, and energy:

$$\frac{\partial\rho^*}{\partial t^*} + \frac{\partial(\rho^* u^*)}{\partial x^*} = 0$$

$$\frac{\partial(\rho^* u^*)}{\partial t^*} + \frac{\partial}{\partial x^*}\left[\rho^* {u^*}^2 + p^* - \frac{4}{3}\mu^*\frac{\partial u^*}{\partial x^*}\right] = 0$$

$$\frac{\partial e^*}{\partial t^*} + \frac{\partial}{\partial x^*}\left[(e^* + p^*)u^* - \frac{4}{3}\mu^* u^*\frac{\partial u^*}{\partial x^*} - k^*\frac{\partial T^*}{\partial x^*}\right] = 0$$

where k^* is the thermal conductivity. For a thermally and calorically perfect gas, we also have the equation of state, the definition of the speed of sound, and the specific heat relation:

$$T^* = \frac{p^*}{\rho^* R^*}$$

$${a^*}^2 = \gamma R^* T^* \quad (\gamma = {c_p}^*/{c_v}^*)$$

$${c_p}^* - {c_v}^* = R^* \qquad R^*/{c_p}^* = (\gamma - 1)/\gamma$$

The laminar viscosity is related to the temperature via Sutherland's law

$$\mu^* = {\mu^*}_{ref}\frac{{T^*}_{ref} + C^*}{T^* \quad + C^*}\left(\frac{T^*}{{T^*}_{ref}}\right)^{3/2}$$

where $C^* = 198.6°\text{R}$ for air.

Substitution of the nondimensional variables defined above into the equation of state and the definition of the speed of sound gives:

$$T = \frac{\gamma p}{\rho} = a^2$$

Sutherland's law in nondimensional terms is given by

$$\mu = \frac{1 + C^*/T^*{}_{ref}}{T + C^*/T^*{}_{ref}} T^{3/2}$$

where C^* is the Sutherland constant ($C^* = 198.6°\text{R}$ for air) and where $T^*{}_{ref}$ is in degrees Rankine.

Substitution of the dimensionless variables into the conservation equations gives, after some rearrangement,

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u)}{\partial x} = 0$$

$$\frac{\partial (\rho u)}{\partial t} + \frac{\partial}{\partial x}\left[\rho u^2 + p - \frac{4}{3}\frac{M_{ref}}{Re_{L_{ref}}}\mu \frac{\partial u}{\partial x}\right] = 0$$

$$\frac{\partial e}{\partial t} + \frac{\partial}{\partial x}\left[(e+p)u - \frac{4}{3}\frac{M_{ref}}{Re_{L_{ref}}}\mu u \frac{\partial u}{\partial x} - \frac{M_{ref}}{Re_{L_{ref}} P_r (\gamma - 1)}\mu \frac{\partial T}{\partial x}\right] = 0$$

where P_r is the Prandtl number (generally assumed to be 0.72 for air)

$$P_r = \frac{c_p^* \, \mu^*}{k^*}$$

and where $Re_{L_{ref}}$, the Reynolds number per unit length in the grid, corresponds to the input variable `reynolds_number` in the `fun3d.nml` file. $Re_{L_{ref}}$ is related to the Reynolds number characterizing the physical problem, $Re_{L^*{}_{ref}}$ by

$$Re_{L_{ref}} = \frac{\rho^*{}_{ref}|V^*|_{ref}(L^*{}_{ref}/L_{ref})}{\mu^*{}_{ref}} = \frac{\rho^*{}_{ref}|V^*|_{ref}L^*{}_{ref}}{\mu^*{}_{ref}}\frac{1}{L_{ref}} = \frac{Re_{L^*{}_{ref}}}{L_{ref}}$$

2.2 Incompressible Equations

$$\begin{aligned}
x &= x^*/(L^*{}_{ref}/L_{ref}) \\
y &= y^*/(L^*{}_{ref}/L_{ref}) \\
z &= z^*/(L^*{}_{ref}/L_{ref}) \\
t &= t^*|V^*|_{ref}/(L^*{}_{ref}/L_{ref})
\end{aligned}$$

$$\begin{array}{llll}
|V| & = |V^*|/|V^*|_{ref} & |V|_{ref} & = 1 \\
u & = u^*/|V^*|_{ref} & u_{ref} & = \cos\alpha\cos\beta \\
v & = v^*/|V^*|_{ref} & v_{ref} & = -\sin\beta \\
w & = w^*/|V^*|_{ref} & w_{ref} & = \sin\alpha\cos\beta \\
p & = p^*/(\rho^*{}_{ref}|V^*|^2_{ref}) & p_{ref} & = 1
\end{array}$$

For incompressible flows, FUN3D does not model any heat sources. The temperature T^* is constant and so is the viscosity μ^*. After dividing through by a constant reference density, the one-dimensional continuity and momentum equations are:

$$\frac{\partial u^*}{\partial x^*} = 0$$

$$\frac{\partial u^*}{\partial t^*} + \frac{\partial}{\partial x^*}\left[u^{*2} + \frac{p^*}{\rho^*{}_{ref}} - \frac{4}{3}\frac{\mu^*{}_{ref}}{\rho^*{}_{ref}}\frac{\partial u^*}{\partial x^*}\right] = 0$$

The fundamental difference between the nondimensionalization of the compressible equations and the incompressible equations is that the sound speed is used in the former and the flow speed in the latter. Substitution of the dimensionless variables defined above into the conservation equations gives, after some rearrangement,

$$\frac{\partial u}{\partial x} = 0$$

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left[u^2 + p - \frac{4}{3}\frac{1}{Re_{L_{ref}}}\frac{\partial u}{\partial x}\right] = 0$$

where, exactly the same as in the compressible-flow path, the Reynolds number per unit length in the grid is

$$Re_{L_{ref}} = \frac{\rho^*{}_{ref}|V^*|_{ref}L^*{}_{ref}}{\mu^*{}_{ref}} = \frac{Re_{L^*{}_{ref}}}{L_{ref}}$$

2.3 Generic Gas Equations

The generic gas path requires all reference quantities (velocity, density, temperature) be entered in the meter-kilogram-second (MKS) system. The transport property nondimensionalization includes the effects of rescaling using the grid length conversion factor. The nondimensionalization of other flow variables follows the practice used to derive the Mach number independence principle. Neither Mach number nor Reynolds number can be used to define reference conditions; these are derived from the fundamental reference quantities. The derived Reynolds number is relative to a one meter reference length. Temperature is never nondimensionalized; it always appears in units of degrees Kelvin.

ρ	$= \rho^*/\rho^*{}_{ref}$	ρ^*_{ref}	[kg/m^3]
u	$= u^*/V^*{}_{ref}$	V^*_{ref}	[m/s]
v	$= v^*/V^*{}_{ref}$	T^*_{ref}	[K]
w	$= w^*/V^*{}_{ref}$	L^*_{ref}	[m]

$$
\begin{aligned}
a &= a^*/V^*{}_{ref} \\
p &= p^*/(\rho^*{}_{ref} V^{*2}{}_{ref}) \\
e &= e^*/V^{*2}_{ref} \\
h &= h^*/V^{*2}_{ref} \\
\mu &= \mu^*(T^*)/\rho^*{}_{ref} V^*_{ref} L^*_{ref}
\end{aligned}
$$

2.4 Unsteady Flows

One of the challenges in unsteady flow simulation is determining the nondimensional time step Δt. The number of time steps at that Δt necessary to resolve the lowest frequency of interest will impact the cost of the simulation and too large a Δt will corrupt the results with temporal errors. Time is nondimensionalized within FUN3D by

$$
\begin{aligned}
t &= t^* a^*{}_{ref}/(L^*{}_{ref}/L_{ref}) && \text{(compressible)} \\
t &= t^* |V^*|_{ref}/(L^*{}_{ref}/L_{ref}) && \text{(incompressible)}
\end{aligned}
$$

where, as in the previous sections, quantities denoted with * are dimensional.

In all unsteady flows, one or more characteristic times $t^*{}_{chr}$ may be identified. In a flow with a known natural frequency of oscillation (e.g., vortex shedding from a cylinder), or in situations where a forced oscillation is imposed (e.g., a pitching wing), a dominant characteristic time is readily apparent. In such cases, if the characteristic frequency in Hz (cycles/sec) is $f^*{}_{chr}$, then

$$t^*{}_{chr} = 1/f^*{}_{chr}$$

In other situations, no oscillatory frequency may be apparent (or not known a priori). In such cases, the time scale associated with the time it takes for a fluid particle (traveling at a nominal speed of $|V^*|_{ref}$) to pass the body of reference length $L^*{}_{ref}$ is often used:

$$t^*{}_{chr} = L^*{}_{ref}/|V^*|_{ref}$$

The corresponding nondimensional characteristic time is therefore given by:

$$
\begin{aligned}
t_{chr} &= t^*{}_{chr} a^*{}_{ref}/(L^*{}_{ref}/L_{ref}) && \text{(compressible)} \\
t_{chr} &= t^*{}_{chr} |V^*|_{ref}/(L^*{}_{ref}/L_{ref}) && \text{(incompressible)}
\end{aligned}
$$

Once the nondimensional characteristic t_{chr} is determined, the user must decide on an appropriate number of time steps (N) to be used for resolving that characteristic time. Then the nondimensional time step may be specified as:

$$\Delta t = t_{chr} \;/\; N$$

The proper value of N must be determined by the user. However, a reasonable rule of thumb for second-order time integration is to take $N = 200$. Note that if there are multiple frequencies requiring resolution in time, the most restrictive should be used to determine Δt.

2.5 Turbulent Flows

The turbulence equations are nondimensionalized by the same reference quantities as the Navier-Stokes equations. The nondimensionalized variables used with the turbulence models are:

$$\begin{aligned}
\mu &= \mu^* / {\mu^*}_{ref} \\
k &= k^* / {a^*}^2_{ref} \\
\omega &= \omega^* \mu^*_{ref} / \rho^*_{ref} {a^*}^2_{ref} \\
P_{\mathrm{k}} &= P^*_{\mathrm{k}} {L^*}^2_{ref} / \mu^*_{ref} {a^*}^2_{ref} \\
P_{\omega} &= P^*_{\omega} {L^*}^2_{ref} / \mu^*_{ref} {a^*}^2_{ref} \\
\tau_{ij} &= {\tau_{ij}}^* {L^*}_{ref} / ({\mu^*}_{ref} {a^*}_{ref})
\end{aligned}$$

where, as in the previous sections, quantities denoted with * are dimensional.

3 Boundary Conditions

This chapter discusses the boundary conditions available in Fun3D. Table 1 lists the integers used to specify Fun3D boundary conditions with a short description. Each grid description subsection in section 4 indicates how these integers are specified. Details of the boundary condition implementation are provided by Carlson. [5] Details of symmetry boundary conditions are provided in section 3.1. Some boundary conditions have required or optionally specified parameters defined in the `&boundary_conditions` namelist, see section B.4.21 for further boundary condition details.

Table 1: FUN3D boundary conditions.

BC	Description	Notes
-1	Overlap	overset grid boundary
3000	Tangency	zero normal velocity, specified via fluxes
4000*	Viscous (strong)	explicitly set the no-slip condition
4100**	Viscous wall function	sets a shear stress condition via a wall function
4110*	Viscous (weak)	implicitly set the no-slip condition
5000	Farfield	Riemann invariants
5026	Extrapolate	supersonic outflow, variables extrapolated from the interior
5050	Freestream	external freestream, specified via fluxes
5051*	Back pressure	specified static pressure (switches to extrapolation boundary condition in the presence of supersonic flow)
5052*	Mach outflow	static pressure outflow boundary condition set via a specified subsonic Mach number (not for boundary layer ingestion)
6021	Symmetry plane 1	symmetry enforced by replacing x-momentum with zero velocity normal to arbitrary boundary plane.
6022	Symmetry plane 2	symmetry enforced by replacing y-momentum with zero velocity normal to arbitrary boundary plane.
6023	Symmetry plane 3	symmetry enforced by replacing z-momentum with zero velocity normal to arbitrary boundary plane.
6100	Periodicity	discrete periodicity, limited to nominally 2D grids extruded across n planes in a third dimension
6661	X-symmetry plane	enforces symmetry for x Cartesian plane
6662	Y-symmetry plane	enforces symmetry for y Cartesian plane
6663	Z-symmetry plane	enforces symmetry for z Cartesian plane
7011*	Subsonic inflow	subsonic inflow ($p_{t,bc} = p_{total,plenum}/p_{static,freestream}$, $T_{t,bc} = T_{total,plenum}/T_{static,freestream}$) for nozzle or tunnel plenum ($\mathsf{M}_{inflow} < 1$)
7012*	Subsonic outflow	subsonic outflow ($p_{bc} = p_{static,inlet}/p_{static,freestream}$ for inlet flow (does not allow for reverse or supersonic flow at the outflow boundary face)
7021*	Reaction control jet plenum	models the plenum of a reaction control system (RCS) jet
7031*	Mass flow out	specification of massflow out of the control volume
7036*	Mass flow in	specification of massflow in to the control volume
7100*	Fixed inflow	fixed primitive variables in to control volume
7101*	Fixed inflow profile	specified profile
7103*	Pulsed supersonic inflow	pulsing supersonic flow
7104*	Ramped supersonic inflow	ramping supersonic flow
7105*	Fixed outflow	specified primitive outflow conditions
7201	Frozen solution	maintains specified solution, whether through restart data or flowfield initialization

* See `&boundary_conditions` namelist in section B.4.21 to specify auxiliary information and for further descriptions.

** See `&turbulent_diffusion_models` namelist in section B.4.10 to specify auxiliary information.

3.1 X-Symmetry, Y-Symmetry, and Z-Symmetry

The symmetry condition in FUN3D enforces discrete symmetry. That is, if the mesh were mirrored about the symmetry plane and run as a full-span simulation, the residuals would be identical. Assuming the reference area input was doubled accordingly, the outputs (such as lift, drag, etc.) would also be identical. The FUN3D automated tests include a case where a mirrored grid is compared to the symmetry boundary condition. Discrete symmetry is also enforced where multiple symmetry condition intersect (i.e., a corner of Y-symmetry and Z-symmetry).

Specifically, the condition enforces:

- Any points on a symmetry plane are first "snapped" to the average coordinate for that plane. Many grid generators will have some small amount of "slop" in their y-coordinates for points on a y-symmetry plane; FUN3D immediately pops all of the points onto the exact same plane, at least to double precision.
- The residual equation corresponding to the momentum normal to the symmetry plane is modified to reflect zero crossflow (i.e., $\rho v = 0$ on a y-symmetry plane).
- The least squares system used to compute gradients for inviscid reconstruction to the cell faces is augmented to include symmetry contributions across the symmetry plane(s).
- All gradients of the velocity normal to the symmetry plane (v for a y-symmetry plane) in the source terms for the turbulence models are zeroed out. Gradients of the tangential velocities are zeroed out normal to the symmetry plane (du/dy, dw/dy).
- No convective flux of turbulence normal to the symmetry plane.
- Grid metrics (e.g., areas, normals) are forced to be symmetric at symmetry planes.

4 Grids

This chapter explains how to supply the proper file formats to FUN3D, but does not cover how to create a mesh. See section 1.3 for grid generation guidance. FUN3D supports a direct reader for many grid formats. The format of the grid is specified in the `&raw_grid` namelist, section B.4.2. In addition to the directly read formats, translators are provided to convert additional grid formats into a format that can be read directly, see section 4.3. FUN3D has the ability to apply rigid body rotations while reading the grid (see section B.4.4 and section B.4.5) and mirror a grid about a symmetry plane (see section 4.5).

4.1 File Endianness

The ordering of bytes within a data item is known as "endianness." If the endianness of a file is different than the native endianness of the computer then a conversion must be performed. The endianness of each grid file format is described in section 4.2. If your compiler supports it, FUN3D will attempt to open binary files with a `open(convert=...)` keyword extension. Consult the documentation of the Fortran compiler you are using to determine if other methods are available. For example, with the Intel® Fortran compiler, the endianness of file input and output can be controlled by setting the `F_UFMTENDIAN` environment variable to `big` or `little`.

4.2 Supported Grid Formats

FUN3D natively supports the grid formats summarized in Table 2.

Table 2: File extensions.

Format	Grid files	BC File
AFLR3	.ugrid	.mapbc
FAST	.fgrid	.mapbc
FieldView	.fvgrid_fmt	.mapbc
	.fvgrid_unf	.mapbc
FUN2D	.faces	.mapbc
VGRID	.cogsg, .bc	.mapbc*
FELISA	.gri, .fro	.bco

* Same suffix, but GridTool format.

The standard FUN3D `.mapbc` file format contains the boundary condition information for the grid. The first line is an integer corresponding to the number of boundary groups contained in the grid file. Each subsequent line in this file contains two integers, the boundary face number and the FUN3D boundary condition integer; these numbers may optionally be followed by a character string that specifies a “family” name for the boundary. The family name is required if the `patch_lumping` option (section B.4.2) is invoked to combine patches into fewer patch families. Below is a sample `.mapbc` file illustrative for all grid formats except GridTool/VGRID, FELISA, and FUN2D, which are described later.

```
13
1       6662  box_ymin
2       5025  box_zmax
3       5050  box_xmin
4       5025  box_ymax
5       5025  box_zmin
6       5025  box_xmax
7       3000  wing_upper
8       3000  wing_lower
9       3000  wing_upper
10      3000  wing_upper
11      3000  wing_lower
12      3000  wing_lower
13      3000  wing_tip
```

4.2.1 AFLR3 Grids

AFLR3, SolidMesh, Pointwise, and GridEx can all produce this format and FUN3D ships with translators that convert Plot3D and CGNS grids to AFLR3 format. The format is documented online at `https://www.simcenter.msstate.edu/software/documentation/ug_io/3d_grid_file_type_ugrid.html`

AFLR3 grid file format types are indicated by file suffixes. The formatted (plain text) style has a `.ugrid` suffix while other types vary according to endianness (see section 4.1) and binary type as shown in Table 3. The boundary

Table 3: AFLR3 non-ASCII grid suffixes.

Type	Little endian	Big endian
Fortran Stream, C Binary, 32-bit integers	`.lb8.ugrid`	`.b8.ugrid`
Fortran Stream, C Binary, 64-bit integers	`.lb8l.ugrid`	`.b8l.ugrid`
Fortran Unformatted, 32-bit integers	`.lr8.ugrid`	`.r8.ugrid`
Fortran Unformatted, 64-bit integers	`.lr8l.ugrid`	`.r8l.ugrid`

conditions are specified via the standard FUN3D `.mapbc` format.

4.2.2 FAST Grids

The `.fgrid` file contains the complete grid stored in ASCII FAST format. The format is documented online at `https://www.simcenter.msstate.edu/`

software/documentation/ug_io/3d_grid_file_type_fgrid.html The boundary conditions are specified via the standard FUN3D .mapbc format.

4.2.3 VGRID Grids

The .cogsg file contains the grid nodes and tetrahedra stored in unformatted VGRID format. The VGRID cogsg files always have big endian byte order regardless of the computer used in grid generation. See section 4.1 for instructions on specifying file endianness.

The .bc file contains the boundary information for the grid, as well as a flag for each boundary face. For viscous grids with a symmetry plane, VGRID is known to produce boundary triangles in the .bc file that are incompatible with the volume tetrahedra. FUN3D provides a repair_vgrid_mesh utility to swap the edges of these inconsistent boundary triangles. If FUN3D reports that there are boundary triangles without a matching volume tetrahedra, use this utility.

VGRID has a different .mapbc boundary condition format. For each boundary flag used in the .bc file, the .mapbc file contains the boundary type information. The VGRID boundary conditions are described at the website: https://tetruss.larc.nasa.gov/usm3d/boundary-conditions. The FUN3D boundary condition integers can also be used in place of the VGRID boundary condition integers. Internally, FUN3D converts the VGRID boundary condition integers to the FUN3D boundary condition integers as indicated in Table 4.

Table 4: Boundary type mapping between VGRID and FUN3D.

VGRID	FUN3D
−1	−1
0	5000
1	6662
2	5005
3	5000
4	4000
5	3000
44	4000
55	3000

4.2.4 FieldView Grids

The .fvgrid_fmt file contains the complete grid stored in ASCII FieldView FV-UNS format, and the .fvgrid_unf file contains the complete grid stored in unformatted FieldView FV-UNS format. Supported FV-UNS file versions are 2.4, 2.5, and 3.0. With FV-UNS version 3.0, the support is only for the grid file in split grid and results format; the combined grid/results format is not supported. Fun3D does not support the arbitrary polyhedron elements of the FV-UNS 3.0 standard. For ASCII FV-UNS 3.0, the standard allows comment lines (line starting with !) anywhere in the file. Fun3D only allows comments immediately after line 1. Only one grid section is allowed. The precision of the unformatted grid format should be specified by the `fieldview_coordinate_precision` variable in the `&raw_grid` namelist, see section B.4.2. The boundary conditions are specified via the standard Fun3D `.mapbc` format.

4.2.5 FELISA Grids

The `.gri` file contains the grid stored in formatted FELISA format. [6] The `.fro` file contains the surface mesh nodes and connectivities and associated boundary face tags for each surface triangle. This file can contain additional surface normal or tangent information (as output from GridEx or SURFACE mesh generation tools), but the additional data is not read by Fun3D. The `.bco` file contains a flag for each boundary face. If original FELISA boundary condition flags (1, 2, or 3) are used, they are translated to the corresponding Fun3D 4-digit boundary condition flag according to Table 5. Alternatively, Fun3D 4-digit boundary condition flags can be assigned directly in this file.

Table 5: Boundary type mapping between FELISA and Fun3D.

FELISA	FUN3D
1	3000
2	6662
3	5000

4.2.6 Fun2D Grids

The `.faces` file contains the complete grid stored in formatted Fun2D format (triangles). Internally, Fun3D will extrude the triangles into prisms in the y-direction and the 2D mode of Fun3D is automatically enabled. Output from the flow solver will include this one-cell wide extruded mesh.

Boundary conditions are contained in the Fun2D grid file as integers 0–8. The mappings to Fun3D boundary conditions are given in Table 6. If Fun3D does not detect a `.mapbc`, it will write a `.mapbc` file that contains the default Table 6 mapping. If you wish to change the boundary conditions from the defaults based on the `.faces` file, simply edit them in this `.mapbc` file and rerun Fun3D. The boundary conditions in the `.mapbc` file have precedence over the `.faces` boundary conditions. If you wish to revert to the boundary conditions in the `.faces` file after modifying the `.mapbc`, you can remove the `.mapbc` and rerun Fun3D.

Table 6: Boundary type mapping between Fun2D and Fun3D.

FUN2D	FUN3D
0	3000
1	4000
2	5000
3	−1
4	4010
5	4010
6	5005
7	7011
8	7012

4.3 Translation of Additional Grid Formats

While Fun3D supports the direct read of multiple formats, utilities are provided to translate additional grid formats into a format that Fun3D can read.

4.3.1 PLOT3D Grids

The utility `plot3d_to_aflr3` converts a PLOT3D structured grid to an AFLR3-format hexahedral unstructured grid. The original structured grid must be 3D multiblock http://www.grc.nasa.gov/WWW/wind/valid/plot3d.html (no iblanking) with the file extension `.p3d` for formatted ASCII or the the file extension `.ufmt` for Fortran unformatted. Only one-to-one connectivity is allowed with this option (no patching or overset). The grid should contain no singular (degenerate) lines or points. A neutral map file with extension `.nmf` is also required. This file gives boundary conditions and connectivity information. The `.nmf` file is described at http://web.archive.org/web/20161202050742/https://geolab.larc.nasa.gov/Volume/Doc/nmf.htm.

Note that the `Type` name in the `.nmf` file must correspond with one of Fun3D's BC types, plus it allows the Type `one-to-one`. If the `Type` is not recognized, you will get errors like:

```
This may be an invalid BC index.
```

An example `.nmf` file is shown here for a simple single-zone airfoil C-grid ($5 \times 257 \times 129$) with six exterior boundary conditions and one `one-to-one` patch in the wake where the C-grid attaches to itself:

```
# ===== Neutral Map File generated by the V2k software of NASA Langley's GEOLAB =====
# ===================================================================================
# Block#   IDIM   JDIM   KDIM
# -----------------------------------------------------------------------------------
     1

     1      5    257    129

# ===================================================================================
# Type            B1  F1  S1   E1  S2   E2   B2  F2   S1   E1   S2  E2  Swap
# ---------------------------------------------------------------------------------
'tangency'         1   3   1  257   1  129
'tangency'         1   4   1  257   1  129
'farfield_extr'    1   5   1  129   1    5
'farfield_extr'    1   6   1  129   1    5
'one-to-one'       1   1   1    5   1   41    1   1    1    5  257 217 false
'viscous_solid'    1   1   1    5  41  217
'farfield_riem'    1   2   1    5   1  257
```

4.3.2 CGNS Grids

Fun3D is distributed with a utility `cgns_to_aflr3` that converts CGNS files http://cgns.github.io/ to AFLR3 grids. This utility will only be built if Fun3D is configured with a CGNS library, see section A.13.13. Only the `Unstructured` type of CGNS files are supported. The following CGNS mixed element types are supported: `PENTA_6` (prisms), `HEX_8` (hexes), `TETRA_4` (tets), and `PYRA_5` (pyramids).

The CGNS file must include `Elements_t` nodes for all boundary faces (type `QUAD_4` or `TRI_3`) to refer to the corresponding boundary elements. Otherwise, the utility cannot recognize what boundaries are present because it currently identifies boundaries via these 2D element types. The `cgns_to_aflr3` utility requires that the BC elements be listed either as a range or a sequential list.

It is also helpful to have separate element nodes for each boundary element of a given BC type. This way, it is easier to interpret the boundaries, i.e., body versus symmetry versus farfield. Visualization tools, such as Tecplot™, can easily distinguish the various boundary condition groups as long as each group has its own node in the CGNS tree. Under `BC_t`, `cgns_to_aflr3` reads these BC names, but ignores additional boundary data (e.g., `BCDataSet`, `BCData`).

Table 7: Boundary type mapping between CGNS and Fun3D.

CGNS	FUN3D
BCSymmetryPlane	6661, 6662, or 6663 via prompt
BCFarfield	5000
BCWallViscous	4000
BCWall	4000
BCWallInviscid	3000
BCOutflow	5026
BCTunnelOutflow	5026
BCInflow	5000
BCTunnelInflow	5000

If the CGNS file is missing BCs (no `BC_t` node), `cgns_to_aflr3` still tries to construct the BCs based on the boundary face `Elements_t` information. If these boundary element nodes have a name listed in Table 7, a `.mapbc` file will be written that contains the Fun3D boundary condition numbers. If the name is not recognized, you will see the message:

```
WARNING: BC type ... in CGNS file not recognized.
```

in which case you will need to fix it by by editing the .mapbc file manually. Always check the `.mapbc` file after the utility has run, to make sure that the boundary conditions have all been interpreted and set correctly. If a translation problem is observed, you should edit the `.mapbc` file before running FUN3D.

4.4 Implicit Lines

The standard implicit solution relaxation scheme in FUN3D is a point-implicit, which inverts the linearization of the residual at each node to compute an update to the solution. A line-implicit relaxation scheme can be used that inverts the linearization of a line of nodes simultaneously. Typically, lines are constructed for the subset of nodes in the boundary layer to address the stiffness inherent in the Navier-Stokes equations on anisotropic grids. Therefore, the use of line-implicit relaxation may improve the convergence of viscous flow simulations. These lines are used in conjunction with the standard point-implicit relaxation scheme. Detailed descriptions of both line and point relaxation schemes are provided by Nielsen et al. [7] Currently, every viscous boundary node must have one and only one associated implicit line. Lines of nodes are specified in a formatted file with the suffix `.lines_fmt` that contains the definitions of lines emanating from viscous boundary nodes as a list of node numbers. The format of the `.lines_fmt` file is

```
[total number of lines] [total number of points in lines]
[min points in a line] [max points in a line]
 [points in line 1]
  [first node of line 1]
  ...
  [last node of line 1]
 [points in line 2]
  ...
```

4.5 Grid Mirroring

A half computational domain with a symmetry plane can be mirrored to form a complete domain. This is a common use case for reflecting half of an aircraft configuration to form a full-span configuration to include the effects of side slip. Use the `--mirror_x`, `--mirror_y`, or `--mirror_z` command line options (section 5.2) to reflect the domain and remove the boundary patch located at $x = 0$, $y = 0$, or $z = 0$, respectively. When one or more of these mirror command line options are provided, FUN3D will start execution on a grid that is mirrored internally during grid processing. The symmetry plane boundary face patch will be removed and each asymmetry face patch will duplicated about the symmetry plane. These operations on boundary face patches will alter the numbering of patches. Therefore, the boundary face indexes used to specify boundary conditions and co-visualization will be modified to track these changes. The user is required to update the indexes in these namelists appropriately. The file `[project_rootname]_mirror*.mapbc` will be created in the current directory to aid this process.

The internally mirrored grid will be discarded when execution is complete. So, provide the same mirror command line option to any restarted executions. Alternatively, the `--write_mesh [new_project]` command line option is available to export the mirrored grid. Utilizing this exported grid would remove the need for the mirror command line option on subsequent runs.

5 Flow Solver, NODET

This chapter covers what is required to run an initial flow solution, how to restart a flow solution, and how to specify what outputs the solver NODET produces.

5.1 Flow Solver Execution

The grid and flow conditions are specified in the file `fun3d.nml`; see section B.4 for the file description. If you configured Fun3D without MPI, the executable is named `nodet`. If you configured Fun3D with MPI, the executable is named `nodet_mpi`. Configuration and installation is explained in detail in section A. The executable `nodet` can be invoked directly from the command line,

```
nodet [fun3d options]
```

but the MPI version `nodet_mpi` will need to be invoked within an MPI environment. The most common method is via

```
[MPI run command] [MPI options] nodet_mpi [fun3d options]
```

The details of the MPI run command and MPI options will depend on the MPI implementation. The MPI run command is commonly `mpirun` or `mpiexec`. The MPI options may contain the number of processors `-np [n]`, a machine file `-machinefile [file]`, or no local `-nolocal`. If a queuing system is used (e.g., PBS) this command will need to be run inside an interactive job or a script. See your MPI documentation or system administrator to learn the details of your particular environment.

If you have provided a grid with boundary conditions and `fun3d.nml`, you will then see the solver start to execute. If an unexpected termination happens during execution (e.g., `SIGSEGV`), especially during grid processing or the first iteration, you may need to set your shell limits to unlimited,

```
$ ulimit unlimited # for bash
$ unlimit          # for c shell
```

A detailed description of the output files is given below.

5.2 Command Line Options

These options are specified after the executable. The majority of the command line options are functionality under development and there is work underway to migrate command line options to namelists. Namelists are the preferred input method. Command line options should be avoided unless they are the only way to activate the functionality you require. These commands are always preceded by `--` (double minus). More than one option may appear on the command line (each option proceeded by a `--`). You can see a listing of the available command line options in any of the codes in the FUN3D suite by using the command line option `--help` after the executable name,

```
./nodet_mpi --help
```

The options are then listed in alphabetical order, along with a short description and a list of any auxiliary parameters that might be needed, and then the code execution stops. Specific examples of the use of command line options are found throughout this, and later, chapters.

5.3 Output Files

These are the output files produced by the flow solver, NODET.

`[project_rootname].flow` This file contains the binary restart information and is read by the solver for restart computations. See the `restart_read` namelist variable in section B.4.14 to control restart behavior.

`[project_rootname]_hist.dat` This file contains the convergence history for the RMS residual, lift, drag, moments, and CPU time, as well as the individual pressure and viscous components of each force and moment. The file is in Tecplot™ format. See section B.4.27 for an improved method to track forces and moments.

`[project_rootname]_subhist.dat` For time accurate computations only. This file contains the subiteration convergence history for the RMS residuals, together with force and moment histories. Force and moment histories reflect the combined viscous and pressure components. The file is in Tecplot™ format and is overwritten each time the code is run.

`[project_rootname].forces` This file contains a breakdown of all the forces and moments acting on each individual boundary group. The totals for the entire configuration are listed at the bottom. See section B.4.27 for an improved method to track forces and moments.

5.3.1 Flow Visualization

There are six basic categories of output: boundary data, sampling data (on entities such as planes, boxes and spheres), volumetric data, slice data, and spatially averaged data controlled by the namelists in Table 8.

Each namelist has a corresponding frequency variable, A positive frequency will cause the output to be generated every frequency time step/iteration. A

Table 8: Solver output types.

Type	Namelist	
domain boundaries	`&boundary_output_variables`	section B.4.31
domain volume	`&volume_output_variables`	section B.4.30
boundary slices	`&slice_data`	section B.4.35
various geometries	`&sampling_parameters` and `&sampling_output_variables`	section B.4.33 and section B.4.32
point	`&sampling_parameters` and `&sampling_output_variables`	section B.4.33 and section B.4.32
spatially averaged profile	`&spatial_avg`	section B.4.34

negative frequency will cause output to be written only at the end of a run. A zero frequency (the default) with produce no output. See the corresponding namelist descriptions for details.

5.3.2 Flow Visualization Output From Existing Solution

If a Fun3D flow solution already exists, visualization files can be produced by setting `steps = 0` or `steps = 1` in the `&code_run_control` namelist (section B.4.14) and setting the `restart_read` variable to something other than `'off'`. One iteration is required to compute gradient quantities (i.e., skin friction). This will allow generation of visualization output without having to repeat the entire calculation.

6 Adjoint Solver, DUAL

This section describes how to execute the adjoint solver, DUAL, directly. Typically, DUAL is executed by scripts that manage the multiple steps required for design optimization (section 9) or grid adaptation (section 8). However, it may be necessary to run DUAL directly to diagnose problems or gain experience during setup including determining input parameters and termination strategies. FUN3D is configured to compile DUAL by default. While the adjoint method is available for most commonly used FUN3D capabilities, only a subset of FUN3D's full capabilities are implemented in the adjoint solver.

6.1 Convergence of the Linear Adjoint Equations

The adjoint solution is dependent on the primal flow solution (and the convergence of the primal flow equations). While the primal solution may have converged enough to give acceptable force and moment results, the flow residuals might still be large, which can cause the adjoint solution scheme to diverge. This divergence issue is most common in turbulent simulations. A divergent adjoint scheme can be improved in some circumstances with the `--outer_loop_krylov` command line option. It is critical to run the flow solver and the adjoint solver with the *same* governing equations and boundary conditions.

The scaling of the adjoint residuals is different from the flow residuals and is dependent on the choice of the adjoint cost functions. The number of iterations `steps` and the residual tolerance `stopping_tolerance` will need to be adjusted, see section B.4.14. The sensitivities should converge at the same rate as your functions (i.e., lift), but an adjoint with some algebraic error may still provide reasonable sensitivities for design and grid adaptation.

6.2 Required Directory Hierarchy and Executing DUAL

The executable `dual` can be invoked directly from the command line,

```
dual [fun3d options]
```

but the MPI version `dual_mpi` will need to be invoked within an MPI environment. The most common method is via

```
[MPI run command] [MPI options] dual_mpi [fun3d options]
```

Any `[fun3d options]` provided to NODET that control the flow solver residual will also be required for the adjoint solver for a consistent adjoint solution and solution scheme. See the flow solver execution instructions for more details, section 5.1.

DUAL expects the cost function description `../rubber.data` to be in the parent directory of the directory from which it is invoked. The input and flow restart files are shared with NODET in the directory `../Flow/`. The flow solver must be run to completion, to provide a flow restart file, before DUAL is invoked. See Table 9 for the required files and locations.

Table 9: Adjoint solver DUAL directory hierarchy.

Relative Path	**Description**
`../Flow/[project_rootname].flow`	Primal flow solution (restart)
`../Flow/fun3d.nml`	Main input namelist file
`../rubber.data`	Description of the adjoint cost function

6.3 `rubber.data`

The minimum required `rubber.data` for running the adjoint (and grid adaptation) can be written with the command

```
f3d function [cost function name]
```

Available cost function names are discussed in section 9.1 and listed in Table 12. See section 9.6.2 for complete details on this file format including the information required for design. The `rubber.data` reader requires the *exact* number of header lines. Be very careful when editing this file.

6.4 Output Files

The adjoint solver will export visualization files in the same manner as the flow solver when requested, see section 5.3.1.

`[project_rootname].adjoint` This file contains the binary restart information and is read by the and adjoint solver for restart computations.

`[project_rootname]_hist.tec` This file contains the convergence history for the RMS residual of the adjoint equations and CPU time. The file is in the same Tecplot™ format as the flow solver produces. History information is truncated when the adjoint solver is restarted.

7 Grid Motion

FUN3D has an extensive capability for grid motion. Grid motion may be rigid, where points in the grid move in unison, or grid motion may result from elastic deformation of the grid to follow the motion of one or more boundaries in the domain. To accommodate both types of grid motion, a distinction is made between 'body motion' and 'grid motion.' The user begins by defining one or more bodies (a *body* being a collection of one or more boundary patches in the grid), and specifies how that body is to move. This section describes how to specify the motion of each body as rigid motion, elastic motion, or a combination of rigid and elastic motions. The motion and deformation of volume grid surrounding the bodies is controlled through the concept of mesh movement (see section 7.2).

Motion requires the introduction of reference frames. FUN3D solves the flow equations in an inertial reference frame (one exception to this, section B.4.8, can not be used with grid motion). Each moving body has its own reference frame; the body frame is taken to coincide with the inertial reference frame at $t = 0$. In addition to the inertial and body reference frames that are automatically bookkept when grid motion is activated, the user may define an 'observer frame' for visualization output. Typically, the 'observer frame' is either the inertial frame (default) or one of the moving body frames. There is the provision for the observer frame to move independently, if required.

FUN3D allows for hierarchical (parent/child) body motions in which the motion of one body follows the motion of another body. The top level of these hierarchical motions is the inertial frame, and the default parent frame of any moving body is taken as the inertial frame (i.e., by default the motion of a body is assumed to be relative to the inertial frame).

For all but a few specialized capabilities discussed below (see section 7.5), grid motion requires that FUN3D be executed in 'time-accurate' mode (see section B.4.15). To activate grid motion, the input variable `moving_grid = .true.` in the `&global` namelist is required. Finally, an additional input file, `moving_body.input`, is needed to establish the details of the motion (see section B.5).

Any grid motion that the user specifies *leaves the original grid file unaltered.* Grid motion always starts from the input grid set in the `&raw_grid` namelist (see section B.4.2). The FUN3D restart file contains enough information so that motion can be continued seamlessly through restarts. If the user would like to obtain a new grid file with the grid in its position at the end of the execution, use the command-line option `--write_mesh [new_project]`, where `[new_project]` is the project name of the new grid file. Note that regardless of the grid format of the original mesh, the new mesh format will be in AFLR stream format.

The distance function needed for any turbulence model is computed once by default, based on the *input* grid. If the entire grid is moved rigidly, the distance function remains unchanged with motion. However, for deforming grids and (to a lesser extent) rigid overset grids, the distance from a point to the nearest surface may change to some extent as the grid is moved from the initial state. For deforming meshes, the grid points nearest to the surface behave as a nearly rigid grid, so the effect on the distance function is minimal. Currently, it is up to the user to decide if the extra cost of recomputing the distance function is justified. As defined in section B.4.29, setting `recompute_turb_dist = .true.` will force the distance function to be recomputed every `turb_dist_update_freq` time steps when grid motion is accomplished by deformation, or if overset grids are used. The distance function is never recomputed for rigid meshes that do not require overset connectivity. These same distance function considerations also apply to the specialized capabilities discussed in section 7.5.

7.1 Overview of `moving_body.input`

Moving-grid cases require the `moving_body.input` input file. Complete details are provided in section B.5 and an overview is provided here. Within this file, basic information required for all moving-grid cases is accomplished through the `&body_definitions` namelist (see section B.5.1). Depending on the type of 'motion driver' selected (i.e., how the body is to be moved), additional namelist input is usually required. The `&forced_motion` namelist (See section B.5.2) allows the user to specify constant translation and/or rotational motion of a body. The `&motion_from_file` namelist (see section B.5.4) specifies body motion by providing a time sequence of 4×4 transform matrices, and is the most general means of specifying rigid-body motion. The `&observer_motion` namelist (see section B.5.3) allows the user to specify motion of an 'observer' reference frame. Visualization (see section 5.3.1) is in the 'observer' frame.

7.2 Choosing the Type of Mesh Movement

The user must choose a means to move the mesh that will accommodate the prescribed or calculated body motion. As described in section B.5.1, the choices are `mesh_movement = 'rigid'`, `mesh_movement = 'deform'`, or a combination of the two (e.g., `mesh_movement = 'rigid+deform'`. Whenever possible, the most robust and fastest executing option is 'rigid', since rigid motion is quickly and efficiently applied via a 4×4 transform matrix (see section 7.3) and is guaranteed to maintain positive cell volumes as the mesh is moved. Mesh deformation, on the other hand, increases the computational time as it requires the solution of an additional system of partial differential equations (linear elasticity). Furthermore, there is no guarantee of positive cell volumes after deformation. FUN3D will automatically try to fix negative volumes during deformation, but this is rarely successful. Execution terminates when negative cell volumes are encountered and cannot be remedied.

Elastic body deformations preclude purely rigid grid motion (i.e., the body itself undergoing deformation, as occurs for aeroelastic bodies). However, some cases can benefit from the combination of rigid and deforming mesh motion. This can allows large-scale motion to be described as rigid motion and reduces the magnitude of the elastic deformation to smaller-scale motions. Reducing the magnitude of body elastic motions can reduce the cost of computing the elastic deformation of the adjacent volume grid. Smaller elastic deformations also increase the robustness by reducing the likelihood of generating negative volumes. An example of combined rigid and elastic deformation is a rotor blade of a rotorcraft simulation, where rigid motion is used to rotate the blades around the shaft axis and elastic deformation is used to deform the the blade under aerodynamic loading.

7.3 4×4 Transform Matrices

All rigid motions (body and/or grid) within FUN3D are accomplished via application of 4×4 matrices to describe affine transformations. [8] These transforms map a body/grid from the position at $t = 0$ to the current position at $t = T$. The inverse of these transforms are used to map a body/grid from its current position back to its position at $t = 0$. Although the user usually does not need to know the details of these transform matrices to use the grid motion capability in FUN3D, they are described below for reference.

The 4×4 transform matrices contain both translation and orthonormal rotation components. Given a point at an initial position $(x, y, z)^T$, application of the transform matrix moves the point to its new position $(x', y', z')^T$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

where the 3×3 submatrix entries in the upper left define an orthonormal rotation about the origin, $(0, 0, 0)^T$, and the last column defines a translation. The last row is always set as $(0, 0, 0, 1)^T$. Application of the inverse of this transform matrix moves the point back to the initial position.

Two often-encountered transforms are a pure translation from the origin to a point $(x_0, y_0, z_0)^T$ and a pure rotation θ in the direction $\hat{\mathbf{n}}$ (unit vector) about the origin

$$[\mathbf{T_0}] = \begin{bmatrix} 1 & 0 & 0 & x_0 \\ 0 & 1 & 0 & y_0 \\ 0 & 0 & 1 & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$[\mathbf{R_0}] = \begin{bmatrix} (1-c\theta){n_x}^2 + c\theta & (1-c\theta)n_x n_y - n_z s\theta & (1-c\theta)n_x n_z + n_y s\theta & 0 \\ (1-c\theta)n_x n_y + n_z s\theta & (1-c\theta){n_y}^2 + c\theta & (1-c\theta)n_y n_z - n_x s\theta & 0 \\ (1-c\theta)n_x n_z - n_y s\theta & (1-c\theta)n_y n_z + n_x s\theta & (1-c\theta){n_z}^2 + c\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $s\theta = \sin(\theta)$ and $c\theta = \cos(\theta)$. The right-hand rule in the direction of $\hat{\mathbf{n}}$ defines the sense of θ. Other forms of the 3×3 rotation submatrix may arise depending on how the motion is specified (e.g., chained rotations such as Euler angles). Note that $[\mathbf{T_0}]^{-1}$ is simply $[\mathbf{T_0}]$ with $(x_0, y_0, z_0)^T$ replaced by $(-x_0, -y_0, -z_0)^T$, and maps $(x_0, y_0, z_0)^T$ to the origin.

An extremely useful feature of the transform matrix approach is that multiple transformations telescope via matrix multiplication. Thus, rotation about a point $(x_0, y_0, z_0)^T$, by an angle θ, in the direction $\hat{\mathbf{n}}$ is effected by first translating to the origin, performing the rotation, and then translating back to $(x_0, y_0, z_0)^T$, which is accomplished via three matrix multiplications to give the complete transform matrix $[\mathbf{T}]$

$$[\mathbf{T}] = [\mathbf{T_0}][\mathbf{R_0}][\mathbf{T_0}]^{-1}$$

The telescoping property of the transform matrices is also useful for tracking motions of one body relative to another. For example, a wing may be translating up and down, while at the same time, a flap may be pitching about a hinge line fixed on the wing. It is natural to describe the pitching motion of the flap in its own coordinate system, which at t=0 is the same as the wing coordinate system. If the transform matrix describing the plunging of the wing relative to its initial position in an inertial reference frame is given by $[\mathbf{T}]_{\mathbf{wing}}$, and the transform matrix of the pitching motion of the flap relative to a hinge line defined in the flap coordinate system is given by $[\mathbf{T}]_{\mathbf{flap}}$, then the position of the flap, relative to the inertial frame, may by computed using the composite transform

$$[\mathbf{T}] = [\mathbf{T}]_{\mathbf{wing}}[\mathbf{T}]_{\mathbf{flap}}$$

The order of matrix multiplication is important: post multiplication of the parent transform takes coordinates from the child system into the parent system, which is then moved relative to the inertial frame according to the parent transform. The example above is for a simple, one-generation, parent-child composite motion, but the concept may be extended to any number of generations.

7.4 Verifying Grid Motion Inputs

When setting up a moving-grid case, it is usually a good practice to verify that the desired grid motion will be performed before an attempt is made with a large and potentially expensive time-dependent flow solution. There are two options that may be used to check that the motion setup is correct; both must be used in conjunction with boundary output to be useful (see section 5.3.1). Both options bypass the flow-solution but not the grid-motion mechanics. Because these options decouple the flow solution and the grid motion, they are generally restricted to cases in which the motion is prescribed by the user. If the motion results from aerodynamic loading, such six degrees of freedom motion or as a result of aeroelastic deformation, there is no complete check but to run the coupled simulation. However, it may be possible to prescribe a motion that is similar to the expected motion so that body definitions and time step sizes may be verified prior to the coupled solution.

The first option is enabled by setting `body_motion_only = .true.` in the `&global` namelist. As the name of the option implies, only the user-defined bodies are moved, not the entire mesh. This is the quickest check, especially for deforming meshes, since the elasticity equations are not solved. Volume and surface elements adjacent to the moving bodies will likely become inverted during this test. So, expect to see inverted elements in visualization of output grids on symmetry boundaries, farfield boundaries, sampling planes, etc.

The second option is more discriminating for deforming grids, and is enabled by setting `grid_motion_only = .true.` in the `&global` namelist. This option runs through the entire grid-motion mechanics, including solution of the linear elasticity equations for deforming grids. This will significantly increase the computational time for this check compared to `body_motion_only = .true.`. For deforming meshes, this option will indicate whether or not the mesh will survive the prescribed motion without negative volumes. All visualization options can be used with this option to observe the surface and volume grid deformation.

Because `body_motion_only = .true.` option is fairly quick even for deforming meshes, the recommended approach for deforming meshes is to start with `body_motion_only = .true.` to verify that the body moves as desired. After the basic motion is verified, the `grid_motion_only = .true.` can be exercised, to ensure the deformation will be successful as the body executes its motion. Both body and grid motion are fast for rigid meshes.

For periodic motions it is strongly recommended to verify that the bodies return to their initial position after the expected number of time steps. Failure to return to the initial position at the expected time likely indicates an error in the input time step or the nondimensional frequency. Insufficient numerical precision of the input values of these quantities can prevent a return to the initial position after one period.

7.5 Static Grid Manipulations

Several options available to move or deform the grid during preprocessing are not considered 'moving grid' options. Therefore, these static grid manipulations do not require the code to be run in a time-accurate manner, the `moving_body.input` file, or `moving_grid = .true.` in the `&global` namelist. These static manipulations result in a one time movement of the mesh at the start of the code execution. As with moving grids, the original grid file specified in the `&raw_grid` namelist (see section B.4.2) remains unaltered. The `body_motion_only` or `grid_motion_only` options do not apply to these static grid manipulations. As for time-accurate moving grids, the rigid-grid option (`&grid_transform`) is preferred over the deforming grid option (`&body_transform`) whenever possible because it is less expensive and more robust.

7.5.1 Grid Transform

This static grid manipulation is governed by the `&grid_transform` namelist in the `fun3d.nml` file (see section B.4.4). It allows the *entire* grid to be rotated, translated, or scaled with a few simple inputs. A more general manipulation of the grid may be obtained through the input of a 4×4 transform matrix.

7.5.2 Body Transform

This static grid manipulation is governed by the `&body_transform` namelist in the `fun3d.nml` file (see section B.4.5). It allows user-defined bodies to be rotated or translated (but not scaled). An example would be to make a small change to the angle of a flap, while leaving the orientation of the wing unchanged. The namelist allows for one or more bodies to be defined as collections of one or more boundaries in the grid, where each body may be manipulated independently of the others. As with the `&grid_transform` option, a 4×4 transform matrix may be input if the simple rotation and translation options provided by the namelist input are insufficient to describe the transform. When this option is used, the specified bodies will be moved according to the input, and the mesh will be deformed to accommodate the new body positions.

7.5.3 Surface From File

This static grid manipulation is governed though the command line option `--read_surface_from_file`. When this command line option is used, a file with the new surface will be read and the mesh deformed to fit this new surface. This surface file must follow the naming convention of `[project]_bodyN.dat`, where `N` is the body number. By default `N=1` is assumed. The file must be a formatted 'MASSOUD' file, which is a formatted Tecplot™ file of type 'FEPOINT', with variables x, y, z, and id, where id is the id of the surface point within the global mesh numbering system. Connectivity information is not required in the `[project]_bodyN.dat` file (any connectivity data in the file is not read by Fun3D); however, including the connectivity data will allow visualization of the surface with Tecplot™ to help ensure the surface is correctly defined and positioned. Should more than one body be necessary, use the `&massoud_output` namelist to specify the number of bodies, and the boundaries in the mesh that define those bodies.

7.6 Aeroelastic Modal Analysis

The modal solver is activated with the `--aeroelastic_internal` command line option and is governed by the `&aeroelastic_modal_data` namelist. In the `moving_body.input` file, `motion_driver(body) = 'aeroelastic'` in the `&body_definitions` namelist declares a body should be driven by the modal solver. `mesh_movement(body) = 'deform'` must also be set because strictly rigid motion is not compatible with deforming surfaces.

The `&aeroelastic_modal_data` in `moving_body.input` defines modal properties. Multiple aeroelastic bodies are permitted in the same simulation with `nmodes` determining how many modes are in each body. The modal mass, `gmass`, and frequency, `freq`, must be defined for each mode. Fun3D's modal solver assumes that the modes are orthogonal; therefore, the modal mass, damping ratios, and frequency in `&aeroelastic_modal_data` correspond to the diagonal entries in a general representation of the matrices.

The structural solver in Fun3D operates in dimensional space. The `uinf` input must be set because this parameter is how Fun3D relates the flow solver time step, which is nondimensional, to the structural time step. The dynamic pressure, `qinf`, dimensionalizes the CFD-based loads on the aeroelastic surfaces. CFD inputs are typically created to be consistent dimensionally with the structural model. For example, if the structural model from which the modal structural model was generated is in standard SI units, a dimensionally consistent CFD mesh would have one grid unit correspond to one meter in physical space. If the CFD grid units do not directly correspond to the units in the structural model, `grefl` is used to specify the scaling factor, and `uinf` and `qinf` must be consistent with the structural model units, not the CFD model.

The `&aeroelastic_modal_data` also controls the temporal integration of the modal structural equations. Options include perturbations to modes for simulating dynamic responses, prescribed motion of modes, controlling the number of fluid-structure interaction iterations per time step, and whether to include skin friction in the computation of forces applied to the structure. See section B.5.7 for more details.

The modal solver can be used for static aeroelastic analysis, where the structural displacements have zero variation at the converged state, or dynamic aeroelastic analysis, where the unsteady behavior of the fluid and structure is of interest. For both types of analyses, the aeroelastic modal solver must be run in a time-accurate mode. Static aeroelastic analysis is performed with the unsteady flow solver by setting the damping ratio for each mode, `damp`, artificially high to a value such as `0.9999`. The aeroelastic simulation is marched in time until the modal displacements converge. Setting `ignore_grid_velocity = .true.` in the `&global` namelist can allow the flow solver to remain stable at larger time steps for faster static aeroelastic convergence.

For restarting aeroelastic simulations, the modal solver writes the structural restart file, `[project_rootname]_modal_structure.restart`, at the same frequency as the flow solver's `[project_rootname].flow` file. The modal solver will read the prior states from this file when `restart_read = 'on'` is set. When performing restarts, perturbations such as `gforce0` in `&aeroelastic_modal_data` will be applied at the first step of the restarted simulation. Therefore, when restarting a dynamic response simulation where a perturbation was applied in the initial simulation, the perturbation inputs should be turned off for the restarted simulation in order to not perturb the system again.

7.6.1 Mode Shape Input Files

The mode shapes input files that FUN3D reads contain the modal shapes mapped onto the aerodynamic surface mesh of the body. The first step to generate these mode shapes is to write the surface definitions from a FUN3D analysis in the form of the `[project_rootname]_massoud_bodyN.dat` file generated from the `&massoud_output` namelist and the `--write_massoud_file` command line option. Next, some software outside of FUN3D is used to interpolate the structural modes onto the `&massoud_output` surfaces. The Radial Basis Function software[1] is an open-source option to perform this interpolation of the mode shapes.

The projected modes shapes are provided to FUN3D in Tecplot™ FEPOINT format files following the naming convention `[project_rootname]_bodyN_modeM.dat`, where `N` is the body number and `M` is the mode number. The FEPOINT format for the mode shape is the same as the `[project_rootname]_massoud_bodyN.dat` (columns of surface x, y, z coordinates and a global id number), with three additional columns of nodal data for the x, y, and z components of the mode shape. Setting `plot_modes` in the `&aeroelastic_modal_data` namelist can be turned on to verify the mode shapes have been read by FUN3D properly.

[1] www.github.com/nasa/rbf

7.6.2 Mode History Files

The history of each mode in an aeroelastic simulation is written to `aehist_bodyN_modeM.dat`. The files contains columns of time and the mode's state: modal displacement, velocity, force, and acceleration. The time in the `aehist_bodyN_modeM.dat` files is in the dimensions consistent with the structural model, not the flow solver time.

7.6.3 Modal Mesh Deformation

Modal mesh deformation takes advantage of the linear nature of both the modal representation of the structure and the elasticity equations in the mesh deformation solver to significantly speed up the deformation process. The tradeoff is the additional memory required to store three scalars per mode per volume node. Modal mesh deformation is required to run aeroelastic cases with the GPU capability.

The modal mesh deformation is activated with `use_modal_deform = .true.` in the `&aeroelastic_modal_data` namelist. At the start of a simulation, a linear elasticity deformation is performed for each mode in the model with the mode amplitude of `modal_ref_amp`. The `modal_ref_amp` values can be selected by estimating the modal displacements expected during the simulation. Alternatively, they can be chosen simply as small nonzero values because regardless of the selected values, the resulting displacements are normalized and stored as unit displacement volume mode shapes. Utilizing the principle of superposition, the volume mesh deformation at each time step is computed as the sum the stored volume mode shapes scaled by the current generalized displacements from the modal structural solver.

Many aeroelastic analyses require simulations of the same model at many flight conditions. To avoid repeating the same volume mode deformations at the start of each simulation, `write_volume_modes = .true.` saves the partitioned volume mode shapes. For subsequent simulations with the same model and same number of MPI ranks, `read_volume_modes = .true.`, skips the initial deformations and instead reads the volume mode shapes from the files generated by the initial simulation.

8 Grid Adaptation

Prior to attempting the use of grid adaptation, it is highly recommended that the user familiarize themselves with the execution of the flow solver on their problem of interest using a fixed, a priori grid. This will provide an opportunity for the user to understand the basic problem physics, execution environment, etc. prior to involving the additional complications associated with an iterative, adaptive process.

Grid adaptation is implemented in the REFINE grid adaptation tool. REFINE is executed in a separate process with files exchanged between FUN3D and REFINE. Geometry can be evaluated to resolve near-wall and surface grid. In a deprecated workflow, an older version of REFINE is called in-core by FUN3D, however, near-wall and surface grid remain fixed. High-level overviews of both workflows are provided, followed by detailed descriptions of each workflow in individual subsections. An outline of the solution-based grid adaptation process is shown in Fig. 5, where the adjoint solve is only used for adjoint-based adaptation.

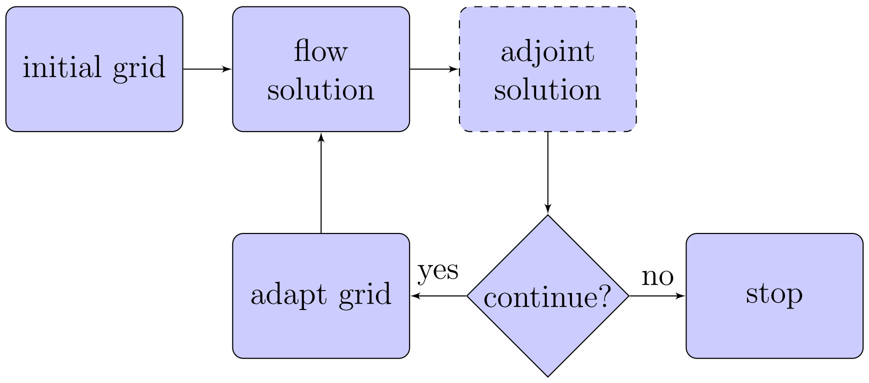


Figure 5: Solution-based grid adaptation process.

The recommended workflow externally couples the FUN3D and REFINE executables via files, and allows the entire grid of tetrahedra and triangles to be modified. Grid adaptation may include modification of the viscous boundary layer and evaluating the original geometry source [9]. In order to modify the surface grid, grid-to-geometry association is required. This association is formed in a “bootstrap” operation, which imports the geometry, constructs initial surface and volume grids, and exports a resulting grid file that also contains geometry and grid-to-geometry association. This grid-to-geometry association requirement excludes starting the process from an arbitrary initial grid if near-wall or surface adaptation is desired. A high degree of automation is possible because the initial grid generation is integrated in the bootstrap process without manual interaction.

Grid adaptation can function without geometry by evaluating a linear representation of the current surface grid or without modification of the near-wall or surface grid. This geometry-free approach may be most useful for an expert-crafted grid with a frozen mixed-element boundary layer where only the off-body tetrahedra are modified.

A deprecated implementation of REFINE grid adaptation is linked into FUN3D. This implementation requires freezing the near-wall grid and limits grid adaptation to the interior of the domain and planar boundary faces without viscous spacing [10]. The deprecated workflow requires an expert-crafted initial grid with suitable surface and near-wall resolution because these regions of the grid are not modified. This workflow is deprecated and not recommended. No future development is planned. The workflow is provided to support existing applications where there is value in capturing important off-body features with fixed near-wall grids.

8.1 Grid Adaptation General Solver Guidance

The grid adaptation approaches described in this chapter control an error estimate based on the current solution. A solution that is a good approximation of a fine-grid solution will accelerate the outer-loop convergence of the grid-adapted solution to the continuous partial differential equation. Inconsistencies between adaptation cycles within the solver will be amplified because these inconsistencies will be interpreted as contributions to the error estimate. Therefore, the user is discouraged from making changes to solver parameters or boundary conditions between adaptation cycles.

While not always possible, attaining low residual error on a given grid accelerates the outer-loop convergence of solution and grid. The baseline nonlinear solver of FUN3D may require lower `schedule_cfl` and `schedule_cflturb` (section B.4.15) on the adapted grids than typical with expert-crafted grids. The adapted grids span large scales from coarse initial grids to resolution of boundary layers, shocks, and singularities in fine grids. The solver namelist parameters may be scheduled with metric complexity, see section 8.2.1 for details, for fast and approximate solves on coarse grids and slow and deep solves on fine grids. The HANIM (section B.4.16) nonlinear solver may improve iterative convergence.

Discretization choices with low dissipation also accelerate outer-loop convergence. For example, the `flux_construction='ldroe'` with eigenvalue smoothing (section B.4.9) results in lower dissipation than the default, `roe`. One might consider using the stabilized finite-element option in FUN3D due to its strengths of exact linearization, strong linear solver, strong nonlinear solver, and low dissipation.

8.2 Scripting Grid Adaptation

A user is welcome to script the execution of grid adaptation in the language of their choice (e.g., bash, Python). A Ruby implementation of this high-level workflow is provided as an example. This Ruby example is named `f3d` and allows for a simple keyword value pair input file that is possible to extend and modify during execution. The FUN3D installation includes the `f3d` Ruby script. The `f3d` script expects to find other components of the FUN3D suite in a user's executable path, i.e., the FUN3D installation `bin` directory. The input file `case_specifics` is described in section 8.2.1. The `case_specifics` file is evaluated continuously by `f3d`, which allows parameters to be changed on-the-fly during adaptation.

Invoking `f3d` without a command will list available commands and the default of variables in the `case_specifics` input file.

```
usage: f3d <command>

  <command>       description
  ---------       -----------
  start           Start adaptation (in background)
  iterate         Start adaptation (blocking)
  view            Echo a single snapshot of stdout
  watch           Watch the result of view
  shutdown        Kill all running fun3d and ruby processes
  clean           Remove output and subdirectories
  function [name] write rubber.data with cost function [name]
```

Execute the `f3d` script in a directory that contains all input files (e.g., grid, `fun3d.nml`, `case_specifics`). The script will create the required `Flow` and `Adjoint` directories to run the case. The command `start` begins adaptation by launching a background job. The command `iterate` begins adaptation and does not exit until adaptation is complete (or an error is thrown). The blocking of `iterate` is helpful in batch queueing systems and the immediate return of `start` is helpful for interactive sessions. The commands `view` and `watch` allow the adaptation progress to be monitored. (Use `Ctrl-C` to escape the `watch` command.) The `shutdown` command kills all Ruby (`f3d`) and FUN3D jobs on the current node. To aid manual termination, a file `f3d.breadcrumbs` is created by `start` or `iterate` that lists the process identifier of `f3d` and the hostname where `f3d` was invoked. The `clean` command removes the `Flow` and `Adjoint` subdirectories, `f3d.breadcrumbs`, `f3d_criteria.dat` state file, and `output` log file. The `function` command creates the `rubber.data` file to define the adjoint cost function.

8.2.1 Input File `case_specifics` for `f3d` Script

The `f3d` script evaluates the input file `case_specifics` as Ruby code. There are recommendations for tailoring this input file in the following subsections, which are specific to external and built-in grid adaptation approaches. Here is an example

```
root_project ''
mpirun_command 'mpiexec'
first_iteration 1
# Any text after a '#' is a comment.
```

where the defaults are listed. Adaptation will be performed from the first grid adaptation iteration (1). The string in quotes next to `root_project` is the project root name.

A two digit iteration number will be appended to the `root_project` for each iteration. The project name for the first adaptation will be `[root_project]01`. All the files required to run the flow solver (e.g., `fun3d.nml`) and refine should be provided in the current directory and the grid filename should include the root project name and iteration number, `[root_project]01`. `Flow` and `Adjoint` subdirectories are created by the script during execution, and the input files are placed in their correct location by the script. A backup copy of `fun3d.nml` is created with the current project (root project name and iteration number) in the `Flow` directory.

Command Line Options Command line options can be passed to the codes via,

```
  all_cl ""
  flo_cl ""
  adj_cl ""
  rad_cl ""
  ref_cl ""
```

where `all_cl` is provided to NODET and DUAL, `flo_cl` is provided to NODET, `adj_cl` is provided to DUAL during the adjoint solve, `rad_cl` is provided to DUAL during error estimation and built-in adaptation, and `ref_cl` is provided to REFINE during external adaptation. For example, the line

```
adj_cl '--outer_loop_krylov'
```

turns on Krylov projection wrapping to stabilize the adjoint solve. Invoke NODET, DUAL, or REFINE with a `-h` for a list of options.

`fun3d.nml` Namelist Modifications The main input file `fun3d.nml` provided in the current directory can be modified by the following commands

```
    all_nl['namelist variable'] = value
    flo_nl['namelist variable'] = value
    adj_nl['namelist variable'] = value
    rad_nl['namelist variable'] = value
```

where `all_nl` changes `fun3d.nml` for NODET and DUAL, `flo_nl` for NODET, `adj_nl` for DUAL during the adjoint solve, and `rad_nl` for DUAL during error estimation and built-in adaptation. The namelist variable must exist in the template `fun3d.nml` where `f3d` is invoked to be substituted. An example is

```
  adj_nl['steps']=500
  adj_nl['stopping_tolerance']=1.0e-12
```

where the termination criteria of the adjoint solver can be specified separately from the flow solver.

`case_specifics` Evaluated as Ruby Script The `case_specifics` is actually executable Ruby code. This allows values to be computed or conditionally executed, but also requires nested quotes for character strings when setting namelist values, e.g.,

```
  rad_nl['adapt_complexity'] = 5000*iteration
  flo_cl "--ascii_tecplot_output" if (iteration<5)
  all_nl['flux_construction'] = "'vanleer'"
```

In Ruby, the contents of single quotes '...' is used verbatim and the contents of double quotes "..." is scanned for expression substitution of the form #{...} where the result of this expression is substituted into the string. If a string does not contain evaluation, either style of quotes can be used. There is an example of string evaluation to change the default `f3d.breadcrumbs` filename.

```
  breadcrumb_filename "#{root_project}-process.txt"
```

`iteration_steps` The steps required to perform a grid adaptation iterations are defined in a function, where the default is set up for external grid adaptation.

```
   def iteration_steps # defines the function to for each iteration
     refdist           # computes exact wall distance, "ref distance"
     flo               # executes the flow solver
     ref_loop_or_exit  # adapt with the external refine executable
   end
```

When performing built-in feature-based adaptation use

```
def iteration_steps
  flo   # executes the flow solver
  adapt # adapts the grid with the refine built-in to nodet
end
```

and performing built-in adjoint-based adaptation use

```
def iteration_steps
  flo # executes the flow solver
  adj # executes the adjoint solver
  rad # adapts the grid with the refine built-in to dual
end
```

8.3 External Grid Adaptation with Geometry Evaluation

Grid adaptation with the external REFINE executable linked to geometry is the recommended workflow. This form of grid adaptation enables automation [9] by reducing manual intervention during initial grid generation and the grid adaptation sequence. While interactive grid generation is eliminated in this automated process, geometry definition issues can be identified that are glossed over in expert-guided grid generation. Incorporating geometry into grid adaptation can place stricter requirements on geometry models because irregularities in the geometry may be identified by the solution error estimation procedure. These geometry issues may not be apparent on coarser grids or with expert-guided grid generation [11].

The interpolation error control scheme denoted the "multiscale metric" in literature has been systematically verified in REFINE and independent implementations [12–14]. Adjoint-based (goal-based) approaches are possible [15] with the external grid adaptation, but are not described here because they have not reached the maturity of interpolation error control. See section 8.3.6 for an example of the multiscale metric applied to a transonic wing.

8.3.1 REFINE Execution

REFINE provides three executables to handle mutually exclusive dependencies, see Table 10. Some MPI implementations and batch queueing systems prevent MPI-enabled software from running on cluster head nodes, even for sequential operation. EGADS and EGADSlite implement identical function signatures, which make them mutually exclusive for static linking. EGADS and OpenCASCADE are required during bootstrap operations, see section 8.3.3. EGADSlite data are cached with the grid in `.meshb` files. Either EGADS or EGADSlite can be used on these `.meshb` files, where EGADS requires an `--egads [root_project].egads` to load the geometry. Invoking `ref` with no arguments will list available subcommands. Help on a particular subcommand is available via a `-h`, i.e., `ref adapt -h`.

Table 10: REFINE capabilities by executable.

Executable	MPI	EGADS with OpenCASCADE	EGADSlite
`ref`		x	
`refmpifull`	x	x	
`refmpi`	x		x

8.3.2 External Grid Adaptation Geometry Preparation

The Engineering Sketch Pad (ESP) [16] contains the Open-Source Constructive Solid Modeler (OpenCSM) [17], Engineering Geometry Aerospace Design System EGADS [18], and EGADSlite [19]. See section A.13.16 for details on installation and configuration. The `serveCSM` executable in the ESP distribution is used to interpret an OpenCSM script (with `.csm` file extension) to build the CFD domain

```
% serveCSM -skipTess [root_project].csm
```

where `-skipTess` runs in a noninteractive mode and avoids the creation of a tessellation for visualization. The OpenCSM script must dump one SolidBody for 3D simulations or one FaceBody for 2D and axisymmetric simulations to an `.egads` file. The OpenCSM `dump` statement creates an EGADS file with the `.egads` file extension for down stream processes. Typically, the root project name is used for the `[root_project].csm` that contains the statement

```
dump [root_project].egads
```

for the single SolidBody or FaceBody. The FaceBody should be in the x-y plane for 2D and axisymmetric, where the y axis is interpreted as radius. There is extensive documentation and tutorials available for OpenCSM that can be installed with ESP, see section A.13.16.

Boundary Conditions ESP has the concept of attributes that follow entities during construction in OpenCSM and evaluation in EGADS. These attributes persist to allow information to be conveyed between geometry creation steps and analysis tools. REFINE uses attributes to create a `.mapbc` file (section 4.2) that contains FUN3D boundary condition numbers (section 3) and a text description. The prerequisite is setting a `bc_name` attribute for every face in 3D or edge in 2D in the OpenCSM `.csm` script. If these attributes are missing, the user is required to create the `.mapbc` file manually. In 3D, set ESP attributes on the faces of the SolidBody

```
select face
attribute bc_name $4000_aircraft_fuselage
```

In 2D and axisymmetric, set ESP attributes on the edges of the FaceBody

```
select edge
attribute bc_name $5000_farfield
```

The OpenCSM `select` command has a rich set of methods for identifying entities and groups of entities in the model. Once applied, these attributes are persisted through subsequent geometry construction operations. It is convenient to apply these attributes as early as possible in the OpenCSM script to individual components (e.g., outer domain, symmetry plane, fuselage, flap) and allow ESP to track these attributes during complex assembly operations. Review the OpenCSM documentation and tutorials for details.

Geometry Creation ESP is optimized for natively constructing geometry convenient for CFD analysis. The import of STEP and IGES geometry is possible, see an overview in Table 11. These geometry creation and importation capabilities can be combined to define the CFD domain. ESP can be used to remove or replace problematic regions of imported geometry, but repairing the geometry in the system that created the geometry is recommended. A detailed description of these ESP capabilities with examples is provided in the ESP documentation and tutorials. The success of Boolean operations can be sensitive to the new surface-surface intersections created. A small perturbation of the input solids or manually adjusting the tolerance can be used to recover from failed Boolean operations.

The watertightness, smoothness, topology, and other properties of imported models can vary wildly. Inconvenient topology and loose boundary representation tolerances may impede geometry construction, initial grid generation, and subsequent grid adaptation [11]. These geometry properties are examined at the completion of the grid bootstrap process (see section 8.3.3) and can provide quick feedback to repair or improve the suitability and convenience of a geometry model. An interim SolidBody or FaceBody can be `dump`ed, bootstrapped, and triaged to debug Boolean operations used to construct complex domains.

EGADS Warnings to Investigate EGADS is used to create an initial tessellation of the geometry for visualization in OpenCSM and to initialize grid bootstrapping. A failure of EGADS to tessellate a face will be indicated with a warning of this form

```
EGADS Warning: Face 304 -> EG_fillTris = -24 (EG_tessThread)!
```

Table 11: ESP capabilities.

Pro	Con
ESP constrained sketcher	
Fastest way to create simple outlines and grow into 3D	Requires planning for robust sketch constraints
ESP manual sketcher	
Most robust way to build geometries in ESP	Requires math to develop the models
ESP solids	
Simple solids and Boolean operations to make geometry	Requires planning of multiple construction steps for increasing complexity
Import STEP	
Quickest way if you already have a model	Needs manual sewing of faces or one or more `MANIFOLD_SOLID_BREP`
Import IGES	
If it is all you have it can work; IGES Type 186, Manifold Solid B-Rep Objects import as solids	May requiring sewing the patches together to get a SolidBody
Discrete grids	
SLUGS [20] (part of ESP) creates a watertight set of surfaces from `.stl`	Manual process

The face indicated should be investigated, because failure to tessellate often indicates an issue with the face topology or edge curve and surface parameterized curve tolerance.

8.3.3 External Grid Adaptation Bootstrap

The external grid adaptation approach requires the grid-to-geometry association, which is built by bootstrapping the initial grid from the geometry. The grid-to-geometry association requirement excludes starting from an arbitrary initial grid. A high degree of automation is possible because the initial grid generation can be automated for valid geometry [9, 21].

Bootstrap the Grid with Geometry Association The bootstrapping and initial grid generation process is invoked for sequential execution by

```
% ref bootstrap [root_project].egads
```

and parallel execution by

```
% mpiexec -np 8 refmpifull bootstrap [root_project].egads
```

A successful bootstrap will produce a coarse initial volume filling grid for a surface grid that resolves curvature and geometric feature size of the SolidBody, `[root_project]-vol.meshb`. A 2D grid is also written to the same filename where the FaceBody geometry is resolved. A smaller number of partitions (8 in this example) should be used for bootstrapping rather than a typical flow solver execution because the grid starts extremely coarse and there are not enough vertices and elements to partition the domain to a larger number of partitions during bootstrap. More complex 3D models with many faces and tidy geometry can bootstrap with more cores than simple models, 2D models, or models with unresolved curvature in the EGADS tessellation.

3D and 2D execution of bootstrap is inferred by the presence of a single SolidBody or FaceBody in the `.egads` file, where the FaceBody is in the x-y plane. Axisymmetric assumes y in the FaceBody is radius and adding `--axi` to bootstrap sets the appropriate boundary conditions in the `.mapbc` file. The initial volume is filled by TetGen `http://tetgen.org` or optionally by AFLR3 `https://www.simcenter.msstate.edu/software/documentation/system/index.html`. See `ref bootstrap -h` for AFLR3 instructions. REFINE assumes that the `tetgen` or `aflr3` executable is in the user's path. The geometry can be triaged after the bootstrap operation and the bootstrap volume grid should be adapted to geometry or the Spalding law of the wall [22] before first flow solve.

Failure to bootstrap The bootstrap process will fail if EGADS cannot create an initial tessellation of *all* faces in the domain. The EGADS tessellation process is controlled through a `.tParams` attribute. REFINE will attempt to iteratively adjust `.tParams` on faces and edges to recover missing faces and report these attempts to standard output. The indexes of these faces and edges indicates places where there is an opportunity to improve the geometry. If this automatic process fails, the attribute `.tParams` can be set manually in OpenCSM and existing attributes will locally deactivate automatic adjustment. The special attribute `.tParams` is a set of 3 parameters. The first is the maximum length of an edge segment or face triangle side. The second limits the deviation between the centroid of the discrete object and underlying geometry. The third is the maximum interior dihedral angle in degrees. A zero for any of these parameters deactivates that constraint. The EGADS tessellation is exported as a surface `[root_project]-init-surf.tec` and discrete geometry `[root_project]-init-geom.tec` at the beginning of a bootstrap. If long triangle sides in the curvature metric remain at the end of the bootstrap, starting with a finer EGADS tessellation can prevent these "stuck" triangle sides. These stuck triangle sides are indicated by large maximum `ratio` indicated to standard output of bootstrap and the `l` variable in the `[root_project]-adapt-prop.tec` Tecplot file.

A list of suggestions to speed up the execution of TetGen is provided to standard output by REFINE when TetGen is invoked. If TetGen requires an unusually long time to complete, consider these suggestions: limit inserted vertices, loosen element shape measure targets, or limit grid optimization. Accommodating a wide range of scales (driven by geometry topology or curvature) between adjacent faces can increase execution time.

Self intersection of the surface will lead to a failure of TetGen or AFLR3 failure. When these grid generation tools fail, the surface is examined and surface triangle-triangle intersection locations are reported to standard output with face indexes. A `[root_project]-intersection.tec` Tecplot file is also written with intersection locations. The intersection file is not written if volume grid generation succeeds.

Triage Geometry Properties Failures in grid adaptation occur due to an interaction of geometry properties, solution error estimation procedure, and grid adaptation protections to maintain a valid grid for the flow solver. These failures are influenced by the grid-adapted solution and cannot be predicted with complete confidence. The following feedback on the geometry is provided to help understand the geometric contribution to failures. Modifications of the geometry require manual interaction, and it is advised to attempt the target or a pathfinder simulation to see if these issues create downstream problems before investing the manual interaction in repair.

The bootstrap process reports locations on geometry with inconvenient edge and face topology, unusually high curvature, small feature size, or large boundary representation tolerance [11]. Sliver faces are marked with `# sliver`, short edges are marked with `# short edge`, and curvature is ignored at locations with `# curve/tol`. These locations are reported to standard output and a Tecplot file `[root_project]-adapt-triage.tec`. The `[root_project]-adapt-geom.tec` Tecplot file renders the discrete surface, edges, and parameter curves of the geometry with the boundary representation in the `gap` variable. Other geometry variables include entity parameterization (`p0`, `p1`) and curvature (`k0`, `k1`). The discrete representation of the geometry can be used to scan the model for loose boundary representation tolerances (distance between `edge` and `pcurve` vertices). The `gap` variable can be filtered to show locations larger than the target viscous spacing (e.g., $y^+ = 1$). The tolerance may have a component tangential to the edge curve and a component perpendicular to the edge curve, where the perpendicular component is more likely to create a step in the surface grid. A large perpendicular component of face surface and edge curve mismatch is likely to cause downstream grid adaptation failures. When vertices are identified by a `gap` larger than target viscous spacing, examining the discrete `edge` and `pcurve` Tecplot zones provides more context for the large tolerance.

The other forms of inconvenient geometry are isolated to efficiency concerns (i.e., require additional refinement to resolve geometry features that are not required by the solution error estimate). The information provided by triage of the geometry can be synergized to prioritize required repairs of the geometry for analysis or potential improvements to the geometry to make grid generation more efficient [11]. For example, a short edge in the geometry is accommodated by resolving the nearby surface and volume grid to incorporate this small edge with limits on grid gradation (smooth size variation). Modifying the geometry to eliminate small edges may improve efficiency by removing the constraint of this edge. In practice, these small geometry features are mostly an aesthetic concern. Loose boundary representation tolerances may be a larger concern for subsequent grid adaptation operations [11].

Improve Initial Grid for Flow Solver The initial volume grid created in the bootstrap process does not conform to the anisotropic curvature and geometry feature size metric used to create the surface grid. The interpolated geometry-based metric is suitable to start inviscid grid adaptation. Boundary layer resolution suitable for viscous simulation can be built implicitly during grid adaptation or boundary layer formation can be accelerated by adapting to the u^+ of the Spalding law of the wall [22].

To adapt the volume grid to interpolated geometry curvature and feature size (add `--axi` to export an axisymmetric `.lb8.ugrid` from 2D input)

```
% mpiexec ... refmpi adapt [root_project]-vol.meshb \
                           -x [root_project]01.meshb \
                           -x [root_project]01.lb8.ugrid
```

or

```
% ref adapt [root_project]-vol.meshb \
         -x [root_project]01.meshb \
         -x [root_project]01.lb8.ugrid \
    --egads [root_project].egads
```

A metric that controls the interpolation error of u^+ is added with the options

```
--spalding [y+=1] [complexity] --fun3d-mapbc [root_project]-vol.mapbc
```

where an a priori estimate of the $y^+ = 1$ normal wall spacing and the complexity of the metric (one half the target number of vertices) is provided with the `.mapbc` extracted from OpenCSM attributes during bootstrap (or created manually). The metric complexity takes priority over the $y^+ = 1$ normal wall spacing, so larger normal wall spacing will result from a low complexity.

Pathfinder examination of the geometry properties can be made by adapting to the Spalding metric at high complexity values. This examination can be performed without the flow solver, where the target complexity should be approached via increasing complexity with a series of grid adaptations.

8.3.4 External Grid Adaptation Required Namelists

A template `fun3d.nml` must be created to be used by the adaptation procedure. These input options should be available in the template `fun3d.nml`. Their values will be substituted by `f3d`.

```
&project project_rootname = '' /
&special_parameters distance_from_file = '' /
&flow_initialization import_from = '' /
&code_run_control restart_read = '' /
```

to be substituted by the `f3d` script. The `.meshb` grid format is required by REFINE because it contains the EGADSlite geometry model and the grid-to-geometry association. AFLR3 format `.lb8.ugrid` is exported for FUN3D

```
&raw_grid grid_format = 'aflr3' /
```

The `&volume_output_variables` namelist in section B.4.30 is used to provide the solution from the flow solver to REFINE. The solution is interpolated to provide an initial condition for the next grid. Output options required for compressible and incompressible are

```
&volume_output_variables
 export_to='solb'
 x = .false.
 y = .false.
 z = .false.
 primitive_variables = .true.
 turb1 = .true.
 turb2 = .true.
/
```

and the output options required for generic gas are

```
&volume_output_variables
 export_to = 'solb'
 x = .false.
 y = .false.
 z = .false.
 primitive_variables = .false.
 u = .true.
 v = .true.
 w = .true.
 rho_i(1:99) = 99*.true.
 tt = .true.
 tv = .true.
 turb1 = .true.
 turb2 = .true.
/
```

For compressible gas simulations, Mach number is computed from the volume output file in REFINE by default. See the `--interpolant` option of `ref loop -h` for other options including reading a sensor field exported from FUN3D with

```
&sampling_parameters
 number_of_geometries = 1
 sampling_frequency(1) = -1
 type_of_geometry(1) = 'partition'
 export_to(1) = 'solb'
 variable_list(1) = 'temperature'
/
```

Where `temperature` can be any variable in the `&sampling_output_variables` namelist described in section B.4.32. Providing an exported sensor field is required for generic gas calculations because REFINE only has gas thermodynamic equations for a perfect gas of $\gamma = 1.4$. A `&sampling_parameters` solb partition can also be used in lieu of `&volume_output_variables` to export the solution for interpolation, if the order of the volume output variables is respected.

The fixed-point metric can process snapshots of the flow to produce the metric that reduces estimated interpolation error over a time window composed of these snapshots [23]. The `&volume_output_variables` namelist in section B.4.30 is used to provide a interpolated solution as an initial condition for the next grid. The sensor is exported with a positive `sampling_frequency` to produce a series of snapshots

```
&sampling_parameters
 number_of_geometries = 1
 sampling_frequency(1) = 10
 type_of_geometry(1) = 'partition'
 export_to(1) = 'solb'
 variable_list(1) = 'mach'
/
```

Typically, 20–100 snapshots are used in the fixed point metric per characteristic oscillation period or convective time period. The fixed-point metric can also track a deforming grid [23], where the vertex locations are required (before the sensor) with each snapshot

```
 variable_list(1) = 'x,y,z,mach'
```

The REFINE options for fixed-point metric with static and deforming grids are provided in section 8.3.5.

8.3.5 External Grid Adaptation Workflow via `f3d` and `case_specifics`

A general overview of `f3d` and its input file `case_specifics` is in section 8.2.1. The directory where `f3d` is invoked should contain the grid (`[project_root]01.meshb` and `[project_root]01.lb8.ugrid`), the boundary conditions (`[project_root]01.mapbc`), and input files (e.g., `fun3d.nml`, `tdata`). The steps for calling REFINE externally inside an iteration are the default

```
def iteration_steps # defines the function to for each iteration
  refdist # computes exact wall distance via "ref distance"
  flo # executes nodet
  ref_loop_or_exit # ref loop or exit when reached schedule_max_complexity
end
```

Where `ref loop` is the "adapt grid" box in Fig. 5, which includes reading the current grid in `.meshb` format, reading the current flow solution in `.solb` format, initializing EGADSlite/EGADS, computing the multiscale metric, adapting the grid to the metric with geometry evaluation, interpolating the solution, saving the grid in `.meshb` format, exporting the grid in `.lb8.ugrid` format, and exporting the interpolated solution in `.solb` format.

The `f3d` script schedules complexity using fewer subiterations when there is low variation in the forces [24]. The complexity is a measure of the metric that forms a sharp estimate of half the number of vertices in the adapted grid. The complexity starts at `schedule_initial_complexity`. Complexity is increased on the `schedule_subiteration_limit` iteration at the current complexity. If the `schedule_force` (defaults to `cd`) shows less difference between the max and the min divided by the mean over the last three grids than `schedule_relative_tol`, the complexity is increased. Lowercase total force coefficients for all boundaries are parsed from the `.forces` file at the end of the `flo` step and are available to define alternate `schedule_force` methods, e.g.,

```
def schedule_force; cl/cd; end
```

The `schedule_filename` tracks the current complexity, which defaults to `f3d_criteria.dat`. The last line of this schedule state file is used to infer the complexity for the next iteration, including situations where `f3d` execution is interrupted and restarted. This process continues until `last_iteration` or `schedule_max_complexity` is reached.

The number of cores increases with the ratio of `schedule_complexity_per_core` until `schedule_max_cores`. If `schedule_max_cores` is not set, it will be inferred from PBS or SLURM environment variables. The default behavior is to delete grid and solution files at the completion of each iteration unless `schedule_clean false`.

A full list of command line options is available via `ref loop -h`. Axisymmetric simulations should include

```
ref_cl("--axi")
```

The defaults for the multiscale metric are suitable for most simulations. Subsonic and transonic drag appears to converge more rapidly for external aerodynamics of streamline vehicles when gradation is increased and the interpolation error is controlled in the L_4-norm

```
ref_cl("--norm-power 4 --gradation 10")
```

When exporting the sensor field (required for generic gas) the sensor used to control interpolation error is set via

```
ref_cl("--interpolant #{project}_sampling_geom1.solb")
```

where `project` is set automatically by `f3d`. Fixed-point iterations are used to control interpolation error over a series of sensor snapshots [23]. The fixed-point multiscale is enabled with

```
ref_cl("--fixed-point _sampling_geom1_timestep 10 10 200")
```

where the first argument is set by sampling, the first number is the first iteration of the averaging window, the second number of the frequency of sampling, and the last number is the final iteration of the averaging window. The middle string of the sampling file output can be changed with the `label` option of `&sampling_parameters`, see section B.4.33. When the grid is moving or deforming, the fixed-point sensor field should begin with `x,y,z` and `--deforming` should be added

```
ref_cl("--fixed-point _sampling_geom1_timestep 10 10 200 --deforming")
```

8.3.6 ONERA M6 Grid Adaptation Tutorial

An ONERA M6 wing example is presented to review external grid adaptation process with geometry. This case is set up to run quickly as an illustrative example. Accurate fine-grid results are expected with sufficient resources [13]. The case is described on the Turbulence Modeling Resource website https://turbmodels.larc.nasa.gov/onerawingnumerics_val.html. This website contains a link for downloading the geometry in STP format or

```
% wget https://turbmodels.larc.nasa.gov/Onerawingnumerics_grids/AileM6_with_sharp_TE.s
```

where the `wget` argument should be a single line with no spaces. The `om6ste.csm` OpenCSM file is

```
1  despmtr radius 100.0
2  sphere 0 0 0 radius # outer boundary
3  select face
4  attribute bc_name $5000_farfield
5  import AileM6_with_sharp_TE.stp # wing step file
6  select face
7  attribute bc_name $4000_wing
8  scale 1.0/810.491484086        # scale to unit root chord
9  subtract # wing from sphere
10 box -2*radius 0 -2*radius 4*radius 2*radius 4*radius
11 select face
12 attribute bc_name $6662_ysymm
13 intersect # half domain
14 dump om6ste.egads
```

The outer boundary is described as a sphere with a radius of 100 root chord lengths (line 2). The far field (5000) boundary condition is applied to the sphere (line 4). The ONERA M6 STEP file is imported (line 5). The no slip (4000) boundary condition is applied to the wing (line 7). The model, described in mm, is scaled to have a unit root chord (line 8). The wing is subtracted from the sphere (line 9). A box is constructed to describe a half domain with symmetry (line 10). The y-symmetry boundary condition is applied to the box (line 12), which is used as a tool to restrict the wing and sphere domain to half-plane symmetry via an intersection (line 13). The `intersect` Boolean operation propagates the symmetry boundary condition to the resulting symmetry face. The resulting domain is exported with `dump` (line 14). The OpenCSM script is executed with

```
% serveCSM -batch om6ste.csm
```

The bootstrap process is used to obtain the initial grid, which is intentionally coarse to reduce the runtime of the tutorial. It is required that tetgen be available in the user's `$PATH` in order to run the `refmpi` step below (if AFLR3 is to be used, it must be in the `$PATH` and the `refmpi` command line must be modified accordingly).

```
% ref bootstrap om6ste.egads
% cp om6ste-vol.mapbc om6ste01.mapbc
% mpiexec refmpi adapt om6ste-vol.meshb \
    --spalding 0.01 4000 \
    --fun3d-mapbc om6ste-vol.mapbc \
    -x om6ste01.meshb \
    -x om6ste01.lb8.ugrid
```

A more typical starting complexity is 100,000 to 1,000,000 depending on the shape, curvature, and number of faces in the model. The $y^+ = 1$ length is typically 0.0001 of the characteristic length of the model and will depend on Reynolds number. The grid is required in two formats. The `.meshb` contains EGADSlite geometry and grid-to-geometry association records, which supports adaptation with REFINE. FUN3D cannot read the `.meshb` format, so the grid is exported to `.lb8.ugrid` to compute the flow solution. The grid can optionally be viewed in Tecplot with

```
% ref translate om6ste01.meshb om6ste01.plt
```

A template `fun3d.nml` must be created to be used by the adaptation procedure. Some fields of the template will differ from a standard `fun3d.nml` file as they will be substituted by `f3d`. The input file portion of the `fun3d.nml` is

```
&project
  project_rootname = '' ! required so it can be filled by f3d
/
&special_parameters
  distance_from_file = '' ! required so it can be filled by f3d
/
&flow_initialization
  import_from = '' ! required so it can be filled by f3d
/
&raw_grid
  grid_format = 'aflr3'
/
```

The reference conditions are

```
&force_moment_integ_properties
  area_reference  = 1.15315084119231
  x_moment_length = 0.801672958512342
  y_moment_length = 1.47601797621980
/
&reference_physical_properties
```

```
    mach_number       = 0.84
    angle_of_attack   = 3.06
    reynolds_number   = 14.6e6
  /
```

The solver parameters are set with

```
  &nonlinear_solver_parameters
    schedule_cfl       = 1.0 10.0
    schedule_cflturb   = 1.0 5.0
  /
  &code_run_control
    steps              = 100
    restart_read       = 'off'
  /
```

Typically, CFL numbers in the range of 5–10 are used to prevent flow solver divergence. These low CFL numbers are offset with subsequent interpolated initial conditions that result in overall faster convergence. The `steps` should be increased to 1,000–10,000, depending on the desired level of convergence. An atypical value of 100 steps is used here to speed up the case. Output is specified with

```
  &global
    boundary_animation_freq=-1
    volume_animation_freq=-1
  /
  &boundary_output_variables
    number_of_boundaries = -1 ! compute from following list
    boundary_list        = '1-12'
  /
  &volume_output_variables
   export_to='solb'
   primitive_variables = .true.
   turb1 = .true.
   x = .false.
   y = .false.
   z = .false.
  /
```

Volume output is required to provide the solution to REFINE for computing the Mach multiscale sensor field and the interpolated restart. Boundary output is optional to visualize the adapted grid and solution.

This example `case_specifics` file is setup for a quick example.

```
  root_project 'om6ste'
  last_iteration 4
```

```
schedule_max_cores 4
schedule_initial_complexity 4000
ref_cl("--norm-power 4 --gradation 10")
```

Typically, `last_iteration` is omitted and `schedule_max_complexity` is used to terminate the adaptive sequence. Typically, `schedule_max_cores` is omitted so that available cores can be inferred from the batch queueing system. A manual specification of `schedule_max_cores` may be required on a laptop or workstation. The `schedule_initial_complexity` should match the complexity used in the Spalding adaptation step and a larger value is typical. The multiscale norm and gradation specification shown here improves results for transonic drag. The defaults would be more appropriate for higher-speed flows. Adaptation is started with

```
% f3d start
```

and observed with

```
% f3d watch
```

FUN3D output files are available in the `Flow` directory as each grid is processed.

8.4 External Grid Adaptation without Geometry

External grid adaptation without geometry is also possible. New vertices will be inserted on a linear representation of the current surface grid and boundary layer structure will be altered. Element types other than tetrahedra and triangles will be frozen. Mixed-element boundary layer elements can be used to maintain the boundary layer structure of the initial grid, and metric conformity may degrade where the frozen mixed elements meet adapted tetrahedra and triangles. An expert-crafted grid with mixed-element boundary layer should be converted to `[project_root]01.meshb` and `[project_root]01.lb8.ugrid` formats via the `ref translate` utility. The `ref loop` command expects `.meshb` format, which will be missing the geometry records inserted by bootstrap. The flow solver inputs (section 8.3.4) and scripting (section 8.2.1) will then proceed in the same fashion as external grid adaptation with geometry. External grid adaptation without geometry has been applied to compute heating of blunt bodies at large freestream Mach numbers [25].

8.5 Deprecated Grid Adaptation without Geometry

A deprecated workflow provides grid adaptation without geometry using an old version of REFINE linked as a library and called through NODET and DUAL executables. FUN3D computes the metric and the REFINE library modifies the grid. These older REFINE libraries do not obtain the metric conformity nor do they produce smoothness like the external version of REFINE. The first step is to construct a metric that describes the desired size and anisotropy of the adapted grid elements. The second step is to produce an adapted grid that is based on this metric.

Feature-based adaptation constructs the metric based on properties of the flow solution. Adjoint-based adaptation constructs the metric from the flow and adjoint solutions to reduce estimated errors in a specified output function. The namelist `&adapt_metric_construction` (section B.4.38) specifies the metric construction method. The related namelist description and documentation provides the mathematical formulation of the metric.

FUN3D supports a number of grid adaptation libraries. The namelist `&adapt_mechanics` (section B.4.39) specifies the grid adaptation library and its options. The REFINE grid adaptation library is distributed and linked with FUN3D by default.

8.5.1 Built-in Grid Adaptation Plane Specification and Grid Freezing

When adapting a grid with REFINE version one (`adapt_library = 'refine/one'` in `&adapt_mechanics`), all boundary faces must be specified as frozen or a geometry definition mush be provided via FAUXGeom. Use the default `patch_lumping='none'` in the `&raw_grid` namelist, as lumping will change boundary patch indexes making it more difficult to specify geometry.

No Geometry, Where the Surface Nodes are Frozen The `adapt_library = 'refine/one'` option cannot preserve the high aspect ratio structures within viscous layers, and so viscous layers must be frozen for a specified distance away from the surface to maintain grid quality. This is invoked with the `adapt_freezebl` command within the `&adaptation_mechanics` namelist, see section B.4.39 for more details and for guidance in specifying this distance.

Additionally, specific surfaces that do not have a viscous boundary condition can be frozen by listing the surface numbers (one per line) in a file named `[root_project].freeze`. For example, `[root_project].freeze` that contains

```
5
7
```

will freeze points on boundary patches 5 and 7. This is also useful for boundary surfaces that do not have an analytical definition handled by FAUXGeom.

FAUXGeom for Planar Boundaries For viscous problems, where the grid on the complex geometry of the body is frozen, FAUXGeom can be used to provide an analytical definition of planar symmetry, farfield, and propulsion boundary surfaces. This allows adaptation to occur on these planar surfaces of the grid, even when the boundary layer grid is frozen. This is a particularly important capability for symmetry planes. At present, FAUXGeom can only handle planar surfaces.

FAUXGeom reads the file `faux_input`. Here is an example file:

```
4
 5 xplane -5.0
 3 yplane -1.0
 1 zplane  1.0
16 general_plane 2.0
   0.707 0.707 0.0
```

The first line is how many faux surfaces are being defined. The subsequent lines have a face number, type of face, and a distance associated with the particular geometry. In this example, the first faux face defined corresponds to surface 5 in the grid and is an $x = -5.0$ constant plane. Faux faces are similarly defined for the z and y planes of surfaces 3 and 1. Surface 16 is a plane perpendicular to a $(0.707, 0.707, 0.0)$ normal that is located 2.0 away from the origin in the direction of the normal; the plane passes through the point $(1.414, 1.414, 0.0)$.

8.5.2 Built-in Feature-Based and Interpolation-Error Adaptation

The `&adapt_metric_construction` variable `adapt_feature_scalar_form` defines the operator that is applied to the `adapt_feature_scalar_key` to compute an adaptation intensity. This intensity is raised to the `adapt_exponent` power to produce a scaling of an isotropic element size estimate on the current grid. The anisotropy of the metric is introduced by the Hessian of the `adapt_hessian_key` variable. The multiscale interpolation error metric can be emulated with `adapt_feature_scalar_form = 'none'`, where the `adapt_complexity` must be set and `adapt_output_tolerance` is ignored. See section B.4.38 for complete details and mathematical description.

Set `restart_read='on'` in section B.4.14 to read the flow solution. Run NODET with the `--adapt` command line option in the directory with the flow restart. The result will be a new grid and interpolated solution file with the `adapt_project` project name. After adaptation, the flow solver can now be restarted with this new grid and interpolated solution by changing the `project_rootname` namelist variable.

`iteration_steps` for `f3d` To perform built-in feature-based adaptation use

```
def iteration_steps
  flo   # executes nodet
  adapt # executes nodet --adapt
end
```

8.5.3 Built-in Adjoint-Based Adaptation

Adjoint-based adaptation requires that a flow solution be calculated in the `Flow` directory and an adjoint solution be calculated in the `Adjoint` directory. See section 6 for more information on obtaining an adjoint solution. The adjoint solution is based on the functional defined in `rubber.data` and this is the same functional targeted for grid adaptation.

Adaptation is performed by executing DUAL with the command line options `--rad --adapt`. The adjoint solver reads the `fun3d.nml` in the `../Flow` directory), so this is the place to specify `&adapt_metric_construction` and `&adapt_mechanics` options. The freeze and FAUXGeom files are read in the current directory, `Adjoint`.

The result will be a new grid and interpolated solution restart file in the `../Flow` directory and an interpolated adjoint restart in the `Adjoint` directory. The project name of these new files is `adapt_project`.

`iteration_steps` for `f3d` To perform build-in adjoint-based adaptation use

```
def iteration_steps
  flo # executes nodet
  adj # executes dual
  rad # executes dual --rad --adapt
end
```

Examining Convergence of Built-in Adjoint-Based Adaptation For feature-based adaptation, there are no tools available to study the convergence of the adaptive grid process, only the solution, grid size, and functional output of the flow solver are available. The practitioner is left to examine the results and exercise engineering judgment.

For output-based adaptation, the items available for feature-based adaptation are augmented with a remaining error estimate and an error corrected functional for `adapt_error_estimation = 'embed'`. While the error corrected functional is printed to standard output for `adapt_error_estimation = 'single'`, it should not be used for monitoring, because it is correcting the algebraic error of the reconstructed solution (not the discretization error as in the embedded grid method).

The system `grep` command can be used to extract this information. For example the adapted grid size is

```
grep nnodesg Adjoint/*_rad_out
```

where the last argument is the file where standard output is stored. The remaining error estimate is extracted with

```
grep remaining_error Adjoint/*_rad_out
```

and both the linear and high order corrected functional is extracted with

```
grep function_corrected Adjoint/*_rad_out
```

where the linear is listed before the higher order. The higher order corrected functional is preferred to examine convergence, and the linear corrected function is provided to flag any problems with interpolation. Again, these error corrected functionals are not valid for `adapt_err_estimation = 'single'`.

8.5.4 Built-in 2D Grid Adaptation

There are a number of considerations for performing grid adaptation of 2D geometries. A 3D grid should be used with one layer of spanwise prism elements between symmetry planes at $y = 0$ and $y = 1$. The adjoint solver lacks a 2D-specific mode, so set `twod_mode = .false.` in the `&raw_grid` namelist. To properly construct the metric from the 3D grid and solution, set `adapt_twod = .true.` in the `&adapt_metric_construction` namelist. An adaptation library that supports 2D should be selected from `adapt_library` in `&adapt_mechanics`.

9 Design Optimization

The FUN3D design framework uses a gradient-based optimization procedure. One potential approach to obtaining the required sensitivity derivatives is a conventional forward mode of differentiation, such as finite-differencing, complex-variable formulations, operator overloading, or direct differentiation. Since the cost of these techniques scales directly with the number of input parameters, these methods are most efficient for problems where the number of outputs is considerably larger than the number of inputs. For such problems, FUN3D provides a complex variable formulation as described in section 9.15. However, for most aerodynamic design problems, the converse is true; the number of design variables is typically much larger than the number of objective functions and/or constraints. In this context, an adjoint, or reverse mode of differentiation is preferred.

FUN3D provides a discrete adjoint capability to efficiently determine the sensitivities required by a gradient-based design procedure. The adjoint approach enables the user to compute sensitivity derivatives of an output function with respect to an unlimited number of design variables at a cost equivalent to a single additional flow solution. For a general review of sensitivity analysis techniques, see [26] and [27].

The adjoint approach used in FUN3D relies on discrete linearizations of the relevant components of the flow solver. Most of FUN3D's compressible perfect gas and incompressible capabilities are accounted for within the adjoint-based framework. Discretely consistent sensitivities have been demonstrated for both steady and unsteady inviscid, laminar, and turbulent flows based on the one-equation model of Spalart and Allmaras. Grid topologies may contain any combination of element types and may also contain overset grid discretizations. Grids may be static, noninertial, or may contain any combination of static, rigidly-moving, or deforming overset component grids. Both compressible and incompressible formulations are available. The most commonly-used boundary conditions are implemented in the adjoint framework, and a broad range of objective/constraint functions is also available. However, the user is encouraged to review the latest release notes or contact `Fun3D-Support@lists.nasa.gov` to determine if a specific analysis capability is currently supported by the adjoint implementation. For a detailed overview of the adjoint-based procedure used in FUN3D and examples of its use for design optimization, see [28] and the references contained therein.

Users are encouraged to gain extensive experience using FUN3D for analysis purposes before attempting design optimization. This experience will aid in properly setting up optimization cases, understanding the steps involved, and interpreting the results.

The adjoint-based algorithms are very efficient, but a typical optimization will still require the equivalent of $\mathcal{O}(20)$ typical analyses. Therefore, securing sufficient computational resources is critical to performing realistic high-fidelity design. It should also be noted that the various optimization packages supported by FUN3D may behave very differently for a given design problem; moreover, the optimal algorithm is generally problem-dependent.

At this time, the documentation provided here is aimed at design optimization of steady flows. The extension to simulations involving unsteady flows is available for general use (e.g., see [29]), but is not currently covered here. Please contact `Fun3D-Support@lists.nasa.gov` if interested in using this capability.

9.1 Objective/Constraint Functions

To perform a gradient-based optimization, the user must specify at least one objective function to quantify the merit of the configuration. In the FUN3D design infrastructure, such objective functions may take a very general form as described here. Note that all of the supported optimization packages always seek to *minimize* the chosen objective function. Care should be taken to pose the objective function accordingly. Multiple outputs may be accounted for in a variety of ways. Constraints may be included implicitly within the objective function(s) as penalty terms. Explicit constraint functions may also be posed, as either equality or inequality constraints.

Note that the primary limitation in posing the problem statement is the general ability of the chosen optimization package to handle the design problem posed by the user. For example, the PORT optimization software does not support the use of explicit constraints. KSOPT is the only supported optimization package that supports the use of more than one objective function; however, FUN3D offers several approaches to scalarize multiple objectives for other packages. Multipoint design is also supported in several forms. See section 9.9 and section 9.10 for specific details on these capabilities.

The FUN3D flow and adjoint solvers do not distinguish between objective functions and constraints. The solvers themselves merely provide function values and their sensitivities for use during the optimization procedure. The actual optimization packages are the only components in the design framework that make a distinction between objective functions and constraint functions.

9.1.1 Terminology

It is useful to establish some basic terminology when composing the design problem statement. Within the FUN3D design infrastructure, the user specifies one or more *component* functions based on typical solver outputs. These *component* functions are then combined to form a single *composite* function. Multiple *component* functions may be used to form *composite* functions, and in turn, multiple *composite* functions may ultimately be specified. The user then classifies each *composite* function, designating it either an objective function or a constraint function. Again, this distinction is solely for the optimization algorithm; FUN3D simply evaluates and linearizes each of the *composite* functions in a generic sense and provides them to the optimization scheme.

The adjoint formulation requires a separate adjoint solution for each composite function. For example, a drag-minimization problem with an explicit lift constraint will generally require two adjoint solutions at each step of the design procedure (one based on drag and one based on lift). Rather than performing separate adjoint executions for each function, FUN3D's adjoint solver is implemented such that multiple adjoint solutions may be computed simultaneously by cycling through a series of right-hand side vectors. In this manner, much of the computational overhead associated with discretizing the adjoint system is amortized over the collection of specified functions, and each additional function only increases the overall computational cost by approximately 40%. See [7] for further details on this aspect of the implementation.

9.1.2 Functional Form

Composite functions take the following general form in FUN3D:

$$f_i = \sum_{j=1}^{J_i} \omega_j (C_j - C_j^*)^{p_j} \tag{1}$$

Here, the index J_i corresponds to the number of individual component functions comprising composite function i. The factor ω_j represents a user-specified weighting coefficient in the summation; C_j is a FUN3D scalar output quantity, C_j^* is a user-specified target value for that output quantity, and p_j is a user-specified exponent. The currently available FUN3D output functions that may be posed as C_j are listed in Table 12. Though not explicitly represented in Eq. 1, the implementation also allows the user to only use specific boundary contributions to C_j and not all boundaries if desired. This could be used to focus the optimization function on forces acting on the wing or tail only. Note that when composing an objective or constraint function it is often helpful to scale the expected value to an $\mathcal{O}(1)$ quantity. This can be readily done using the ω_j factor. Additional details relevant to more complex functions are covered in section 9.2. The specific input mechanism for providing each of the component/composite function parameters will be discussed at length in section 9.6.2.

To demonstrate the use of the general functional form given by Eq. 1, several examples are given here:

Unconstrained Drag Minimization For an unconstrained problem in which the user wishes solely to minimize drag, one potential approach might

Table 12: Objective/constraint component function keywords.

Keyword	Function
`cl`, `cd`	Lift, drag coefficients
`clp`, `cdp`	Lift, drag coefficients: pressure contributions
`clv`, `cdv`	Lift, drag coefficients: shear contributions
`cmx`, `cmy`, `cmz`	x/y/z-axis moment coefficients
`cmxp`, `cmyp`, `cmzp`	x/y/z-axis moment coefficients: pressure contributions
`cmxv`, `cmyv`, `cmzv`	x/y/z-axis moment coefficients: shear contributions
`cx`, `cy`, `cz`	x/y/z-axis force coefficients
`cxp`, `cyp`, `czp`	x/y/z-axis force coefficients: pressure contributions
`cxv`, `cyv`, `czv`	x/y/z-axis force coefficients: shear contributions
`powerx`, `powery`, `powerz`	x/y/z-axis power coefficients
`clcd`	Lift-to-drag ratio
`fom`	Rotorcraft figure of merit
`propeff`	Rotorcraft propulsive efficiency
`rtr_thrust`	Rotorcraft thrust function
`pstag`	RMS of stagnation pressure in cutting plane disk
`distort`	Engine inflow distortion
`boom_targ`	Near-field p/p_∞ pressure target
`sboom`	Coupled sBOOM ground-based noise metrics
`ae`	Supersonic equivalent area target distribution
`press_box`	RMS of pressure in user-defined box, also pointwise dp/dt, $d\rho/dt$
`cpstar`	Target pressure distributions

be to specify a single composite function consisting of a single component function with $\omega_1 = 1.0$, $C_1 =$ `cd`, $C_1^* = 0.0$, and $p_1 = 2$. In this manner, the objective function is simply

$$f = C_D^2, \tag{2}$$

where the quadratic form has been chosen to provide a convex function space.

Drag Minimization with Lift Penalty To add an interior penalty term accounting for a lift equality constraint of 0.5, one might instead use two component functions within the same single composite function where $\omega_1 = 10.0$, $\omega_2 = 1.0$, $C_1 =$ `cd`, $C_2 =$ `cl`, $C_1^* = 0.0$, $C_2^* = 0.5$, and $p_1 = p_2 = 2$. These parameters yield

$$f = 10C_D^2 + (C_L - 0.5)^2. \tag{3}$$

In this case, any deviation of the lift coefficient from its target value of 0.5 will "penalize" the objective function. The weighting parameters ω_j have been selected based on typical magnitudes of C_D and C_L, so as to produce roughly equivalent contributions to the objective function. Note that the choice of these weighting parameters is heuristic in nature and often troublesome in practice.

Drag Minimization with Explicit Lift Constraint In this example, the lift constraint $C_L = 0.5$ is instead posed as an explicit constraint for the optimizer. Here, two composite functions are formed, each with a single component function. First, an objective function is specified as in Eq. 2 with $\omega_1 = 1.0$, $C_1 =$ `cd`, $C_1^* = 0.0$, and $p_1 = 2$. As before, this yields

$$f_1 = C_D^2. \tag{4}$$

However, an additional composite function for the lift constraint is also specified with $\omega_1 = 1.0$, $C_1 =$ `cl`, $C_1^* = 0.5$, and $p_1 = 1$, which gives

$$f_2 = C_L. \tag{5}$$

This explicit form of the lift constraint is generally preferred in practice.

9.2 Some Details on Specific Objective/Constraint Functions

Many of the scalar functions shown in Table 12 and designed to be used as the term C_j in Eq. 1 are straightforward. For example, the keyword `cd` is sufficient to characterize a drag-based component function. However, some of the scalar functions listed in Table 12 require the user to be aware of specific requirements and/or to provide additional auxiliary data. In this section, scalar functions requiring further data and/or explanation are covered.

9.2.1 Lift-to-Drag Ratio (Keyword: `clcd`)

This function must be specified with a `0` for its boundary index, i.e., it must be applied to the entire configuration and is not available for individual boundary patches. It is defined as

$$f = \frac{C_L}{C_D}. \tag{6}$$

9.2.2 Rotorcraft Figure of Merit (Keyword: `fom`)

This function is defined as

$$f = \frac{C_L^3}{2C_{M_z}^2}. \tag{7}$$

Note that this functional form assumes that the rotor axis of rotation is in the $+z$ direction. The definition also represents the square of the traditional Figure of Merit function. See [30] for a motivation for this modified form. This function must be specified with a `0` for its boundary index, i.e., it must be applied to the entire configuration and is not available for individual boundary patches.

9.2.3 Rotorcraft Propulsive Efficiency (Keyword: `propeff`)

This function is defined as

$$f = \frac{-C_z}{C_{M_z}}. \tag{8}$$

Note that this functional form assumes that the rotor axis of rotation is in the $+z$ direction. The minus sign has been introduced to yield a positive efficiency since C_{M_z} is negative. This function must be specified with a `0` for its boundary index, i.e., it must be applied to the entire configuration and is not available for individual boundary patches.

9.2.4 Rotorcraft Thrust (Keyword: `rtr_thrust`)

This function is defined as the force normal to the plane of a rotor system:

$$f = C_L \cos(\theta_T) - C_D \sin(\theta_T) \tag{9}$$

where the angle from $+z$-direction θ_T in radians is the variable `thrust_angle` that must be set in the source file `LibF90/custom_transforms.f90`.

9.2.5 RMS of Stagnation Pressure (Keyword: `pstag`)

This function computes the RMS of stagnation (total) pressure in a circular disk that passes through the grid in a specified location and orientation. This is commonly employed to quantify engine inlet distortion. The user must specify the variables in the `&pstag_function` namelist within `fun3d.nml`. See section B.4.45 for details related to this namelist. This function is only available for compressible flows.

9.2.6 Engine Inflow Distortion (Keyword: `distort`)

This function provides a metric for quantifying inflow distortion at the fan face of an aircraft engine. A circumferential sector of a specified angle is incrementally rotated about the center of the fan face to locate the sector with the smallest average value of total pressure. The function is defined as

$$f = \frac{\bar{p}_t - \bar{p}_{t,min}}{\bar{q}} \tag{10}$$

where $\bar{p}_t$ and $\bar{q}$ are the average total pressure and average dynamic pressure over the entire fan face, respectively; and $\bar{p}_{t,min}$ is the average total pressure in the circumferential sector with the minimum value. For further details on the background of this approach, see [31].

The user must specify the sector and incremental angles in the `&fan_distortion` namelist within `fun3d.nml`, and the averaging performed in Eq. 10

may be weighted using either area or mass flow. See section B.4.46 for details related to these namelist inputs. Any number of engine faces may be accommodated, with each appearing as a component function within one or more composite functions. Each engine face must appear as a single boundary patch in the mesh, and the patch number must be specified in the first column of data associated with the relevant component function (see section 9.6.2).

When Fun3D evaluates objective or constraint functions involving this metric, two files will be written for each specified fan face. The file `[project]_distortion.i.dat` provides the value of average total pressure in each sector as a function of the sector angle Θ in Tecplot™ format for the i-th fan face specified in `rubber.data` (see section 9.6.2). Similarly, the file `[project]_worstsector.i.dat` provides Tecplot™ geometric line data indicating the $\Theta = 0°$ direction in black and the sector corresponding to $\bar{p}_{t,min}$ in red for the i-th fan face. This data may be superimposed on contour plots of the fan face solution to assist in visualizing the distortion field. An example of such data is shown in the figure below. The angle Θ is measured counterclockwise when looking into the engine.

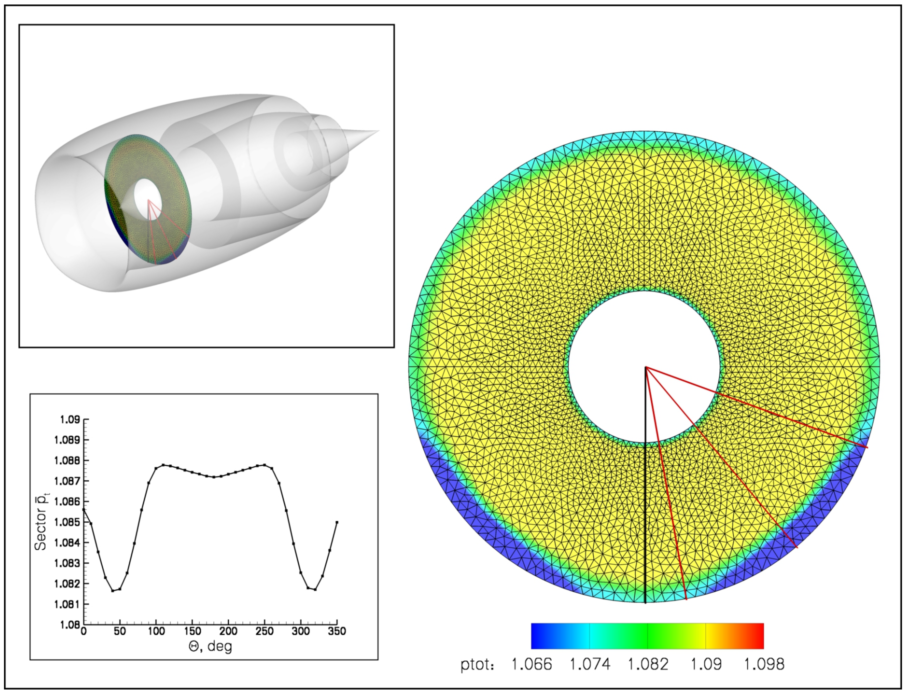


Figure 6: Contours of total pressure on an engine fan face. Black line represents $\Theta = 0°$ and red lines represent centerline and boundaries of the 60° sector with minimum average total pressure.

The current implementation of this function requires that each fan face remain static, and may not move as a result of kinematics, aeroelasticity, changes due to design optimization, or any other mechanism. Fan face patches are assumed to be circular and planar, but may lie in any general orientation. A fan face may be cut by at most a single symmetry plane. This function is only available for compressible flows.

9.2.7 Near-field p/p_∞ Pressure Target (Keyword: `boom_targ`)

This function allows inverse design of near-field pressure signatures, which is a commonly used tactic for creating low sonic boom configurations. This function is only available for compressible flows. The user specifies `yz`-coordinate pairs through which rays are passed parallel to the `x`-axis. Off-body pressure distributions in the vicinity of an aircraft are nominally oriented parallel to the freestream velocity vector. In the case of a nonzero angle of attack, the rays are rotated about a user-specified center of rotation to align them with the freestream direction. The user also provides the minimum and maximum `x`-extent for the rays. A user-specified number of points are evenly distributed along each ray and the grid element containing each point is identified. See section B.4.41 for guidance on the required namelist inputs.

The functional form is given by

$$f = \sum_{i=1}^{N} \omega_i \left(\left.\frac{p}{p_\infty}\right|_i - \left.\frac{p}{p_\infty}\right|_i^* \right)^2 \tag{11}$$

where p is the local static pressure. The summation takes place over all points in the rays defined by the user, and the values of p are evaluated at the centroids of the enclosing elements. The values of ω_i and $\left.\frac{p}{p_\infty}\right|_i^*$ are user-supplied pointwise weighting coefficients and target values of p/p_∞, respectively, which must be provided in a file named `pressure_target.dat`. If this file is not present, the target values of p/p_∞ are set to 1.0 and the weighting coefficients are set to 1.0. Note that with the above functional form, the target and exponent parameters present in Eq. 1 are usually set to `0.0` and `1`, respectively.

A template for `pressure_target.dat` is typically generated by first extracting a set of p/p_∞ distributions for a known configuration by running the optimization driver with `what_to_do = 1` in the `design.nml &design` namelist, see section 9.5.1. Note that the input value `weight` must be set to `.true.` and the desired ray extraction (y, z) coordinate pairs must be specified in the `&sonic_boom` namelist in `fun3d.nml`. This operation produces a file named `pressure_signatures.dat`, which uses the same file format intended for the `pressure_target.dat` target input file. (Note that the file format is amenable to Tecplot™ usage.) The user may then use the `pressure_signatures.dat` file to develop a `pressure_target.dat` input file by modifying the existing pressures to reflect their target values as desired. Note that by specifying `weight=.true.` in the `&sonic_boom` namelist, a column of data representing pointwise weighting coefficients (all initially set to

1.0) will be provided in pressure_signatures.dat. This column of data is required to be present in pressure_target.dat. The individual weights may be left as 1.0, or they may be modified on an individual basis to optionally weight a specific region of the signature more or less in the final objective function. A brief example of this file format for a case involving two off-body signatures is shown below. Note that target distributions need not have the same number of locations as, nor line up with, the eventual sampling locations along the extraction rays. FUN3D will linearly interpolate between input target values to obtain values at the sampling locations.

```
VARIABLES = "x", "y", "z", "p/pinf", "weight"
zone t="Signal 1"
  -0.500E+01 0.100E-11 0.826E+00 0.110010E+01 0.100E+01
  -0.472E+01 0.100E-11 0.835E+00 0.110011E+01 0.100E+01
  -0.415E+01 0.100E-11 0.855E+00 0.110012E+01 0.100E+01
  -0.354E+01 0.100E-11 0.876E+00 0.110016E+01 0.100E+01
zone t="Signal 2"
  -0.500E+01 0.100E-11 0.182E+01 0.102008E+01 0.100E+01
  -0.472E+01 0.100E-11 0.183E+01 0.102008E+01 0.100E+01
  -0.414E+01 0.100E-11 0.185E+01 0.102009E+01 0.100E+01
  -0.356E+01 0.100E-11 0.187E+01 0.102012E+01 0.100E+01
  -0.335E+01 0.100E-11 0.188E+01 0.105013E+01 0.100E+01
  -0.320E+01 0.100E-11 0.188E+01 0.105014E+01 0.100E+01
  -0.264E+01 0.100E-11 0.190E+01 0.105017E+01 0.100E+01
```

9.2.8 Coupled sBOOM Ground-Based Signatures, Noise Metrics, and Equivalent Areas (Keyword: sboom)

This option uses the adjoint capability of the Burgers equation boom propagation code sBOOM [32] to inversely design ground pressure signatures, optimize a ground-based noise metric, or match equivalent area distributions. [33–35] FUN3D must be configured and built with the sBOOM library as described in section A.13.14 to use this capability.

In the coupled FUN3D-sBOOM implementation, FUN3D is responsible for computing pressure signals in the immediate vicinity of an aircraft (typically within 10 body lengths). The sBOOM tool then propagates these disturbances to the ground using an augmented Burgers equation that considers effects such as nonlinearity, thermoviscous absorption, and any number of molecular relaxation phenomena during the propagation of waveforms through the atmosphere. In this manner, the user can directly simulate ground-based noise metrics such as A-weighted loudness or compute other loudness measures (e.g., Perceived Level) from the computed ground signatures. In a similar fashion, a coupled adjoint problem is used to determine the discrete sensitivities of the ground-based metrics with respect to any of FUN3D's typical design parameters which may then be used to optimize the configuration.

sBOOM can generate off-track signatures based on ray theory using user input azimuthal angles. sBOOM can also predict the sonic boom signatures in the presence of wind, turn rate (changing heading angle), climb rate, climb angle, and acceleration (dMach/dt). During maneuvering flight, boom focusing is possible. The current version sBOOM in not able model focusing and will exit with an appropriate message if focusing occurs.

Equivalent area distributions are computed with reversed augmented Burgers equation (when `rs`< (`alt`−`hg`)) or a direct conversion of off-body pressures (when `rs`> (`alt`−`hg`)). This is different than the Mach cut equivalent area matching approach in section 9.2.9. The discrete sensitivities of the difference between a target and the computed equivalent areas are provided to Fun3D. The target area is specified with the `target_dpress` and `target_xx` variables in the `&sboom` namelist.

The user must provide inputs relevant to the nearfield pressure signal extraction (see section B.4.41) as well as parameters specific to the sBOOM library (see section B.4.42). Note that when the `sboom` keyword is used as the component function name, the actual form of the objective/constraint component function is determined entirely within sBOOM. In this case, the values of ω, C^*, and p in Eq. 1 are ignored. This function is only available for compressible flows.

9.2.9 Supersonic Mach Cut Equivalent Area Distribution (Keyword: `ae`)

This function aims to match a target Mach cut equivalent area distribution for supersonic flows. The Mach cut equivalent area distribution is directly computed from surface pressures and geometry for this function. This is a different approach than the equivalent area computation of sBOOM in section 9.2.8. The function is defined as

$$f = \sum_{i=1}^{N} \omega_i (L_i + V_i - A_i^*)^2 \tag{12}$$

where N represents the total number of longitudinal stations used to sample the solution and geometry for the current azimuth, and L_i and V_i are the lift and volume contributions, respectively, to the current equivalent area. The term A_i^* represents the user-supplied target equivalent area distribution. The ω_i enables the user to locally weight individual segments of the distribution if desired. Note that with the above functional form, the target and exponent parameters present in Eq. 1 are usually set to `0.0` and `1`, respectively. This function is only available for compressible flows, and the configuration is assumed to align with the `x`-axis.

Any number of desired azimuthal (centerline/off-track) locations may be specified and used as individual component functions. The user must provide the data indicated in the `&equivalent_area` namelist in `fun3d.nml` as described in section B.4.43. A centerline symmetry plane may be used to reduce computational expense; in this case, the cutting planes at each longitudinal station will be correctly accounted for on the virtual side of the aircraft. A file `ae_target.dat` must also be provided, which describes the (optionally weighted) target equivalent area profiles.

A template for `ae_target.dat` is typically generated by first extracting a set of equivalent area distributions for a known configuration by running the optimization driver with `what_to_do = 1` in the `design.nml &design` namelist, see section 9.5.1. This operation will produce a Tecplot™ file `[project_rootname]_ae.dat` which uses the same file format intended for the target input file `ae_target.dat`. The user may then use the `[project_rootname]_ae.dat` file to develop a `ae_target.dat` input file by modifying the existing equivalent areas to reflect their target values as desired. Note that the file `[project_rootname]_ae.dat` contains a column of data representing the pointwise weighting coefficients ω_i (all initially set to `1.0`). This column of data is required to be present in `ae_target.dat`. The individual weights may be left as `1.0`, or they may be modified on an individual basis to optionally weight a specific region of the distributions more or less in the final objective function. A brief example of this file format for a case involving three azimuthal signatures is shown below. Note that target distributions need not have the same number of locations as, nor line up with, the longitudinal sampling locations. FUN3D will linearly interpolate between input target values to obtain values at the sampling locations. Also note that in the input file `ae_target.dat`, the second and third columns of the format are ignored.

```
VARIABLES = "x", "V", "L", "Ae", "weight"
zone t="Ae Function 1"
  -0.01000E+00  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+01
   0.13839E+01  0.25482E-01 -0.26289E-02  0.22853E-01  0.10000E+01
   0.27678E+01  0.47548E-01 -0.64155E-02  0.41133E-01  0.10000E+01
   0.41517E+01  0.76165E-01 -0.10361E-01  0.65804E-01  0.10000E+01
zone t="Ae Function 2"
  -0.01000E+00  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+01
   0.14018E+01  0.25700E-01 -0.26610E-02  0.23039E-01  0.10000E+01
   0.28036E+01  0.48215E-01 -0.64628E-02  0.41752E-01  0.10000E+01
   0.42054E+01  0.77379E-01 -0.10358E-01  0.67020E-01  0.10000E+01
   0.56072E+01  0.11457E+00 -0.14045E-01  0.10052E+00  0.10000E+01
   0.98126E+01  0.30728E+00 -0.25726E-01  0.28156E+00  0.10000E+01
zone t="Ae Function 3"
  -0.01000E+00  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+01
   0.14155E+01  0.26009E-01 -0.26166E-02  0.23392E-01  0.10000E+01
   0.28310E+01  0.48902E-01 -0.62883E-02  0.42614E-01  0.10000E+01
   0.42465E+01  0.78591E-01 -0.10011E-01  0.68579E-01  0.10000E+01
```

Finally, the solver will also provide the user with a Tecplot™ output file `[project_rootname]_ae_cuts_i.dat` for the `i`th specified equivalent area function. These files contain the actual cross-sectional slices of the aircraft that were generated for each azimuthal function.

9.2.10 RMS of Pressure in User-Defined Box, Pointwise dp/dt, $d\rho/dt$ (Keyword: `press_box`)

This function computes the RMS of a quantity in Cartesian region, which is typically used to indicate a region of the flow is important for grid adaptation or that fluctuations in a region should be minimized in a design. These functions rely on the inputs associated with the `&press_box_function` namelist; see section B.4.44 for details. The function may be composed of the RMS value of the pressure within a user-defined box in the domain, or for unsteady simulations, the time derivative of pressure or density (for compressible flows) at a single grid point.

9.2.11 Target Pressure Distributions (Keyword: `cpstar`)

FUN3D has an inverse design capability where the objective function may be composed of target pressure distributions. The file containing the `j`th target distribution must be named `cpstar.data.j`. However, setup is tedious, primarily due to the difficulty in specifying pressure distributions on a three-dimensional configuration. If this capability is of interest, please contact `Fun3D-Support@lists.nasa.gov` for more detailed guidance.

9.3 Geometry Parameterizations

In order to perform shape optimization, FUN3D must be provided with a set of design variables describing the geometric shape of the configuration. FUN3D is currently set up to interface directly with geometry parameterizations provided by MASSOUD [36], Bandaids [37], or Sculptor™. MASSOUD and Bandaids are software packages developed by Jamshid Samareh of NASA Langley. Users may request copies of these packages on the NASA Software Catalog website; tutorial information for these tools is available on the FUN3D website. These packages allow the user to parameterize completely arbitrary shapes using a free-form deformation technique. The packages are very efficient, robust, and also provide analytic Jacobians of the parameterization, which are necessary for FUN3D-based design. Sculptor™ is a popular commercial package developed by Optimal Solutions and also provides the necessary data for FUN3D-based design. Note that any combination of parameterizations based on these tools may be used within the context of a single optimization. For example, the planform of a wing or tail surface may be best treated using MASSOUD, while Bandaids or Sculptor™ may be most appropriate for a wing-body fillet region or a feature such as a fuselage protuberance. Finally, the user may also use a parameterization scheme of their choosing; see section section 9.3.4 for further details.

9.3.1 Surface Grid Extraction

To parameterize a surface grid using any of the above tools, it must first be extracted to a Tecplot™ file. To do this, add a `&massoud_output` namelist to `fun3d.nml` to group all of the required boundary patches for a body to be parameterized into a single body (see also section B.4.40):

```
&massoud_output
  n_bodies = 2                ! parameterize 2 bodies: wing and tail
  nbndry(1) = 6               ! # of bounds that comprise wing
  boundary_list(1) = '3-8'    ! wing bounds (account for lumping!)
  nbndry(2) = 3               ! # of bounds that comprise tail
  boundary_list(2) = '9,10,12' ! tail bounds (account for lumping!)
/
```

Note that the boundary indices shown here must reflect any patch lumping that may have been requested in the `&raw_grid` namelist (see also section B.4.2). A single iteration of the flow solver should now be executed with the `--write_massoud_file` command line option. This will generate a `[project_rootname]_massoud_bndry#.dat` file for each of the boundary groups present in the `&massoud_output` namelist. These files contain the information necessary to parameterize the surface grid using any of the aforementioned tools. See the documentation for those packages for further instructions on how to construct the actual parameterization.

9.3.2 Access to Executables

If MASSOUD, Sculptor™, or a user-defined executable is being used for parameterizations, the executable for those packages must be available in the runtime `PATH`. The executables for MASSOUD and Sculptor™ must be named `massoud` and `sculptor`, respectively. If using a user-defined parameterization package (see section 9.3.4), the executable must be named according to the input provided in the `&design` namelist in `design.nml` (see section 9.5.1). The optimization driver supplied with FUN3D will attempt to call these executables if such parameterization types are present. If Bandaids are being used, no additional executables must be supplied; all Bandaid evaluations are handled internally by FUN3D.

9.3.3 Notes on Using Sculptor™

If Sculptor™ is being used, FUN3D will invoke Sculptor™ in batch (non-GUI) mode during the course of the optimization. However, current versions of

Sculptor™ will still attempt to communicate with an X server, even when run in this fashion. If the system does not run an X server (such as compute nodes on a cluster), then a fake X server such as `Xvfb` is recommended. You will need to execute the fake server prior to running the design optimization. For example, a run script may have the following commands:

```
Xvfb :1 &
export DISPLAY=:1.0 # for bash
setenv DISPLAY :1.0 # for c shell
[any command that uses Sculptor]
```

The syntax here may vary; if this does not allow the optimization driver to run Sculptor™ in batch mode successfully on the system, the user should get in touch with Sculptor™ support for assistance.

In addition, the parameterization of all bodies treated using Sculptor™ must be bookkept within a single set of Sculptor™ input files. For example, in the wing-tail example above, both bodies must be contained in a single instance of Sculptor™ files. Therefore, the `&massoud_output` namelist described above should group all of the desired boundaries necessary to describe the geometry(s) of interest into a single body:

```
&massoud_output
  n_bodies = 1                ! wing and tail grouped into a body
  nbndry(1) = 9               ! # of tail and wing bounds
  boundary_list(1) = '3-10,12' ! wing and tail boundaries
/
```

Each of the desired bodies may be worked on independently within Sculptor, but they must ultimately appear as a single body to FUN3D.

9.3.4 Using Other Parameterization Packages

FUN3D provides a generic interface for user-defined external geometric parameterization packages. The user-defined tool must provide the surface mesh coordinates as a function of some vector of design variables for the body of interest. The partial derivatives of these coordinates with respect to the design variables must also be supplied. See [38] for an example of such an approach.

To invoke a user-defined parameterization package for one or more bodies present in the simulation, the user must tag individual bodies appropriately (see section 9.6.2) and provide the executable name to be invoked by FUN3D's design driver at run-time via the `&design` namelist in `design.nml` (see section 9.5.1). This may be a binary executable or simply a script that invokes other user-defined operations that may be necessary to evaluate the parameterization and its sensitivities.

When FUN3D requires an evaluation of the user-defined parameterization for a body (or its sensitivities), it will first write a file named `customDV.i`, where the `i` suffix corresponds to the body index for which FUN3D is requesting updated surface data. The format of the `customDV.i` file is as shown below.

```
#User defined design variables
#Number of DVs
         3
  1.907460000000000E+00
  0.000000000000000E+00
 -0.002469800000000E+00
```

The first two lines are comment lines, and the third specifies the total number of design variables in the parameterization for the current body (whether the user may have designated some active and others inactive in `rubber.data`; see section 9.6.2). The remaining lines in the file contain the current value of each design variable, with one value per line.

After writing the `customDV.i` file, FUN3D will then invoke the user-specified executable command. FUN3D will append a space and a single integer to this command, where the integer corresponds to the body index for which FUN3D requires an evaluation of the parameterization. The user-provided executable (or script) must be capable of parsing this integer command-line option in order to process the requested body.

After the external package has completed the evaluation of the parameterization and its sensitivities, the data must be supplied to FUN3D via a Tecplot™ file named `model.tec.i.sd1`, where the integer `i` in the filename represents the current body index. The format of this file must be as follows:

```
TITLE = "PARAMETERIZATION DATA"
VARIABLES = "X" "Y" "Z" "ID" "XD1" "YD1" "ZD1"  "XD2" "YD2" "ZD2"  "XD3" "YD3" "ZD3"
ZONE T = group0, I = 4, J = 1, F=FEPOINT
0.0 1.0 0.0 235 1.234 -23.0 892.1 -23.0 82.123 -90.2 -905.2 857.12 348.2
1.0 1.0 0.0 872 4.14 -0.123 -0.324 23.13 2978.2 -0.114 -982.4 -3.22 0.1185
1.0 0.0 0.0 912 -0.34 0.938 8.45 78.23 -35.2 -0.023 8.32 -0.009 -0.92
0.0 0.0 0.0 455 9.01 -8.23 -0.456 2.56 1.21 0.0 -0.091 -1.22 0.0088
1 2 3 4
```

The trivial (and completely contrived) example shown here provides surface mesh point locations and the corresponding sensitivities for a single quad element parameterized by three design variables. The title in the first line may contain anything the user desires. The variables identified in line 2 represent the x-, y-, and z-coordinates for the current surface mesh point, the node index in the global grid for the current surface mesh point, and the sensitivity derivatives of the x-, y-, and z-coordinates of the current surface mesh point with respect to each of the design variables provided by FUN3D in `customDV.i` above. The file should contain a single zone, indicated by the third line of the example file shown. Here, the zone title specified by `T =` may be anything the user desires. The `I =` value corresponds to the number of surface mesh points for the current body, while the `J =` value specifies the number of surface elements (triangles or quads) contained in the surface mesh for the current body. FUN3D will only read the `I =` value; the surface mesh topology is immaterial in this context.

In this example, the floating point values have been truncated for readability; users are strongly encouraged to provide double-precision values in practice. The connectivity information at the end of the file is not used by FUN3D. It may be omitted if desired; however, it is often instructive to load this file into Tecplot™ for visualization. In that context, the connectivity data (including the appropriate `J =` value in the file header) will be required.

For very large surface meshes and/or parameterizations containing a very large number of design variables, I/O of this ASCII format can be inefficient. There is an alternative C-binary/Fortran stream format that may be used in its place; interested users should get in touch with `Fun3D-Support@lists.nasa.gov` for further details on this option.

To support execution of the user-defined parameterization tool, auxiliary files may be provided in the `description.i` directory; the filenames should be provided in the file `user_def_param_files.data` (see section 9.6.1).

9.4 Design Optimization Directory Structure

The optimization driver `opt_driver` requires a very specific directory structure. It can be established by running `opt_driver` in an interactive mode with the `--setup_design` command line option. The number of design points should be 1 for single-point design or greater than 1 for multipoint design.

```
opt_driver --setup_design [number of design points]
```

This interactive command will prompt the user for several directory paths required by the optimization, namely the paths to the FUN3D source code, the configuration directory where FUN3D was configured and built, and the path to the location where the design will be performed. Here, directories should be provided as absolute paths contained in single quotes, with trailing slashes omitted, i.e.,

```
'/absolute/path'
```

At the completion of this setup procedure, a summary of the files required from the user will be echoed to the screen. The directories created in the specified run location are shown in Table 13. The `i` suffix in `description.i` and `model.i` represents the design point index. For single-point design, this will be `1`; for multipoint design, this value will range from `1` to the number of user-specified design points. The setup procedure will populate the various directories with links to the required FUN3D executables and templates for various input files described below.

Table 13: Directory structure required for FUN3D-based design.

Directory	Description
ammo	Location where optimization will be executed
description.i	Location of all baseline input files describing design point i
model.i	Location where analysis & sensitivity analysis of design point i will be performed

9.5 Contents of the ammo Directory

The ammo directory will contain files related to the optimization procedure itself. This includes the design.nml input file described in section 9.5.1 and a link to the opt_driver executable.

9.5.1 &design namelist in design.nml

This namelist contains variables that specify inputs relevant to running design optimization. While many of the default values for the variables in this namelist may be acceptable, at a minimum, the user will need to specify what_to_do and provide the base_directory name in which the case is to be executed.

```
&design
  what_to_do                 = 1
  restart_optimization       = .false.
  previous_evaluations       = 0
  max_design_cycles          = 10
  base_directory             = ''
  n_design_pts               = 1
  design_pt_weight(:)        = 1.0
  opt_algorithm              = 4
  max_function_evals         = 20
  tau_subproblem             = 1.0e-4
  feas_tol_val               = 1.0e-3
  dot_method                 = 1
  user_def_executable        = ''
  body_grouping              = .false.
  actuator_disk_grouping     = .false.
  n_body_transforms          = 0
  body_transforms(:)         = 0
  mpirun_prefix              = 'mpiexec'
  adjoint_nproc              = 0
  snopt_optimality_tolerance = 1.e-4
/
```

what_to_do = 1

This is the operation performed by the optimization driver.

`'1'` only performs the function evaluation.

`'2'` performs the function evaluation and sensitivity analysis.

`'3'` performs optimization.

<u>`restart_optimization = .false.`</u>

This logical specifies whether to start the optimization from the baseline problem description or to restart the optimization from a previous run already executed in this directory.

<u>`previous_evaluations = 0`</u>

Number of previous evaluations when `restart_optimization=.true.` for which history files have been saved. Several files are saved with each function/sensitivity evaluation (convergence history, surface history, etc.) by appending the evaluation number to the end of the file name. This `previous_evaluations` allows the previous history files to remain to maintain a contiguous history for the optimization. See section 9.6.7 for details on this archiving scheme.

<u>`max_design_cycles = 10`</u>

This scalar integer sets an upper limit on the number of design cycles the optimizer may perform.

<u>`base_directory = ''`</u>

This string should be an absolute path to the design location specified during the setup procedure. Path should be enclosed in single quotes and contain no trailing slashes.

<u>`n_design_pts = 1`</u>

This scalar integer is the number of design points to be considered during the optimization. The value should be at least `1` and less than or equal to the number of design points specified during the setup procedure.

<u>`design_pt_weight(:) = 1.0`</u>

A nonnegative real-valued scalar should be specified for each design point. The value represents the weighting to be applied in the linear combination of objective functions from each individual design point as used to construct the final composite objective function (see section 9.10). For single point optimization, a single value of `1.0` should be specified.

<u>`opt_algorithm = 4`</u>

This integer specifies the optimizer to be used. FUN3D must be configured and built with the specified optimization library. See section A.13 for more information on obtaining installing the optimizers.

`‘1’` utilizes the DOT/BIGDOT™ optimizer.

`‘3’` utilizes the KSOPT optimizer.

`‘4’` utilizes the PORT optimizer.

`‘5’` utilizes the NPSOL™ optimizer.

`‘6’` utilizes the SNOPT™ optimizer.

`max_function_evals = 20`

This scalar integer is only relevant for PORT-based optimizations and sets an upper limit on the number of flow solutions allowed during the design.

`tau_subproblem = 1.0e-4`

This scalar real value is only relevant for DOT/BIGDOT™- and PORT-based optimizations and specifies the relative function convergence criterion for which the optimization will terminate.

`feas_tol_val = 1.0e-3`

This scalar real value is only relevant for optimizations based on the DOT/BIGDOT™, NPSOL™, and SNOPT™ packages. The value specifies the feasibility tolerance for constraints.

`dot_method = 1`

This scalar integer is only used for DOT/BIGDOT™-based optimization and specifies the optimization method to be used with DOT/BIGDOT™. See the DOT/BIGDOT™ documentation for further information.

`user_def_executable = ''`

This string enclosed in single quotes represents the name of the executable (or script) to be launched if user-defined geometric parameterizations are to be used. The executable should be available in the users `PATH`. See also section 9.3.4.

`body_grouping = .false.`

This logical specifies whether any body grouping should be applied. See also section 9.6.4.

`actuator_disk_grouping = .false.`

This logical specifies whether any actuator disk grouping should be applied. This is specified in a manner identical to body grouping. See also section 9.6.4.

`n_body_transforms = 0`

This scalar integer specifies the number of MASSOUD-parameterized bodies for which spatial transforms should be applied. See also section 9.6.10.

`body_transforms(:) = 0`

These integers specify the MASSOUD-parameterized bodies to which spatial transforms are to be applied. There should be `n_body_transforms` entries supplied. If `n_body_transforms` is zero, this line of data should not be present. See also section 9.6.10.

`mpirun_prefix = 'mpiexec'`

This string enclosed in single quotes will be used as a prefix when running MPI programs. This is usually `mpirun` or `mpiexec`, depending on the MPI implementation, or perhaps `aprun` on Cray® systems.

`adjoint_nproc = 0`

This scalar integer specifies the number of processors on which to execute the adjoint solver. Most often, this is the same number of processors requested for the job and the value used for the flow solver. However, in the event that a split communicator is being used for the flow solver (e.g., for simultaneous execution of Suggar++, VisIt, dedicated file I/O, etc), the actual flow solution will be performed on some subset of the cores dedicated to the overall job. In this case, the adjoint solver must be forced to execute on the same reduced number of cores, which is specified using this input. If the value of `adjoint_nproc` is set to `0`, then the adjoint solver will be executed on the total number of processors allocated to the overall job.

`snopt_optimality_tolerance = 1.e-4`

This real parameter declares the tolerance required to achieve optimality when running with SNOPT™.

9.6 Contents of the `description.i` Directory

The `description.i` directory serves as a repository for the baseline files for the CFD model, the geometric parameterization, and several other input files related to the computational model for the `i`th design point. These files must be set up by the user prior to the run and will not be modified by Fun3D during execution. During the initial setup procedure, templates for several input files will be placed in this location to aid in setting up the case. During the actual optimization, the optimization driver will copy files from this directory into the `model.i` directory as needed.

Any files normally required by the flow solver must be present in this directory. This would typically include the grid and boundary condition files and `fun3d.nml`. If the mesh uses overset grids assembled with the Suggar++ utility, the Suggar++ DCI file must be present as well. The optional file `remove_boundaries_from_force_totals` (section B.3) may also be present, if desired.

In addition to the files normally required by the flow solver, a number of other files must also be present to perform the design optimization, some of which are optional. These are described below.

9.6.1 Geometry Parameterization Files

If performing shape optimization, the user must provide the relevant parameterization files for each body in the mesh to be modified. The specific set of files required for each body depends on the parameterization package(s) being used.

MASSOUD Parameterizations For MASSOUD parameterizations, the MASSOUD parameterization files should be named `design.gp.j`, where `j` is the index of the body to be designed. The files specifying the values of the raw MASSOUD variables should be named `design.j` for each of the bodies to be designed. For FUN3D-based design, the custom design variable linking feature of MASSOUD must be used. If the raw MASSOUD variables are intended to be used as-is, simply set the linking matrix as the identity matrix in the MASSOUD `.usd` file. These files specifying the design variable linking for each body should be named `design.usd.j`.

The MASSOUD control file specifies the names of the files outlined above for MASSOUD and must be provided as `massoud.j` for the `j`th body. The files listed in the MASSOUD control file must reflect these names. The first line of the MASSOUD control file(s) must have a positive integer equal to the number of custom design variables. If the intent is simply to use the raw MASSOUD variables as-is, this value is simply the number of raw MASSOUD variables for that body. For the in/out-of-core parameter, use in-core (0). The file name for Tecplot™ output viewing must be named `model.tec.j` for the `j`th body. The design variable grouping file specified should be named `designVariableGroups.j` for the `j`th body. The FAST output file name can be named anything the user wishes; the FUN3D tools do not use this MASSOUD output file. Finally, the user design variable file for the `j`th body should be named `customDV.j`. In summary, a `massoud.j` control file for the `j`th body should look like the following:

```
#MASSOUD INPUT FILE
# runOption 0-analysis, >0-sd users dvs, -1-sd massouds    dvs
```

```
52
# core 0-incore solution, 1-out of core solution
0
# input parameterized file
design.gp.1
# design variable input file
design.1
# input sensitivity file - used for runOption > 0
design.usd.1
# output file grid file
newframe.fast.1
# output Tecplot file for viewing
model.tec.1
# file containing the design variables group
designVariableGroups.1
# user design variable file
customDV.1
```

Bandaid Parameterizations For Bandaid parameterizations, the input files created by the Bandaid setup tool should be named `bandaid.data.j` for the `j`th body. Because Bandaid parameters behave linearly, the sensitivities contained in these files are constant and this input is all that is required during the course of a design.

Sculptor™ Parameterizations For Sculptor™ parameterizations, the user must provide `[project_rootname].mdf`, `[project_rootname].sd1`, `[project_rootname].vol`, and `[project_rootname].stu` files. See the Sculptor™ documentation for more details on each of these files. A file named `[project_rootname].def` must also be provided. An example `[project_rootname].def` file for a simple two-body parameterization is shown below:

```
set_mdf [project_rootname].mdf
default 1 DV1-T1 0.00
default 1 DV1-T2 0.00
default 1 DV1-T3 0.00
default 1 DV1-T4 0.00
default 1 DV1-T5 0.00
default 2 DV2-T1 0.00
default 2 DV2-T2 0.00
default 2 DV2-T3 0.00
export model.tec.1
exit
```

The filename specified for the `export` command must be `model.tec.1`. The remainder of the file is dictated by the specific parameterization developed in the Sculptor™ application.

After the configuration has been parameterized using Sculptor™ and all of the appropriate files have been assembled for FUN3D-based design, a copy of the original `[project_rootname]_massoud_bndry#.dat` file must also be placed in the `description.i` directory, but with the file name changed to `[project_rootname].sd1`. Sculptor™ requires this baseline file during the optimization.

Finally, prior to performing the design, the `[project_rootname].sd1` file must be read into Sculptor™ in GUI mode as "Import Mesh/CFD as Tecplot Point FE." Following this, the Sculptor volumes need to be imported onto the `[project_rootname].sd1` file, and then the model must be saved again. Once this is done, the command `export model.tec.1` within the `[project_rootname].def` batch script will generate a `model.tec.1.sd1` file as needed for FUN3D-based design optimization.

User-Defined Parameterizations In the event that a user-defined geometric parameterization package is to be used, the user must provide a file `user_def_param_files.data`. Since FUN3D will not be aware of any auxiliary files that may be needed by the user-defined parameterization package, those files should be listed here, with a single file name per line. Each file named here must be provided by the user. At run time, FUN3D will move the named files to the appropriate location prior to execution of the parameterization executable indicated by the user in the `&design` namelist in `design.nml`. See also section 9.3.4 and section 9.5.1.

9.6.2 `rubber.data`

This section describes how to set up each block of the design control file `rubber.data`. The template provided in the `Adjoint` directory of the source code distribution is installed in the `description.i` directory during setup. This file serves as the primary control file during the course of the optimization and stores all of the high-level information relevant to the design. The file is repeatedly read and updated by the various tools during the design procedure. A simple example of this file to be used for discussion purposes is shown below.

```
################################################################################
########################### Design Variable Information ########################
################################################################################
Global design variables (Mach number, AOA, Yaw, Noninertial rates)
  Var Active         Value               Lower Bound            Upper Bound
 Mach    0   0.800000000000000E+00  0.000000000000000E+00  0.900000000000000E+00
  AOA    1   1.000000000000000E+00  0.000000000000000E+00  5.000000000000000E+00
  Yaw    0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
xrate    0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
yrate    0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
```

```
zrate    0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
Number of bodies
    2
Rigid motion design variables for 'wing'
  Var Active         Value               Lower Bound          Upper Bound
RotRate  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotFreq  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotAmpl  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotOrgx  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotOrgy  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotOrgz  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotVecx  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotVecy  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotVecz  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
TrnRate  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
TrnFreq  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
TrnAmpl  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
TrnVecx  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
TrnVecy  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
TrnVecz  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
Parameterization Scheme (Massoud=1 Bandaids=2 Sculptor=4 User-Defined=5)
    1
Number of shape variables for 'wing'
    3
Index Active         Value               Lower Bound          Upper Bound
    1    1   1.000000000000000E+00  0.000000000000000E+00  2.000000000000000E+00
    2    1   1.000000000000000E+00  0.000000000000000E+00  2.000000000000000E+00
    3    1   1.000000000000000E+00  0.000000000000000E+00  2.000000000000000E+00
Rigid motion design variables for 'tail'
  Var Active         Value               Lower Bound          Upper Bound
RotRate  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotFreq  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotAmpl  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotOrgx  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotOrgy  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotOrgz  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotVecx  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotVecy  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
RotVecz  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
TrnRate  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
TrnFreq  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
TrnAmpl  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
TrnVecx  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
TrnVecy  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
TrnVecz  0   0.000000000000000E+00  0.000000000000000E+00  0.000000000000000E+00
Parameterization Scheme (Massoud=1 Bandaids=2 Sculptor=4 User-Defined=5)
    2
Number of shape variables for 'tail'
    2
Index Active         Value               Lower Bound          Upper Bound
    1    1   2.000000000000000E+00 -1.000000000000000E+00  5.000000000000000E+00
    2    1   2.000000000000000E+00 -1.000000000000000E+00  5.000000000000000E+00
################################################################################
############################### Function Information ###########################
################################################################################
Number of composite functions for design problem statement
    2
################################################################################
Cost function (1) or constraint (2)
    1
If constraint, lower and upper bounds
    0.0 0.0
Number of components for function   1
    1
Physical timestep interval where function is defined
```

```
    1 1
Composite function weight, target, and power
1.0 0.0 1.0
Components of function   1: boundary id (0=all)/name/value/weight/target/power
    0 cl            0.000000000000000    1.000   10.00000 2.000
Current value of function   1
   0.000000000000000
Current derivatives of function wrt global design variables
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
Current derivatives of function wrt rigid motion design variables of body   1
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
Current derivatives of function wrt shape design variables of body   1
   0.000000000000000
   0.000000000000000
   0.000000000000000
Current derivatives of function wrt rigid motion design variables of body   2
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
Current derivatives of function wrt shape design variables of body   2
   0.000000000000000
   0.000000000000000
################################################################################
Cost function (1) or constraint (2)
    2
If constraint, lower and upper bounds
    -0.03 -0.01
Number of components for function   2
    1
Physical timestep interval where function is defined
    1 1
Composite function weight, target, and power
1.0 0.0 1.0
Components of function   2: boundary id (0=all)/name/value/weight/target/power
```

```
    0 cmy           0.000000000000000    1.000    0.00000 1.000
Current value of function   2
   0.000000000000000
Current derivatives of function wrt global design variables
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
Current derivatives of function wrt rigid motion design variables of body   1
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
Current derivatives of function wrt shape design variables of body   1
   0.000000000000000
   0.000000000000000
   0.000000000000000
Current derivatives of function wrt rigid motion design variables of body   2
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
Current derivatives of function wrt shape design variables of body   2
   0.000000000000000
   0.000000000000000
```

Global Design Variable Data This section of `rubber.data` lays out global design variables for the computation. These include the freestream Mach number, angle of attack, and angle of yaw. For simulations using a noninertial reference frame, the noninertial rotation rates about each of the three Cartesian coordinate axes are also available as design variables.

Quantities associated with each variable are specified on their own row in the file and have several attributes that must be set by the user. The first column is a dummy index and is merely to assist the user in quickly navigating through the file. The second column is a toggle to activate the design variable.

If this value is a `1`, the variable will be allowed to change during the design. If the value is assigned a `0`, this variable will be held constant at the value specified. For incompressible flows and mixed-element grids, the Mach number must be declared inactive. Similarly, for flows posed in the inertial reference frame, the noninertial rates must be declared inactive.

The third column in the design variable block is the current value for this design variable. The values of any active variables in this file will take precedence over other input decks during design. For example, the flow solver will run an angle of attack of `1` degree in this case, regardless of what may be specified in `fun3d.nml`. Columns four and five specify the upper and lower bounds for the current design variable.

Body-Specific Design Variable Data The next input following the Mach number and angle of attack entries specifies the number of bodies for which the user has provided shape parameterizations. Note that not every body in the grid must be included here. If the wing of an aircraft is the sole focus of the optimization, there is no need to account for other boundaries such as the tail or fuselage here.

Following the number of bodies, there should be two blocks of design variables for each desired body, namely a list of rigid motion variables controlling the dynamics of the body, and a set of shape parameters controlling the shape of the body. The columns of inputs are identical to those described above for Mach number and angle of attack.

The bodies present in the computation may be listed in any order; however, the order of their appearance in this control file must match the integer suffix on their parameterization files that are provided in the `description.i` directory, as well as files such as `body_grouping.data`, `transforms.j`, etc.

The first section for the current body specifies design variables governing rigid body motion and is only applicable for time-dependent problems. For optimization of steady flows and/or static geometries, the rigid motion data is irrelevant but must be present in this file. These variables should be set as inactive in these cases.

The next block of inputs relates to the shape parameterization for the current body. First, the parameterization scheme is identified by a scalar integer. The following values are available: (1) MASSOUD, (2) Bandaids, (4) Sculptor™, and (5) User-Defined. The next input specifies the number of parameterized shape variables on the current body and the subsequent lines lay out the design variable information for that body. A row of data must be provided for every variable in the parameterization, whether it is active or not. (Note however, that internally, the optimizer is only made aware of the variables marked as active.) If a parameterization contains 25 variables, then 25 rows must appear in this corresponding block of `rubber.data`, even if only a subset is active. If the design variable linking feature in MASSOUD or

Bandaids has been used to create additional derived variables, they must also appear here. Note that the "Active" attribute for shape variables may take values of not only 0 or 1, but also -1 in certain multipoint design scenarios (see section 9.10).

Care should be taken in choosing upper and lower bounds for shape variables. Optimizers tend to fully explore the design space, which may result in infeasible shapes (or extreme shapes the mesh movement/solvers cannot handle robustly). Set these limits conservatively; one can always restart a design with less restrictive bounds.

As noted previously, when using Sculptor™ parameterizations for multiple bodies, all such design variables must appear as a single concatenated body in `rubber.data`.

Cost Function/Constraint Specification The first line following the design variable block specifies the total number of composite functions to be used as objectives or constraints for the current design point. Multiple composite objective functions may be specified in certain cases; see section 9.9. Otherwise, a single composite objective function must be specified. The example file shown here contains a single composite objective function based on the lift coefficient and a single explicit constraint based on the pitching moment. Note that explicit constraints may only be specified if the optimization package chosen in the `&design` namelist in `design.nml` supports them.

Following the scalar value specifying the total number of composite objectives and constraints, each composite function and/or constraint will have a block of data associated with it. Objective functions and constraints may be specified in any order.

The first two inputs in the composite function block specify a scalar integer indicating how the current function is to be viewed by the optimizer. The two subsequent inputs represent lower and upper bounds on the function if it is to be used as a constraint. If the function is an objective function, the first input value should be `1`, and the lower and upper bounds must be present but their values are irrelevant. However, if the current function is to be used as a constraint, special attention must be paid to these inputs depending on the optimization package being used.

Constraints Using NPSOL™ and SNOPT™ If the current function is to be used as an inequality constraint, the first input should be `2`, and the lower and upper bounds should be set to their appropriate values. If the current function is to be used as an equality constraint, the first input should be `2`; however, the lower and upper bounds should both be set equal to the desired constraint value.

Constraints Using KSOPT and DOT/BIGDOT™ These optimization packages assume constraint functions of the form $f \leq 0$, such that the bound of the feasibility region is implicit in the function definition and the lower and upper bound inputs must be present but are not used. If the current function is intended as an inequality constraint, the first input should be `2`. If the current function is intended as an equality constraint, the first input value should be `3`. In this case, FUN3D's design driver will provide the current function to the optimizer as an inequality constraint, but will also bookkeep an equal and opposite function as an additional inequality constraint. In this manner, an equality constraint is achieved by only allowing the intersection of the two inequality constraints as feasible.

Following the classification of the current function, the next line states how many component functions comprise the current composite function. This can be any positive integer greater than or equal to `1`. Following the number of component functions, the user must specify the physical time step interval over which the function is to be applied. This input is only relevant to optimization of unsteady flows. For steady flows, the values of these two inputs are ignored but must be present.

The weight, target, and power to be applied to the current composite function are specified next. These values are only relevant when combining multiple composite objective functions into a single global objective function (see section 9.9). For all other cases, these values should be specified as `1.0`, `0.0`, and `1.0`, respectively.

At this point, each component function that contributes to the current composite function has a line specifying several pieces of data. The first column is the boundary patch over which to apply the current component. This index corresponds directly to the boundary patches in the CFD grid, and must reflect any patch lumping that is indicated in the `&raw_grid` namelist in `fun3d.nml` (see section B.4.2). If a component function is to be used over the entire grid (total drag, for example), simply put a `0` in this column. Alternatively, if a single boundary patch is to be targeted, one might apply the component function to only that patch. Several patches may be targeted by including a component function for each. The next column is the keyword for the aerodynamic quantity to be used for the current function component. For a list of available keywords, see section 9.1. The next column contains the current value of the current function component. This is an output value during the optimization and need not be set by the user. The final three columns in the row correspond to the weight, target value, and power to be applied to the current component function in constructing the overall composite function.

Current Function Value and Sensitivities Following the specification of the component functions, the next line of `rubber.data` contains the current value of the composite function. This is an output and need not be set by the user.

The remaining lines in the current function block contain the sensitivity derivatives with respect to all of the design variables listed in the top half of the file. This section is divided into derivatives with respect to the global design variables, as well as the rigid motion and shape design variables for each of the bodies laid out in the top portion of the file. These derivatives are outputs set by FUN3D and not by the user. However, a line for each design variable (both global variables as well as body-specific variables) must be provided in each composite function block present. The values do not matter, but the solvers need positions available in the file to store the current values.

9.6.3 `ae_target.dat` (optional)

If the function keyword `ae` is specified anywhere in `rubber.data`, the file `ae_target.dat` must be present prior to performing the optimization. This file provides the target equivalent area distribution(s) for each of the azimuthal locations specified in the `&equivalent_area` namelist in `fun3d.nml` (in the same order). See section 9.2.9 and section B.4.43.

9.6.4 `body_grouping.data` (optional)

This file is used to specify body grouping information. For example, if the objective function is the Figure of Merit FM for a three-bladed rotor, then the three blades (each typically specified as a separate parameterized body in `rubber.data`) should be associated into one group, so that sensitivity derivatives will reflect a composite $\partial(FM)/\partial D$ for all three blades. This capability requires that the bodies to be associated all have identical parameterizations (same number of design variables on each body, etc). The format of the `body_grouping.data` file is as follows:

```
Number of groups to create
1
Number of bodies in group, list of bodies
3
1 2 3
```

The first scalar integer specifies the number of groups to create (i.e., one rotor). The next set of inputs specifies the number of bodies in each group, followed by the bodies that comprise that group (i.e., each of the three rotor blade bodies).

9.6.5 `command_line.options` (optional)

The `command_line.options` file specifies any command line options to be used with the flow solver, the adjoint solver, or the MPI job launcher (`mpirun`, `mpiexec`, `aprun`, etc). An example of this file is shown below.

```
3
1 flow
'--rmstol 1.e-7'
1 adjoint
'--rmstol 1.e-3'
2 mpirun
'-nolocal'
'-machinefile ../machinefile'
```

The first line of the file specifies the number of programs for which command line options are being provided. The subsequent line must contain an integer followed by a keyword. The integer specifies how many command line options are being provided for the code identified by the keyword. The valid keywords are `flow`, `adjoint`, and `mpirun`. This line is followed by a line for each of the command line options provided for the code identified by the keyword. Each command line option should appear in single quotation marks on its own line. The specified programs and their associated command line options may appear in any order.

If a `-np x` option is provided as a command line option to the MPI job launcher to specify the number of processes to run (given by the value `x`), this value will override the value specified for execution of the adjoint solver in the `&design` namelist in `design.nml` (see section 9.5.1).

9.6.6 `cpstar.data.j` (optional)

Fun3D has an inverse design capability where the objective function may be composed of target pressure distributions. The file containing the `j`th target distribution must be named `cpstar.data.j`. However, setup is tedious, primarily due to the difficulty in specifying pressure distributions on a three-dimensional configuration. If this capability is of interest, please contact `Fun3D-Support@lists.nasa.gov` for more detailed guidance.

9.6.7 `files_to_save.data` (optional)

If desired, users may indicate specific files produced by the flow and adjoint solvers to be archived during an optimization. For example, custom visualization files may be produced at each design iteration (see section 5.3.1) to enable animations of the design history after the optimization is complete.

To specify certain files to be archived during the course of a design execution, the user may provide the optional file `files_to_save.data` in the `description.i` directory. Each line of the file consists of one of two keywords, either "Flow" or "Adjoint", followed by the specific name of a file to be archived. For example, the following inputs could be used to archive Tecplot™ files containing solution data on boundaries for both the flow and adjoint solvers, while also saving two additional Tecplot™ files containing user-specified sampling data for the flow solution:

```
Flow [project_rootname]_tec_boundary.dat
Flow [project_rootname]_tec_sampling_geom1.dat
Flow [project_rootname]_tec_sampling_geom2.dat
Adjoint [project_rootname]_tec_boundary.dat
```

At the end of each function evaluation (i.e., flow solution) for the `i`th design point, the files `[project_rootname]_tec_boundary.dat`, `[project_rootname]_tec_sampling_geom1.dat`, and `[project_rootname]_tec_sampling_geom2.dat` will be stored in the `model.i/Flow/SaveFiles` directory, with an integer corresponding to the current design iteration appended to each of the filenames. Similarly, the file `[project_rootname]_tec_boundary.dat` will be stored in the `model.i/Adjoint/SaveFiles` directory at the completion of each sensitivity analysis (i.e., adjoint solution).

9.6.8 `machinefile` (optional)

If the optimization will be executed in an environment which requires an explicit list of machines on which the MPI jobs will be executed, this file must be present. It should take the format required of the particular MPI implementation being used. If the optimization will be executed in an automated queuing environment, the job scheduler normally assigns the machines to be used at runtime and this file is therefore not required.

9.6.9 `pressure_target.dat` (optional)

If the function keyword `boom_targ` is specified anywhere in `rubber.data`, the file `pressure_target.dat` must be present prior to performing the optimization. This file provides the nearfield target p/p_∞ distribution(s) for each of the off-body locations specified in the `&sonic_boom` namelist in `fun3d.nml` (in the same order). See section 9.2.7 and section B.4.41.

9.6.10 `transforms.i` (optional)

Since MASSOUD uses a coordinate system specific to an assumed aircraft orientation, it is sometimes necessary to reorient a body from its physical position to a MASSOUD-aligned coordinate system and vice-versa. Examples might include a vertical tail or the various blades of a rotor system. The file describing the transform for the `i`th body should be included as `transforms.i`. The format of a typical `transforms.i` file is as follows:

```
ROTATE 0.0 0.0 1.0 -120.0
```

This would rotate the MASSOUD parameterization for the `i`th body by -120 degrees about a unit vector in the $+z$ direction. The commands `TRANSLATE` and `SCALE` are also available.

9.7 Contents of the `model.i` Directory

Just as for the `description.i` directories, the `i` in the `model.i` naming convention represents the design point index. This value is `1` for single point design or the design point index for multipoint design. The `model.i` directory contains the subdirectories `Flow`, `Adjoint`, and `Rubberize`. During the course of the design procedure, FUN3D will evaluate the relevant parameterizations and perform flow and adjoint solutions within these locations. These subdirectories are populated during the initial setup procedure with links to the required executables from the user's FUN3D installation. Files in the `model.i` subdirectories should not be modified by the user, although one may wish to observe various solver output files during the course of the optimization. All user-provided inputs are confined to files located in the `description.i` and `ammo` directories.

9.8 Running the Optimization

Once all of the required inputs and files have been provided, the user should first request a single function evaluation from the optimization driver. This is strongly recommended in order to identify any potential issues in the various inputs. To perform this check, set the value of `what_to_do` in the `&design` namelist in `design.nml` to `1`, and execute the optimization driver from the `ammo` subdirectory:

```
opt_driver > screen.output &
```

Here, the output from the optimization driver has been redirected to a file called `screen.output`. This file is very useful if a problem needs to be diagnosed with the execution. It is also good practice to include this file with help requests to `Fun3D-Support@lists.nasa.gov`.

At the completion of the function evaluation, the user should check for the desired/expected result. This is also a good opportunity to establish reasonable values for the number of time steps to run, the residual tolerance at which the solver should quit, and so forth. Such run control parameters may be set in `fun3d.nml` or via the `command_line.options` file.

Once the function evaluation procedure has been verified, the user should perform the same test for a sensitivity analysis by setting `what_to_do` in the `&design` namelist in `design.nml` to `2` and re-executing the optimization driver. Similar checks on convergence parameters, etc for the adjoint solver should be noted and applied to the relevant input files.

With successful function and gradient evaluations in hand, an actual optimization may be initiated. The value of `what_to_do` in the `&design` namelist in `design.nml` should be set to `3`, and the optimization driver can be executed as before. The user should closely monitor the screen output as the process proceeds, especially during the first several design cycles when input parameters may first cause problems. The largest changes in design variables often occur early on as well, which can cause issues with mesh movement operations, solver convergence, and other aspects. It is also very useful to occasionally filter the screen output for the current function value(s) reported at the completion of each flow solution in order to monitor overall progress:

```
grep "Current value of function" screen.output
```

When an optimization completes, the optimizer will report the reason for the termination to the screen, which may be a local minimum, or some problem encountered during the simulation. A summary of the optimization is provided by each optimization package in the file(s) noted in Table 14. The final set of design variables and function/constraint values determined by the optimizer will be available in `model.i/rubber.data`. To track the history of the optimization, a backup of all intermediate copies of `rubber.data` are stored in the directory `model.i/Rubberize/surface_history`. Intermediate copies of the surface grids developed during the design process are also stored in this location as `model.tec.j.sd1.iter`, where `j` is the body index, and `iter` is the design iteration. These files may be used to produce animations of the surface history if desired. Using the broad range of visualization output options for the solvers, the user has great freedom to produce customized animations of the design history (see section 9.6.7).

Table 14: Files containing design summary from various optimization packages.

Optimization Package	Summary File(s)
DOT/BIGDOT™	dot.output
KSOPT	ksopt.output
PORT	port.output
NPSOL™	npsol.printfile, npsol.summaryfile
SNOPT™	snopt.printfile, snopt.summaryfile

9.8.1 Filesystem Latency Problems

Design optimization using some cached file systems may experience problems due to the rapid execution of the various tools during the design process. In some cases, a file system lag may cause some processes to receive older/stale versions of files during execution. Specifying the `--sleep_delay [seconds]` command line option to the `opt_driver` executable will pause the optimization process with a sleep duration of `seconds` between subsequent code executions to allow the file system to perform correctly. On older systems, delays as large as 60 seconds are sometimes necessary; more recent systems seem to perform considerably better and values of 5-10 seconds are often sufficient.

9.9 Multiobjective Design

KSOPT, PORT, and SNOPT™ are the only packages currently supported for use with multi-objective design. Details on the usage for each package are provided below.

9.9.1 KSOPT

KSOPT is the only supported optimization package with explicit support for multiple objective functions. When using KSOPT, the user may designate any number of composite functions as objective functions in `rubber.data`.

9.9.2 PORT, SNOPT™

The FUN3D design driver offers a simple approach to scalarizing multiple user-specified objective functions for use with the PORT or SNOPT™ packages. If multiple composite functions are specified in `rubber.data`, the FUN3D design driver will combine them using the weight, target, and power values specified at the *composite* function level (i.e., the input values that appear just before the *component* function data is specified in `rubber.data`, see section 9.6.2). If N composite functions f are labeled as objective functions in `rubber.data`, the scalarized objective function F to be provided to the optimization procedure will take the form

$$F = \alpha_1(f_1 - f_1^*)^{p_1} + \alpha_2(f_2 - f_2^*)^{p_2} + ... + \alpha_N(f_N - f_N^*)^{p_N} \qquad (13)$$

where α_i, f_i^*, and p_i are the weight, target, and power values associated with each *composite* function in `rubber.data`.

9.10 Multipoint Design

The FUN3D design infrastructure offers several approaches to multipoint optimization. This refers to design problems where the user may wish to simultaneously optimize a configuration for operations at two different conditions — perhaps the beginning and end of a cruise segment for example, where the aircraft weight may be substantially different. The user may also wish to design for cruise and takeoff or landing (or all three). The various design points may be characterized by different flow conditions (i.e., speed, angle of attack, etc), or more generally, by the geometries (and therefore grids) at each point. For example, one design point may consist of a cruise geometry operating at Mach 0.8, while another design point may be a landing configuration operating at Mach 0.2 with a high-lift system deployed. For examples of FUN3D-based multipoint design in practice, see the studies in Refs. [30] and [39]. In these references, a tilt-rotor geometry has been optimized for a set of several blade collective settings as well as hover and forward flight conditions.

To perform a multipoint optimization, the user must request the desired number of design points when setting up the directory structure where the design will be performed (see section 9.4). The user must populate each of the `description.i` directories for each design point `i` just as in the single-point design context. The order of the design points does not matter. The value of `n_design_pts` in the `&design` namelist in `design.nml` should be set appropriately. Ultimately, FUN3D provides several ways to formulate the multipoint design problem. These approaches are outlined below.

In general, the optimizer will be seeking a unique set of design variables to simultaneously achieve goals at all of the design points. For this reason, a consistent set of design variables across all design points must be used. This applies to the global variables Mach number and angle of attack as well as any body-specific variables such as shape parameters. For example, if a set of 15 thickness variables is provided for a wing shape in cruise, other design points (again, perhaps a landing configuration as an example) must utilize the same set of 15 thickness variables. Moreover, the same subset of design variables must be active at each design point.

Multivalued Design Variables In some situations, the user may desire different optimal values of a design variable at different design points. For example, consider power minimization for a rotor in hover at two different weight conditions, where each of the two design points may have different minimum thrust coefficients posed as constraints. In addition to other design variables that may be present, the user may have a shape parameter controlling the blade collective setting (blade pitch). However, rather than constraining the optimal blade collective to a single unique value, the user may desire separate, optimal values for each design point. As another example, consider a configuration with an ability to actively morph its outer mold line. In this case, the user may wish to determine optimal values of the shape parameters that are unique to different design points.

To accommodate such multivalued design variables, the user may set the

"Active" attribute for individual shape design variables to −1 in `rubber.data` (see section 9.6.2). If this is done, it must be applied consistently for that same variable across all design points. For variables with this attribute, the FUN3D design driver will internally bookkeep separate values of the variable for each design point. This feature is currently only available for use with the SNOPT™ package.

9.10.1 Linear Combination of Objective Functions

The most straightforward approach to multipoint design is to linearly combine individual objective functions f_i from each of the N design points `i` into a single global objective function f_{mp}:

$$f_{mp} = \alpha_1 f_1 + \alpha_2 f_2 + \alpha_3 f_3 + ... + \alpha_N f_N \tag{14}$$

To perform the optimization in this fashion, a single composite objective function should be posed in each `description.i/rubber.data` file. Each of the α_i weighting coefficients must be specified in the `design_pt_weight(:)` array in the `&design` namelist in `design.nml`, in the corresponding order.

This form of multipoint design is supported by PORT, DOT/BIGDOT™, and SNOPT™. Note that PORT and SNOPT™ will also combine multiple objective functions within each design point as described in section 9.9 if desired. Explicit constraints can be posed at each design point when using DOT/BIGDOT™ or SNOPT™; such constraints are each treated individually.

9.10.2 Combination of Objective Functions using the Kreisselmeier-Steinhauser Function

Another alternative for performing multipoint design is the approach inherent in the KSOPT package. In this approach, all objective functions and constraints are combined using the Kreisselmeier-Steinhauser (KS) function. The user is referred to [40] for the details of this formulation. Here, the FUN3D design driver gathers any number of objective and constraint functions across all design points and provides them to KSOPT, which internally constructs its KS function for the actual optimization problem.

9.10.3 Single-Point Objective Function with Off-Design Constraint Functions

In this approach to multipoint design, a single objective function is provided to the optimizer, while all other functions are treated as explicit constraints. Here, the user should designate a single composite function across all of the `description.i/rubber.data` input files as an objective function. Any other composite functions at each design point should be designated as constraint functions. KSOPT, SNOPT™, and NPSOL™ support this form of multipoint optimization; SNOPT™ can also construct the final objective function by linearly combining multiple objective functions within the desired design point as described in section 9.9.

9.11 Optimization of Two-Dimensional Geometries

While the Fun3D flow solver supports a 2D mode of operation, this capability is not currently available from within the design infrastructure. Instead, the optimization must be performed as a pseudo-3D case. The user should provide a nominally two-dimensional grid, with a single layer of elements in the spanwise (`y`) direction. The mesh should consist of either prisms or hexahedra (or both), but should contain no pyramids or tetrahedra. Follow the same procedure used for 3D cases to extract the surface grid for parameterization. The surface should be parameterized just as for a 3D simulation; however, the parameterization should allow no spanwise asymmetries in the geometry to develop. When using MASSOUD or Bandaids, this is readily accomplished by linking the raw parameters with an equal weighting across the span into a single set of design variables that operate in a chordwise fashion. Note that the sidewalls should use `symmetry_y` boundary conditions so that only in-plane mesh deformation occurs during the optimization. The design may now be executed as usual, with the 2D nature of the problem enforced implicitly through the parameterization.

9.12 Rotor Optimization with Actuator Disk Modeling

Fun3D is capable of performing adjoint-based sensitivity analysis and optimization for rotors simulated with steady actuator disk models. The disk loading supported in the design context includes the constant thrust and blade-element-based loading described in section B.6. Since the actuator disk models only approximate the overall rotor influence on the configuration without relying on discrete grids for detailed blade geometry, the methods are relatively inexpensive and can rapidly achieve improved rotor designs. Several related output functions have been implemented and can be used as design objectives or constraints. The corresponding function keywords are provided in Table 15 and the component functional definition follows the form introduced in section 9.1.2. The disk power is defined as

$$f_P = \int_A \mathbf{F} \cdot \mathbf{u} \, dA \tag{15}$$

and is further decomposed into the normal and tangential components, f_{P_N}

Table 15: Objective/constraint component function keywords for actuator disk models.

Keyword	Function
`ad_power`	disk power
`ad_propeff`	propeller efficiency
`ad_x_force`, `ad_y_force`, `ad_z_force`	x-/y-/z-component force
`ad_xmoment`, `ad_ymoment`, `ad_zmoment`	x-/y-/z-component moment

and f_{P_T}, respectively:

$$f_{P_N} = \int_A F_n u_n dA \tag{16}$$

$$f_{P_T} = \int_A F_t u_t dA \tag{17}$$

where $\mathbf{F}$ is the vector of disk forces; $\mathbf{u}$ is the vector of flow velocities obtained by interpolating the CFD solution from the nearest element to the actuator disk source location; F_n and F_t are the normal and tangential disk force components; u_n and u_t are normal and tangential flow velocities; and A denotes integration over the disk mesh. For the constant-thrust loading, the tangential power component vanishes and `ad_power` is equal to the normal power. For the blade-element disk loading, `ad_power` accounts for the shaft (rotational) power, f_{P_T}. The dimensional shaft power is obtained by multiplying the nondimensional shaft power `ad_power` by the conversion factor $\rho^*_{ref}(a^*_{ref})^3$ for $L_{ref} = 1$, where ρ^*_{ref} is the dimensional freestream density; a^*_{ref} is the dimensional speed of sound; and L_{ref} is the physical length corresponding to one unit in the computational grid.

The output function, `ad_propeff`, is defined as the ratio of the normal power to tangential power components given by

$$f_\eta = \frac{f_{P_N}}{f_{P_T}}. \tag{18}$$

The x-, y-, and z-component forces, namely `ad_x_force`, `ad_y_force`, and `ad_z_force`, are the integrated aerodynamic forces acting on the aircraft and rotor disks, where the disk forces are normalized by $(1/2\rho_{ref}V^2_{ref}S_{ref})$ similarly to the normalization of lift and drag. The total rolling, pitching, and yaw moments contributed from the airframe and rotor disks are specified using the output functions, `ad_xmoment`, `ad_ymoment`, and `ad_zmoment`, respectively. The disk moments are normalized by $(1/2\rho_{ref}V^2_{ref}S_{ref}c_{ref})$ consistent with the normalization for aerodynamic moments. Here, c_{ref} represents the reference length used to nondimensionalize moments and is set internally based on `x_moment_length` or `y_moment_length` in the `&force_moment_integ_properties` namelist. These aerodynamic force and moment functions are often used as constraints to ensure force equilibrium on an optimized configuration. Following the composite function definition in Eq. (1), the user can also set multiple component functions for individual rotors identified by the actuator disk index in `rotor.input` and apply desired weighting coefficients to scale component functions if needed. For example, if two actuator disks with the supported disk loading are specified in `rotor.input` and the disk power from `rotor 1` is desired to be scaled by one order of magnitude greater than `rotor 2`, the component functions in `rubber.data` can be specified as,

```
Components of function   1: boundary id (0=all)/name/value/weight/target/power
   1 ad_power  0.100000000000E+01 0.100000000000E+02  0.000000000000E+00  0.200000000000E+01
   2 ad_power  0.100000000000E+01 0.100000000000E+01  0.000000000000E+00  0.200000000000E+01
```

where the first integer for each component function is the actuator disk index, determined by the order of rotor definitions in `rotor.input` (section B.6). In this example, the weighting coefficient is 10.0 for `rotor 1` and 1.0 for `rotor 2`. If the actuator disks are desired to be equally scaled, a single component function can be defined with `boundary id` set to `0`.

The adjoint solver in FUN3D can be used to compute discretely-consistent sensitivity derivatives for the output functions given in Table 15 with respect to a set of actuator disk design variables. These variables include rotor thrust, disk orientation, tip speed ratio, blade radius, disk location, the collective pitch control angle, and twist and chord length at specified radial stations. Enabling the actuator disk design variables requires the use of an additional design control file, `actuator_disk.data`. This file stores the actuator disk design variables and sensitivity derivatives of the output functions. The FUN3D optimization driver reads and updates this file throughout the design procedure. A sample is provided in the `Adjoint` directory of the source code distribution and is shown below for discussion purposes.

```
################################################################################
########################### Design Variable Information ########################
################################################################################
Number of rotors
    2
Number of variables for rotor 1
    14
Index Active         Value                Lower Bound            Upper Bound
    1     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    2     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    3     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    4     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    5     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    6     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    7     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    8     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    9     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
   10     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
   11     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
   12     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
   13     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
   14     1    0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
```

```
Number of variables for rotor 2
    14
Index Active         Value               Lower Bound           Upper Bound
    1    1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    2    1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    3    1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    4    1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    5    1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    6    1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    7    1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    8    1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    9    1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    10   1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    11   1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    12   1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    13   1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
    14   1   0.100000000000000E+01  0.000000000000000E+00  0.500000000000000E+01
################################################################################
################################ Function Information ############################
################################################################################
Number of composite functions for design problem statement
    2
################################################################################
Current derivatives of function wrt design variables of rotor  1
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
Current derivatives of function wrt design variables of rotor  2
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
################################################################################
Current derivatives of function wrt design variables of rotor  1
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000000
```

```
     0.000000000000000
     0.000000000000000
     0.000000000000000
     0.000000000000000
 Current derivatives of function wrt design variables of rotor  2
     0.000000000000000
     0.000000000000000
     0.000000000000000
     0.000000000000000
     0.000000000000000
     0.000000000000000
     0.000000000000000
     0.000000000000000
     0.000000000000000
     0.000000000000000
     0.000000000000000
     0.000000000000000
     0.000000000000000
     0.000000000000000
```

The first block defines the number of rotors and the design variables for individual rotors. The design variables must be ordered as shown in Table 16, where n denotes the number of radial stations specified in the input file, `bladegeom_rotor.dat` (c.f., section B.6.10). For example, if two radial stations have been specified for the blade-element based actuator disk model, a total of 14 design variables should be specified for each rotor in the design control file, as shown in the example. The second column specifies the status of the design variables. If the status is set to `1`, the variable will be allowed to change during the design. If the status is set to `0`, this variable will be held constant at the value specified in the third column. (Also see the discussion on maintaining asymmetric values via this status input at the end of this section.) For constant thrust disk loading, the twist and chord design variables must be declared inactive. Similarly, for the blade-element disk loading, the thrust coefficient design variable must be declared inactive. Note that a row of data must be provided for every actuator disk design variable in this control file, whether active or not. The third column provides the values of design variables corresponding to the baseline configuration. These values must be specified carefully, as they will be copied to the primary `rotor.input` file at the beginning of an optimization. The fourth and fifth columns contain the respective upper and lower bounds for each individual design variable of each rotor.

The second block in the `actuator_disk.data` file specifies the number of composite functions, which must be consistent with the number of composite functions specified in `rubber.data`. The subsequent entries are sensitivity derivatives of each composite function with respect to all actuator disk design variables, whether active or not. The initial values are set to zero and these entries are updated automatically during the design process.

A procedure similar to body grouping as discussed in section 9.6.4 can be applied to actuator disks. When multiple rotors are grouped, the sensitivity derivatives with respect to the design variables are combined as composite

Table 16: Actuator disk design variables (n represents the number of radial stations at which twist and chord length design variables are specified).

Index	Design variable
1	thrust coefficient, `ThrustCoff`
2	the first Euler angle describing the disk orientation, `phi1`
3	the second Euler angle describing the disk orientation, `phi2`
4	the third Euler angle describing the disk orientation, `phi3`
5	tip speed ratio, `Vt_Ratio`
6	blade radius, `TipRadius`
7	x-coordinate of the hub center of rotation, in grid units, `X0_rotor`
8	y-coordinate of the hub center of rotation, in grid units, `Y0_rotor`
9	z-coordinate of the hub center of rotation, in grid units, `Z0_rotor`
10	collective pitch control angle, `theta0`
11	twist at radial station 1, in degrees
...	...
10+n	twist at radial station n, in degrees
11+n	chord at radial station 1, in grid units
...	...
10+2n	chord at radial station n, in grid units

derivatives from all rotors defined within one group. Consequently, the changes in the design variables are applied uniformly to the grouped rotors. This grouping capability is particularly useful when a set of rotors need to maintain the same geometric specifications during an optimization. The grouping information is specified in the input file, `actuator_disk_grouping.data`, and a sample is shown below for demonstration.

```
Number of groups to create
1
Number of rotors in group, list of rotors
2
1 2
```

Note that for rotors symmetrically installed on the aircraft, certain actuator disk design variables such as the tip speed ratio and y-coordinate of the hub center of rotation may be present with the same magnitude but opposite sign. To allow the FUN3D optimization driver to identify these asymmetric design variables and enforce valid changes, set the status of such design variables to 2 in `actuator_disk.data` (the second column in the first block) so that asymmetric values can be maintained.

9.13 Using a Different Optimization Package

In a CFD-based design context, the term "function" implies an evaluation of the geometric parameterization, mesh movement (both surface and volume), a flow solution, and an evaluation of the output function/constraint for a given set of design variables. The file manipulations and solver operations necessary to achieve this are not trivial. For users interested in using the tools as “black boxes” providing function data for an optimization package, a wrapper has been provided in the `LibF90` directory of the distribution named `analysis.f90`. This module contains a subroutine called `perform_analysis()` which performs the extensive set of tasks involved with producing the final desired function output.

To obtain sensitivities, the FUN3D package relies on a discrete adjoint formulation. As with function evaluations, the low-level operations required to perform an adjoint-based sensitivity analysis are numerous. A wrapper routine called `perform_sensitivity_analysis()` in the `LibF90/sensitivity.f90` module will perform an adjoint solution for the flow field, an adjoint solution for the mesh movement scheme, an evaluation of the linearized geometric parameterization, and finally produce the desired sensitivity derivatives.

The FUN3D design driver uses the wrappers `perform_analysis()` and `perform_sensitivity_analysis()` to greatly simplify function and gradient evaluations when connecting to off-the-shelf optimization packages. If the user wishes to implement a new optimization strategy, it is highly recommended that these wrappers be used in a similar fashion. A review of the existing modules in the `Design` directory of the FUN3D source code distribution, which implement the currently available optimization interfaces, is also strongly suggested. Users may contact `Fun3D-Support@lists.nasa.gov` for further guidance in leveraging FUN3D’s capabilities from within their own existing design framework.

9.14 Implementing New Cost Functions/Constraints

Implementation of new cost functions or constraints is not a trivial undertaking and requires extensive modification of FUN3D source code. Experience in Fortran 2003, unstructured-grid discretizations, development in a domain-decomposed/distributed-memory environment, and general CFD methods are essential. Routines to evaluate the proposed function and linearizations of the function with respect to both the flow field variables and grid are ultimately required. The complex-variable form of FUN3D (see section 9.15) is invaluable in verifying the accuracy of these linearizations. It is highly recommended that the user contact `Fun3D-Support@lists.nasa.gov` for guidance prior to attempting the implementation of a new cost function or constraint.

9.15 Forward Mode Differentiation Using Complex Variables

The reverse, or adjoint, mode of differentiation is primarily used for design with FUN3D. A forward mode of differentiation is also provided based on the use of complex variables [41–43]. This capability is useful for design problems containing few design variables and many cost functions or constraints. To generate and build a complex-variable FUN3D executable, see section A.10.

The complex-valued flow solver reads the usual real-valued grid files and is set up to compute derivatives of every output variable with respect to Mach number, angle of attack, shape parameters, noninertial rotation rates, or the x, y, or z coordinate of a single grid point (others are trivial to add). This choice is controlled by the file `perturb.input`. A template for this file is provided in the `[install-prefix]/share/fun3d/Flow` directory[2] and is shown here:

```
     PERTURB   EPSILON GRIDPOINT
           2    1.e-50       666

 0 = No perturbation
 1 = Mach number
 2 = Alpha
 3 = Shape
 4 = x-rotation rate
 5 = y-rotation rate
 6 = z-rotation rate
 7 = Grid point x
 8 = Grid point y
 9 = Grid point z
 10 = Yaw
 11 = error transport (truncation error)
 12 = RCS jet plenum pressure, p0
 100+ = add an imaginary source term to equation
          PERTURB-100 of node GRIDPOINT
          (to verify the adjoint lambda value)
```

The value of `PERTURB` specifies the variable for which sensitivities will be taken with respect to. The valid integer values are as shown above. The input `EPSILON` specifies the magnitude of the imaginary perturbation to be applied. The recommended value is `1.e-50`. If the value of `PERTURB` is greater than six, the value of `GRIDPOINT` specifies the grid point index to perturb. The remaining lines in `perturb.input` are not read; they are simply reminders of the valid inputs just described. The complex-valued flow solver may then be executed in a manner similar to the real-valued flow solver:

[2]See section A.3 for `[install-prefix]`.

```
mpirun ./complex_nodet_mpi
```

To compute derivatives with respect to a shape parameterization variable, the sensitivities of the parameterization must first be evaluated in the directory `model.i/Rubberize` using the relevant parameterization software. The value of `PERTURB` should be set to `3` in `perturb.input`.The complex-valued flow solver can then be executed in the following fashion:

```
mpirun ./complex_nodet_mpi --dv_index [body] [dv] --snap_grid
```

Here, the values of `body` and `dv` specify the body and design variable index in `rubber.data` to which to apply the imaginary perturbation. The `--snap_grid` argument forces the flow solver to propagate the surface sensitivities into the volume mesh using FUN3D's elasticity-based deformation mechanics.

At the completion of the complex-valued flow solve, outputs will contain both real and imaginary parts. The imaginary part represents the sensitivity of that output with respect to the perturbation variable that was specified in `perturb.input`.

10 Accelerating Solutions Using Graphics Processing Units

10.1 GPU Quick Start Guide

This quick start guide assumes the user is running a configuration with homogeneous compute nodes and default numbering of processing elements/devices (see section 10.7). It also assumes a standard MPI launcher such as `mpirun` or `mpiexec` and not a batch launcher like `aprun` (Cray® systems) or `jsrun` (some IBM® systems). It also assumes each compute node has at least 4 physical CPU cores per GPU. If not, lower the multiplier below appropriately.

1. Determine if your simulation options are supported (see section 10.4).
2. Determine if your hardware is supported (see sections 10.3 and 10.6).
3. Determine the number of MPI ranks needed by placing one on each desired GPU.
4. If writing visualization output more often than once per simulation, add `cuda_start_mps=.true.` to the `&gpu_support` namelist and multiply the number of MPI ranks by 4 (see section 10.10).
5. Set `use_cuda=.true.` in the `&gpu_support` namelist and `mixed=.true.` in the `&code_run_control` namelist.
6. Launch FUN3D with the computed number of MPI ranks.

NAS FUN3D users: For a guide to using the NAS prebuilt FUN3D modules on GPUs, please enter

```
module help /path/to/FUN3D/module
```

at the shell prompt. This will direct you to a file containing the current instructions specific to NAS GPU nodes.

10.2 Background

As of FUN3D v13.7, a capability to accelerate flow solutions using NVIDIA® Graphics Processing Units (GPUs) is provided and will be included in a FUN3D installation if an appropriate CUDA installation is specified as described in section A.9. Throughout this documentation, the terms CPU and host are used interchangeably, as are GPU and device.

10.3 Requirements

Use of the FUN3D GPU capability requires access to NVIDIA® Tesla P100, V100, or A100 GPU hardware. The host CPU may be an Intel® or AMD® x86_64 processor, an IBM® POWER® processor, or a compatible ARM processor. The default provided library supports x86_64 processors; please contact `Fun3D-Support@lists.nasa.gov` for an IBM® POWER® or ARM processor supported library. Multiple GPUs per host and execution across multiple GPU-accelerated nodes are supported. An NVIDIA® NVLink® interconnect may provide slightly better performance but is not essential. Version 10.2 or newer of the NVIDIA® CUDA Toolkit must be installed on the target system and specified according to section A.9. The toolkit and driver follow a set of compatibility requirements listed here.

The FUN3D development team routinely builds binaries for GPU execution using recent versions of the Intel®, IBM® XL, and GNU compilers, although any compiler with full Fortran/C interoperability support should be sufficient. If using GFortran, version 5.0 or newer is required. FUN3D execution across multiple GPUs has been successfully tested using OpenMPI, Intel MPI, HPE/SGI MPT, IBM Spectrum MPI, MVAPICH, and Cray MPI.

To build FUN3D with GPU support enabled, please see section A.9.

With regards to memory, the requirements primarily vary based on the gas model. For perfect gas RANS simulations, approximately 3-3.5 million grid points can fit in 16 GB of GPU memory. For chemically reacting flows, the memory requirement scales roughly quadratically with the number of equations. For 5 species RANS simulations, approximately 1-1.5 million grid points can fit in 16 GB of GPU memory. It is recommended to utilize at least 30% of the available memory on each GPU, regardless of how many MPI ranks are assigned to the GPU (see section 10.5). This should provide enough computational work to realize optimal efficiency for the GPU, although smaller workloads will still outperform CPU-based simulations of similar size. Memory at initialization can be printed out by setting `print_crude_dev_mem=.true.` in the `&gpu_support` namelist. Some additional memory is allocated during the first time step; it is recommended to utilize at most 95% of the available GPU memory at initialization.

10.4 Supported Options

Only a subset of FUN3D's overall capabilities is available when running on GPUs. If an option has not been implemented, your run will not work correctly. All GPU-based simulations on tetrahedral grids must use the command line option `--mixed`, which may also be enabled using the `mixed` option in the `&code_run_control` namelist. Grids with any other element types present will automatically use this option.

In general, all flow paths with RANS with options for DES have been implemented at this time. Laminar and inviscid flows may also be computed. Internal checks attempt to catch invalid user inputs but are by no means exhaustive. Solution visualization options are valid; however, most such options execute on the CPU (see section 10.5).

All of the usual output files will be generated at the requested frequency (for example, `project.forces`, `project_hist.dat`, `project.flow`, visualization data, etc).

10.4.1 Perfect Gas Path Options

Supported options currently include:

- Fluxes: Roe, LDFSS, Low-dissipation Roe, HLLC, DLDFSS, van Leer, HLLE++
- Limiters: Barth, hvanalbada, hvanleer, hvenkat, hminmod
- Jacobians: van Leer, LDFSS
- Turbulence modeling: SA/SA-neg models with DES, DDES, MDDES, Dacles-Mariani, and RC options, SST, SST-V
- QCR2000
- Boundary conditions: 3000, 4000 (including adiabatic wall, constant temperature wall, specified surface velocities), 5000, 5026, 5050, 5051, 5052, 6661/2/3, 7011, 7012, 7100, 7201
- Time integration: Backward Euler for steady flows, all unsteady BDF schemes, local time-stepping
- Time-averaging statistics
- Specified rigid grid motion
- Aeroelastic analysis using internal modal solver with modal mesh deformation
- Asynchronous native volume and surface pressure outputs
- Actuator surface models for rotor/propeller modeling: Actuator disk models (`rotor_type=1` in `rotor.input`), including load types 1, 2, 3, and 7 (constant/linear pressure jumps, blade-element loading with/without radial variations). Note that when executing such models on the GPU, the preprocessing necessary to associate source locations with CFD grid points is performed on the CPU; MPS can be very helpful in this context (see section 10.10).

10.4.2 Incompressible Gas Path Options

Supported options currently include:

- Fluxes: Roe
- Jacobians: Roe, dfduc3
- Turbulence modeling: SA/SA-neg models with DES, DDES, MDDES, Dacles-Mariani, and RC options, SST, SST-V
- Boundary conditions: 3000, 4000, 5000, 5005, 5010, 5020, 5025, 5026, 5050, 5051, 6661/2/3, 7010
- Time integration: Backward Euler for steady flows, all unsteady BDF schemes, local time-stepping
- Time-averaging statistics
- Specified rigid grid motion
- Asynchronous native volume and surface pressure outputs
- Actuator surface models for rotor/propeller modeling: Actuator disk models (`rotor_type=1` in `rotor.input`), including load types 1, 2, 3, and 7 (constant/linear pressure jumps, blade-element loading with/without radial variations). Note that when executing such models on the GPU, the preprocessing necessary to associate source locations with CFD grid points is performed on the CPU; MPS can be very helpful in this context (see section 10.10).

10.4.3 Generic Gas Path Options

The generic gas path solves the thermochemical nonequilibrium flow equations with various turbulence models. The current library supports up to 16 species, 2 temperatures, and 2 turbulence equations. Larger number of species are supported if desired (please contact `Fun3D-Support@lists.nasa.gov`), although require significant amount of resources due to memory constraints. Supported options currently include:

- Gas Models: perfect gases, thermochemical nonequilibrium gases
- Temperature Models: One and two temperature models
- Fluxes: Roe, stvd, HLLE++
- Limiters: hvanalbada, hvanleer, hvenkat, hminmod, minmod gg (for stvd)
- Coupled Turbulence modeling: SA Catris with DES option (`des` option in the `&spalart` namelist), KW-SST
- Experimental Decoupled Turbulence modeling (see `use_decoupled_turbulence_`

`gen`): SA/SA-neg models with DES, DDES, MDDES, Dacles-Mariani, and RC options, SST, SST-V

- Boundary conditions: 3000, 4000 (including adiabatic wall, constant temperature wall, radiative equilibrium wall, specified surface velocities), 5000, 5026, 5050, 5051, 6661/2/3, 7021, 7100
- Catalytic Wall Models: All but equilibrium-catalytic
- Time integration: Backward Euler for steady flows, all unsteady BDF schemes, local time-stepping
- Time-averaging statistics
- Specified rigid grid motion
- Aeroelastic analysis using internal modal solver with modal mesh deformation
- Asynchronous native volume outputs

10.5 GPU Execution Strategy and Concerns

Due to current bandwidth limitations between the CPU and GPU, the GPU implementation of Fun3D is designed to execute entirely on the GPU with as little data motion to/from the host as possible. In this approach, every relevant kernel within the main time-stepping loop has been ported for GPU execution. The use of some CPU kernels (such as visualization output) is still valid, but requires host/device data motion and may execute slowly relative to the GPU kernels.

In the most basic approach, the user requests one MPI rank per GPU. For example, on a dual-socket processor with 40 physical cores, a CPU-based simulation will typically use 40 MPI ranks, with one rank on each physical core. However, for a GPU-based simulation where 4 GPUs reside locally on the node alongside the CPU, the user would request just four MPI ranks to be placed on the node. Ideally, the first two MPI ranks would be placed on two physical cores on one CPU socket and the next two on the second CPU socket (consult your MPI or system documentation on how to pin MPI ranks to individual cores). This assumes two GPUs are associated with each socket, which is the most common placement. During execution, each of the four MPI ranks will shepherd an individual GPU *and the other 36 CPU cores will remain idle.*

GPU-based simulations benefit greatly from a minimal number of interruptions to perform CPU-based operations such as visualization output. If such options are required, the user is highly encouraged to use multiple MPI ranks per GPU (see section 10.10). This approach enables the use of more CPU cores to perform these relatively expensive auxiliary kernels. (An alternative implementation for highly efficient GPU-based volume output is available. Here, data is asynchronously moved to the CPU and then to disk behind the GPU-based PDE solution kernels to hide such overhead. This approach has been successfully used to store as much as 200 terabytes of data to disk with no degradation in performance. Contact `Fun3D-Support@lists.nasa.gov` for further information on using this approach.)

Since grid preprocessing operations at startup are also entirely CPU-based, this phase may also appear relatively inefficient when running a single MPI rank per GPU for relatively brief simulations (e.g., steady-state cases that converge very quickly or unsteady cases of very brief duration). This is another aspect of execution that will benefit from using multiple MPI ranks per GPU (see section 10.10). If multiple cases will be run using the same grid, the user may also greatly benefit from storing the grid partitions, and optionally the distance function, to disk for such cases. See the `&partitioning` namelist inputs `write_grid_partitions` and `read_grid_partitions` (section B.4.48) as well as the `&special_parameters` namelist inputs `write_slen` and `read_slen` (section B.4.47). These options enable the user to store the partitioned mesh and distance function to disk, avoiding the need to perform these steps for subsequent simulations. Note that there is a similar option available for storing volume mode shapes to disk when running modal aeroelastic solutions (see section B.5.7).

10.5.1 Running a Case

To perform a GPU-based simulation, add the option `use_cuda=.true.` to the `&gpu_support` namelist (section B.4.52). You may also use the command line option `--use_cuda`. Launch FUN3D with a number of MPI ranks which is a multiple of the total number of GPUs. If the multiple is greater than 1, please see section 10.10 for additional guidance.

10.6 Device Information

At a Linux shell prompt on any compute node, the command `nvidia-smi` will provide information about the devices on the node. Some useful commands follow:

- `nvidia-smi topo -m` – Display the node topology. This will show a matrix of information containing the location of each device and NIC/HCA, as well as the links data must traverse between each device.
- `nvidia-smi -L` – Print the device UUIDs.
- `nvidia-smi --query-gpu=index,gpu_name,gpu_bus_id,uuid --format=csv` – Print the GPU device number, name, bus ID, and UUID for each GPU.

- `nvidia-smi -q` – Query all attributes for all GPUs.

For more `nvidia-smi` information, see here.

The command `cat /proc/driver/nvidia/gpus/gpu_bus_id/information` contains some useful device information. `gpu_bus_id` is replaced with the GPU bus id, which can be obtained from the `nvidia-smi` useful command list.

10.7 Device Selection

By default, FUN3D will automatically determine, based on the number of GPUs and MPI ranks on each node, how to assign ranks to GPUs. This can be disabled by setting `use_auto_device_select=.false.` in the `&gpu_support` namelist. Automatic device selection assumes the following.

- A maximum of one MPI rank is pinned to each physical processing element
- MPI ranks are spread evenly across sockets and numbered sequentially (e.g., ranks 0 and 1 reside on cores 0 and 1 on socket 0 and ranks 2 and 3 reside on cores $\frac{ncores}{2}$ and $\frac{ncores}{2}+1$ on socket 1)
- GPUs are spread evenly across sockets and numbered sequentially (e.g., GPUs 0 and 1 reside on socket 0 and GPUs 2 and 3 reside on socket 1)
- You are launching MPI ranks from a traditional MPI launcher such as `mpirun` or `mpiexec`
- You are not running on a batch system which uses a launcher similar to the IBM® `jsrun` utility. If you are, consult their documentation for how to select devices and use CUDA MPS if appropriate (see section 10.10)

If any of these assumptions is violated, performance may suffer. Automatic device selection also accepts a comma-separated list of device numbers by way of `device_list` in the `&gpu_support` namelist. For example, a reverse ordering of 4 GPUs would set `device_list='3,2,1,0'`. This would place the first $\frac{ranks\_per\_node}{gpus\_per\_node}$ ranks on GPU 3.

Another option is to set `use_setdevice=.true.` and `gpus_per_node` in the `&gpu_support` namelist. This will supersede auto-selection and assign a rank to GPU ($rank$ % $gpus\_per\_node$), where % is the `modulo` operator. This is a legacy option and does not support running multiple MPI ranks per GPU.

If both `use_auto_device_select=.false.` and `use_setdevice=.false.`, FUN3D does nothing, which means each rank will choose the first available device, which is generally the GPU with device ordinal 0, unless the environment variable `CUDA_VISIBLE_DEVICES` is set, in which case the first GPU listed is selected.

For help with nonstandard or advanced system configurations, please contact `Fun3D-Support@lists.nasa.gov`.

10.8 Expected Performance

Performance is variable depending on the physics, numerical methods, grid, and partitioning and thus it is difficult to generate exact comparison numbers. Overall, performance scales with the hardware memory bandwidth. GPU strong scaling is also better than CPU strong scaling using a pure MPI approach due to the GPU using fewer ranks than the CPU. This being said, the values discussed below are what should be expected.

For perfect gas simulations, a single Tesla V100 GPU should provide a speedup of 4.5-5.0× over a dual-socket Xeon Skylake containing a total of 40 physical cores. Equivalently, a Tesla V100 GPU should provide the throughput of approximately 180-200 Xeon Skylake cores. Similarly, nodes containing 4 or 8 Tesla V100 GPUs should provide the performance of approximately 700-800 or 1400-1600 Skylake cores, respectively. For generic gas simulations, a single Tesla V100 GPU should provide a speedup of 8.0-10.0× over a dual-socket Xeon Skylake containing a total of 40 physical cores (see [44] for more details). Currently, several generic gas CPU kernels are not optimal and algorithmic improvements are being investigated. It is recommended to use the command line option `--time_timestep_loop` to monitor performance for both CPU- and GPU-based simulations to ensure the expected performance is being observed. Performance for the Tesla P100 should be approximately half that of the Tesla V100, while the Tesla A100 should be approximately 60-70% faster than that of the Tesla V100.

To verify your execution, you may wish to run your job once on CPUs and then again on GPUs to compare outputs. In general, this should be done using the same number of MPI ranks for each of the two executions since the order in which FUN3D's linear algebra is performed depends directly on the partitioning of the problem. Note these differences only appear in the linear system whose purpose is to provide approximate updates to the outer nonlinear problem —any partitioning of the linear system is a valid one and will yield consistent nonlinear updates that converge to the same steady-state solution. Alternatively, if the linear system is solved to machine precision (say, using a very large number of relaxation sweeps), the effects of partitioning can be effectively eliminated and the nonlinear systems should converge identically.

10.9 Using CUDA-Aware MPI and GPUDirect™

The term CUDA-aware MPI describes the ability of an MPI implementation to accept device memory buffers as arguments to MPI functions. Most major MPI implementations now support this feature and FUN3D can accommodate both host-based and CUDA-aware MPI calls. Use of device buffers is enabled by setting `use_cuda_mpi=.true.` in the `&gpu_support` namelist or through the command line option `--use_cuda_mpi`. Ideally, send and receive data stored in GPU buffers would be exchanged directly, avoiding the need to stage data through host memory. NVIDIA® GPUDirect™ RDMA provides a data path between a GPU and another device (GPU, NIC, etc) over PCI Express (or NVLink®). The terms CUDA-aware MPI and GPUDirect™ are often used interchangeably to indicate the use of device buffers in MPI functions and that also occurs in this document.

In practice, larger messages than those most commonly used in FUN3D benefit most from GPUDirect™ and thus the default, host-based MPI performs best. Under optimal conditions, however, GPUDirect™ may prevail. Currently, it is advised to install the latest Mellanox OFED, OpenMPI, and UCX stack with the addition of the NVIDIA® gdrcopy library to maximize GPUDirect™ performance. Correctly configuring this stack is nontrivial and requires root privileges. OpenMPI should be run using the rendezvous protocol by way of `--mca btl_sm_eager_limit 1`.

Different MPI implementations may require a special setting to activate CUDA-aware MPI and GPUDirect™. Table 17 summarizes the settings for common MPI implementations. Note that the implementation must be compiled with CUDA support for these features to function.

Table 17: MPI settings to enable CUDA-aware MPI/GPUDirect™.

MPI	Setting
IBM® Platform MPI	`PMPI_GPU_AWARE=1`
IBM® Spectrum MPI	`mpirun -gpu`
Intel® MPI	Not likely supported
MPICH / Cray® MPT	`MPICH_RDMA_ENABLED_CUDA=1`
MVAPICH2	`MV2_USE_CUDA=1`
OpenMPI	Default enabled, but see here
SGI/HPE MPT	`MPI_USE_CUDA=true`

In general, a user will need to experiment with an MPI implementation when using GPUDirect™ to find the optimal configuration. For example, SGI/HPE MPT prefers disabling the rendezvous protocol ("single copy") by setting the environment variable `MPI_DEFAULT_SINGLE_COPY_OFF=true`, but OpenMPI appears to prefer the opposite.

GPUDirect™ remains an active area of vendor and FUN3D research. For guidance from NVIDIA® please see here and here.

10.10 Running Multiple MPI Ranks per GPU

The NVIDIA® Multi-Process Service, or MPS, is a service provided by NVIDIA® which allows multiple MPI ranks to share a GPU efficiently. FUN3D CPU-based functionalities such as preprocessing and visualization are generally serial processes within an MPI rank. Therefore, if only a single MPI rank is used per GPU, these kernels may be slow compared to conventional CPU executions with 1 MPI rank per CPU core. MPS can help mitigate this. MPS may decrease or increase the speed of GPU execution by a small amount; this behavior is case-dependent. However, if the duration of the run is short enough, the improved efficiency of the CPU-based kernels should outweigh any GPU performance loss.

When using MPS, FUN3D should always be run with the number of MPI ranks which is a multiple of the number of GPUs. In general, 4 MPI ranks per GPU should work well. The user is encouraged to experiment with this value to find the optimal configuration for a particular case, especially if the case will be run multiple times (e.g., long temporal durations, parameter studies for database generation, etc). If help with a nonstandard configuration (e.g., a number of ranks which is not a multiple of the number of GPUs) is required, please contact `Fun3D-Support@lists.nasa.gov`.

To use MPS, the `nvidia-cuda-mps-control` daemon must be running. It can be started by the shell command `nvidia-cuda-mps-control -d`, assuming `nvidia-cuda-mps-control` is in `PATH`. FUN3D can also start the daemon by specifying `cuda_start_mps=.true.` in the `&gpu_support` namelist. When the daemon is running properly, creating a GPU context will spawn a process named `nvidia-cuda-mps-server` on each GPU (visible by running the command `nvidia-smi` on the node where FUN3D is executing; see section 10.6). Upon completion, FUN3D will kill the MPS daemon if and only if the same instance of FUN3D initially started it. Note that GPUs must be in the `Default` compute mode, as indicated by `Compute M.` on the output of `nvidia-smi`.

By default, FUN3D uses the current environment value of `CUDA_VISIBLE_DEVICES`, which may limit the MPS server to a subset of the GPUs. To ignore this variable, either unset it before launching FUN3D or set `ignore_cuda_visible_devices=.true.` in the `&gpu_support` namelist.

One important consideration for the use of MPS is the value of `CUDA_MPS_PIPE_DIRECTORY`. This environment variable sets the directory through which processes communicate with the CUDA MPS server. It **must** be the same value when the MPS daemon is started and for each process that communicates with the daemon. This directory must be writable and unique to each node. Generally, `/tmp` suffices. If `CUDA_MPS_PIPE_DIRECTORY` is unset, this directory defaults to `/tmp/nvidia-mps`. By default, FUN3D takes the current value of `CUDA_MPS_PIPE_DIRECTORY`. This is equivalent to setting `cuda_mps_pipedir='env'` in the `&gpu_support` namelist. The recommended approach is to set `cuda_mps_pipedir=''` in the `&gpu_support` namelist, which unsets the value of `CUDA_MPS_PIPE_DIRECTORY`.

Of less importance is the value of `CUDA_MPS_LOG_DIRECTORY`. This controls the location of the output files `control.log` and `server.log`, which are used by the MPS daemon/server. This defaults to the current environment variable, as with `CUDA_MPS_PIPE_DIRECTORY`. If unset, the location defaults to `/var/log/nvidia-mps`.

As a rule of thumb, a steady-state perfect gas turbulent-flow simulation will take approximately 0.15 seconds per time step (per subiteration for unsteady flow) when one million grid points reside on each V100 GPU. This time generally scales linearly with the size of the mesh. Preprocessing for such a case requires approximately 9 seconds on a 40-core Skylake Gold 6148 for 40 MPI ranks and 20 seconds for 4 MPI ranks. However, preprocessing time does not necessarily scale linearly and may take much longer using 4 ranks for large meshes. Assuming MPS increases the cost of each subiteration to roughly 0.165 seconds, these figures may be used to compute when it might be useful to use MPS. If visualization output is active (particularly advanced options such as sampling) and is required more often than once per simulation, use of MPS is highly encouraged.

For more details, see the NVIDIA® MPS documentation.

10.11 Implications of Asynchronous Execution

Given the same compiled executable and mesh partitioning over MPI ranks, a simulation using Fun3D on CPUs is deterministic; it will perform the same order of operations every time. However, one of the underlying reasons GPU hardware is computationally efficient is its ability to dynamically schedule its work. For the implementation in Fun3D, this implies that the order of operations for successive GPU runs will be different. For this reason, outputs will be slightly different between GPU runs. Since Fun3D uses single-precision arithmetic during its linear algebra (and recent work has even leveraged FP16, or half-precision, for these operations on the GPU [45]), the relative difference in solution norms will differ between successive runs by typically no more than 1.0×10^{-6}. It is important to note, however, that when driven to a steady-state, this relative difference will be no larger than 1.0×10^{-13} since Fun3D solves the discrete equations in correction form. In other words, all paths to a fully-converged steady-state solution are consistent to double-precision accuracy (barring the unusual situation of nonunique solutions).

There are instances in which results can be especially sensitive to this change in order of operations, such as flows requiring flux limiters and realizability constraints. It is important to remember, however, that slight differences in results due to order of operations is not a GPU-specific problem. Since the order of CPU execution is typically fixed, one does not usually observe these fluctuations in practice. However, note that operation order in the CPU code can be arbitrarily permuted to show the same behavior as observed for GPU execution.

10.12 Troubleshooting

FUN3D checks for CUDA errors before and after most GPU operations. If you encounter a CUDA error you cannot resolve, please contact `Fun3D-Support@lists.nasa.gov` and include the full screen output including `stderr`. After encountering a CUDA error, a FUN3D MPI rank will print a stack trace and call `MPI_Abort`. Please be sure to include this trace in any support request. If there is no trace, please inform `Fun3D-Support@lists.nasa.gov` as well, as this too indicates a problem. Due to the asynchronous nature of the host/device relationship, the point at which FUN3D encounters an error may not be accurate unless the environment variable `CUDA_LAUNCH_BLOCKING=1` is set prior to execution. **Note that this option will considerably hamper execution speed**.

Common CUDA errors in FUN3D can be grouped into several classes as shown in Table 18. The groupings are only designed to aid in understanding.

Table 18: Common CUDA errors in FUN3D.

Error type	Error string reported by FUN3D
Out of memory	"out of memory"
Host problem	"invalid argument" "invalid configuration argument" "too many resources requested for launch" "invalid configuration argument"
Device problem	"an illegal memory access was encountered" "unspecified launch failure in prior launch" "unspecified launch failure"
System/environment problem	"CUDA driver version is insufficient for CUDA runtime version" "all CUDA-capable devices are busy or unavailable" "no CUDA-capable device is detected" "invalid device ordinal"
Other	"unknown error"

- **Out of memory** – See section 10.2. To aid in diagnosing and preventing this error, you may set `print_crude_dev_mem=.true.` in the `&gpu_support` namelist.
- **Host problem** – These errors may indicate memory corruption on the host. Ensure that the MPI environment is sound and that `PATH` and `LD_LIBRARY_PATH` are as expected. Ensure no other processes are running on the GPUs. You may try setting `use_cuda_pin=.false.` in the `&gpu_support` namelist.
- **Device problem** – These errors may indicate memory corruption on the device or a legitimate bug. Ensure all the steps taken in **Host**

problem are followed. If MPS is used, ensure it is running as indicated in section 10.10. You may try running FUN3D with `mpirun cuda-memcheck nodet_mpi` to pinpoint the error. On some systems, `use_cuda_pin=.false.` in the `&gpu_support` namelist must be set for `cuda-memcheck` to function without creating further errors, but this is aberrant behavior.

- **System/environment problem** – These errors typically indicate a mismatch between the CUDA driver and toolkit (see section 10.3), an incorrectly set `CUDA_VISIBLE_DEVICES`, or a problem with the GPU hardware. It may also indicate that the batch system has not made the correct GPUs visible to the user (see section 10.6).
- **Other** – This error is normally caused by an anomaly. If the case is rerun, it should not fail with "unknown error" again. If it persists, please contact `Fun3D-Support@lists.nasa.gov`.

10.13 Half-Precision Solver

NVIDIA® GPUs natively support the IEEE 754 FP16 or half-precision floating point data type. As FUN3D performance is typically bound by the bandwidth of main memory, the FP16 format has the potential to greatly reduce the volume of data moved by FUN3D and thus the bandwidth required, increasing performance. By specifying `use_half_precision=.true.` in the `&gpu_support` namelist, FUN3D will leverage FP16 in its linear solver. In practice, this will provide a total speedup of 1.1-1.2×, depending on the case.

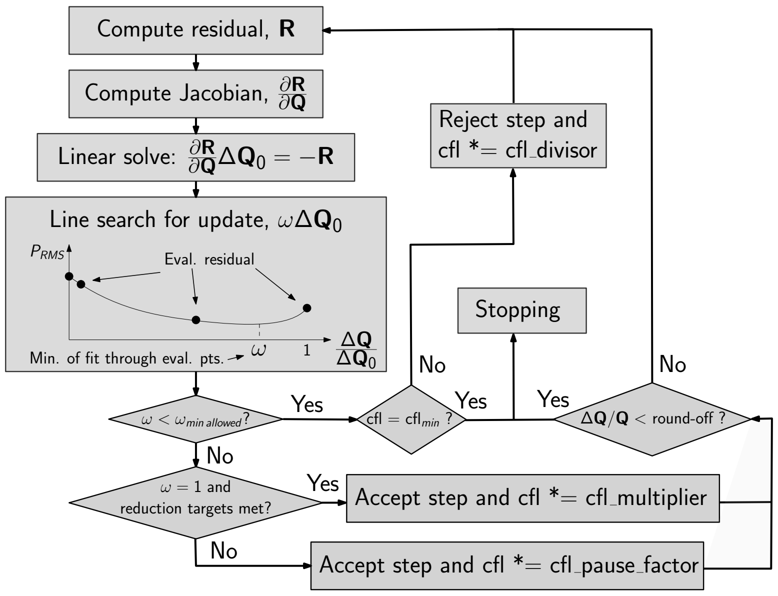
The FP16 solver and experimental results are described in Ref. [45]. When using FP16, solution norms from successive runs will likely show a relative difference of approximately 1.0×10^{-3} if not fully converged. Though early experiments have been successful, please note that this is an experimental option which may negatively impact convergence.

11 Stabilized Finite Elements (SFE)

FUN3D-SFE [46, 47] is a continuous, stabilized finite-element discretization within the FUN3D infrastructure [48, 49]. The addition of finite-elements to FUN3D provides a low-dissipation scheme that is extendible to high-order discretizations, although only linear basis functions, which provide second-order spatial accuracy, are available with this release. The current release includes the capability to obtain steady-state solutions, obtain adjoint solutions, conduct static aeroelastic analysis, and perform flutter analysis using linearized frequency-domain technology [50].

The Streamlined Upwind Petrov-Galerkin (SUPG) finite-element discretization is used to solve for density, velocities, and temperature, as well as the working variable for the turbulence model [51]. Options for both strong and weak boundary conditions are available, and shock capturing is provided via the shock-smoothing approach described in Ref. [52]. Because the stencil for this discretization depends only on nearest neighbors, the scheme is particularly amenable to obtaining, and maintaining, exact linearizations for all components of the solver. This feature enables the scheme to exploit Newton-type solution algorithms, as well as enable other technologies that rely on having accurate linearizations such as sensitivity analysis [53] and adjoint-based adaptation [54–57].

To advance the solution of the Navier-Stokes equations toward a steady state, all the variables are updated in a tightly-coupled, globally convergent Newton-type solver described in Ref. [46]. As mentioned previously, all linearization are discretely consistent, including the boundary conditions. Consistency of the linearization has been verified using expression templates, operator overloading, and with hand-differentiated residuals. To provide clarity to the user, and to later assist in understanding the selection of input variables, Fig. 7 provides an overview of the solution process. As depicted in the upper left-hand corner of the figure, the first step is to compute the residual, followed by computation of the Jacobian matrix, which typically includes the linearization of the spatial residual as well as a pseudo-time term that is adjusted during the iterative process as discussed below. After the residual and linearization have been computed, the Sparse Linear Algebra Toolkit (SLAT) [58] is used to solve the linear system using a Generalized Minimal Residual (GMRES) method, where right-preconditioning with an Incomplete Lower-Upper (ILU) approximate inverse [59] is used on each mesh partition. A feature unique to SLAT is the ability to dynamically reorder the matrix to ensure that the ILU(k) process is stable [60]. Using the update to the solution variables, ΔQ, obtained by solving the linear system, a line search is then conducted to determine whether a full Newton step can be taken, or whether a shorter step length is required to minimize the residual; in some cases, the step may be rejected all together. The determination is made by first comparing the residual computed using the fully updated variables against a targeted reduction factor. If the residual is reduced after taking a full step size, and the resulting residual meets the specified targeted reduction, the full step is accepted and the Courant-Friedrichs-Lewy (CFL) number is increased. In contrast, if the step for minimizing the residual is deemed to be too small, the step is rejected, the CFL number is lowered, and the process is begun again from the solution at the previous time step. In the case where the step size is not small, but does not allow a full Newton step or the targeted residual reduction is not met, the solution variables are updated, but the CFL number is then slightly decreased before proceeding with the next step.



Compute residual, $\mathbf{R}$
Compute Jacobian, $\frac{\partial \mathbf{R}}{\partial \mathbf{Q}}$
Linear solve: $\frac{\partial \mathbf{R}}{\partial \mathbf{Q}} \Delta \mathbf{Q}_0 = -\mathbf{R}$
Line search for update, $\omega \Delta \mathbf{Q}_0$
P_{RMS}
Eval. residual
Min. of fit through eval. pts.
ω
1
$\frac{\Delta \mathbf{Q}}{\Delta \mathbf{Q}_0}$
Reject step and cfl *= cfl_divisor
Stopping
$\omega < \omega_{min\,allowed}$?
Yes
No
cfl = cfl$_{min}$?
$\Delta \mathbf{Q}/\mathbf{Q}$ < round-off ?
$\omega = 1$ and reduction targets met?
Accept step and cfl *= cfl_multiplier
Accept step and cfl *= cfl_pause_factor

Figure 7: Flow chart of SFE nonlinear solution process

It should be noted that several options within the flow solver work in conjunction with the solution strategy described above, and play integral roles in the ability to reliably obtain converged solutions. Most of these options can be left at their default settings and although not described here, more details can be found in the description of `sfe.cfg`. See section B.15.

11.1 Supported Options

- Governing equations: inviscid, laminar, turbulent
- Boundary conditions: 3000, 4000 (optional adiabatic wall), 5000, 5050, 6661, 6662, 6663, 7011, 7012
- Time integration: steady

- Static aeroelastic analysis using the internal modal solver

For turbulent simulations, SFE utilizes the Negative Continuation Spalart-Allmaras [61] model.

11.2 SFE Execution

SFE is enabled by setting `flow_solver = 'sfe'` in `&governing_equations` of `fun3d.nml`. In general, inputs in `fun3d.nml` that are specific to the finite-volume discretization (e.g. flux methods) are ignored by SFE, but non-discretization specific options such as reference conditions, mesh preprocessing, and sampling output are applicable to SFE. SFE-specific options are read from a separate input file, `sfe.cfg`. This file is namelist-like but with zero-based indexing. See section B.15 for detailed description of these inputs.

11.3 SFE Output Files

`[project_rootname]_sfe_hist.dat` The standard `[project_rootname]_hist.dat` computes forces using the finite-volume discretization with SFE's nodal flow state. These loads are less accurate than those computed with the finite-element discretization; therefore, when SFE is enabled, a second history file, `[project_rootname]_sfe_hist.dat`, is written in Tecplot™ format with loads computed consistent with the finite-element method. This file also reports other SFE-specific output such as the linear solver residuals, summary of the nonlinear controller, and timing information.

`[project_rootname]_sfe.out` This file logs detailed information about each SFE iteration including timing information and the full linear solver convergence.

`[project_rootname]_flow.921` and `[project_rootname]_flow.cfl` These files contain more detail of nonlinear controller and line search. This is useful information to send to `Fun3D-Support@lists.nasa.gov`, but otherwise can be ignored.

`sfe_restart.cfg` When a `[project_rootname].flow` restart file is written, a `sfe_restar.cfg` file is also written containing parameters used by SFE's non-linear solver.

11.4 Adjoint

SFE does not utilize the `dual_mpi` executable or the `rubber.data` file for driving the its adjoint process. Instead, it is run as a single time step of the `nodet_mpi` executable, restarted from the steady primal solution. As opposed to the finite-volume adjoint which uses the point-implicit process to converge the adjoint, SFE directly computes the adjoint state using GMRES.

Adjoint mode is enabled by setting `adjoint = .true.` in `sfe.cfg`. See section B.15.10 for available cost functions. `dynamic_reordering = 1` is recommended. For adjoint-based adaptation, `write_solb = .true.` in `sfe.cfg` will generate a binary INRIA Metrix format file that contains primal state and the adjoint state.

Hybrid MPI+OpenMP parallelization creates larger partitions that improve the efficacy of ILU(k) preconditioners in SFE's adjoint. When configuring Fun3D, `--enable-openmp` must be set for hybrid parallelism. In the `&code_run_control` namelist, the `use_openmp` should be set to `.true.`, and at runtime, `OMP_NUM_THREADS` environment variable must be set to the desired number of OpenMP threads per MPI process. Hybrid parallel analysis should use level scheduled implementations of the ILU(k) factorization, which is activated by `preconditioner = LSILUK` in `sfe.cfg`.

11.5 Static Aeroelastic Analysis with SFE

Static aeroelastic analysis with SFE can be performed by fully converging the SFE solution at each intermediate deformed configuration or coupling SFE to the modal structural solver at each time step. For the first approach, set `moddfl=-1` in `&aeroelastic_modal_data` then set `gdisp0` to the current aeroelastic modal displacement for each mode. Once the SFE solver converges at the current deformed state, use the final modal forces in the aeroelastic body history files to update the aeroelastic modal displacements in `gdisp0`. Repeat until the coupled problem converges. Coupling SFE to the modal structural solver at each time step follows the same process as the finite-volume solver, see section 7.6. The aeroelastic modal solver must be run in one of the time domain modes, but static aeroelastic analysis with SFE is best performed with the flow solver running in steady mode. Therefore, to speed up the convergence of either of the coupling approaches, setting `time_accuracy = 0` in `sfe.cfg` overrides the `time_accuracy` from `fun3d.nml` for SFE.

11.6 Linearized Frequency Domain Analysis

The linearized frequency-domain (LFD) method in SFE computes generalized aerodynamic forces (GAFs) of the aeroelastic model [50]. The frequency-domain GAFs are computed from linearized flow responses due to harmonic perturbations of the structural modes at prescribed frequencies. The LFD linear problem in SFE is a complex-valued aeroelastic system, requiring Fun3D to be compiled in complex mode (see section A.10) and run with the internal modal solver active:

```
mpirun complex_nodet_mpi --aeroelastic_internal
```

Users should expect that the LFD solver to require significantly more memory than a time-domain aeroelastic analysis because the solver is run in complex mode. The primary inputs that affect the amount of memory for LFD are the linear solver settings: `krylov_dimension` and `level_of_fill`. Larger `level_of_fill` values can improve the preconditioner effectiveness but increases the memory requirement of the preconditioner matrix. For each `krylov_dimension` an additional vector the size of the flow state is stored in memory. When other selected linear solver parameters are sufficient for linear solver to remain stable, larger values of `max_matvecs` with the same or a lower `krylov_dimension` will allow more GMRES iterations without the additional memory of storing more Krylov vectors. Additionally because each Krylov vector is computed as orthogonal to all the previous active ones, each subsequent Krylov vector is slower to compute. `q_ordering=1` with a low `prune_width` (e.g. 0.1) can improve convergence without the need to increase the `level_of_fill` or `krylov_dimension`.

11.6.1 LFD Inputs

To run LFD, set `time_accuracy = 'lfd'` in `&nonlinear_solver_parameters`. In `&aeroelastic_modal_data`, use `lfd_nfreq` and `lfd_freq` to set the frequencies of motion. LFD is performed by solving the linear problem with a perturbed mode and frequency pair by looping over the modes as the inner loop. The number of `steps` in `&code_run_control` should be equal to the product of the number of perturbed modes and the number of LFD frequencies.

As a linearization about a steady flow field, LFD must be run as a restarted case by setting `restart_read = 'on_nohistorykept'` in `&code_run_control` and providing a restart file. Since SFE uses the same `[project_rootname] .flow` restart file as the finite-volume solver, it is possible to LFD perform linearizing about a finite volume solution steady flow. However, it is strongly recommended to linearize about a steady flow solution generated by SFE because experience has shown that better convergence of the steady flow residuals consistent with those being linearized about (the SFE residuals) leads to less trouble converging the LFD linear problems. Inputs in `sfe.cfg` that aid convergence but are not active at the fully converged solution, such as `smoother_type = ramped`, must be turned off for LFD.

LFD requires `use_modal_deform = .true.` in `&aeroelastic_modal_data`. However, unlike with the modal mesh deformation in the time domain, the solver does not store all of the volume modes in memory at the same time with `time_accuracy = 'lfd'` due to the already significant memory requirement. Instead, `lfd_write_mode_files` in `&aeroelastic_modal_data` can be set to

.false. to recompute the required volume mode every LFD step or .true. to write the volume modules to files when the mode is first perturbed, then reload the volume mode each subsequent perturbation of that mode.

11.6.2 [project_rootname]_gafs.dat

For every LFD linear system solved, the generalized aerodynamic forces (GAFs) are computed for each of the modes present in the model, not only for the set of perturbed modes. The complex-valued GAFs correspond to a unit oscillation amplitude of the perturbed mode. The GAFs are written to the `[project_rootname]_gafs.dat` file. A partial sample output is shown below.

```
ifreq | mode perturbed | Re(GAF(jmode)) | Im(GAF(jmode))
1 1    1.5348657917e-02   -6.9956838836e-01    1.4951474641e-02
1 2   -4.4910442580e+02   -1.2810352222e+00    2.1757726451e+02
2 1    5.1896450250e-02   -3.4974692639e+00    4.4185705557e-02
2 2   -4.4928596732e+02   -6.4541957456e+00    2.1750889556e+02
3 1    1.7525809885e-01   -6.9863549566e+00    1.3359026638e-01
3 2   -4.4952225295e+02   -1.3177186972e+01    2.1724024870e+02
```

Each row corresponds to a single LFD linear problem. The first column is the index of the frequency in `lfd_freq`, and the second is the perturbed mode number. The subsequent columns are the real and imaginary parts of the GAF for mode 1, read and imaginary parts of the GAF for mode 2, etc. for all of the modes in the model. Due to the order in which the GAFs are computed for efficiency, the values in the output file are the transpose of how GAFs are typically represented in matrix form.

11.6.3 Visualization of the Linearized Flow State

The standard sampling options are not usable for LFD solutions because LFD runs in complex mode. To visualize the LFD solution, set `write_solb = .true.` in `sfe.cfg`. The solver will then write the linearized state vectors to binary INRIA Metrix format files, named `[project_rootname]_lfd_[step] .solb`. The state vector in SFE contains the nondimensional ρ, u, v, w, T, and $\hat{\nu}$ for turbulent cases. The first variables in these solb files are the real part of the SFE linearized state vector, then the imaginary parts. The `ref visualize` command from REFINE can be used to convert the solb file to other formats such as Tecplot™. Often the desired visualization output is the linearized pressure coefficient. The pressure coefficient based on the nondimensional pressure is:

$$c_p = \frac{2}{M_{ref}^2}\left(p - p_{ref}\right)$$

Linearizing and applying the equation of state to convert to SFE's ρ and T:

$$c_p' = \frac{2}{\gamma M_{ref}^2} \left(\rho' T + \rho T'\right)$$

12 Yoga

This section will be expanded substantially for the next release of FUN3D.

Yoga is an overset grid assembler [62] that can be used with FUN3D to enable large-scale overset simulations [63]. Yoga uses a modified wall-distance criteria to determine appropriate locations for overset boundaries and has several interpolation strategies available (by default, linear least squares).

References

1. Advisory Group for Aerospace Research and Development: Experimental Data Base for Computer Program Assessment: Report of the Fluid Dynamics Panel Working Group 04. NATO Research and Technology Organisation AGARD AR-138, Jan. 1979.

2. Anderson, W. K.; and Bonhaus, D. L.: An Implicit Upwind Algorithm for Computing Turbulent Flow on Unstructured Grids. *Comp. & Fluids*, vol. 23, no. 1, Jan. 1994, pp. 1–21.

3. Alexandrov, N.; Atkins, H. L.; Bibb, K. L.; Biedron, R. T.; Gnoffo, P. A.; Hammond, D. P.; Jones, W. T.; Kleb, W. L.; Lee-Rausch, E. M.; ; Nielsen, E. J.; Park, M. A.; Raman, V. V.; Roberts, T. W.; Thomas, J. L.; Vatsa, V. N.; Viken, S. A.; White, J. A.; and Wood, W. A.: Team Software Development for Aerothermodynamic and Aerodynamic Analysis and Design. NASA TM-2003-212421, 2003.

4. Hecht, F.: The Mesh Adapting Software: BAMG. INRIA Report, 1998. http://www-rocq1.inria.fr/gamma/cdrom/www/bamg/eng.htm.

5. Carlson, J.-R.: Inflow/Outflow Boundary Conditions with Application to FUN3D. NASA TM-2011-217181, NASA Langley Research Center, Oct. 2011.

6. Peiró, J.; Peraire, J.; and Morgan, K.: FELISA System Version 1.1 Reference Manual Part 2–User Manual. University of Wales/Swansea Technical Report CR/822/94, Aug. 1994.

7. Nielsen, E.; Lu, J.; Park, M.; and Darmofal, D.: An Implicit, Exact Dual Adjoint Solution Method for Turbulent Flows on Unstructured Grids. *Computers and Fluids*, vol. 33, no. 9, 2004, pp. 1131–1155.

8. Schneider, P. J.; and Eberly, D. H.: *Geometric Tools for Computer Graphics*. Morgan Kaufmann, 2002.

9. Kleb, B.; Park, M. A.; Wood, W. A.; Bibb, K. L.; Thompson, K. B.; Gomez, III, R. J.; and Tesch, S. H.: Sketch-to-Solution: An Exploration of Viscous CFD with Automatic Grids. AIAA Paper 2019–2948, 2019.

10. Park, M. A.; and Carlson, J.-R.: Turbulent Output-Based Anisotropic Adaptation. AIAA Paper 2010–168, 2010.

11. Park, M. A.; Haimes, R.; Wyman, N. J.; Baker, P. A.; and Loseille, A.: Boundary Representation Tolerance Impacts on Mesh Generation and Adaptation. 2021–2992, 2021.

12. Galbraith, M. C.; Caplan, P. C.; Carson, H. A.; Park, M. A.; Balan, A.; Anderson, W. K.; Michal, T.; Krakos, J. A.; Kamenetskiy, D. S.; Loseille, A.; Alauzet, F.; Frazza, L.; and Barral, N.: Verification of Unstructured Grid Adaptation Components. *AIAA Journal*, vol. 58, no. 9, Sept. 2020, pp. 3947–3962.

13. Balan, A.; Park, M. A.; Anderson, W. K.; Kamenetskiy, D. S.; Krakos, J. A.; Michal, T.; and Alauzet, F.: Verification of Anisotropic Mesh Adaptation for RANS Simulations over ONERA M6 Wing. *AIAA Journal*, vol. 58, no. 4, Feb. 2020, pp. 1550–1565.

14. Michal, T.; Krakos, J.; Kamenetskiy, D.; Galbraith, M.; Ursachi, C.-I.; Park, M. A.; Anderson, W. K.; Alauzet, F.; and Loseille, A.: Comparing Unstructured Adaptive Mesh Solutions for the High Lift Common Research Model Airfoil. *AIAA Journal*, vol. 59, no. 9, Sept. 2021, pp. 3566–3584.

15. Park, M. A.; Balan, A.; Clerici, F.; Alauzet, F.; Loseille, A.; Kamenetskiy, D. S.; Krakos, J. A.; Michal, T.; and Galbraith, M. C.: Verification of Viscous Goal-Based Anisotropic Mesh Adaptation. AIAA Paper 2021-1362, 2021.

16. Haimes, R.; and Dannenhoffer, III, J. F.: The Engineering Sketch Pad: A Solid-Modeling, Feature-Based, Web-Enabled System for Building Parametric Geometry. AIAA Paper 2013–3073, 2013.

17. Dannenhoffer, III, J. F.: OpenCSM: An Open-Source Constructive Solid Modeler for MDAO. AIAA Paper 2013–701, 2013.

18. Haimes, R.; and Drela, M.: On The Construction of Aircraft Conceptual Geometry for High-Fidelity Analysis and Design. AIAA Paper 2012–683, 2012.

19. Haimes, R.; and Dannenhoffer, III, J. F.: EGADSlite: A Lightweight Geometry Kernel for HPC. AIAA Paper 2018–1401, 2018.

20. Dannenhoffer, III, J. F.: The Creation of a Static BRep Model Given a Cloud of Points. AIAA Paper 2017–138, 2017.

21. Park, M. A.; Kleb, B.; Jones, W. T.; Krakos, J. A.; Michal, T.; Loseille, A.; Haimes, R.; and Dannenhoffer, III, J. F.: Geometry Modeling for Unstructured Mesh Adaptation. AIAA Paper 2019–2946, 2019.

22. Spalding, D. P.: A Single Formula for the 'Law of the Wall'. *Journal of Applied Mechanics*, vol. 28, no. 3, 1961, pp. 455–458.

23. Barral, N.; Alauzet, F.; and Loseille, A.: Metric-Based Anisotropic Mesh Adaptation for Three-Dimensional Time-Dependent Problems Involving Moving Geometries. AIAA Paper 2015–2039, 2015.

24. Alauzet, F.; and Frazza, L.: 3D RANS Anisotropic Mesh Adaptation on the High-Lift Version of NASA's Common Research Model (HL-CRM). AIAA Paper 2019–2947, 2019.

25. Nastac, G.; Tramel, R.; and Nielsen, E.: Improved Heat Transfer Prediction for High-Speed Flows over Blunt Bodies using Adaptive Mixed-Element Unstructured Grids. AIAA Paper 2022–111, 2022.

26. Newman III, J.; Taylor, A.; Barnwell III, R.; Newman, P.; and Hou, G.-W.: Overview of Sensitivity Analysis and Shape Optimization for Complex Aerodynamic Configurations. *Journal of Aircraft*, vol. 36, no. 1, 1999, pp. 87–96.

27. Peter, J.; and Dwight, R.: Numerical Sensitivity Analysis for Aerodynamic Optimization: A Survey of Approaches. *Computers and Fluids*, vol. 39, no. 3, 2010, pp. 373–391.

28. Nielsen, E.; and Diskin, B.: Discrete Adjoint-Based Design for Unsteady Turbulent Flows on Dynamic Overset Unstructured Grids. *AIAA Journal*, vol. 51, no. 6, 2013, pp. 1355–1373.

29. Jones, M.: CFD Analysis and Design Optimization of Flapping Wing Flows. Ph.D. Thesis, North Carolina A&T State University, June 2013.

30. Nielsen, E.; Lee-Rausch, E.; and Jones, W.: Adjoint-Based Design of Rotors in a Noninertial Reference Frame. *AIAA Journal of Aircraft*, vol. 47, no. 2, 2010, pp. 638–646.

31. Bloch, G. S.: An Assessment of Inlet Total-Pressure Distortion Requirements for the Compressor Research Facility. Wright Laboratory Technical Report WL-TR-92-2066, 1992.

32. Rallabhandi, S. K.: Advanced Sonic Boom Prediction Using the Augmented Burgers Equation. *AIAA Journal of Aircraft*, vol. 48, no. 4, July–Aug. 2011, pp. 1245–1253.

33. Rallabhandi, S.: Sonic Boom Adjoint Methodology and its Applications. AIAA Paper 2011–3497, 2011.

34. Rallabhandi, S.; Nielsen, E.; and Diskin, B.: Sonic Boom Mitigation Through Aircraft Design and Adjoint Methodology. AIAA Paper 2012–3220, 2012.

35. Rallabhandi, S.: Application of Adjoint Methodology to Supersonic Aircraft Design Using Reversed Equivalent Areas. AIAA Paper 2013–2663, 2013.

36. Samareh, J. A.: Novel Multidisciplinary Shape Parameterization Approach. *AIAA Journal of Aircraft*, vol. 38, no. 6, Nov.–Dec. 2001, pp. 1015–1024.

37. Samareh, J. A.: Aerodynamic Shape Optimization Based on Free-From Deformation. AIAA Paper 2004–4630, 2004.

38. Nielsen, E.; and Jones, W.: Integrated Design of an Active Flow Control System Using a Time-Dependent Adjoint Method. *Mathematical Modeling of Natural Phenomena*, vol. 6, no. 3, 2011, pp. 141–165.

39. Jones, W.; Nielsen, E.; Lee-Rausch, E.; and Acree Jr., C.: Multi-point Adjoint-Based Design of Tilt-Rotors in a Noninertial Reference Frame. AIAA Paper 2014–290, 2014.

40. Wrenn, G. A.: An Indirect Method for Numerical Optimization using the Kreisselmeir-Steinhauser Function. NASA CR-4220, NASA Langley Research Center, Mar. 1989.

41. Lyness, J. N.: Numerical Algorithms Based on the Theory of Complex Variables. *Proceedings of the ACM 22nd National Conference*, Thomas Book Co., 1967, pp. 124–134.

42. Lyness, J. N.; and Moler, C. B.: Numerical Differentiation of Analytic Functions. *J. Numer. Anal.*, vol. 4, 1967, pp. 202–210.

43. Anderson, W. K.; Newman, J. C.; Whitfield, D. L.; and Nielsen, E. J.: Sensitivity Analysis for the Navier-Stokes Equations on Unstructured Meshes Using Complex Variables. *AIAA J.*, vol. 39, no. 1, Jan. 2001, pp. 56–63. See also AIAA Paper 99-3294.

44. Nastac, G.; Walden, A.; Nielsen, E. J.; and Frendi, K.: Implicit Thermochemical Nonequilibrium Flow Simulations on Unstructured Grids using GPUs. AIAA Paper 2021-0159, 2021.

45. Walden, A.; Nielsen, E.; Diskin, B.; and Zubair, M.: A Mixed Precision Multicolor Point-Implicit Solver for Unstructured Grids on GPUs. *Ninth Workshop on Irregular Applications: Architectures and Algorithms*, IA3 2019, 2019.

46. Anderson, W. K.; Newman, J. C.; and Karman, S. L.: Stabilized Finite Elements in FUN3D. *Journal of Aircraft*, vol. 55, no. 2, Mar. 2018, pp. 696–714.

47. Anderson, W. K.; and Newman, J. C.: High-Order Stabilized Finite Elements on Dynamic Meshes. AIAA Paper 2018-1307, 2018. URL https://doi.org/10.2514/6.2018-1307.

48. Anderson, W. K.; and Bonhaus, D. L.: An Implicit Upwind Algorithm for Computing Turbulent Flows on Unstructured Grids. *Computers and Fluids*, vol. 23, no. 1, 1994, pp. 1–22.

49. Biedron, R. T.; Carlson, J.-R.; Derlaga, J. M.; Gnoffo, P. A.; Hammond, D. P.; Jones, W. T.; Kleb, B.; Lee-Rausch, E. M.; Nielsen, E. J.; Park, M. A.; Rumsey, C. L.; Thomas, J. L.; Thompson, K. B.; and Wood, W. A.: FUN3D Manual: 13.6. NASA TM-2018-220416, Langley Research Center, Oct. 2019.

50. Jacobson, K. E.; Stanford, B. K.; Wood, S. L.; and Anderson, W. K.: Flutter Analysis with Stabilized Finite Elements based on the Linearized Frequency-domain Approach. AIAA Paper 2020-0403, Orlando, Florida, January 6–10 2020. URL https://arc.aiaa.org/doi/abs/10.2514/6.2020-0403.

51. Spalart, P. R.; and Allmaras, S. R.: A One-Equation Turbulence Model for Aerodynamic Flows. *La Recherche Aerospatiale*, vol. 1, no. 1, 1994, pp. 5–21.

52. Holst, K. R.; Glasby, R. S.; Erwin, J. T.; Stefanski, D. L.; and Coder, J. G.: High-Order Shock Capturing Techniques using HPCMP CREATE-AV Kestrel. AIAA Paper 2019-1345, 2019. URL https://arc.aiaa.org/doi/abs/10.2514/6.2019-1345.

53. Jacobson, K.; Stanford, B.; Wood, S. L.; and Anderson, W. K.: Adjoint-based Sensitivities of Flutter Predictions based on the Linearized Frequency-domain Approach. AIAA Paper 2021-0282, 2021. URL https://arc.aiaa.org/doi/abs/10.2514/6.2021-0282.

54. Belme, A.; Alauzet, F.; and Dervieux, A.: An a priori Anisotropic Goal-Oriented Error Estimate for Viscous Compressible Flow and Application to Mesh Adaptation. *Journal of Computational Physics*, vol. 376, Jan. 2019, pp. 1051–1088. URL https://doi.org/10.1016/j.jcp.2018.08.048.

55. Park, M. A.; and Darmofal, D. L.: Parallel Anisotropic Tetrahedral Adaptation. AIAA Paper 2008-0917, 2008. URL https://arc.aiaa.org/doi/abs/10.2514/6.2008-917.

56. Wood, S.; and Anderson, W. K.: Reynolds-Averaged Navier-Stokes Computations of the NASA Juncture Flow Model Using Expert-Crafted

and Adapted Grids. AIAA Paper 2020-2751, 2020. URL https://arc.aiaa.org/doi/abs/10.2514/6.2020-2751.

57. Balan, A.; Park, M. A.; Wood, S. L.; Anderson, W. K.; Rangarajan, A.; Sanjaya, D. P.; and May, G.: A review and comparison of error estimators for anisotropic mesh adaptation for flow simulations. *Computers & Fluids*, vol. 234, 2022, p. 105259. URL https://www.sciencedirect.com/science/article/pii/S0045793021003601.

58. Wood, S. L.; Jacobson, K.; Jones, W. T.; and Anderson, W. K.: Sparse Linear Algebra Toolkit for Computational Aerodynamics. AIAA Paper 2020-0317, 2020. URL https://arc.aiaa.org/doi/abs/10.2514/6.2020-0317.

59. Saad, Y.: *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2 ed., 2003.

60. Anderson, W. K.; Wood, S.; and Jacobson, K. E.: Node Numbering for Stabilizing Preconditioners Based on Incomplete LU Decomposition. AIAA Paper 2020-3022, 2020. URL https://arc.aiaa.org/doi/abs/10.2514/6.2020-3022.

61. Allmaras, S. R.; Johnson, F. T.; and Spalart, P. R.: Modifications and Clarifications for the Implementation of the Spalart-Allmaras Turbulence Model. *Seventh International Conference on Computational Fluid Dynamics (ICCFD7)*, 2012.

62. Druyor, Jr., C. T.: Advances in Parallel Overset Domain Assembly. Ph.D. Thesis, University of Tennessee at Chattanooga, Aug. 2016.

63. Druyor, C.: Enhancing Scalability for FUN3D Rotorcraft Simulations with Yoga: an Overset Grid Assembler. AIAA Paper 2021-2746, 2021.

64. Blanco, J. L.; and Rai, P. K.: nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees. https://github.com/jlblancoc/nanoflann, 2014.

65. Boman, E. G.; Catalyurek, U. V.; Chevalier, C.; and Devine, K. D.: The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering, and Coloring. *Scientific Programming*, vol. 20, no. 2, 2012, pp. 129–150.

66. Biedron, R. T.; and Thomas, J. L.: Recent Enhancements to the FUN3D Flow Solver For Moving-Mesh Applications. AIAA Paper 2009–1360, 2009.

67. Anderson, W. K.; Rausch, R. D.; and Bonhaus, D. L.: Implicit/Multigrid Algorithms for Incompressible Turbulent Flows on Unstructured Grids. *J. Comput. Phys.*, vol. 128, no. 2, 1996, pp. 391–408.

68. Sutton, K.; and Gnoffo, P. A.: Multi-Component Diffusion with Application to Computational Aerothermodynamics. AIAA Paper 98-2575, June 1998.

69. Nishikawa, H.; and Liu, Y.: Third-Order Edge-Based Scheme for Unsteady Problems. AIAA Paper 2018-4166, 2018.

70. Park, M. A.: Anisotropic Output-Based Adaptation with Tetrahedral Cut Cells for Compressible Flows. Ph.D. Thesis, Massachusetts Institute of Technology, Sept. 2008.

71. Venkatakrishnan, V.: Convergence to Steady State Solutions of the Euler Equations on Unstructured Grids with Limiters. *Journal of Computational Physics*, vol. 118, no. 1, 1995, pp. 120–130.

72. Nishikawa, H.: New Unstructured-Grid Limiter Functions. AIAA Paper 2022–1374, 2022.

73. Turkel, E.: Preconditioning Techniques in Computational Fluid Dynamics. *Annual Review of Fluid Mechanics*, vol. 31, 1999, pp. 385–416.

74. Catris, S.; and Aupoix, B.: Density Corrections for Turbulence Models. *Aerospace Science and Technology*, vol. 4, no. 1, Jan. 2000, pp. 1–11.

75. Spalart, P. R.; Jou, W.-H.; Strelets, M.; and Allmaras, S. R.: Comments on the Feasibility of LES for Wing and on a Hybrid RANS/LES Approach. *Proceedings of the First ASOSR Conference on DNS/LES*, Aug. 1997.

76. Menter, F. R.: Two-Equation Eddy-Viscosity Turbulence Models for Engineering Applications. *AIAA Journal*, vol. 32, no. 8, Aug. 1994, pp. 1598–1605.

77. Menter, F. R.: Improved two-equation k-omega turbulence models for aerodynamic flows. NASA TM-103975, Oct. 1992.

78. Wilcox, D. C.: Reassessment of the Scale-Determining Equation for Advanced Turbulence Models. *AIAA Journal*, vol. 26, no. 11, Nov. 1988, pp. 1299–1310.

79. Wilcox, D. C.: Formulation of the k-ω Turbulence Model Revisited. *AIAA Journal*, vol. 46, no. 11, Nov. 2008, pp. 2823–2838.

80. Strelets, M.: Detached Eddy Simulation of Massively Separated Flow. AIAA Paper 2001–879, 2001.

81. Woodruff, S. L.: Coupling Turbulence in Hybrid LES-RANS Techniques. *Seventh International Symposium on Turbulence and Shear Flow Phenomena, Ottawa,Canada*, 2010.

82. Lynch, C. E.; and Smith, M. J.: Hybrid RANS-LES Turbulence Models on Unstructured Grids. AIAA Paper 2008–3854, 2008.

83. Lynch, C. E.: Advanced CFD Methods for Wind Turbine Analysis. Ph.D. Thesis, Georgia Institute of Technology, May 2011.

84. Langtry, R. B.; and Menter, F. R.: Correlation-Based Transition Modeling for Unstructured Parallelized Computational Fluid Dynamics Codes. *AIAA Journal*, vol. 47, no. 12, Dec. 2009, pp. 2894–2906.

85. Abdol-Hamid, K. S.; Carlson, J.-R.; and Rumsey, C. L.: Verification and Validation of the k-kL Turbulence Model in FUN3D and CFL3D Codes. NASA TM-2015-218968, Nov. 2015.

86. Spalart, P. R.: Strategies for Turbulence Modelling and Simulations. *International Journal of Heat and Fluid Flow*, vol. 21, no. 3, June 2000, pp. 252–263.

87. Rumsey, C. L.; and Gatski, T. B.: Recent Turbulence Model Advances Applied to Multielement Airfoil Computations. *AIAA Journal of Aircraft*, vol. 38, no. 5, Sept.–Oct. 2001, pp. 904–910.

88. Spalart, P. R.: Trends in Turbulence Treatments. *Fluids 2000 Conference and Exhibit*, 2000.

89. Dacles-Mariani, J.; Zilliac, G. G.; Chow, J. S.; and Bradshaw, P.: Numerical/Experimental Study of a Wingtip Vortex in the Near Field. *AIAA Journal*, vol. 33, no. 9, Sept. 1995, pp. 1561–1568.

90. Dacles-Mariani, J.; Kwak, D.; and Zilliac, G.: On Numerical Errors and Turbulence Modeling in Tip Vortex Flow Prediction. *International Journal for Numerical Methods in Fluids*, vol. 30, no. 1, 1999, pp. 65–82.

91. Shur, M. L.; Strelets, M. K.; Travin, A. K.; and Spalart, P. R.: Turbulence Modeling in Rotating and Curved Channels: Assessing the Spalart-Shur Correction. *AIAA Journal*, vol. 38, no. 5, May 2000, pp. 784–702.

92. Spalart, P. R.; Deck, S.; Shur, M. L.; Squires, K. D.; Strelets, M. K.; and Travin, A.: A New Version of Detached-eddy Simulation, Resistant to Ambiguous Grid Densities. *Theoretical and Computational Fluid Dynamics*, vol. 20, no. 3, July 2006, pp. 181–195.

93. Vatsa, V. N.; and Lockard, D. P.: Assessment of Hybrid RANS/LES Turbulence Models for Aeroacoustics Applications. AIAA Paper 2010–4001, 2010.

94. Carpenter, M. H.; Viken, S. A.; and Nielsen, E. J.: The Efficiency of High Order Temporal Schemes. AIAA Paper 2003–86, 2003.

95. Vatsa, V. N.; Carpenter, M. H.; and Lockard, D. P.: Re-evaluation of an Optimized Second Order Backward Difference (BDF2OPT) Scheme for Unsteady Flow Applications. AIAA Paper 2010–122, 2010.

96. Gottlieb, S.; Shu, C. W.; and Tadmor, E.: Strong Stability-Preserving High-Order Time Discretization Methods. *SIAM Review*, vol. 43, no. 1, 2001, pp. 89–112.

97. Jacobson, K.; Stanford, B.; Wood, S.; and Anderson, W. K.: Flutter Analysis with Stabilized Finite Elements based on the Linearized Frequency-domain Approach. AIAA Paper 2020-0403, 2020.

98. Nishikawa, H.; and Liu, Y.: Accuracy-Preserving Source Term Quadrature for Third-Order Edge-Based Discretization. *Journal of Computational Physics*, vol. 341, Sept. 2017, pp. 595–622.

99. Liu, Y.; and Nishikawa, H.: Third-Order Inviscid and Second-Order Hyperbolic Navier-Stokes Solvers for Three-Dimensional Unsteady Inviscid and Viscous Flows. AIAA Paper 2017-738, 2017.

100. Pandya, M. J.; Diskin, B.; Thomas, J. L.; and Frink, N. T.: Improved Convergence and Robustness of USM3D Solutions on Mixed-Element Grids. *AIAA Journal*, vol. 54, no. 9, Sept. 2016, pp. 2589–2610.

101. Pandya, M. J.; Diskin, B.; Thomas, J. L.; and Frink, N. T.: Assessment of USM3D Hierarchical Adaptive Nonlinear Method Preconditioners for Three-Dimensional Cases. *AIAA Journal*, vol. 55, no. 10, Oct. 2017, pp. 3409–3424.

102. Wang, L.; Diskin, B.; Nielsen, E. J.; and Liu, Y.: Improvements in Iterative Convergence of FUN3D Solutions. AIAA paper, 2021.

103. Eisenstat, S. C.; Elman, H. C.; and Schultz, M. H.: Variational Iterative Methods for Nonsymmetric Systems of Linear Equations. *SIAM Journal on Numerical Analysis*, vol. 2, Apr. 1983, pp. 345–357.

104. Saad, Y.; and Schultz, M. H.: GMRES: A Generalized Minimum Residual Method for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, 1986, pp. 856–869.

105. Knopp, T.; Alrutz, T.; and Schwamborn, D.: A Grid and Flow Adaptive Wall-Function Method for RANS Turbulence Modelling. *Journal of Computational Physics*, vol. 220, no. 1, Dec. 2006, pp. 19–40.

106. Lee-Rausch, E. M.; Frink, N. T.; Mavriplis, D. J.; Rausch, R. D.; and Milholen, W. E.: Transonic Drag Prediction on a DLR-F6 Transport Configuration Using Unstructured Grid Solvers. AIAA Paper 2004–556, 2004.

107. Boyd, I.: Predicting Breakdown of the Continuum Equations Under Rarefied Flow Conditions. *AIP Conference Proceedings*, vol. 663, May 2003, pp. 899–906.

108. Alauzet, F.; and Loseille, A.: High-Order Sonic Boom Modeling Based on Adaptive Methods. *Journal of Computational Physics*, vol. 229, no. 3, 2010, pp. 561–593.

109. Barth, T. J.: Recent Developments in High Order K-Exact Reconstruction on Unstructured Meshes. AIAA Paper 93–668, 1993.

110. Park, M. A.; Loseille, A.; Krakos, J. A.; and Michal, T.: Comparing Anisotropic Output-Based Grid Adaptation Methods by Decomposition. AIAA Paper 2015–2292, 2015.

111. Alauzet, F.: Size Gradation Control of Anisotropic Meshes. *Finite Elements in Analysis and Design*, vol. 46, no. 1–2, 2010, pp. 181–202.

112. Venditti, D. A.: Grid Adaptation for Functional Outputs of Compressible Flow Simulations. Ph.D. Thesis, Massachusetts Institute of Technology, 2002.

113. Loseille, A.; Dervieux, A.; and Alauzet, F.: Fully Anisotropic Goal-Oriented Mesh Adaptation for 3D Steady Euler Equations. *Journal of Computational Physics*, vol. 229, no. 8, 2010, pp. 2866–2897.

114. Cheatwood, F. M.; and Gnoffo, P.: *User's Manual for the Langley Aerothermodynamic Upwind Relaxation Algorithm (LAURA)*. NASA TM-4674, 1996.

115. Michal, T.; and Krakos, J.: Anisotropic Mesh Adaptation Through Edge Primitive Operations. AIAA Paper 2012–159, 2012.

116. Shenoy, R.: Overset Adaptive Strategies for Complex Rotating Systems. Ph.D. Thesis, Georgia Institute of Technology, May 2014.

117. Ffowcs Williams, J. E.; and Hawkings, D. L.: Sound Generated by Turbulence and Surfaces in Arbitrary Motion. *Philosophical Transactions of the Royal Society*, vol. A264, no. 1151, 1969, pp. 321–342.

118. Brentner, K. S.; Lopes, L. V.; Chen, H. N.; and Horn, J. F.: Near Real-Time Simulation of Rotorcraft Acoustics and Flight Dynamics. 59th Annual Forum, AHS International, Alexandria, VA, 2003.

119. Lopes, L. V.; and Burley, C. L.: Design of the Next Generation Aircraft Noise Prediction Program: ANOPP2. AIAA Paper 2011–2854, 2011.

120. Waithe, K. A.: Source Term Model for Vortex Generator Vanes in a Navier-Stokes Computer Code. AIAA Paper 2004-1236, Jan. 2004.

121. Bartels, R. E.: Development, Verification and Use of Gust Modeling in the NASA Computational Fluid Dynamics Code FUN3D. NASA TM-2012-217771, NASA Langley Research Center, Oct. 2012.

122. Biedron, R. T.; Jacobson, K. E.; Jones, W. T.; Steven J. Massey, E. J. N.; Kleb, W. L.; and Zhang, X.: Sensitivity Analysis for Multidisciplinary Systems (SAMS). NASA TM-2018-220089, 2018.

123. Edwards, J. W.; Bennett, R. M.; Whitlow, Jr., W.; and Seidel, D. A.: Time-Marching Transonic Flutter Solutions Including Angle-of-Attack Effects. *AIAA J.*, vol. 20, no. 11, 1983, pp. 899–906.

124. O'Brien, D. M.: Analysis Of Computational Modeling Techniques For Complete Rotorcraft Configurations. Ph.D. Thesis, Georgia Institute of Technology, 2006.

125. Biedron, R. T.; and Lee-Rausch, E. M.: Rotor Airloads Prediction Using Unstructured Meshes and Loose CFD/CSD Coupling. AIAA Paper 2008–7341, 2008.

126. Goldstein, S.: On the Vortex Theory of Screw Propellers. *Proceeding of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 792, no. 123, Apr. 1929, pp. 440–465.

127. Stern, F.; Kim, H. T.; and Patel, V. C.: A Viscous-Flow Approach to the Computation of Propeller-Hull Interaction. *Journal of Ship Research*, vol. 32, no. 4, Dec. 1988, pp. 246–262.

128. Gaasbeek, J. R. V.: Rotorcraft Flight Simulation Computer Program C81 with Datamap Interface. USAAVRADCOM-TR-80-D-38A, U.S. Army Research And Technology Laboratories, Oct. 1981.

129. Srinivasan, S.; Tannehill, J. C.; and Weilmuenster, K. J.: Simplified Curve Fits for the Thermodynamic Properties of Equilibrium Air. NASA RP-1181, June 1987.

130. Prabhu, R. K.; and Erickson, W. D.: A Rapid Method for the Computation of Equilibrium Chemical Composition of Air to 15,000 K. NASA RP-2792, Mar. 1988.

131. McBride, B. J.; and Gordon, S.: Computer Program for Calculation of Complex Chemical Equilibrium Compositions and Applications. NASA RP-1311, June 1996.

132. Gnoffo, P. A.; Gupta, R. N.; and Shinn, J. L.: Conservation Equations and Physical Models for Hypersonic Air Flows in Thermal and Chemical Nonequilibrium. NASA TP-2867, Feb. 1989.

133. Gupta, R.; Yos, J.; Thompson, R. A.; and Lee, K.: A Review of Reaction Rates and Thermodynamic and Transport Properties for an 11-Species Air Model for Chemical and Thermal Nonequilibrium Calculations to 30,000 K. NASA RP-1232, Aug. 1990.

134. Millikan, R. C.; and White, D. R.: Systematics of Vibrational Relaxation. *Chem. Phys.*, vol. 39, no. 12, Dec. 1963, pp. 3209–3213.

135. Ali, A. W.: The Harmonic and Anharmanic Models for Vibrational Relaxation and Dissociation of the Nitrogen Molecule. U.S. Navy NRL Memo 5924, Dec. 1986.

136. Wright, M.: Recommended Collision Integrals for Transport Property Computations Part 1: Air Species. *AIAA J.*, vol. 43, no. 12, 2005, pp. 2558–2564.

137. Wright, M.: Recommended Collision Integrals for Transport Property Computations Part 2: Mars and Venus Entries. *AIAA J.*, vol. 45, no. 1, 2005, pp. 281–288.

138. Johnston, C. O.; Hollis, B. R.; and Sutton, K.: Spectrum Modeling for Air Shock-Layer Radiation at Lunar-Return Conditions. *Journal of Spacecraft and Rockets*, vol. 45, no. 6, Nov–Dec 2008, pp. 865–878.

139. Johnston, C. O.; Hollis, B. R.; and Sutton, K.: Non-Boltzmann Modeling for Air Shock-Layer Radiation at Lunar-Return Conditions. *Journal of Spacecraft and Rockets*, vol. 45, no. 6, Nov–Dec 2008, pp. 879–890.

140. Johnston, C. O.; Gnoffo, P. A.; and Sutton, K.: The Influence of Ablation on Radiative Heating for Earth Entry. *Journal of Spacecraft and Rockets*, vol. 46, no. 3, May–June 2009, pp. 481–491.

141. Cunto, W.: TOPbase at the CDS. *Astronomy and Astrophysics*, vol. 275, 1993, pp. L5–L8.

142. Gally, T.: Development of Engineering Methods for Nonequilibrium Radiative Phenomena about Aeroassisted Entry Vehicles. Ph.D. Thesis, Texas A&M, 1992.

143. Johnston, C. O.; Hollis, B. R.; and Sutton, K.: Radiative Heating Methodology for the Huygens Probe. *Journal of Spacecraft and Rockets*, vol. 44, no. 5, Sep–Oct 2007, pp. 993–1002.

144. Cuthill, E.; and McKee, J.: Reducing the Bandwidth of Sparse Symmetric Matrices. *Proceedings of the 1969 24th National Conference*, ACM '69, ACM, New York, NY, USA, 1969, pp. 157–172. URL http://doi.acm.org/10.1145/800195.805928.

145. Spalart, P.: Strategies for turbulence modelling and simulations. *International Journal of Heat and Fluid Flow*, vol. 21, no. 3, 2000, pp. 252 – 263. URL http://www.sciencedirect.com/science/article/pii/S0142727X00000072.

146. Kamenetskiy, D.; Bussoletti, J.; Hilmes, C.; Johnson, F.; Venkatakrishnan, V.; and Wigton, L.: *Numerical Evidence of Multiple Solutions for the Reynolds-Averaged Navier-Stokes Equations for High-Lift Configurations*. URL https://arc.aiaa.org/doi/abs/10.2514/6.2013-663.

147. Linde, T.; and Roe, P. L.: On Multidimensional Positively Conservative High-Resolution Schemes. *Barriers and Challenges in Computational Fluid Dynamics*, V. Venkatakrishnan, M. D. Salas, and S. R. Chakravarthy, eds., Springer Netherlands, vol. 6 of *ICASE/LaRC Interdisciplinary Series in Science and Engineering*, 1998, pp. 299–313.

148. Barth, T. J.: Aspects of Unstructured Grids and Finite-Volume Solvers for the Euler and Navier-Stokes Equations. VKI Lecture Series 1994-05, 1994.

149. Diskin, B.; and Thomas, J. L.: unpublished notes, 2013.

Appendix A

Installation

FUN3D is distributed as gzipped archive of source code. The GNU build system is used to package and install FUN3D. The required installation steps are detailed in this section. Due to the large range of capabilities, configuring the dependent packages is the most involved step and is the focus this section.

As was illustrated in the Quick Start section, four basic steps are required:

1. Extract the source code from the gzipped tarball archive with `tar`
2. Configure the desired dependencies and compiler options with `configure`
3. Compile via `make`
4. Install the compiled binaries and supporting scripts via `make install`

If any difficulties arise with the installation process, please run the script `collect_logs.sh` from within the configuration directory. This script is located in the top level source directory. This will produce a compressed tar file, `config_logs.zgt`. Please send this file along with the full stdout and stderr of `make` to `Fun3D-Support@lists.nasa.gov`. The user is *strongly* advised against editing the `configure` script or any `Makefile` it produces. We are unable to assist users who have edited these files.

A.1 Extracting Files

After downloading the source code as a gzipped tarball, the user can unpack it with

```
tar zxf fun3d_intg-14.0.1-*.tar.gz
```

which will create the directory `fun3d_intg-14.0.1-*`. (The `*` represents a code that the FUN3D uses internally to version the code.) If you have do not have a GNU-compatible `tar`, you may have to insert a separate decompression step, i.e.,

```
gzip -d fun3d_intg-14.0.1-*.tar.gz | tar zxf -
```

A.2 Configure Introduction

The FUN3D suite of tools is configured and built via the GNU build system and must be configured first. Change to this directory, e.g., `cd fun3d_intg-14.0.1-*`, and execute

```
./configure --help
```

to see a list of available compilation options. To see a list of the available compilation options from all FUN3D components, execute

```
./configure --help=recursive
```

When `configure` is invoked, detailed results of all the tests it performs are written to the file `config.log`. The FUN3D source is organized into multiple independent components, each of which is configured and generates it's own `config.log`. A script, `collect_logs.sh`, is provided which will collect these logs for shipment to `Fun3D-Support@lists.nasa.gov` if problems are encountered.

Some features of the configure step that have caused problems for users are:

- An incorrect spelling of a `--enable-*` or `--with-*` option is silently ignored. This will result in the intended option not being included in the compiled executable.
- Option values containing spaces must be quoted to be correctly interpreted by the shell (i.e., `FCFLAGS='-option1 -option2'`).
- If the `configure` command is executed more than once with different options, `make clean` is required before the `make` step, so that changes to the configuration are correctly reflected in the compiled executable.

A.3 Alternative Installation Path

The path to the installation directory is specified by the `--prefix=` option. The default is to install to `/usr/local` with executables placed in `/usr/local/bin`. This default location may not be available if the user does not have write permission to this directory (without root or administrator privileges).

To install to an alternative path (e.g., `$HOME/local`), use the `--prefix=` option to set the installation path

```
./configure --prefix=$HOME/local
```

Finally, to include the FUN3D executables in the command search path, add

```
setenv PATH $HOME/local/bin:$PATH
```

to the `~/.cshrc` file or the equivalent for your shell.

A.4 Fortran Compiler Option Tuning (FTune)

By default, `configure` will use compiler and linker options chosen by the FUN3D team. The process is referred to as "FTune." The users `PATH` is searched in a predefined order until the first FUN3D-compatible compiler is found. When configured with MPI, the build will use `mpif90` located in the `bin` directory of the given MPI installation.[A1] However, the user can explicitly specify the desired Fortran compiler via the `FC` environment variable.

To directly specify the compiler and linker options, use the `FCFLAGS` and `LDFLAGS` environment variables. The default behavior is to append their values to the options defined by FTune. If the `--disable-ftune` option is given to `configure`, FTune will be disabled and the values given by `FCFLAGS` and `LDFLAGS` will be used explicitly. For example, to ensure that the Intel® Fortran compiler `ifort` is used with only the `-O3`, `-ip`, and `-lm` options,

```
./configure --disable-ftune \
  FC=ifort \
  FCFLAGS='-O3 -ip' \
  LDFLAGS='-lm'
```

Do **not** set the compiler variables to MPI compiler wrappers (ex: mpif90) as such will cause problems when detecting underlying compiler options. These variables, if set, should be set to the underlying compiler used by the MPI compiler wrappers (ex: `mpif90 -show`). The order of variables and options are inconsequential, and single quotation marks (') are used to protect values with spaces from the shell. Some FTune options may be *unconditionally required* for a given compiler, as in the case of linking with the math library `-lm` above.

A.5 C++ Compiler Requirements

FUN3D contains C++ source that relies on the C++11 standard. As such, a C++11 compiler is required for build. Some commercial compilers (ex: Intel®) depend on the GNU tools of the target system. These may include the GNU header files, the GNU linker, and associated libraries. Therefore, the commercial compiler will behave like the GCC version on your system regardless of the version of the commercial compiler (see for example Intel). The C++11 requirement will require a `gcc` version of 5 or better.

A.6 C++ Compiler Option Tuning (CXXTune)

The "FTune" process for setting compiler and linker options chosen by the FUN3D team has been extended to C++ compilers. The users `PATH` is searched in a predefined order until the first FUN3D-compatible C and C++ compilers

[A1]To see what the underlying compiler is, use `mpif90 -show`.

are found. When configured with MPI, the build will use `mpicc` (`mpicxx` for C++) located in the `bin` directory of the given MPI installation.[A2] However, the user can explicitly specify the desired C compiler via the `CC` environment variable, and/or the C++ compiler via the `CXX` environment variable.

To directly specify the compiler and linker options, use the `CFLAGS` (`CXXFLAGS` for C++) and `LDFLAGS` environment variables. The default behavior is to append their values to the options defined by CXXTune. If the `--disable-cxxtune` option is given to `configure`, CXXTune will be disabled and the values given by `FCFLAGS` and `LDFLAGS` will be used explicitly. For example, to ensure that the GNU C and C++ compilers `gcc` and `g++` are used with only the `-O3`, `-march=core-avx2`, and `-lm` options,

```
./configure --disable-cxxtune \
  CC=gcc \
  CXX=g++ \
  CFLAGS='-O3 -march=core-avx2' \
  CXXFLAGS='-O3 -march=core-avx2' \
  LDFLAGS='-lm'
```

The order of variables and options for the configure command are inconsequential, and single quotation marks (') are used to protect values with spaces from the shell. Some CXXTune options may be *unconditionally required* for a given compiler, as in the case of linking with the math library `-lm` above.

A.7 Support for Multiple Simultaneous Configurations

FUN3D supports the concurrent configuration of multiple variants within a single source tree. *This is the preferred method of configuration* and supports side-by-side building of differing executables (different enabled capabilities, compilers, MPI communication layers, etc.) using a single copy of the source. To exercise such, simply create a subdirectory within the top-level of the FUN3D source and, from within it, reference the configure script at the top-level.

```
mkdir _build-gfortran
cd _build-gfortran
../configure --prefix=$HOME/local/fun3d-gfortran FC=gfortran
make
make install
```

This will result in an executable with the specified capabilities, etc..

Note, it is recommended that you use this method on a fresh, unpolluted source tree. However, if you have previously configured FUN3D at the top-level, you will need execute `make distclean` at the top-level to remove the

[A2] To see what the underlying compiler is, use `mpicc -show` (`mpicxx -show` for C++).

products of `configure` and `make` before attempting to configure in a subdirectory.

A.8 Specialized AVX-512 Support

FUN3D provides specialized linear algebra routines based on the use of low-level intrinsics, which can outperform compiler-generated assembly code. This capability is only compatible with the Intel® compiler suite processors supporting the AVX-512 Advanced Vector Extensions SIMD instructions, e.g., Skylake and later for Xeon, Knights Landing for Xeon Phi. AVX-512 support is only available for the 3D calorically perfect gas path.

To enable AVX-512 optimizations, FUN3D should be configured with the option `--enable-avx512` and the following C++ compiler flags should be specified (Fortran compiler flags do not require modification),

```
./configure --enable-avx512 \
  CXXFLAGS='-O3 -xMIC-AVX512 -fno-alias -mcmodel=large -DL1P=1 \
            -DL2P=1 -std=c++0x -qno-opt-prefetch'
```

At runtime, the user must also set `use_vector_intrinsics = .true.` in the `&code_run_control` namelist (see section B.4.14).

A.9 GPU Support

FUN3D provides GPU acceleration of many common simulation options (see section 10.4). Currently, NVIDIA® GPUs are supported and section 10.3 recommends a minimum CUDA toolkit version for use with FUN3D. Earlier versions may work. Configure with the options

```
--with-cuda=/path/to/CUDA
--with-libfluda=/path/to/FLUDA
```

The CUDA directory `/path/to/CUDA` should contain the subdirectories `lib64`, `include`, and `bin`. The FLUDA directory `/path/to/FLUDA` is the directory to a precompiled binary of the FUN3D GPU library. It should contain subdirectories `lib` and `include`. Precompiled binaries for x86_64 are available in the FUN3D `fluda_binaries` folder.

A.10 Complex Variable Version

The FUN3D suite can be compiled with the real variables in the code replaced with complex variables by a source translation tool. This permits the computation of forward-mode sensitivities, see section 9.15 for details. To enable, add the `--enable-complex` configure option to the `configure` script. The complex-valued code can be compiled with `make complex`; and a `make install` will place the complex-valued executables in the `bin` installation directory. Enabling the complex variable version will increase the compile time.

A.11 SLAT

The Sparse Linear Algebra Toolkit [58] provides access to Krylov methods, preconditioning methods, and reordering methods. The –without-slat option will disable this library.

A.12 Yoga

FUN3D configure options for building with Yoga:

```
--enable-yoga
--with-nanoflann=/path/to/nanoflann
```

Yoga depends on the third party header-only library *nanoflann* [64] to build an approximate distance field for each component grid. *nanoflann* has a BSD license and can be obtained from GitHub [64]. The v1.3.0 of *nanoflann* is needed. Please check out this version from GitHub. Yoga is currently incompatible with the later versions.

A.13 External Libraries

Fun3D relies on external libraries to enable some of its advanced applications. Use Table A1 to determine which set of external libraries are necessary for your applications of interest. Discussions of each external library are found in the following sections.

It is highly recommended that Fun3D is configured to use a parallel execution (MPI and one of the partitioner libraries) if you plan to perform any advanced calculations. SUGGAR++ and DiRTlib are only required if overset (chimera) meshes will be used. The 6-DOF library is only required if six degrees of freedom simulations will be performed (trajectories determined by integrating the equation of motion). KSOPT, PORT, SNOPT™, NPSOL™, and DOT/BIGDOT™ are optimization libraries. At least one of these optimization libraries is required for performing design optimization.

A.13.1 MPI

MPI provides Fun3D's capability to communicate between processors. Configure with the option

```
--with-mpi=/path/to/MPI
```

Table A1: Configuration options.

Option	Parallel Execution	Overset Motion	Computed Trajectories	Unconstrained Design	Constrained Design	Binary Tecplot™ Output	CGNS	Sonic Boom Propagation	Mesh Deformation	Mesh Adaptation
MPI	x									
ParMETIS	x									
Zoltan	x									
SUGGAR		x								
DiRTlib		x								
6-DOF			x							
KSOPT					x					
PORT				x						
SNOPT™					x					
NPSOL™					x					
DOT/BIGDOT™					x					
Tecplot™						x				
CGNS							x			
sBOOM								x		
SPARSKIT									x	
ESP										x

where `/path/to/MPI` is the directory where MPI is installed.

In addition, FUN3D must be executed in an environment that represents the same MPI installation used for configuration/compilation (e.g. same mpiexec, mpirun, etc.). Failure to provide such consistency will result in undefined behavior and undetermined segmentation faults.

In some cases, MPI may already be installed on the target machine. If it is not, OpenMPI or MPICH can be used and the option of static MPI libraries is recommended. Also, if building OpenMPI or MPICH from source, it is important to maintain consistency with compilers (both vendor and version) throughout the build and execution of FUN3D and its dependent libraries. For example, if OpenMPI is built with gfortran version 4.4.7, then the execution environment for FUN3D must reflect the same vendor and version, gfortran 4.4.7. FUN3D execution must also employ the same `mpiexec` from the OpenMPI installation used to build the software.

Some high performance computing environments use a proprietary MPI implementation that does not provide `mpif90`. It that situation, the configure option `--without-mpif90` may be required in combination with the `FC` environment variable to explicitly set the compiler.

Verifying the MPI Implementation Functionality A simple Fortran program is included in the FUN3D distribution to verify that the MPI implementation is functional. This is very helpful for quickly troubleshooting issues with the MPI implementation. It is located in `utils/MPIcheck`. From within that directory you should be able to

```
mpif90 -DHAVE_MPI -o mpi_hello_world mpi_hello_world.F90
```

and execute on two processors

```
mpiexec -np 2 ./mpi_hello_world
     0 says, "Hello World!"  5 = 5
     1 says, "Hello World!"  5 = 5
```

To verify the Fortran compiler that MPI is built with, try

```
mpif90 -show
```

if the MPI implementation supports it.

A.13.2 ParMETIS

Website: http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview

The ParMETIS library performs domain decomposition to enable parallel execution of FUN3D. The license for ParMETIS is found at http://glaros.dtc.umn.edu/gkhome/metis/parmetis/download. It is critical that

FUN3D and ParMETIS are compiled with *exactly* the same MPI installation and compilers. This includes the C compiler used to compile MPI, ParMETIS, and FUN3D.

When configuring FUN3D, use

```
--with-parmetis=/path/to/ParMETIS
```

where `/path/to/ParMETIS` is the directory of the ParMETIS installation. FUN3D expects the `/path/to/ParMETIS` directory to contain the following files in `lib` and `include` subdirectories,

```
/path/to/ParMETIS/lib/libmetis.a
/path/to/ParMETIS/lib/libparmetis.a
/path/to/ParMETIS/include/metis.h
/path/to/ParMETIS/include/parmetis.h
```

These required libraries includes a version of the sequential execution tool METIS packaged with ParMETIS. This included version of METIS is required and must also be built. The included version is incompatible with the independently distributed METIS version.

See the `Install.txt` instructions in the ParMETIS distribution for build instructions. FUN3D requires both `libmetis.a` and `libparmetis.a` libraries and their accompanying header files. There is an example of commands to build both libraries,

```
cd parmetis-4.*
 make config prefix=/path/to/ParMETIS
 make install
 cd metis
  make config prefix=/path/to/ParMETIS
  make install
```

where `/path/to/ParMETIS` matches the FUN3D configure argument.

New Website: `https://github.com/KarypisLab/ParMETIS`

The distribution on ParMETIS has migrated to GitHub, although as of this writing, the version remains unchanged at 4.0.3. The ParMETIS license remains unchanged and is available at `https://github.com/KarypisLab/ParMETIS/blob/main/LICENSE`.

There are now three separate repositories required for the build and installation of ParMETIS: libGKlib, libmetis, libparmetis. FUN3D works with both distributions of ParMETIS in the same way, by specifying

```
--with-parmetis=/path/to/ParMETIS
```

where `/path/to/ParMETIS` is the directory of the ParMETIS installation.

The following provides an example of the commands to build all three libraries from the GitHub distribution.

```
mkdir ParMETIS
cd ParMETIS
  git clone https://github.com/KarypisLab/GKlib.git
  git clone https://github.com/KarypisLab/METIS.git
  git clone https://github.com/KarypisLab/ParMETIS.git
  set parmetis_path=/path/to/ParMETIS
  cd GKlib
    make config prefix=${parmetis_path} CFLAGS="-D_POSIX_C_SOURCE=199309L -fPIC"
    make -j && make install
  cd ../METIS/
    make config prefix=${parmetis_path} CFLAGS="-fPIC"
    make -j && make install
  cd ../ParMETIS/
    make config prefix=${parmetis_path} CFLAGS="-fPIC"
    make -j && make install
  cd ../..
```

where `/path/to/ParMETIS` matches the FUN3D configure argument.

A.13.3 Zoltan

Website: http://www.cs.sandia.gov/Zoltan/

The Zoltan library [65] performs domain decomposition to enable parallel execution of FUN3D. Note: FUN3D and Zoltan must be compiled with *exactly* the same MPI installation and compilers. This includes the C/Fortran compilers used to compile MPI, Zoltan, and FUN3D.

When configuring FUN3D, use

```
--with-zoltan=/path/to/Zoltan
```

where `/path/to/Zoltan` is the directory of the Zoltan installation. FUN3D expects the `/path/to/Zoltan` directory to contain the following files in `lib` and `include` subdirectories,

```
/path/to/Zoltan/lib/libzoltan.a
/path/to/Zoltan/include/zoltan.h
/path/to/Zoltan/include/zoltan.mod
```

The header file `zoltan.h` recursively includes a number of header files in the `include` path.

See the User's Guide on the Zoltan website for "Building the Zoltan Library" instructions. The Autotools stand-alone build environment is recommended over the CMake option for use with FUN3D. To use Zoltan with FUN3D, the Zoltan library *must* be built with Fortran support. The same C/Fortran compilers and MPI implementation used for FUN3D should be in your path. Here is an example of configuring and building Zoltan for use with FUN3D,

```
cd Zoltan_v3.*
 mkdir build
 cd build
  ../configure \
    --prefix=/path/to/Zoltan \
    --enable-mpi \
    --with-mpi-compilers \
    --enable-f90interface \
    --with-gnumake
  make everything
  make install
```

While the Zoltan installation procedure supports builds against available ParMETIS installations, FUN3D does not require this. If the user wishes to use ParMETIS for domain decomposition, FUN3D should be built directly with ParMETIS support (see section A.13.2).

A.13.4 SUGGAR++-1.0.10 or Higher

Website: `http://celeritassimtech.com`

SUGGAR++ is used for overset (chimera) applications and assembles composite meshes, cuts holes, determines interpolation coefficients, etc. If configuring with SUGGAR++, FUN3D must also be configured with DiRTlib v1.40 or higher.

SUGGAR++ may be compiled as a stand-alone executable and/or as a library. For static overset meshes you will need the stand-alone compilation; for moving body simulations you will need to compile both the stand-alone executable and the library. See the documentation that comes with SUGGAR++ for more information on how to compile the software.

When configuring FUN3D, use

```
--with-suggar=/path/to/SUGGAR++
```

where `/path/to/SUGGAR++` is the directory where SUGGAR++ library archive files (`.a` files) reside. In this directory, there must be an archive file called `libsuggar.a`, which is the serial compilation of SUGGAR++, and there must also be an archive file called `libsuggar_mpi.a`, which is the MPI compilation of SUGGAR++.

A.13.5 DiRTlib v1.40 or higher

Website: `http://celeritassimtech.com`

The DiRTlib library must be linked to FUN3D in order to use the overset connectivity data computed by SUGGAR++-1.0.10 or Higher. See the documentation that comes with DiRTlib for more information on how to compile the software.

When configuring FUN3D, use

```
--with-dirtlib=/path/to/DiRTlib
```

where `/path/to/DiRTlib` is the directory where DiRTlib library archive files (.a files) reside. In this directory, there must be an archive file called `libdirt.a`, which is the serial compilation of DiRTlib, and there must also be an archive file called `libdirt_mpich.a`, which is the MPI compilation of DiRTlib.

A.13.6 6-DOF

Contact: Nathan.C.Prewitt@erdc.dren.mil

The 6-DOF libraries provide trajectory tracing. When configuring FUN3D, use

```
--with-sixdof=/path/to/sixdof
```

where `/path/to/sixdof` is the directory where your 6-DOF installation resides.

A.13.7 KSOPT

Contact: Gregory.A.Wrenn@nasa.gov

The KSOPT [40] library is used for multi-objective and constrained FUN3D-based design optimization. If you configure FUN3D to link to KSOPT, you must use the Fortran 90 implementation of KSOPT with its object files gathered into a library called `libksopt.a`.

When configuring FUN3D, use

```
--with-KSOPT=/path/to/ksopt
```

where `/path/to/ksopt` is the directory where your KSOPT installation resides.

A.13.8 PORT

Website: `http://www.netlib.org/port`

The PORT library is used for unconstrained FUN3D-based design optimization. The Netlib site offers a tarball of the PORT library with a `Makefile`. Download the tarball from Netlib, but replace the original `Makefile` with the file included inside the FUN3D distribution as `Design/PORT.Makefile`. If you install both the PORT and NPSOL™ libraries, you may have to comment out low-level BLAS routines in one of the two packages because the linker will report the duplicate versions of these routines.

When configuring FUN3D, use

```
--with-PORT=/path/to/port
```

where `/path/to/port` is the directory where your PORT installation resides.

A.13.9 SNOPT™

Website: http://www.sbsi-sol-optimize.com

The SNOPT™ library is used for Fun3D-based design optimization. By default the SNOPT™ package builds a shared library. Either build SNOPT™ with the `--disable-shared` option, or add the the SNOPT™ install directory to your `LD_LIBRARY_PATH` environment variable to ensure Fun3D can find the shared library at run time.

When configuring Fun3D, use

```
--with-SNOPT=/path/to/snopt
```

where `/path/to/snopt` is the directory where your SNOPT™ installation resides.

A.13.10 NPSOL™

Website: http://www.sbsi-sol-optimize.com

The NPSOL™ library is used for constrained Fun3D-based design optimization. If you install both the PORT and NPSOL™ libraries, you may have to comment out low-level BLAS routines in one of the two packages because the linker will report the duplicate versions of these routines.

When configuring Fun3D, use

```
--with-NPSOL=/path/to/npsol
```

where `/path/to/npsol` is the directory where your NPSOL™ installation resides.

A.13.11 DOT/BIGDOT™

Website: http://www.vrand.com/products.html

The DOT/BIGDOT™ library is used for unconstrained or constrained Fun3D-based design optimization. When configuring Fun3D, use

```
--with-DOT=/path/to/dot
```

where `/path/to/dot` is the directory where your DOT/BIGDOT™ installation resides.

A.13.12 Tecplot™

Website: http://www.tecplot.com

By default, any Tecplot™ output generated from within the flow solver itself is written as a text file. If you have a copy of Tecplot™ library `tecio.a` (or `teciompi.a` for mpi versions) that allows for binary output,[A3] you may configure the Fun3D suite to use the library via:

```
--with-tecio=/path/to/tecio
```

With this option, Tecplot™ solution data written out from the flow solver will be in binary form. This results in smaller file sizes and faster importation into Tecplot™. The file extension is .plt for binary Tecplot™ 2013 format and earlier, and .szplt for new versions. If you have compiled against new standard TecIO, you can get .plt via the `--plt_tecplot_output` command line option. (This option does not work with TecIO-MPI.)

If you have compiled against the Tecplot™ `tecio` library, you can still request text output via the `--ascii_tecplot_output` command line option.

A.13.13 CGNS

Website: http://www.cgns.org

The CGNS library is used for working with files written in CGNS format. CGNS is a convention for writing machine-independent, self-descriptive data files for CFD and includes implementation software. Fun3D has the capability to translate and write CGNS files. The translation utilities are only compiled when CGNS is configured. Version 2.5 or greater of the CGNS library is required. To include CGNS, use

```
--with-CGNS=/path/to/cgns
```

where `/path/to/cgns` is the directory where the CGNS installation resides.

A.13.14 sBOOM

Contact: Sriram.Rallabhandi@NASA.gov

This package propagates a computed pressure signature to the ground for sonic boom simulations. Atmospheric variations are included, and an adjoint version is available for coupling into design and grid adaptation. sBOOM is distributed as a standalone executable or a static library. Fun3D is not able to interact with the standalone executable; the static library must be linked.

You may configure the Fun3D suite to use the library via:

```
--with-SBOOM=/path/to/sBOOM
```

where `/path/to/sBOOM` is the directory where the sBOOM installation resides.

[A3]The `tecio` library that was shipped with Tecplot360-2008 had a bug that will result in error messages when the binary files are written. You must get an updated version of the library.

A.13.15 SPARSKIT

Website: http://www-users.cs.umn.edu/~saad/software/SPARSKIT/index.html

The SPARSKIT library is a basic tool-kit for sparse-matrix computations. In particular, SPARSKIT contains an implementation of Yousef Saads' Generalized Minimum Residual (GMRES) method for solving asymmetric linear systems. GMRES is the recommended method for solving the linear elasticity equations when mesh deformation is used in FUN3D. If FUN3D is not configured with SPARSKIT, then the SLAT implementation of GMRES in FUN3D is the default solver for the linear elasticity equations, Version 2 of the SPARSKIT library is required. To include SPARSKIT, use

```
--with-SPARSKIT=/path/to/sparskit
```

where `/path/to/sparskit` is the directory where the SPARSKIT installation (libskit.a) resides.

A.13.16 ESP

Website: https://acdl.mit.edu/ESP/

The Engineering Sketch Pad (ESP) [16] is a geometry creation and manipulation system designed specifically to support the analysis and design of aerospace vehicles. While it is possible to compile ESP, users are *strongly* encouraged to install the pre-built binary distribution. ESP contains the Open-Source Constructive Solid Modeler (OpenCSM) [17] for the development and import of geometry models. REFINE links to the Engineering Geometry Aerospace Design System (EGADS) [18] and EGADSlite [19] components of ESP. EGADS has a dependency on OpenCASCADE, and a "hardened" version of OpenCASCADE is included in the pre-built binary distribution. EGADS is required to bootstrap the mesh adaptation process and EGADSlite allows efficient evaluation of the geometry model on high-performance computing systems. You may configure the Fun3D suite to use ESP when building REFINE via:

```
--with-EGADS=/path/to/ESP/EngSketchPad
--with-OpenCASCADE=/path/to/ESP/OpenCASCADE
```

where `EngSketchPad` and `OpenCASCADE` are directories in the ESP distribution. The `OpenCASCADE` path may include a version (e.g., `OpenCASCADE-7.4.1`).

Appendix B

FUN3D Input Files

There are a variety of input files necessary for the various codes that make up the FUN3D suite. Table B1 lists frequently used input files with a short description. This chapter will describe the basic formats of each of these files and meaning of the specific inputs they contain.

Table B1: FUN3D input files.

File	Description
`stop.dat`	signals a change to the number of iterations / time steps that were requested at the beginning of the run
`remove_boundaries_from_force_totals`	omits boundary faces from total force integration
`[project_rootname].flow`*	flow field solution
`fun3d.nml`	primary Fortran namelist (required)
`moving_body.input`	body motion Fortran namelist
`rotor.input`	describes the rotor actuator disk model
`tdata`	specifies the generic gas model
`kinetic_data`	specifies the possible chemical reactions in the generic gas model
`species_transp_data`	specifies generic gas model species collision cross sections
`species_transp_data_0`	specifies a higher-order generic gas model species collision cross sections
`hara_namelist_data`	controls the radiation models used by the HARA library
`boundary_condition_control`	controls select inflow and outflow boundary condition parameters

* The `[project_rootname]` is a `&project` namelist variable, see section B.4.1.

FUN3D utilizes Fortran namelists for a large portion of input specification because it is defined in the Fortran 90 standard. With all Fortran namelists, leaving out or misspelling any namelist (defined with an ampersand preceding its name) will result in default values being used for all of the parameters within that namelist. For example, if the namelist name **`linear_solver_parameters`** were to be misspelled as **`linear_solver_parameter`** (missing **`s`**), then all parameters within that namelist would be ignored and retain their default values. Leaving out any parameter within a namelist results in the default value for that parameter being used. Misspelling or misusing any particular parameter will typically cause FUN3D to issue an error and stop. While an effort is made to keep the namelists backwards compatible, namelists and namelist entries may be modified or deleted over time. See the release notes on the FUN3D website for entries that have been modified, deleted, or marked for deprecation.

B.1 stop.dat

This optional file either halts or extends the execution of the solver. The `stop.dat` plain text file contains a single integer. After every iteration, the solver will check to see if this file exists. If the file is found in the directory that the solver was invoked, the integer is read and a message is printed to the standard output stream of the form, "stop.dat file found, user requested stop at cycle N." If the integer is greater than zero and less than or equal to the current iteration, the solver will write the current solution, delete the `stop.dat` file, and halt execution. If the integer is less than zero, the solver will *not* write the restart file, but will delete the `stop.dat` file and halt execution. If the integer is zero, `stop.dat` will be ignored.

The integer is with respect to the `steps` variable in the `&code_run_control` namelist and not with respect to a cumulative number of steps that may result from a restarted run. The value of the integer may be greater than the number of `steps` specified in `fun3d.nml`'s `&code_run_control` namelist, so that `stop.dat` can be used to extend the execution of the solver. Some environments, especially ones with network-mounted filesystems (e.g., NFS), may exhibit a delay in the `stop.dat` file being read or being deleted.

B.2 [project_rootname].flow

The optional `[project_rootname].flow` binary file contains flow solution and checkpoint information. The `[project_rootname]` is a `&project` namelist variable, see section B.4.1. This file is read by the solver to restart computations from a previously computed flow solution. The contents vary due to the checkpoint requirements of the simulation. The file contains a minimum of the current solution and convergence history. It can also contain working variables for the turbulence model, solutions from previous iterations for time accurate cases, or previous grid positions and velocities for deforming grids.

B.3 remove_boundaries_from_force_totals

The optional `remove_boundaries_from_force_totals` file is for specifying boundaries that are *not* to be included in the calculation of force and moment totals. This file is useful, for example, in situations where there may be a mounting sting on a wind tunnel model, but only the forces on the model are actually of interest. The forces on the specified boundaries are still computed and appear in the `[project_rootname].forces` file. However, they are not included in the totals. The position of the text lines in this file is significant. So, follow this template carefully:

```
Remove selected boundaries from the total forces
Number of boundaries to turn off
2
Boundaries to turn off (boundary lumping changes indexes)
12
15
```

The third line is the number of boundaries to exclude. The fifth and subsequent lines are the patch indexes of the excluded boundaries.

B.4 `fun3d.nml`

The main input namelist file, `fun3d.nml`, is described in detail below, with defaults listed before the descriptions. The namelist file contains a large number of input variables. In general, it is not necessary to specify them all because they have suitable default values. Only those variables that are *different* from the defaults need to be specified. An overview of tasks and their associated namelists are listed below.

The project name and grid information:

&project (B.4.1)

&raw_grid (B.4.2)

&force_moment_integ_properties (B.4.3)

&grid_transform (B.4.4)

&body_transform (B.4.5)

The equation set and reference conditions:

&governing_equations (B.4.6)

&reference_physical_properties (B.4.7)

&noninertial_reference_frame (B.4.8)

The inviscid flux discretization:

&inviscid_flux_method (B.4.9)

Turbulence model:

&turbulent_diffusion_models (B.4.10)

&turbulence (B.4.11)

&spalart (B.4.12)

&gammaretsst (B.4.13)

Number of and size of time steps or steady iterations:

&code_run_control (B.4.14)

&nonlinear_solver_parameters (B.4.15)

&hanim (B.4.16)

Linear relaxation controls:

&linear_solver_parameters (B.4.19)

&elasticity_gmres (B.4.20)

Boundary conditions and transition:

&boundary_conditions (B.4.21)

&periodicity (B.4.22)

&two_d_trans (B.4.23)

&three_d_trans (B.4.24)

&filters (B.4.25)

Flowfield initialization:

&flow_initialization (B.4.26)

Force tracking and visualization:

&component_parameters (B.4.27)

&time_avg_params (B.4.28)

&global (B.4.29)

&volume_output_variables (B.4.30)

&boundary_output_variables (B.4.31)

&sampling_output_variables (B.4.32)

&sampling_parameters (B.4.33)

&spatial_avg (B.4.34)

&slice_data (B.4.35)

Overset grid systems and rotorcraft simulation:

&overset_data (B.4.36)

&rotor_data (B.4.37)

Grid adaptation:

&adapt_metric_construction (B.4.38)

&adapt_mechanics (B.4.39)

Design optimization cost functions:

&massoud_output (B.4.40)

&sonic_boom (B.4.41)

&sboom (B.4.42)

&equivalent_area (B.4.43)

&press_box_function (B.4.44)

&pstag_function (B.4.45)

&fan_distortion (B.4.46)

Other:

&special_parameters (B.4.47)

&partitioning (B.4.48)

&fwh_acoustic_data (B.4.49)

&vortex_generator (B.4.50)

&gust_data (B.4.51)

&gpu_support (B.4.52)

B.4.1 &project

This namelist allows the user to specify the rootname of the project, which forms the majority of input and output filenames.

```
&project
  project_rootname = 'default_project'
/
```

project_rootname = 'default_project'

The project rootname is the root for the grid, restart, and visualization files. The manual refers to it as `[project_rootname]`. The `'default_project'` can be replaced with any filename allowed by the file system.

B.4.2 &raw_grid

This namelist specifies details of the grid format.

```
&raw_grid
  grid_format                     = 'vgrid'
  data_format                     = 'default'
  twod_mode                       = .false.
  y_coplanar_tol                  = 1.0e-11
  swap_yz_axes                    = .false.
  fieldview_coordinate_precision = 'double'
  patch_lumping                   = 'none'
  ignore_euler_number             = .false.
  mesh_diagnostics                = .false.
/
```

`grid_format = 'vgrid'`

This specifies the grid file format. See section 4 for the details on these formats. The currently supported values are:

`'fast'` for FAST .fgrid/.mapbc files.

`'vgrid'` for single- and multisegmented VGRID .cogsg/.bc/.mapbc files.

`'fun2d'` for Fun2D.faces files.

`'aflr3'` for AFLR3 formatted, unformatted, or C-binary/Fortran-stream files.

`'felisa'` for Felisa grid files.

`'fieldview'` for FieldView formatted or unformatted .fvgrid_fmt/.fvgrid_unf/.mapbc files.

`data_format = 'default'`

This provides the encoding of the grid file. A particular `grid_format` may only support a subset of encodings. Fun3D will stop with an error message if the `data_format` is inconsistent with the `grid_format`. The `'default'` value is changed to an admissible value based on `grid_format` as noted next to each value,

`'ascii'` ASCII text grid file. It is the default for `'felisa'` and `'fun2d'` grids.

`'unformatted'` Fortran unformatted grid file. It is the default for `'fast'`, `'vgrid'`, and `'fieldview'` grids.

`'stream'` C-binary/Fortran-stream grid file. It is the default for `'aflr3'` grids.

`'stream64'` 64-bit integer C-binary/Fortran-stream grid file.

`twod_mode = .false.`

Turns on two-dimensional mode for a single layer prism or hex grid. If `grid_format = 'fun2d'`, `twod_mode` is automatically `.true.`. Note that if `grid_format` is not `'fun2d'`, then the grid *must* have two distinct boundaries declared to be y-symmetry boundaries. A grid with both symmetry planes lumped into one boundary will not work.

`y_coplanar_tol = 1.0e-11`

This is the tolerance used to check whether points lie on a plane when `twod_mode = .true.`. If problems are encountered running a mesh in two-dimensional mode, *and* the mesh has two distinct y-symmetry boundaries as noted in `twod_mode`, try increasing this tolerance.

`swap_yz_axes = .false.`

When `.true.`, this swaps the y- and z-axes for the grid. This option can be used to rotate the grid so the z-axes is in the FUN3D convention for angle of attack and lift—see section 2. Note: The boundary conditions are applied after the rotation, which implies that symmetry and other boundary conditions should be specified for the boundary orientation of the swapped axes.

`fieldview_coordinate_precision = 'double'`

This specifies floating point precision of reals for FieldView meshes only.

`'double'` for double precision coordinates.

`'single'` for single precision coordinates.

`patch_lumping = 'none'`

This enables boundary patch lumping. It combines the grid patches into fewer patches to ease the bookkeeping of patch groups, but will effect all features that reference boundary patch numbers (e.g., `&boundary_conditions`). The .mapbc files for any of the supported grid formats may contain an optional third column of data, which specifies a family name. The exception is the VGRID.mapbc file, where the family name is mandatory and appears in the sixth column. If family names are not present in the .mapbc file, `patch_lumping` can not be `family`.

`'none'` for no patch lumping.

`'bc'` for physical boundary condition lumping.

`'family'` for family name lumping

`ignore_euler_number = .false.`

This will permit the use of grids with a failing Euler number check. See section C.9 for a description of the Euler number and its implications. Ignoring the Euler number check is not recommended.

`mesh_diagnostics = .false.`

When `.true.`, this option activates analysis of mesh elements geometry and how the shape or configuration of mesh elements could impact reconstructed gradient accuracy.

B.4.3 &force_moment_integ_properties

Reference lengths and area are defined in this namelist to scale aerodynamic force and moment data. These data are reported in the `[project].forces` file, relative to the grid Cartesian axis system. Optionally, a 'body-axis' coordinate system may be defined in which aerodynamic force and moment data are also computed. If this optional system is defined, aerodynamic force and moment data relative to this body-axis system is reported in a secondary forces file, `[project]_body_frame.forces`. In both the Cartesian axis system and the optional body-axis system, moments are computed about the moment center specified via this namelist, i.e. the origin of the optional body-axis system is taken as the specified moment center. The optional body axes may be defined by *one* of the following methods: 1) a rotation vector (single angle, plus direction via a unit Cartesian vector), or 2) Euler angle rotations (pitch, roll, yaw), or 3) an input 4x4 transform matrix.

```
&force_moment_integ_properties
  area_reference            = 1.0
  x_moment_length           = 1.0
  y_moment_length           = 1.0
  x_moment_center           = 0.0
  y_moment_center           = 0.0
  z_moment_center           = 0.0
  body_axes_theta           = 0.0
  body_axes_tx              = 1.0
  body_axes_ty              = 0.0
  body_axes_tz              = 0.0
  body_axes_x0              = x_moment_center
  body_axes_y0              = y_moment_center
  body_axes_z0              = z_moment_center
  euler_angle_order         = 'ypr'
  body_axes_yaw             = 0.0
  body_axes_pitch           = 0.0
  body_axes_roll            = 0.0
  echo_body_axes_transform  = .false.
  body_axes_transform(1,1:4) = 1.0, 0.0, 0.0, 0.0
  body_axes_transform(2,1:4) = 0.0, 1.0, 0.0, 0.0
  body_axes_transform(3,1:4) = 0.0, 0.0, 1.0, 0.0
  body_axes_transform(4,1:4) = 0.0, 0.0, 0.0, 1.0
/
```

`area_reference = 1.0`

This area is used for nondimensionalization of forces and moments, specified in grid units squared. For a semi-span model, use half of the full configuration reference area.

`x_moment_length = 1.0`

This length in x-direction is used to nondimensionalize moments about y (pitching moment), specified in grid units.

`y_moment_length = 1.0`

This length in y-direction is used to nondimensionalize moments about x (rolling moment) and z (yawing moment), specified in grid units.

`x_moment_center = 0.0`

This specifies the x-coordinate location of moment center, in grid units.

`y_moment_center = 0.0`

This specifies the y-coordinate location of moment center, in grid units.

`z_moment_center = 0.0`

This specifies the z-coordinate location of moment center, in grid units.

`body_axes_theta = 0.0`

This is the body axes rotation angle (in degrees). Used when specifying a rotation vector. A positive rotation is applied by the right hand rule with the thumb pointing in direction of the rotation axis. Used in conjunction with `body_axes_tx, body_axes_ty, body_axes_tz`.

`body_axes_tx = 1.0`

This is the x-component of the body axes rotation unit vector. Used when specifying a rotation vector.

`body_axes_ty = 0.0`

This is the y-component of the body axes rotation unit vector. Used when specifying a rotation vector.

`body_axes_tz = 0.0`

This is the z-component of the body axes rotation unit vector. Used when specifying a rotation vector.

`body_axes_x0 = x_moment_center`

This is the x-component of the body axes origin

`body_axes_y0 = y_moment_center`

This is the y-component of the body axes origin

`body_axes_z0 = z_moment_center`

This is the z-component of the body axes origin `euler_angle_order = 'ypr'`

This is the order of application of the Euler angles; each successive transform is applied to the axis system resulting from the previous transform. There are six possibilities:

`'ypr'` first yaw about the z-axis, then pitch about the y'-axis, then roll about the x" axis (primes denote new axes after previous rotation(s)). Right hand rule applies to rotations.

`'rpy'` first roll about the x-axis, then pitch about the y'-axis, then yaw about the z" axis

`'pyr'` first pitch about the y-axis, then yaw about the z'-axis, then roll about the x" axis

`'ryp'` first roll about the x-axis, then yaw about the z'-axis, then pitch about the y" axis

`'pry'` first pitch about the y-axis, then roll about the x'-axis, then yaw about the z" axis

`'yrp'` first yaw about the z-axis, then roll about the x'-axis, then pitch about the y" axis

`body_axes_yaw = 0.0`

This is the yaw angle of the body axes (in degrees).

`body_axes_pitch = 0.0`

This is the pitch angle of the body axes(in degrees).

`body_axes_roll = 0.0`

This is the roll angle of the body axes (in degrees).

`echo_body_axes_transform = .false.`

This is a flag to write the final transform(s) (and inverse) to the screen. Primarily a developer tool, but potentially useful for setup debugging.

`body_axes_transform(1,1:4) = 1.0, 0.0, 0.0, 0.0`

This is a 4x4 transform matrix; user must ensure the transform provides rotations about the moment center specified via this namelist.

B.4.4 &grid_transform

This namelist defines a constant grid translation or rotation that is applied before the start of the flow solution. For example, the original grid may be rotated to position the geometry at a different angle of attack. Rotation of the grid may be accomplished by specifying *either* a rotation vector (single angle, plus direction via a unit Cartesian vector), or three Euler angle rotations (pitch, roll, yaw). Rotation(s) are applied first, followed by translation. The translation and rotation input have limited capability to reposition the mesh; if a more complex repositioning is required, input an appropriate transform matrix instead of the simple translation or rotation parameters. A scale factor (default 1.0) is applied as the last step to determine the final transform matrix applied to the grid. Note that if `&grid_transform` is used in conjunction with `&body_transform`, the transforms specified by `&grid_transform` are applied first.

```
&grid_transform
  n_transforms         = 0
  mesh_id(:)           = 0
  body_name(:)         = ''
  ds(:)                = 0.0
  sx(:)                = 1.0
  sy(:)                = 0.0
  sz(:)                = 0.0
  theta(:)             = 0.0
  tx(:)                = 1.0
  ty(:)                = 0.0
  tz(:)                = 0.0
  x0(:)                = 0.0
  y0(:)                = 0.0
  z0(:)                = 0.0
  yaw(:)               = 0.0
  pitch(:)             = 0.0
  roll(:)              = 0.0
  euler_angle_order(:) = 'ypr'
  scale(:)             = 1.0
  echo_transform       = .false.
  transform(1,1:4,1)   = 1.0, 0.0, 0.0, 0.0
  transform(2,1:4,1)   = 0.0, 1.0, 0.0, 0.0
  transform(3,1:4,1)   = 0.0, 0.0, 1.0, 0.0
  transform(4,1:4,1)   = 0.0, 0.0, 0.0, 1.0
/
```

`n_transforms = 0`

Number of static transforms to be applied; *unless the mesh is overset*, `n_transforms > 1` *is not allowed.*

`mesh_id(:) = 0`

This selects the *component* mesh within the *composite* mesh to apply the transform to. For non-overset cases, there is only one component, and the terms 'component' and 'composite' are synonymous. Use the default for non-overset cases. A value of `mesh_id > 0` is valid only if `overset_flag = .true.`, and `mesh_id` must be unique for each of the `n_transforms` specified - multiple transforms cannot be applied to the same component mesh. For more complex transforms, input the `transform` explicitly.

`body_name(:) = ''`

Specifies a body name to associate with this transform; needed only for overset meshes.

`ds(:) = 0.0`

This is the translation distance; translation is applied after rotation.

`sx(:) = 1.0`

This is the x-component of a unit vector in the translation direction.

`sy(:) = 0.0`

This is the y-component of a unit vector in the translation direction.

`sz(:) = 0.0`

This is the z-component of a unit vector in the translation direction.

`theta(:) = 0.0`

This is the rotation angle (in degrees). Used when specifying a rotation vector. A positive rotation is applied by the right hand rule with the thumb pointing in direction of rotation axis. Rotation is applied before translation.

`tx(:) = 1.0`

This is the x-component of the rotation axis unit vector. Used when specifying a rotation vector.

`ty(:) = 0.0`

This is the y-component of the rotation axis unit vector. Used when specifying a rotation vector.

`tz(:) = 0.0`

This is the z-component of the rotation axis unit vector. Used when specifying a rotation vector.

`x0(:) = 0.0`

This is the x-coordinate of the rotation origin. Used when specifying a rotation vector or a set of Euler angles.

`y0(:) = 0.0`

This is the y-coordinate of the rotation origin. Used when specifying a rotation vector or a set of Euler angles.

`z0(:) = 0.0`

This is the z-coordinate of the rotation origin. Used when specifying a rotation vector or a set of Euler angles.

`yaw(:) = 0.0`

This is the yaw Euler angle (in degrees). Euler angle rotations are applied before translation.

`pitch(:) = 0.0`

This is the pitch Euler angle (in degrees).

`roll(:) = 0.0`

This is the roll Euler angle (in degrees).

`euler_angle_order(:) = 'ypr'`

This is the order of application of the Euler angles; each successive transform is applied to the axis system resulting from the previous transform. There are six possibilities:

`'ypr'` first yaw about the z-axis, then pitch about the y'-axis, then roll about the x" axis (primes denote new axes after previous rotation(s)). Right hand rule applies to rotations.

`'rpy'` first roll about the x-axis, then pitch about the y'-axis, then yaw about the z" axis

`'pyr'` first pitch about the y-axis, then yaw about the z'-axis, then roll about the x" axis

`'ryp'` first roll about the x-axis, then yaw about the z'-axis, then pitch about the y" axis

`'pry'` first pitch about the y-axis, then roll about the x'-axis, then yaw about the z" axis

`'yrp'` first yaw about the z-axis, then roll about the x'-axis, then pitch about the y" axis

`scale(:) = 1.0`

This is a scale factor applied to all coordinate values.

`echo_transform = .false.`

This is a flag to write the final transform(s) (and inverse) to the screen. Primarily a developer tool, but potentially useful for setup debugging.

`transform(1,1:4,1) = 1.0, 0.0, 0.0, 0.0`

This is a 4x4 transform matrix (see for example [66]). Do not include a scale factor in the transform matrix; if scaling is desired, input the desired scaling using scale(:), as above.

B.4.5 &body_transform

This namelist defines a constant body translation or rotation that is applied before the start of the flow solution. A body is defined by the user to be comprised of one or more boundaries in the mesh. For example, the original body may be rotated to position the geometry at a different effective angle of attack. Rotation of the body may be accomplished by specifying *either* a rotation vector (single angle, plus direction via a unit Cartesian vector), or three Euler angle rotations (pitch, roll, yaw). Rotation(s) are applied first, followed by translation. The translation and rotation input have limited capability to reposition the body; if a more complex repositioning is required, input an appropriate transform matrix instead of the simple translation or rotation parameters. Note that if `&body_transform` is used in conjunction with `&grid_transform`, the transforms specified by `&grid_transform` are applied first. After repositioning the body, the mesh is deformed once to reflect the new position of the body, before beginning the solution. Body translations and rotations are thereby restricted to ranges that will result in a deformed mesh with positive volumes - case dependent.

```
&body_transform
  n_bodies            = 0
  mesh_id(:)          = 0
  nbndry(:)           = 0
  boundary_list(:)    = ''
  ds(:)               = 0.0
  sx(:)               = 1.0
  sy(:)               = 0.0
  sz(:)               = 0.0
  theta(:)            = 0.0
  tx(:)               = 1.0
  ty(:)               = 0.0
  tz(:)               = 0.0
  x0(:)               = 0.0
  y0(:)               = 0.0
  z0(:)               = 0.0
  yaw(:)              = 0.0
  pitch(:)            = 0.0
  roll(:)             = 0.0
  euler_angle_order(:) = 'ypr'
  echo_transform       = .false.
  transform(1,1:4,1)   = 1.0, 0.0, 0.0, 0.0
  transform(2,1:4,1)   = 0.0, 1.0, 0.0, 0.0
  transform(3,1:4,1)   = 0.0, 0.0, 1.0, 0.0
  transform(4,1:4,1)   = 0.0, 0.0, 0.0, 1.0
/
```

n_bodies = 0

This is the number of bodies that are to be repositioned at the start of the computation.

mesh_id(:) = 0

This selects the *component* mesh within the *composite* mesh that the body belongs to. For non-overset cases, there is only one component, and the terms 'component' and 'composite' are synonymous. Use the default for non-overset cases. A value of `mesh_id > 0` is valid only if `overset_flag = .true.`. The value of `mesh_id` is used to determine if this body should inherit a static grid transform from a transform specified in the `&grid_transform` namelist, which has the same value of `mesh_id`.

nbndry(:) = 0

This is the number of boundary patches listed for a given body.

boundary_list(:) = ''

This is a list of boundary patch numbers for a given body. Commas and dashes can be used to specify ranges, i.e., `'1,2,5-7'`.

ds(:) = 0.0

This is the translation distance.

sx(:) = 1.0

This is the x-component of a unit vector in the translation direction.

sy(:) = 0.0

This is the y-component of a unit vector in the translation direction.

sz(:) = 0.0

This is the z-component of a unit vector in the translation direction.

theta(:) = 0.0

This is the rotation angle (in degrees). A positive rotation is applied by the right hand rule with the thumb pointing in direction of rotation axis. Used when specifying a rotation vector.

tx(:) = 1.0

This is the x-component of the rotation axis unit vector. Used when specifying a rotation vector.

ty(:) = 0.0

This is the y-component of the rotation axis unit vector. Used when specifying a rotation vector.

`tz(:) = 0.0`

This is the z-component of the rotation axis unit vector. Used when specifying a rotation vector.

`x0(:) = 0.0`

This is the x-coordinate of the rotation origin. Used when specifying a rotation vector or a set of Euler angles.

`y0(:) = 0.0`

This is the y-coordinate of the rotation origin. Used when specifying a rotation vector or a set of Euler angles.

`z0(:) = 0.0`

This is the z-coordinate of the rotation origin. Used when specifying a rotation vector or a set of Euler angles.

`yaw(:) = 0.0`

This is the yaw Euler angle (in degrees). Euler angle rotations are applied before translation.

`pitch(:) = 0.0`

This is the pitch Euler angle (in degrees).

`roll(:) = 0.0`

This is the roll Euler angle (in degrees).

`euler_angle_order(:) = 'ypr'`

This is the order of application of the Euler angles; each successive transform is applied to the axis system resulting from the previous transform. There are six possibilities:

`'ypr'` first yaw about the z-axis, then pitch about the y'-axis, then roll about the x" axis (primes denote new axes after previous rotation(s)). Right hand rule applies to rotations.

`'rpy'` first roll about the x-axis, then pitch about the y'-axis, then yaw about the z" axis

`'pyr'` first pitch about the y-axis, then yaw about the z'-axis, then roll about the x" axis

`'ryp'` first roll about the x-axis, then yaw about the z'-axis, then pitch about the y" axis

`'pry'` first pitch about the y-axis, then roll about the x'-axis, then yaw about the z" axis `‘yrp’` first yaw about the z-axis, then roll about the x’-axis, then pitch about the y” axis

`echo_transform = .false.`

This is a flag to write the final transform(s) (and inverse) to the screen. Primarily a developer tool, but potentially useful for setup debugging.

`transform(1,1:4,1) = 1.0, 0.0, 0.0, 0.0`

This is a 4x4 transform matrix (see for example [66]).

B.4.6 &governing_equations

This namelist specifies the equation set that describes underlying physics of the problem.

```
&governing_equations
  eqn_type                  = 'compressible'
  artificial_compress       = 15.0
  viscous_terms             = 'turbulent'
  chemical_kinetics         = 'finite-rate'
  thermal_energy_model      = 'non-equilib'
  prandtlnumber_molecular = 0.72
  schmidt_number            = -1.
  gas_radiation             = 'off'
  rad_use_impl_lines        = .false.
  multi_component_diff      = .false.
  flow_solver               = 'fun3d'
/
```

`eqn_type = 'compressible'`

This specifies the set of governing equations to be solved.

`'compressible'` for compressible, calorically perfect gas. See section 2.1 for equations and nondimensionalization.

`'incompressible'` for incompressible, calorically perfect gas. [67] See section 2.2 for equations and nondimensionalization. The incompressible solution is affected by the choice of `artificial_compress`, see the description of this parameter for details.

`'generic'` for multispecies, reacting gas simulations. See section 2.3 for nondimensionalization. The `tdata` input file is required.

`artificial_compress = 15.0`

This is the artificial compressibility factor, β, which is only used by the `eqn_type = 'incompressible'`. This parameter must be in the range of $(100, 1)$. See Anderson, Rausch, and Bonhaus [67] for details. The sensitivity of the solution to this parameter will decrease with mesh refinement, so consider a refined grid if an unacceptable amount of sensitivity is experienced. A high sensitivity to this parameter can also indicate that the problem is actually compressible and the user is encouraged to check the incompressible solution by performing a low Mach compressible simulation.

`viscous_terms = 'turbulent'`

This describes the modeling of the viscosity term in the governing equations.

`‘inviscid’` no viscosity, for inviscid flow.

`‘laminar’` apply laminar viscosity, to model laminar flow.

`‘turbulent’` include laminar viscosity and model turbulent flow with a `&turbulent_diffusion_models`.

<u>`chemical_kinetics = ’finite-rate’`</u>

This describes the chemical kinetics, only used when `eqn_type = ’generic’`.

`‘frozen’` for frozen chemical compositions.

`‘finite-rate’` for finite-rate reacting gases.

<u>`thermal_energy_model = ’non-equilib’`</u>

This describes the thermal energy model, only used when `eqn_type = ’generic’`.

`‘frozen’` for frozen chemical compositions.

`‘non-equilib’` for nonequilibrium gases.

<u>`prandtlnumber_molecular = 0.72`</u>

This is the molecular Prandtl number. It must be greater than zero.

<u>`schmidt_number = -1.`</u>

This is the Schmidt number used in the generic gas path. If the user wants to override the default path of computing a variable Schmidt number from collision cross sections then use this parameter to specify the constant Schmidt number.

<u>`gas_radiation = ’off’`</u>

This controls flow field radiation coupling. When active, this option will compute radiation source terms and surface heat fluxes after the first time step. Radiation source terms are not further updated during the rest of the time steps. Radiation source terms are not stored in the restart file, so they need to be recalculated when restarting simulations that include flow field radiation. Only for `eqn_type = ’generic’`.

`‘off’` no radiation calculations.

`‘uncoupled’` will use the HARA program to compute radiative surface heat fluxes, but radiation source terms would not be included in the flow-field governing equations.

`‘coupled’` will include radiation source terms in the governing equations, with the divergence of the radiative flux being computed by the HARA program. Requires `rad_use_impl_lines = .true.`; only valid if all of the domain nodes are included in one and only one line as defined by the implicit lines file.

`rad_use_impl_lines = .false.`

For `gas_radiation`, the mesh nodes in the lines of sight are read from the implicit lines file. Coupled radiation must use this option. For uncoupled radiation, the lines of sight can either be read from the implicit lines file when `.true.`, or the lines of sight will be generated during the FUN3D run when `.false.` Only for `eqn_type = 'generic'`.

`multi_component_diff = .false.`

When `.true.`, engage multicomponent diffusion using subiteration of Stefan-Maxwell Equations as described by Sutton and Gnoffo [68]. Otherwise, use binary diffusion with mass fraction averaged correction to force sum of diffusion flux to equal zero. Only for `eqn_type = 'generic'`.

`flow_solver = 'fun3d'`

This defines the name of the flow solver plug-in to use. The default value is 'fun3d' which represents the internal flow solver. This allows for an alternative flow solver plug-in to be run within the FUN3D framework.

B.4.7 &reference_physical_properties

This namelist is used to specify reference conditions and nominal freestream flow conditions in a user-defined unit system. It is also used to convert between grid units and flow solver units.

```
&reference_physical_properties
  dim_input_type        = 'nondimensional'
  gridlength_conversion = 1.0
  mach_number           = 0.0
  vinf_ratio            = 1.0
  reynolds_number       = 0.0
  velocity              = 0.0
  density               = 0.0
  temperature           = 273.0
  temperature_units     = 'Kelvin'
  angle_of_attack       = 0.0
  angle_of_yaw          = 0.0
  gamma                 = 1.4
  molecular_weight      = molecular_weight_air
  sutherland_constant   = -1.0
/
```

`dim_input_type = 'nondimensional'`

This is the system of measurement for the reference conditions. Currently, it must be `'dimensional-SI'` for `eqn_type = 'generic'` and `'nondimensional'` otherwise. This input is intended for future expansion. The temperature is always input as a dimensional quantity.

`'nondimensional'` requires `mach_number` and `reynolds_number` to be defined.

`'dimensional-SI'` requires dimensional `velocity` and `density` to be defined.

`gridlength_conversion = 1.0`

For `dim_input_type = 'dimensional-SI'`, this is the conversion factor to scale the grid and it should be set to meters per grid unit. It is used for providing heat flux in proper units and other tasks. For `dim_input_type = 'nondimensional'`, this should be set to 1.0, because the grid is already in nondimensional grid units.

`mach_number = 0.0`

This is the reference Mach number defined as velocity/speed-of-sound. It is only allowed for `dim_input_type` = `'nondimensional'` and `eqn_type = 'compressible'`. It must be set to a positive value.

`vinf_ratio = 1.0`

This is a multiplicative factor that is applied to the reference velocity. The effective freestream velocity for the simulation is then `vinf_ratio` $\times$ `reference_velocity`. The default value of `vinf_ratio` gives a freestream velocity identical to the reference velocity. Note that for `eqn_type = 'compressible'`, the reference velocity has magnitude `mach_number`, while for `eqn_type = 'incompressible'`, the reference velocity has magnitude 1. For either `eqn_type`, a value of `vinf_ratio` different from unity allows distinct freestream and reference conditions. This is often used, for example, in rotorcraft simulations where the tip Mach number is taken as the reference Mach number, while the freestream Mach number reflects the forward speed. `vinf_ratio` is not applicable to `eqn_type = 'generic'`

`reynolds_number = 0.0`

This is the reference Reynolds number, per one unit of the grid. Not correctly accounting for the unit of the grid has been a point of confusion in the past. For example, when the grid units are feet, Reynolds number should be specified per foot. This input is only used if `dim_input_type = 'nondimensional'` and is ignored by `eqn_type = 'generic'`. It must be set to a positive value.

`velocity = 0.0`

This is the reference velocity, in m/s. Only used for `dim_input_type` = `'dimensional-SI'` and `eqn_type = 'generic'`.

`density = 0.0`

This is the reference density, in kg/m^3. Only used for `dim_input_type` = `'dimensional-SI'` and `eqn_type = 'generic'`.

`temperature = 273.0`

This is the reference temperature, in units of `temperature_units`.

`temperature_units = 'Kelvin'`

The units used to specify `temperature`.

`angle_of_attack = 0.0`

This is the freestream angle of attack in degrees.

`angle_of_yaw = 0.0`

This is the freestream angle of yaw (side-slip) in degrees.

`gamma = 1.4`

This is the ratio of specific heats for the reference fluid. It is only allowed for `eqn_type = 'compressible'`. It must be set to a positive value. The default value is for a calorically perfect gas, `1.4`.

`molecular_weight = molecular_weight_air`

This is the ratio of average molecular weight of the fluid. It is only allowed for `eqn_type = 'compressible'`. It must be set to a positive value. The default value is the Standard Atmosphere reference for air, `28.964 kg/kmol`.

`sutherland_constant = -1.0`

This is the Sutherland constant in degrees Rankine. Setting this to a positive value will override the air default which is 198.6 ° Rankine.

B.4.8 &noninertial_reference_frame

FUN3D can perform simulations in noninertial reference frame rotating at a constant rate, Ω. The noninertial reference frame simulation can be run as a steady state problem if the freestream velocity crossed with the rotation vector is zero, $\mathbf{U}_\infty \times \Omega = 0$. In a practical sense, freestream velocity should be zero or parallel to the axis of rotation. Using a standard inertial reference frame requires the same problem to be run as an unsteady simulation at a larger computational cost. Typical uses would be the simulation of an isolated rotor in hover (without forward motion) or an aircraft performing a steady-state pitching maneuver or constant roll about the wind axis.

```
&noninertial_reference_frame
  noninertial              = .false.
  rotation_center_x        = 0.0
  rotation_center_y        = 0.0
  rotation_center_z        = 0.0
  rotation_rate_x          = 0.0
  rotation_rate_y          = 0.0
  rotation_rate_z          = 0.0
  noninertial_frame_output = .true.
/
```

noninertial = .false.

When .true., use a noninertial reference frame. The default is the inertial reference frame.

rotation_center_x = 0.0

This is the x of the steady rotation rate center point.

rotation_center_y = 0.0

This is the y of the steady rotation rate center point.

rotation_center_z = 0.0

This is the z of the steady rotation rate center point.

rotation_rate_x = 0.0

This is the steady noninertial rotation rate (nondimensional) about the rotation center x-axis. For `eqn_type = 'compressible'`, the nondimensional rate is $\omega = \omega^* \frac{L^*_{ref}}{a^*_{ref} L_{ref}}$, where ω^* is the dimensional rotation rate about the rotation center x-axis, in rad/sec. For `eqn_type = 'incompressible'`, the nondimensional rate is $\omega = \omega^* \frac{L^*_{ref}}{V^*_{ref} L_{ref}}$; see also section 2.

`rotation_rate_y = 0.0`

This is the steady noninertial rotation rate (nondimensional) about the rotation center y-axis. For `eqn_type = 'compressible'`, the nondimensional rate is $\omega = \omega^* \frac{L^*_{ref}}{a^*_{ref} L_{ref}}$, where ω^* is the dimensional rotation rate about the rotation center y-axis, in rad/sec. For `eqn_type = 'incompressible'`, the nondimensional rate is $\omega = \omega^* \frac{L^*_{ref}}{V^*_{ref} L_{ref}}$; see also section 2.

`rotation_rate_z = 0.0`

This is the steady noninertial rotation rate (nondimensional) about the rotation center z-axis. For `eqn_type = 'compressible'`, the nondimensional rate is $\omega = \omega^* \frac{L^*_{ref}}{a^*_{ref} L_{ref}}$, where ω^* is the dimensional rotation rate about the rotation center z-axis, in rad/sec. For `eqn_type = 'incompressible'`, the nondimensional rate is $\omega = \omega^* \frac{L^*_{ref}}{V^*_{ref} L_{ref}}$; see also section 2.

`noninertial_frame_output = .true.`

When `.true.`, visualization output is left in the noninertial reference frame. In this frame, the velocity on (viscous) solid surfaces will be zero, and farfield boundaries will have velocity components that vary as $\omega \times R$, where R is the distance from the rotation axis. When `.false.`, visualization output is converted to the inertial frame. In the inertial frame, the velocity on (viscous) solid surfaces will vary as $\omega \times R$, and farfield boundaries will have (nominally) zero velocity components.

B.4.9 &inviscid_flux_method

This namelist controls the construction of the inviscid fluxes and flux Jacobians.

```
&inviscid_flux_method
  flux_construction          = 'roe'
  flux_construction_lhs      = 'vanleer'
  kappa_umuscl               = -1.0
  flux_limiter               = 'none'
  smooth_limiter_coeff       = 1.0
  freeze_limiter_iteration   = -1
  first_order_iterations     = 0
  multidm_option             = 1
  fixed_direction            = .true.
  recalc_dir_freq            = 1
  adptv_entropy_fix          = .false.
  rhs_u_eigenvalue_coef      = 0.
  lhs_u_eigenvalue_coef      = 0.
  rhs_a_eigenvalue_coef      = 0.
  lhs_a_eigenvalue_coef      = 0.
  entropy_fix                = .false.
  temperature_fix            = .false.
  re_min_vswch               = 50.
  re_max_vswch               = 500.
  pole_gradient              = .false.
  lowmach_prec               = .false.
  prec_mach_star             = 1.0
  prec_mach_trans1           = 1.0
  prec_mach_trans2           = 1.0
  prec_ff_bc                 = .false.
  print_frozen_limiter_fixes = .false.
  shock_sensor               = .false.
  adaptive_shock_sensor      = .true.
/
```

flux_construction = 'roe'

This specifies the inviscid flux residual construction method.

'roe' for Roe flux difference splitting.

'ldroe' for L2Roe flux with low dissipation for low mach number flow. [69]

'lmroe' for low mach roe, one variation of Ld-roe

'vanleer' for van Leer flux vector splitting.

'hllc' for HLLC.

`‘hlle++’` for HLLE++.

`‘aufs’` for AUFS.

`‘ldfss’` for LDFSS.

`‘dldfss’` for dissipative LDFSS.

`‘aldfss’` for LDFSS with an adaptive entropy fix.

`‘roe_ec’` for entropy-consistent Roe scheme.

`‘stvd’` for Yee’s symmetric total variation diminishing scheme. Not compatible with turbulence for `eqn_type = ’compressible’`.

`‘stvd_modified’` for a modified version of `stvd`.

`‘central_difference’` for central difference.

`‘multidm’` for Gnoffo’s multidimensional scheme.

<u>`flux_construction_lhs = ’vanleer’`</u>

This specifies the inviscid flux Jacobian construction method. A `‘consistent’` method yields the best asymptotic iterative convergence rate of the non-linear residual, but a more diffusive flux may stabilize a poorly converging or diverging linear system iterative scheme. Note that this parameter is set internally to `‘consistent’` if HANIM (section B.4.16) is used.

`‘consistent’` for a consistent linearization with the residual construction method.

`‘vanleer’` for Van Leer.

`‘roe’` for Roe linearization.

`‘hllc’` for HLLC.

`‘aufs’` for AUFS.

`‘ldroe’` for LD-Roe.

`‘lmroe’` for low mach Roe

`‘ldfss’` for LDFSS.

<u>`kappa_umuscl = -1.0`</u>

Controls the amount of upwinding in the unstructured-grid MUSCL reconstruction scheme. The default will adjust `kappa_umuscl` internally to 0.5 for 3D mixed-element grids or 0.0 for all other grid types. 0.0 is the upwind-biased (Fromm) discretization, 1.0 is the (unstable) central-difference discretization, and the range [0,1] is a blend of the two.

<u>`flux_limiter = ’none’`</u>

This selects the flux limiter. The limiters that begin with the letter `h`, `’barth’`, and `’venkat’` are stencil-based limiters (they apply a limiter to each edge in a node's the reconstruction stencil and store the most restrictive edge limiter value at the node). Other limiters are evaluated in a strictly edge-based manner. The `h`-series of limiters automatically turns on a heuristic pressure based limiter that is used to augment the selected flux limiter. [70] The node-based limiters can be frozen with the `freeze_limiter_iteration` namelist variable described below. Note: The adjoint solver is only compatible with a frozen limiter flow solution. For hypersonic flows computed using the calorically perfect gas path, the `hvanleer` or `hvanalbada` flux limiters are recommended. The smooth class of limiters use the `smooth_limiter_coeff` as noted in the limiter descriptions.

`'none'` for no limiter.

`'barth'` for the Barth limiter.

`'venkat'` for the Venkatakrishnan [71] limiter. This limiter relies on a user-tunable parameter, `smooth_limiter_coeff`.

`'nishikawa'` for the Nishikawa [72] limiter. This limiter relies on a user-tunable parameter, `smooth_limiter_coeff`. usage is the same as in Venkat limiter.

`'hminmod'` for the stencil-based min-mod limiter augmented with a heuristic pressure limiter.

`'hvanleer'` for the stencil-based van Leer limiter augmented with a heuristic pressure limiter.

`'hvanalbada'` for the stencil-based van Albada limiter augmented with a heuristic pressure limiter. This limiter relies on a user-tunable parameter, `smooth_limiter_coeff`.

`'hvenkat'` for the Venkatakrishnan limiter augmented with a heuristic pressure limiter. This limiter relies on a user-tunable parameter, `smooth_limiter_coeff`.

`'minmod'` for the min-mod limiter.

`'vanleer'` for the van Leer limiter.

`'vanleer_gg'` for van Leer limiter that also turns on Green-Gauss gradients for inviscid reconstruction.

`'vanalbada'` for the van Albada limiter.

`smooth_limiter_coeff = 1.0`

This is the coefficient used to tune smooth limiters, such as `'venkat'`. Larger values of `smooth_limiter_coeff` reduce the effect of the limiter, tending toward the unlimited solution, with (typically) better convergence properties at the expense of increased oscillations near shocks.

This class of smooth limiters assume that the geometry in the mesh is O(1), i.e., that the grid is normalized to a characteristic length of your model. When the geometry is not O(1), a suggested value for `smooth_limiter_coeff` is proportional to the reciprocal of a characteristic length, e.g., proportional to 1/(Mean Aerodynamic Chord). By way of example, assume you have a grid in which the mean aerodynamic chord is 1.0, and you were satisfied with the solution properties and convergence with a value of `smooth_limiter_coeff` = 3.0. Then assume you scale this mesh so that the mean aerodynamic chord is 10.0. Running the scaled mesh at the same flow conditions would require that `smooth_limiter_coeff` = 3.0/10.0 to achieve the same solution properties and convergence as the unscaled mesh.

`freeze_limiter_iteration = -1`

The node-based limiters can be frozen with `freeze_limiter_iteration` equal to zero or an iteration number (subiteration number if time accurate). A negative value does not freeze the limiter. Zero is used to retain the limiter field contained in the restart file for all iterations (if steady state) or subiterations (if time accurate) of the current run. Freezing a limiter can reduce nonlinear iterative convergence "ringing." [71] Freezing is typically engaged at 3/4 of the iterations required for convergence. The steady decline of the low frequency residual modes tend to continue at similar rate with or without freezing, you just can't see it in the ringing residual norm. The later you wait to freeze the limiter, the closer the frozen limiter solution is to the converged answer. If a unrealizable reconstruction (negative density or pressure) is encountered the limiter field will be locally updated at the node experiencing the problematic reconstruction. The adjoint solver is only compatible with a frozen limiter.

`first_order_iterations = 0`

This is the number of iterations to use first-order spatial accuracy prior to using second-order spatial accuracy. If second-order spatial accuracy in not required, set this to a value larger than the number of `steps`. This option is useful for starting difficult supersonic flow simulations. For time accurate cases (`time_accuracy` not equal to `'steady'`), this is the number of first-order accurate subiterations to run for each time step.

`multidm_option = 1`

This controls the `multidm` reconstruction weighting.

`'1'` virtual node averaging.

`'2'` weighted average of edges.

`fixed_direction = .true.`

This specifies the use of Cartesian directions in `multdm` reconstruction.

`recalc_dir_freq = 1`

This sets the frequency of direction recalculation in the `multidm` scheme.

`adptv_entropy_fix = .false.`

This activates the adaptive entropy fix for Roe's scheme.

`rhs_u_eigenvalue_coef = 0.`

This is the contact/shear eigenvalue smoothing coefficient for the adaptive entropy fix and the `roe` residual. It increases dissipation to improve robustness with the penalty of reduced solution accuracy.

`lhs_u_eigenvalue_coef = 0.`

This is the contact/shear eigenvalue smoothing coefficient for the adaptive entropy fix and the `roe` Jacobian. It increases dissipation to improve robustness of the linear solve with the potential penalty of reduced nonlinear convergence if it is different from `rhs_u_eigenvalue_coef`.

`rhs_a_eigenvalue_coef = 0.`

This is the acoustic eigenvalue smoothing coefficient for the adaptive entropy fix and the `roe` residual. It increases dissipation to improve robustness with the penalty of reduced solution accuracy.

`lhs_a_eigenvalue_coef = 0.`

This is the acoustic eigenvalue smoothing coefficient for the adaptive entropy fix and the `roe` Jacobian. It increases dissipation to improve robustness of the linear solve with the potential penalty of reduced nonlinear convergence if it is different from `rhs_a_eigenvalue_coef`.

`entropy_fix = .false.`

This activates the entropy fix for the `stvd` flux.

`temperature_fix = .false.`

This activates the temperature fix for undershoots with the `stvd` flux.

`re_min_vswch = 50.`

For the `stvd` flux, eigenvalue limiting is turned off below this cell Reynolds number.

`re_max_vswch = 500.`

For the `stvd` flux, eigenvalue limiting is fully engaged above this cell Reynolds number.

`pole_gradient = .false.`

If true, use limiting form of continuity equation across pole in association with `symmetry_1_strong`, `symmetry_2_strong`, or `symmetry_3_strong`.

`lowmach_prec = .false.`

This option turns the low-Mach preconditioning formulation on or off. If active, the value of `flux_construction` must be set to `'roe'`. The options for low-Mach preconditioning are based on the formulation presented in Ref. [73]. The input parameters `prec_mach_star`, `prec_mach_trans1`, and `prec_mach_trans2` (with `prec_mach_trans1` <= `prec_mach_trans2`) are used to define the scaling factor, beta, which is expected to be proportional to the square of the local Mach number, but is not allowed to become zero. If the local Mach number is between `prec_mach_trans1` and `prec_mach_trans2`, beta is computed according to a function that smoothly connects the limiting values at the lower and upper bounds.

`prec_mach_star = 1.0`

This input corresponds to the nominal value of the Mach number, to be used with the low-Mach preconditioning scheme.

`prec_mach_trans1 = 1.0`

This input defines the lower Mach number at which the lower bound cutoff is applied. If the local Mach number is less than `prec_mach_trans1`, the parameter beta is set to a lower-bound cutoff defined as $beta = prec\_mach\_star^2$.

`prec_mach_trans2 = 1.0`

This input defines the upper limit on the local Mach number at which the low-Mach preconditioner is active. If the local Mach number is greater than `prec_mach_trans2`, the parameter beta is set to 1, which implies no low-Mach preconditioning.

`prec_ff_bc = .false.`

This parameter controls the application of the low-Mach preconditioning at the far-field boundary. A value of `.false.` implies no preconditioning, while `.true.` applies preconditioning.

`print_frozen_limiter_fixes = .false.`

This variable controls the printing of the global number of nodes at which a frozen limiter is updated every iteration.

`shock_sensor = .false.`

This activates a shock sensor for the `stvd` flux.

`adaptive_shock_sensor = .true.`

This enables an adaptive shock sensor for HLLE++ and `stvd` fluxes based on freestream Mach number.

B.4.10 &turbulent_diffusion_models

When `viscous_terms = 'turbulent'`, this namelist is used to set the form of the turbulence model.

```
&turbulent_diffusion_models
  turbulence_model                = 'sa'
  turb_model                      = 'deprecated-use-turbulence_model'
  turb_intensity                  = -0.001
  turb_viscosity_ratio            = -0.001
  reynolds_stress_model           = 'linear'
  use_nonlinear_stress_jacobians = .false.
  turb_compress_model             = 'off'
  turb_conductivity_model         = 'off'
  prandtlnumber_turbulent         = 0.9
  schmidtnumber_turbulent         = 1.
  miv_hrles                       = .false.
  use_decoupled_turbulence_gen    = .false.
  dw_rans                         = 0.2
  dw_les                          = 0.4
  strelets_des                    = .false.
/
```

`turbulence_model = 'sa'`

This selects the form of the turbulence model. The naming convention of http://turbmodels.larc.nasa.gov/ is used for the models described on the website.

`'sa'` for Spalart-Allmaras model. [51] See the `&spalart` namelist in section B.4.12 for additional controls.

`'sa-catris'` for Spalart-Allmaras Catris-Aupoix model. [74] Available only for `eqn_type='generic'`.

`'des'` for Spalart-Allmaras based DES model. [75] See the `&spalart` namelist in section B.4.12 for additional controls. Not available for `eqn_type='generic'`.

`'sa-neg'` for Spalart-Allmaras model with negative turbulence variable provisions. [61] Not available for `eqn_type='generic'`.

`'des-neg'` for Spalart-Allmaras based DES model with negative turbulence variable provisions. [61,75] Not available for `eqn_type='generic'`.

`'menter-sst'` option is no longer valid, use `sst` or `sst-v`

`'bsl'` Menter Baseline Two-Equation Model. [76]

`'sst'` Menter SST Two-Equation Model with strain source term. [76]

`'sst-v'` Menter SST Two-Equation Model with vorticity source term. [77] Not available for `eqn_type='generic'`.

`'wilcox1988'` Wilcox (1988) k-omega Two-Equation Model. [78] Not available for `eqn_type='generic'`.

`'wilcox1988-v'` Wilcox (1988) k-omega Two-Equation Model with vorticity-based production source term. [78] Not available for `eqn_type='generic'`.

`'wilcox2006'` Wilcox (2006) k-omega Two-Equation Model. [79] Not available for `eqn_type='generic'`.

`'wilcox2006-v'` Wilcox (2006) k-omega Two-Equation Model with vorticity source term. [79] Not available for `eqn_type='generic'`.

`'kw-des'` for SST-kω based DES model, the kω-based Strelets DES [80] model, and the model invariant hybrid RANS-LES method of Woodruff [81].

`'hrles'` for Menter SST-based hybrid-RANS/LES model. [82, 83] Does not work with `eqn_type='generic'`.

`'gamma-ret-sst'` for Langtry and Menter SST transition model. [84] See the `&gammaretsst` namelist in section B.4.13 for additional controls. Not available for `eqn_type='generic'`.

`'k-kL-MEAH2015'` for Menter/Egorov and Abdol-Hamid two-equation k-kL model. [85] Not available for `eqn_type='generic'`.

`'baldwin-lomax'` Baldwin-Lomax algebraic model. Available only for `eqn_type='generic'`. Requires implicit lines file `.lines_fmt`. Search this manual for `implicit_lines` to find instructions to use the utility `aflr3_line_extraction` in order to generate the implicit lines file. Also requires `partition_lines=.true.` in namelist `&partitioning`.

`'cebeci-smith'` Cebeci-Smith algebraic model. Available only for `eqn_type=generic`. Requires implicit lines file `.lines_fmt`. Search this manual for `implicit_lines` to find instructions to use the utility `aflr3_line_extraction` in order to generate the implicit lines file. Also requires `partition_lines=.true.` in namelist `&partitioning`.

`turb_model = 'deprecated-use-turbulence_model'`

This is a deprecated namelist variable for `turbulence_model`. It is included for backwards compatibility with fun3d.nml files, but may be removed in a future version.

`turb_intensity = -0.001`

This sets the freestream turbulence intensity, $\sqrt{\frac{2k}{3u_\infty^2}}$, where k is the turbulent kinetic energy. Negative value forces `turb_intensity` $= \sqrt{\frac{0.01\mu_0}{1.5}}$

and `turb_viscosity_ratio` $= \rho_0 0.005$ yielding $(\rho k)_0 = 0.01\mu_0$ and $(\rho\omega)_0 = 2$. Only applies to `eqn_type='generic'`.

<u>`turb_viscosity_ratio = -0.001`</u>

This sets the freestream ratio of turbulent viscosity to molecular viscosity. Negative value forces `turb_intensity` $= \sqrt{\frac{0.01\mu_0}{1.5}}$ and `turb_viscosity_ratio` $= \rho_0 0.005$ yielding $(\rho k)_0 = 0.01\mu_0$ and $(\rho\omega)_0 = 2$. Only applies to `eqn_type='generic'`.

<u>`reynolds_stress_model = 'linear'`</u>

This controls the mean stress-strain constitutive relation.

`'linear'` indicates the use of the linear Boussinesq assumption.

`'qcr2000'` activates the Quadratic Constitutive Relationship (QCR) 2000 version of Spalart [86] or one of its later versions. This nonlinear model is not valid for explicit algebraic Reynolds stress or full Reynolds stress transport models (e.g., ASM, EASM, RSM models).

`'cubic_equation'` solves a cubic equation for the stress-strain assumption at each point in the flow field based on Rumsey and Gatski [87]. It is only implemented for k-kL turbulence models.

<u>`use_nonlinear_stress_jacobians = .false.`</u>

This indicates whether to include linearizations of nonlinear Reynolds stress terms in the meanflow. Computing these linearizations increases execution time. Including these terms has not shown an improvement in convergence for all cases. This option is provided for testing in situations where convergence is difficult without these terms.

<u>`turb_compress_model = 'off'`</u>

This controls the turbulence compressibility model.

`'off'` for no correction.

`'ssz'` for the modified SSZ correction that is vorticity-based instead of strain-based [88] (use with Spalart-Allmaras models).

`'zeman'` for Zeman (use with $k - \epsilon$ models).

`'wilcox'` for Wilcox (use with SST-based models).

`'sarkar'` for Sarkar (use with $k - \epsilon$ models).

<u>`turb_conductivity_model = 'off'`</u>

This controls whether a turbulence conductivity model is employed. Only applies to `eqn_type='generic'`.

`'off'` to turn off a turbulence conductivity model.

`'on'` to turn on a turbulence conductivity model.

`prandtlnumber_turbulent = 0.9`

This is the turbulent Prandtl number.

`schmidtnumber_turbulent = 1.`

This is the turbulent Schmidt number. Only applies to `eqn_type='generic'`.

`miv_hrles = .false.`

This invokes the model invariant $k-\omega$ based hybrid RANS-LES method of Woodruff [81]. This option can only be combined with `turbulence_model='kw-des'`.

`use_decoupled_turbulence_gen = .false.`

Experimental: This enables decoupled turbulence models for generic gas. Currently supports `'sa'`,`'sa-neg'`,`'des'`,`'des-neg'`, `'sst'`, and `'sst-v'`.

`dw_rans = 0.2`

This is the distance from wall at which the RANS region ends for `miv_hrles=.true.`, the model invariant hybrid RANS-LES method of Woodruff [81].

`dw_les = 0.4`

This is the distance from wall at which the LES region starts for `miv_hrles=.true.`, the model invariant hybrid RANS-LES method of Woodruff [81].

`strelets_des = .false.`

This invokes the Strelets kw-des model [80] when set to `.true`. This option can only be combined with `turbulence model = 'kw-des'`.

B.4.11 &turbulence

This namelist provides auxiliary information to the turbulence boundary conditions used in for the turbulent transport equations. Refer to section 3 for the definition of boundary numbers.

```
&turbulence
  use_least_squares_gradients = .false.
  limit_crossd                = .false.
/
```

use_least_squares_gradients = .false.

The default method for calculating the velocity gradients used in the multiequation turbulence models is the edge-based Green-Gauss. For linear functions, the Green-Gauss routines will provide exact gradients for tetrahedral grids. Reconstruction of the gradients on nontetrahedral elements may have some error associated with them. Similarly, the default method for calculating gradients of the turbulence variables is edge-based Green-Gauss. Again, for linear functions on tetrahedral grids the gradients will be exact. When the mesh has other element types, such as hexahedra, prisms or pyramids, the edge-based Green-Gauss may not produce sufficiently accurate gradients. The accuracy is very dependent upon the exact shape of the elements, the nature of the flow and simulation, and the placement in the mesh overall. Setting `use_least_squares_gradients = .true.`, the code will calculate the velocity gradients used in the turbulence models and turbulence variable gradients via weighted least squares and least squares routines, respectively. For linear functions, the least squares methods should produce exact gradients with mixed element types.

limit_crossd = .false.

`limit_crossd = .true.` will limit the value of the cross-derivative term in the kw-SST model to between ±1., as shown in the following equations.

$$-1.0 < CD_{k\omega} < 1.0, \text{ where } CD_{k\omega} = 2\frac{\rho\sigma_{\omega 2}}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j} \quad \text{(B1)}$$

During the initial phase of solution development, depending on the configuration, the grid and the solution process, large excursions in the magnitude of the cross-diffusion term can occur. Limiting this term could improve solution stability until the transients have diminished, after which the limiting should be turned off. The default is `limit_crossd = .false.`.

B.4.12 &spalart

This namelist is used to modify details of the SA and SA based DES turbulence models.

```
 &spalart
   turbinf        = 3.0
   dacles_mariani = .false.
   sarc           = .false.
   sa_lre         = .false.
   ddes           = .false.
   ddes_mod1      = .false.
   cddes          = 0.975
   des            = .false.
   noft2          = .false.
   crot           = 2.0
 /
```

<u>turbinf = 3.0</u>

This is the freestream turbulence value for the SA model.

<u>dacles_mariani = .false.</u>

This activates the Dacles-Mariani [89, 90] rotation correction (denoted SA-R by http://turbmodels.larc.nasa.gov/).

<u>sarc = .false.</u>

This activates the rotation/curvature correction [91] (denoted SA-RC by http://turbmodels.larc.nasa.gov/).

<u>sa_lre = .false.</u>

This activates SA-LRe (low Reynolds number fix from Spalart and Garbaruk, AIAA Journal, Vol. 58, No. 5, May 2020, pp. 1903-1905).

<u>ddes = .false.</u>

This changes the `turbulence_model='des'` into Delayed DES [92] (DDES).

<u>ddes_mod1 = .false.</u>

This changes the `turbulence_model='des'` into Modified Delayed DES [93] (MDDES). It also requires `ddes = .true`. This option should be used with caution. Most experience with this model is for large separated flows encountered in wake regions of bluff bodies, such as cylinders and landing gears. Further validation is required for other situations. There is sensitivity to the parameter `cddes`.

<u>cddes = 0.975</u>

This is C_{MDD} in Ref. [93], which is used when `ddes_mod1 = .true.`.

`des = .false.`

Activates DES for sa-catris (only available for `eqn_type = 'generic'`).

`noft2 = .false.`

Activate SA-noft2 turbulence model if .true., only for SA

`crot = 2.0`

Parameter for SA-R and SA-neg-R models, default is 2, but it can be 1

B.4.13 &gammaretsst

This namelist modifies the details of the `turbulence_model = 'gamma-ret-sst'` transition turbulence model of Langtry and Menter. [84] Correctly setting the freestream levels of turbulence is key to the transition location of this model. Ideally, freestream turbulence levels are measured in an experiment.

```
&gammaretsst
  gammaretsst2003     = .false.
  crossflow_LM2015    = .false.
  crossflow_CFHE      = .false.
  h_rough             = 0.0
  extra_4eqn_limiters = .false.
/
```

`gammaretsst2003 = .false.`

The default for the gamma-ret-sst model is currently association with the 1994 version of SST. Setting this to true will change the association to the 2003 version of SST.

`crossflow_LM2015 = .false.`

This turns on the crossflow transition capability of Langtry-Menter (AIAA-2015-2474). When used in combination with gammaretsst2003 set to true, the resulting model is SST-2003-LM2015.

`crossflow_CFHE = .false.`

This turns on the crossflow transition capability of Grabe et al. (AIAA-2016-3572). When used in combination with gammaretsst2003 set to true, the resulting model is SST-2003-LM2009-CFHE.

`h_rough = 0.0`

This is used for one of the correlations in the Langtry-Menter crossflow model, which accounts for surface roughness induced stationary crossflow. It is the nondimensional roughness height relative to the unit length of the computational grid; i.e., if the grid is in meters, then this value should also be specified in meters. This term only has an effect when `crossflow_LM2015` is set to true.

`extra_4eqn_limiters = .false.`

This turns on additional 4-eqn model limiters that are not a part of the official gamma-ret-sst equations, but have been found to help at times with robustness. This limits the production term of the gamma equation and the destruction term of the Rethetat equation. Using this may have some impact on the converged solution.

B.4.14 &code_run_control

This namelist controls the length of the simulation. Restart options, Jacobian update strategy, and angle of attack continuation can also be specified.

```
&code_run_control
  steps                          = 500
  stopping_tolerance             = 1.e-15
  duration_limit_in_minutes      = -1.0
  no_restart                     = .false.
  append_timestep_to_restart_name = .false.
  history_write_freq             = -1
  restart_write_freq             = 250
  restart_read                   = 'on'
  smart_jupdate                  = .true.
  jacobian_eval_freq             = 0
  jupdate_startup_steps          = 10
  jupdate_amut_max_change        = 0.10
  jupdate_cfl_inv_change         = 5.0
  dfduc3_jacobians               = .false.
  alpha_sweep                    = .false.
  cycle_increment                = 50
  alpha_increment                = 0.25
  alpha_max                      = 180.0
  alpha_min                      = -180.0
  alpha_switchbacks              = 0
  write_steady_restart           = .false.
  sd_file_format                 = 'ascii'
  use_openmp                     = .false.
  use_vector_intrinsics          = .true.
  ebv                            = .false.
  mixed                          = .false.
/
```

`steps = 500`

This is the number of time steps or steady iterations to perform.

`stopping_tolerance = 1.e-15`

This instructs the solver to terminate before all `steps` are complete when the root mean square (RMS) of every equation (continuity, energy, etc.) is less than this tolerance.

`duration_limit_in_minutes = -1.0`

This is the maximum run duration limit in minutes (a negative value is unlimited). This limit can terminate the solver before all `steps` are complete, which may be helpful if the solver is run as a batch system job with a time limit. Additional time is required to complete the current iteration and write restart file. So, allow an extra time margin for code shutdown. MPI required.

`no_restart = .false.`

When this is `.true.`, no restart checkpoint file is written.

`append_timestep_to_restart_name = .false.`

When this is `.true.`, restart filenames will contain the current timestep index as a suffix. This prevents the loss of existing restart data should the file system fail while writing restart data. The user must manually remove the integer suffix prior to subsequent runs when intending to restart from this data.

`history_write_freq = -1`

The history and forces files will be written to disk every `history_write_freq` time steps of the current run.

`restart_write_freq = 250`

The restart checkpoint will be written to disk every `restart_write_freq` time steps of the current run.

`restart_read = 'on'`

This defines the solution at the first time step.

`'on'` to initialize the simulation with a solution read from the restart file. The current convergence history will be concatenated with the prior solution history.

`'on_nohistorykept'` to initialize the simulation with a solution read from the restart file. The previous history (e.g., residuals, forces, moments) will be discarded.

`'off'` for no restart file read. The solution will be initialized as freestream or as specified in the `&flow_initialization` namelist.

`smart_jupdate = .true.`

This option allows the code to automatically adjust the Jacobian update frequency based on residual reduction.

`jacobian_eval_freq = 0`

This is the frequency of Jacobian evaluation based on time steps. It should be set to zero when `smart_jupdate = .true.`

`jupdate_startup_steps = 10`

The Jacobians are evaluated at every time step for the first `jupdate_startup_steps`, which aids robustness during initial start transients.

`jupdate_amut_max_change = 0.10`

For turbulent flow when the maximum eddy viscosity is greater than 1.5, the Jacobians are refreshed when the maximum eddy viscosity changes by this amount from the maximum eddy viscosity corresponding to the last jacobian evaluation.

`jupdate_cfl_inv_change = 5.0`

The Jacobians are refreshed when the time term associated with the cfl of the adaptive strategy and the time term of the last jacobian evaluation differ by this amount.

`dfduc3_jacobians = .false.`

This option only affects `eqn_type = 'incompressible'`. When `.true.`, approximate Jacobians are computed that may improve the convergence of some cases.

`alpha_sweep = .false.`

This option activates a procedure to adjust `angle_of_attack` during a simulation. It can be used to calculate a drag polar in a single execution or explore a hysteresis loop. The starting `angle_of_attack` for the sweep is the angle specified in the `&reference_physical_properties` namelist. The sweep is controlled by the following options.

`cycle_increment = 50`

When `alpha_sweep=.true`,

`cycle_increment` < 0 increments `angle_of_attack` after residuals have reached the `stopping_tolerance`.

`cycle_increment` > 0 is the number of iterations between increments to alpha.

`cycle_increment` $= 0$ is an inadmissible value.

`alpha_increment = 0.25`

When `alpha_sweep=.true.`, increment `angle_of_attack` by these many degrees at a point controlled by `cycle_increment`.

`alpha_max = 180.0`

When `alpha_sweep=.true.`, this is the maximum value of `angle_of_attack`.

`alpha_min = -180.0`

When `alpha_sweep=.true.`, this is the minimum value of `angle_of_attack`.

`alpha_switchbacks = 0`

When `alpha_sweep=.true.`, this is the number of directional changes in the `angle_of_attack` sweep. When `alpha_switchbacks` > 0, `alpha_increment` is changed in sign after reaching `alpha_max` or `alpha_min`. This allows exploration of hysteresis loops.

`write_steady_restart = .false.`

When `write_steady_restart=.true.`, and running a time-accurate simulation (`itime\=0`), this will cause a steady-state restart file to be written, rather than an unsteady restart file. This steady-state restart file allows the steady-state solver to be used on subsequent executions. Warning, solution time history will be discarded.

`sd_file_format = 'ascii'`

Specifies the format of mesh sensitivity files. See section 9.3 for details.

`'ascii'` for ASCII formatted mesh sensitivity files.

`'stream'` for Fortran-stream (C-binary) formatted mesh sensitivity files.

`use_openmp = .false.`

Instructs FUN3D to use OpenMP directives. Currently only supported for SFE. If the compiler does not provide OpenMP support or it was not invoked when FUN3D was built, `use_openmp` will be forced to `.false`. Similarly, if an MPI library is linked but does not provide thread support, `use_openmp` will be forced to `.false`.

`use_vector_intrinsics = .true.`

This option instructs FUN3D to use optimized AVX-512 vector intrinsics for AVX-512-enabled processors. FUN3D must be configured and built as described in section A.8 to use this option. FUN3D will detect AVX-512 support at runtime if built with the Intel compiler and disable this option if not supported. If not so built, no detection will occur, which will result in an error if not running on an AVX-512-capable machine. As such, if FUN3D is not built as described in section A.8 or built without the Intel compiler, this option is .false. by default.

`ebv = .false.`

ebv = .true. Activates the edge-based viscous (EBV) method for the viscous kernel of the compressible Navier-Stokes equations, the SA/SA-neg and the sst turbulence models with or without qcr2000 option, and chemistry models, applied in the perfect and generic gas paths and for the CPU and GPU implementations. For EBV formulation for tetrahedral grids, see AIAA-2021-2728; For EBV extension to mixed-element grids, see AIAA-2022-4083; For EBV accuracy analysis and verification, see ICCFD11-1403, 2022;

`mixed = .false.`

Forces the solver to discretize pure tetrahedral grids using the general mixed-element formulation.

B.4.15 &nonlinear_solver_parameters

This namelist defines the temporal accuracy of the solution advancement scheme. The subiterations and time step size of time accurate simulations can also be specified. The ramping of the pseudo time advancement CFL number is also set. Density and pressure floors on the update and relation factors are available.

```
&nonlinear_solver_parameters
  time_accuracy               = 'steady'
  time_step_nondim            = 0.0
  time_step_dpsi              = -1.0
  rotor_sets_time_step        = 1
  subiterations               = 0
  temporal_err_control        = .false.
  temporal_err_floor          = 0.1
  legacy_temp_err_checks      = .true.
  schedule_iteration(1:2)     = 1, 50
  schedule_cfl(1:2)           = 200.0, 200.0
  schedule_cflturb(1:2)       = 50.0, 50.0
  f_allow_minimum_m           = 0.01
  invis_relax_factor          = 1.0
  visc_relax_factor           = 1.0
  divergence_source_unsteady = .false.
  force_half_steps            = .true.
  hanim                       = .false.
/
```

`time_accuracy = 'steady'`

This defines the temporal scheme.

`'steady'` for steady state calculations. This is a local time step pseudotime advancement scheme that is not time accurate.

`'1storder'` is a first-order backward differencing scheme (backward Euler) for time-accurate temporal time integration.

`'2ndorder'` is a second-order backward differencing scheme (BDF2 in Ref. [94]) for time-accurate temporal time integration.

`'2ndorderOPT'` is an optimized second-order backward differencing (BDF2opt in Ref. [95]) for time-accurate temporal time integration. This scheme is second-order accurate in time but has an order-of-magnitude lower leading coefficient than standard BDF2.

`'3rdorder'` is a third-order backward differencing scheme (BDF3 in Ref. [94]) for time-accurate temporal integration.

`‘4thorderMEBDF4’` is a fourth-order modified extended backward differencing scheme (MEBDF4 in Ref. [94]) for time-accurate temporal integration.

`‘3rdorderSSPRK’` is a third-order Strong Stability Preserving (SSP) Runge-Kutta scheme [96] for time-accurate temporal integration. This option is not currently available when using a turbulence model.

`‘lfd’` is the linearized frequency domain method in the stabilized finite-element solver [97].

`time_step_nondim = 0.0`

This is the nondimensional time step for time accurate simulations. It is ignored when `time_accuracy = 'steady'`. The nondimensionalization of this parameter depends on `eqn_type`. When `eqn_type = 'compressible'`, it is $dt\frac{a_{ref}}{L}$, where a_{ref} is the reference speed of sound, and L is unit 1 of the grid. When `eqn_type = 'incompressible'` or `'generic'`, it is $dt\frac{u_{ref}}{L}$, where u_{ref} is the reference velocity. See section 2.4 for more details and guidance on appropriate values.

`time_step_dpsi = -1.0`

This is a rotorcraft-specific time step that is specified in degrees of rotation of the rotor (the first rotor if more than one) per timestep. `time_step_dpsi` is ignored if negative; if `time_step_dpsi` is positive, the value of `time_step_nondim` is replaced with a value calculated from `time_step_dpsi` and the rotor rotation rate. If more than one rotor is present, `rotor_sets_time_step` may be used to select which rotor's rotation speed is used to set the time step.

`rotor_sets_time_step = 1`

This sets which rotor is used to set the time step when `time_step_dpsi` is used. This option is only for rotorcraft simulation.

`subiterations = 0`

Number of subiterations applied to solve the implicit time integration. It is ignored when `time_accuracy = 'steady'`. A constant CFL time step is used in each subiteration. By the end of a *convergent* subiteration process the pseudo time term drops out, giving the correct temporal discretization.

`temporal_err_control = .false.`

This governs whether the specified number of `subiterations` are run for each time step (`.false.`), or, if the temporal error is monitored and the subiterations are stopped when a specified tolerance is reached (`.true.`). It is ignored when `time_accuracy = 'steady'`.

`temporal_err_floor = 0.1`

This sets the tolerance for which time-accurate subiterations are stopped. The tolerance is given as a multiplicative factor of the flow residuals (mean and turbulence). It is ignored when `time_accuracy = 'steady'`.

`legacy_temp_err_checks = .true.`

This governs the equations that are monitored for temporal error convergence. Historically, *only* the x-momentum equation and *only* the first turbulence equation have been checked for convergence. When `legacy_temp_err_checks = .false.`, *all* mean flow equations and *all* turbulence equations are checked, so all equations must satisfy the convergence criterion set by the `temporal_err_floor` value before the subiterations are considered to be converged for the current time step. Furthermore, the legacy checks required a minimum of five subiterations to be performed, regardless of whether both x-momentum and first turbulence equations satisfied the convergence criterion. No minimum is imposed when `legacy_temp_err_checks = .false.`.

`schedule_iteration(1:2) = 1, 50`

These are the iteration or subiteration numbers at which CFL numbers are specified. When `time_accuracy = 'steady'`, this controls the CFL number of the pseudotime terms over iterations. When running time-accurately, this controls the CFL number of the pseudotime terms of the linear system over subiterations. The parameter `schedule_iteration(1)` must be one, because it defines the starting CFL number at the first iteration or subiteration. The actual CFL number is determined by a linear ramp from `schedule_cfl(1)` at `schedule_iteration(1)` to `schedule_cfl(2)` at `schedule_iteration(2)`. The CFL number is held constant at `schedule_cfl(2)` after `schedule_iteration(2)`.

`schedule_cfl(1:2) = 200.0, 200.0`

This controls the ramping and final CFL number of the meanflow equations. See the description for `schedule_iteration`.

`schedule_cflturb(1:2) = 50.0, 50.0`

This controls the ramping and final CFL number of the turbulence model equations. See the description for `schedule_cfl` in `schedule_iteration`.

`f_allow_minimum_m = 0.01`

This limits the solution update to prevent pressure and density from dropping below this fraction of their freestream values. Applied to `eqn_type = 'compressible'` only.

invis_relax_factor = 1.0

This is the relaxation factor of inviscid terms. It scales the nonlinear update of the inviscid terms by this fraction and is only used for eqn_type = 'generic'.

visc_relax_factor = 1.0

This is the relaxation factor of viscous terms. It scales the nonlinear update of the viscous terms by this fraction and is only used for eqn_type = 'generic'.

divergence_source_unsteady = .false.

This uses the compact formulation of accuracy-preserving source term quadrature for third-order edge-based discretization. [98, 99]

force_half_steps = .true.

This forces the time integration to advance the meanflow prior to linearizing and advancing the turbulence for compressible and incompressible perfect gas simulations. The .true. option emulates the behavior of 13.4 and previous versions. The .false. option reduces communication to improve parallel scaling.

hanim = .false.

Setting hanim = .true. activates the HANIM solution algorithm for primal analysis. Parameters used in the HANIM solver are defined in the &hanim namelist described in section B.4.16.

B.4.16 &hanim

This namelist specifies input parameters associated with a Hierarchical Adaptive Nonlinear Iteration Method (HANIM) [100–102] that aims to improve the robustness and efficiency of the standard nonlinear solver described in section B.4.15.

HANIM is based upon a hierarchy of modules such as preconditioner, a generalized conjugate residual (GCR) method, [103] realizability checks, a nonlinear controller, and CFL adaptation. A flowchart illustrating key elements of the HANIM modules is shown in Fig. B1. Each HANIM iteration starts from a realizable solution. The preconditioner generates a preliminary solution update. A matrix-free Krylov approach and Frechet derivatives are used to construct the GCR search directions. The GCR method optimizes solution update by solving a linear system that represents the exact linearization of the coupled meanflow and turbulence model equations. The GCR solution update is checked to provide a realizable nonlinear solution. HANIM modules report success or failure based on the level of residual reduction and realizability of the updated solution. The CFL number used in the pseudotime-stepping method is adaptively controlled based on success or failure of the HANIM modules.

HANIM is currently valid only for primal flow computations on CPUs with `eqn_type = 'compressible'`; work is ongoing to extend HANIM to GPU computations. The supported turbulence models include the one-equation Spalart-Allmaras (SA) model and its negative variant (SA-neg); the SA-neg model is preferable from the HANIM performance point of view. More details about the solver output and diagnostics are provided after the HANIM namelist. The HANIM algorithm is activated by setting `hanim = .true.` in the `&nonlinear_solver_parameters` namelist, see section B.4.15. Default options for parameters used in the various HANIM modules are set according to the following list.

```
&hanim
  meanflow_relaxations                 = 300
  turbulence_relaxations               = 300
  preconditioner_tolerance_m           = 0.2
  preconditioner_tolerance_t           = 0.2
  preconditioner_max_growth            = 100.0
  minimum_preconditioner_relaxations = 1
  hanim_cfl                            = 1.0
  hanim_max_cfl                        = 1.0e10
  hanim_min_cfl                        = 1.0e-12
  cfl_decrease_factor                  = 0.1
  cfl_increase_factor                  = 2.0
  cfl_factor_turb                      = 1.0
  gcr_search_directions                = 5
  mu_gcr                               = 0.92
  gcr_restarts                       = 0
  nominal_step_size                  = 1.0e-7
  node_weighting                     = 2
  weight_viscous_mf                  = 1.0
  weight_viscous_t                   = 1.0
  weight_symmetry_mf                 = 1.0
  weight_symmetry_t                  = 1.0
  weight_farfield_mf                 = 0.0
  weight_farfield_t                  = 0.0
  hanim_output_flag                  = .false.
  hanim_checker                      = .false.
/
```

`meanflow_relaxations = 300`

This is the maximum number of relaxation sweeps used for the meanflow preconditioning. The recommended range is from 100 to 500.

`turbulence_relaxations = 300`

This is the maximum number of relaxation sweeps used for the turbulence preconditioning. The recommended range is from 100 to 500.

`preconditioner_tolerance_m = 0.2`

This is the desired relative reduction in the root-mean-square (RMS) norm of the meanflow preconditioner residual. The recommended range is from 0.5 to 0.1.

`preconditioner_tolerance_t = 0.2`

This is the desired relative reduction in the RMS norm of the turbulence preconditioner residual. The recommended range is from 0.5 to 0.1.

`preconditioner_max_growth = 100.0`

This is the maximum allowable growth of the RMS norm of preconditioner residuals. If the RMS norm of the meanflow or turbulence preconditioner residual exceeds the maximum allowable growth, the solution update is rejected and CFL is reduced. The recommended value is 100.

`minimum_preconditioner_relaxations = 1`

This is the minimum number of relaxation sweeps used for the meanflow and turbulence preconditioning.

`hanim_cfl = 1.0`

This is the starting value of the CFL number. The CFL number is adaptively controlled in HANIM during the solution process. If a realizability issue occurs in the beginning of nonlinear iterations, lowering `hanim_cfl` may be helpful.

`hanim_max_cfl = 1.0e10`

This is the maximum value of the CFL number.

`hanim_min_cfl = 1.0e-12`

This is the minimum value of the CFL number.

`cfl_decrease_factor = 0.1`

This is the CFL number decrease factor. If a HANIM module reports failure, the CFL number is decreased as $\text{CFL}^{\text{new}} = \text{CFL}^{\text{old}} \times$ `cfl_decrease_factor`. The CFL number is bounded by the minimum limit, `hanim_min_cfl`.

`cfl_increase_factor = 2.0`

This is the CFL number increase factor. If all HANIM modules report success, the CFL number is increased as $\text{CFL}^{\text{new}} = \text{CFL}^{\text{old}} \times$ `cfl_increase_factor`. The CFL number is bounded by the maximum limit, `hanim_max_cfl`.

`cfl_factor_turb = 1.0`

This is the multiplier for the CFL number applied to the turbulence model equation relative to the CFL number used for the meanflow equations. When `cfl_factor_turb = 1.0`, the CFL number is identical for the meanflow and turbulence model equations.

`gcr_search_directions = 5`

This is the maximum number of search directions for the GCR algorithm.

`mu_gcr = 0.92`

This is the relative reduction in the RMS norm of the GCR residual required for success of the GCR module. If the maximum number of search directions has been reached but the target GCR residual reduction has not been achieved, the GCR module reports failure and the CFL number is decreased. A greater GCR reduction level requires a tighter tolerance for the preconditioner residuals, more preconditioning relaxations, and more search directions.

`gcr_restarts = 0`

This controls the number of restarts for the GCR algorithm.

`nominal_step_size = 1.0e-7`

This is the nominal step size used in real-valued Frechet derivatives. The actual perturbation applied is chosen adaptively based on the RMS norm of the solution and the L_2 norm of the solution update. For turbulent flows at moderate and high speeds, `nominal_step_size=1.0e-7` is suitable. For low-speed turbulent or laminar (and inviscid) flows, `nominal_` `step_size=1.0e-5` provides better accuracy of the Frechet derivatives and may improve iterative convergence.

<u>`node_weighting = 2`</u>

This parameter globally controls the type of the weighted norm used in HANIM. The weighting for points on domain boundaries can be modified using the boundary weighting parameters described in this section such as `weight_viscous_mf`, `weight_viscous_t`, and etc.

'`1`' applies uniform weighting for all points of a CFD grid.

'`2`' uses $1/\sqrt[3]{V}$ for nodal weighting; V denotes dual-cell volume.

'`3`' uses $\min(1,\, 1/V)$ for nodal weighting.

'`4`' uses $1/d^3$ for the weighting applied to interior points of the turbulence field; d is distance to the nearest solid wall. The uniform weight of one is applied to the meanflow fields including surface points.

<u>`weight_viscous_mf = 1.0`</u>

This is a multiplier to modify the weighted norm of meanflow residuals in HANIM for points on viscous walls. If `weight_viscous_mf = 1.0`, the weights applied to the meanflow residual norm for those points are purely controlled by `node_weighting`.

<u>`weight_viscous_t = 1.0`</u>

This is a multiplier to modify the weighted norm of turbulence residuals in HANIM for points on viscous walls. If `weight_viscous_t = 1.0`, the weights applied to the turbulence residual norm for those points are purely controlled by `node_weighting`.

<u>`weight_symmetry_mf = 1.0`</u>

This is a multiplier to modify the weighted norm of meanflow residuals in HANIM for points on symmetry planes. If `weight_symmetry_mf = 1.0`, the weights applied to the meanflow residual norm for those points are purely controlled by `node_weighting`.

<u>`weight_symmetry_t = 1.0`</u>

This is a multiplier to modify the weighted norm of turbulence residuals in HANIM for points on symmetry planes. If `weight_symmetry_t = 1.0`, the weights applied to the turbulence residual norm for those points are purely controlled by `node_weighting`.

<u>`weight_farfield_mf = 0.0`</u>

This is a multiplier to modify the weighted norm of meanflow residuals in HANIM for points on farfield boundaries. If `weight_farfield_mf =`

`1.0`, the weights applied to the meanflow residual norm for those points are purely controlled by `node_weighting`.

<u>`weight_farfield_t = 0.0`</u>

This is a multiplier to modify the weighted norm of turbulence residuals in HANIM for points on farfield boundaries. If `weight_farfield_t = 1.0`, the weights applied to the turbulence residual norm for those points are purely controlled by `node_weighting`.

<u>`hanim_output_flag = .false.`</u>

This parameter controls the form of HANIM screen output. The option `hanim_output_flag = .true.` can be used to print detailed information about each individual HANIM module throughout the solution procedure. The option `hanim_output_flag = .false.` provides a concise format.

<u>`hanim_checker = .false.`</u>

A switch to turn on HANIM checks for boundary conditions and outputs of CFL number adaption and solution smoothing. This parameter is mainly used for debugging to attain additional info on the HANIM solution process. The default is `.false.` to avoid frequent file outputs.

Output of the HANIM solver HANIM is built upon a hierarchy of modules and thus additional information is printed to the screen as compared to the standard solver. The screen output indicates the progress and convergence of individual HANIM modules and can be used to diagnose the behavior of each module, particularly if there is a convergence issue. Suitable adjustments to the input parameters may result in better convergence.

The option `hanim_output_flag = .true.` in the `&hanim` namelist can be used to print detailed information about each individual HANIM module throughout the solution procedure (see the output example and discussion at the end of this section). To avoid excessive screen output, however, only a summary is printed with the default value `hanim_output_flag = .false.` Sample screen output of a steady-flow simulation is shown below.

```
10  0.647138820058768E-06  0.39754E-05  0.73966E+00 -0.10000E+01 -0.66408E-02
    0.101666651073165E-03  0.10023E-02 -0.23564E+03 -0.10000E+01  0.94906E+03
    10 HANIM 1 Prec(S mf 30 0.320E+00 t 5 0.465E+00) GCR(S 1 0.928E+00) Realz(S) Contrl(S) CFL 0.256E+03
    Lift  0.500835263487714E+01         Drag  0.573650518188323E+01
```

The first two lines show regular FUN3D output of the RMS norms of the residuals, the maximum residual values, and the locations at the current (10^{th}) nonlinear iteration. The third line contains HANIM output in a concise format. Here, the first integer is the nonlinear iteration number (for unsteady flow simulations, both the time step number and the subiteration index are printed).

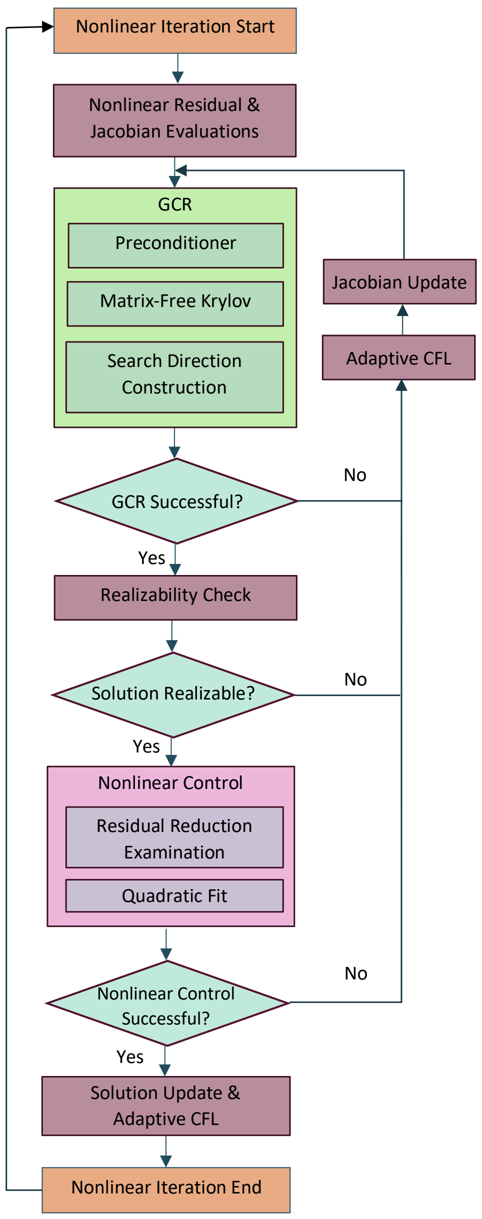

Nonlinear Iteration Start
Nonlinear Residual & Jacobian Evaluations
GCR
Preconditioner
Matrix-Free Krylov
Search Direction Construction
Jacobian Update
Adaptive CFL
GCR Successful?
No
Yes
Realizability Check
Solution Realizable?
No
Yes
Nonlinear Control
Residual Reduction Examination
Quadratic Fit
Nonlinear Control Successful?
No
Yes
Solution Update & Adaptive CFL
Nonlinear Iteration End

Figure B1: Flowchart illustrating key elements of HANIM algorithm.

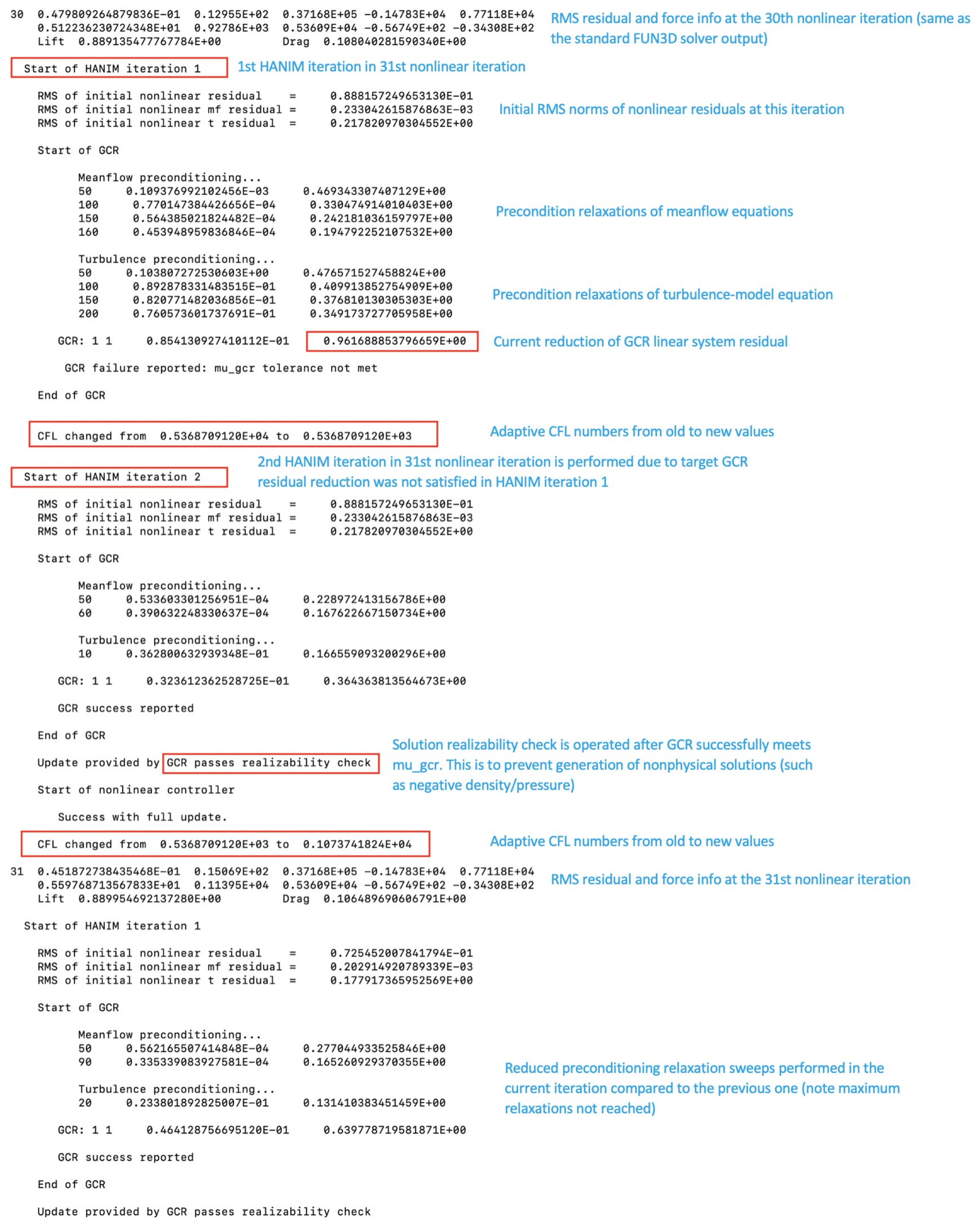
The notation "Prec" represents the preconditioner module; included in the parenthesis are its status, the actual number of preconditioning relaxations, and relative reductions in the preconditioner residuals. In this example, a total of 30 and 5 preconditioning relaxations were performed for the meanflow and turbulence-model equations, respectively, leading to relative reductions of `0.320E+00` and `0.465E+00` in the respective preconditioner residuals. The parenthesis following "GCR" include the GCR status, the total number of constructed search directions, and the relative reduction in the GCR residual. The notation “Realz” represents the realizability violation check and the value in parenthesis indicates its status. The notation “Contrl” represents the nonlinear controller module, and similarly, its status. The operational CFL number at the current step is output at the end of the line. The status indicators are defined in Table B2. In the case of failure, the solution is restored to the state at the beginning of the nonlinear iteration and the CFL number used in the pseudotime-stepping method is reduced. In the case of success, the solution is updated and the CFL number is adjusted accordingly.

Table B2: Status keywords of HANIM modules.

Preconditioner (Prec)	
S	Success in meanflow and turbulence preconditioning
Fm	Failure in meanflow equation preconditioning
Ft	Failure in turbulence equation preconditioning
GCR	
S	Success in GCR
F	Failure in GCR
Realizability Violation Check (Realz)	
S	Success in nodal realizability check
F	Failure in nodal realizability check
P	Solution smoothing to resolve realizability violation
Nonlinear Controller (Contrl)	
S	Success with full solution update
Sw	Success with partial solution update
Sr	Success with full/partial solution update, roundoff errors detected
F	Failure in the nonlinear controller, no update

Sample output with the option of `hanim_output_flag = .true.` is shown in Fig. B2; the blue text describes different blocks of HANIM output. The CFL number used in the pseudotime-stepping method is updated and printed out at each iteration.

Diagnosis of the HANIM solver performance When a convergence issue is detected, one should first check the CFL number shown near the end of the nonlinear iterations. One possible issue is a small CFL number (e.g., close to the minimum) that cannot recover. Potential causes include failures of the preconditioner or the GCR procedure, or frequent realizability violations. The preconditioner reports success if either the maximum relaxation sweeps have been reached or the target reduction of the preconditioner residual has been achieved. In general, at least a half to one order of magnitude reduction is recommended for the preconditioner. If preconditioner cannot attain the target residual reduction, allowing more preconditioner relaxation sweeps by increasing `meanflow_relaxations` and/or `turbulence_relaxations` may be


RMS residual and force info at the 30th nonlinear iteration (same as the standard FUN3D solver output)
1st HANIM iteration in 31st nonlinear iteration
Initial RMS norms of nonlinear residuals at this iteration
Precondition relaxations of meanflow equations
Precondition relaxations of turbulence-model equation
Current reduction of GCR linear system residual
Adaptive CFL numbers from old to new values
2nd HANIM iteration in 31st nonlinear iteration is performed due to target GCR residual reduction was not satisfied in HANIM iteration 1
Solution realizability check is operated after GCR successfully meets mu_gcr. This is to prevent generation of nonphysical solutions (such as negative density/pressure)
Adaptive CFL numbers from old to new values
RMS residual and force info at the 31st nonlinear iteration
Reduced preconditioning relaxation sweeps performed in the current iteration compared to the previous one (note maximum relaxations not reached)

```
30  0.479809264879836E-01  0.12955E+02  0.37168E+05 -0.14783E+04  0.77118E+04
    0.512236230724348E+01  0.92786E+03  0.53609E+04 -0.56749E+02 -0.34308E+02
    Lift  0.889135477767784E+00         Drag  0.108040281590340E+00

  Start of HANIM iteration 1

    RMS of initial nonlinear residual    =     0.888157249653130E-01
    RMS of initial nonlinear mf residual =     0.233042615876863E-03
    RMS of initial nonlinear t residual  =     0.217820970304552E+00

    Start of GCR

          Meanflow preconditioning...
          50      0.109376992102456E-03     0.469343307407129E+00
          100      0.770147384426656E-04     0.330474914010403E+00
          150      0.564385021824482E-04     0.242181036159797E+00
          160      0.453948959836846E-04     0.194792252107532E+00

          Turbulence preconditioning...
          50      0.103807272530603E+00     0.476571527458824E+00
          100      0.892878331483515E-01     0.409913852754909E+00
          150      0.820771482036856E-01     0.376810130305303E+00
          200      0.760573601737691E-01     0.349173727705958E+00

       GCR: 1 1     0.854130927410112E-01      0.961688853796659E+00

        GCR failure reported: mu_gcr tolerance not met

    End of GCR

    CFL changed from  0.5368709120E+04 to  0.5368709120E+03

  Start of HANIM iteration 2

    RMS of initial nonlinear residual    =     0.888157249653130E-01
    RMS of initial nonlinear mf residual =     0.233042615876863E-03
    RMS of initial nonlinear t residual  =     0.217820970304552E+00

    Start of GCR

          Meanflow preconditioning...
          50      0.533603301256951E-04     0.228972413156786E+00
          60      0.390632248330637E-04     0.167622667150734E+00

          Turbulence preconditioning...
          10      0.362800632939348E-01     0.166559093200296E+00

       GCR: 1 1     0.323612362528725E-01     0.364363813564673E+00

       GCR success reported

    End of GCR

    Update provided by GCR passes realizability check

    Start of nonlinear controller

       Success with full update.

    CFL changed from  0.5368709120E+03 to  0.1073741824E+04
31  0.451872738435468E-01  0.15069E+02  0.37168E+05 -0.14783E+04  0.77118E+04
    0.559768713567833E+01  0.11395E+04  0.53609E+04 -0.56749E+02 -0.34308E+02
    Lift  0.889954692137280E+00         Drag  0.106489690606791E+00

  Start of HANIM iteration 1

    RMS of initial nonlinear residual    =     0.725452007841794E-01
    RMS of initial nonlinear mf residual =     0.202914920789339E-03
    RMS of initial nonlinear t residual  =     0.177917365952569E+00

    Start of GCR

          Meanflow preconditioning...
          50      0.562165507414848E-04     0.277044933525846E+00
          90      0.335339083927581E-04     0.165260929370355E+00

          Turbulence preconditioning...
          20      0.233801892825007E-01     0.131410383451459E+00

       GCR: 1 1     0.464128756695120E-01     0.639778719581871E+00

       GCR success reported

    End of GCR

    Update provided by GCR passes realizability check
```

Figure B2: Sample output of the HANIM solver.

helpful. If the GCR procedure fails frequently, verify that the flux constructions used for the left- and right-hand sides are consistent. When simulating supersonic or hypersonic flows, a flux limiter is generally needed to avoid under- or over-shoots in the flow solution. If a flux limiter is needed, one of the pressure-based limiters (i.e., the h-series of limiters) should be selected; only h-series of limiters are currently supported by HANIM.

If a convergence issue occurs at a high CFL number and all HANIM modules function normally but steady-state residual appears to be oscillatory, then this may indicate a limit cycle (i.e., the solution remains trapped within a local basin) or some flow unsteadiness. Possible remedies include changing the initial condition, restarting from a different CFL number, tuning the spatial discretization, allowing more dissipation with eigenvalue smoothing, or using a tighter tolerance for the GCR module and/or more search directions. Sometimes, the Frechet derivative may fail to provide sufficient accuracy. Generally, the default value of `nominal_step_size` works well for turbulent flows at moderate or high speeds. For low-speed ($M \leq 0.01$) turbulent or laminar (and inviscid) flows, increasing the nominal step size by 3 orders of magnitude (`nominal_step_size=1.e-4`) may provide better accuracy of the real-valued Frechet derivatives and can benefit convergence.

B.4.17 &update_limits

This namelist specifies the limits in the update step and chemical source term for the generic gas path. All options in this namelist only apply for `eqn_type = 'generic'`.

```
&update_limits
  temperature_max                = 50000.0
  temperature_min                = 1.0
  density_min                    = 1.0e-06
  mass_fraction_min              = 1.0e-20
  freeze_source_frac             = 1.0e-10
  max_frac_target                = 0.5
  max_number_of_resets           = 5
  cpiv_min_factor                = 0.0001
  augment_kinetics_limiting      = .false.
  implicit_rate_limiting         = .true.
  disable_thermodynamic_warnings = .false.
  therm_fac_lim                  = 1.e+20
  omega_min                      = -1.0
/
```

`temperature_max = 50000.0`

The maximum temperature allowed, in units specified by `temperature_units`.

`temperature_min = 1.0`

The minimum temperature allowed, in units specified by `temperature_units`.

`density_min = 1.0e-06`

The minimum mixture density allowed, as a fraction of the freestream density.

`mass_fraction_min = 1.0e-20`

The minimum species mass fraction allowed.

`freeze_source_frac = 1.0e-10`

Freeze the chemical source term if a species mass fraction is below this value and the source term is depleting it further.

`max_frac_target = 0.5`

Limit the fractional update of positive definite quantities by this factor in the generic gas path.

`max_number_of_resets = 5`

The maximum percentage of nodes that can hit a maximum or minimum limit, before the code terminates. Note, an integer is expected here, and is converted to a percentage (i.e., "5" means 5 percent).

`cpiv_min_factor = 0.0001`

This variable sets the minimum value of the vibrational-electronic heat capacity as a fraction of the translational-rotational heat capacity for each species `i`. In some cases, ramping this value up to 0.01 can help suppress undershoot of vibrational temperature upstream of a strong shock. The vibrational-electronic heat capacity must be positive for stability.

`augment_kinetics_limiting = .false.`

When `.true.`, augment chemical kinetic source term limiting.

`implicit_rate_limiting = .true.`

When `.true.`, limit chemical rates if extrema exist in formulation.

`disable_thermodynamic_warnings = .false.`

In the generic gas path, warnings for temperatures and densities out of normal range as well as failure of some thermodynamic subiterations to converge are output by default. In some cases, these warnings overwhelm the output file for an issue that is transient or expected. This flag allows the user to disable such warnings.

`therm_fac_lim = 1.e+20`

This factor limits the magnitude of the thermal relaxation rate. The default represents an essentially unlimited value. For simulations where thermal nonequilibrium is only evident at the shock front or in a highly expanded wake, the user may observe convergence problems associated with stiffness because the major part of the flowfield is in thermal equilibrium. In these cases, the stiffness can be alleviated by setting this parameter to a value which is large enough to maintain thermal equilibrium where it is appropriate but small enough that stiffness is removed. If used, a starting value of 5.e+05 is recommended. If this limiting factor is too large, stiffness is not alleviated. If this limiting factor is too small, thermal equilibration is prevented in a nonphysical manner.

`omega_min = -1.0`

The minimum allowed value of the second turbulence variable in a 2-eqn turbulence model, as a fraction of the value at reference conditions. This should be less than one. Negative values or zero will be reset as: $\omega_{\min} =$

1×10^{-3} Pa/$(q *_{ref}\ k_{ref})$, where k_{ref} is the nondimensional, freestream turbulent kinetic energy, and $q{*}_{ref}$ is the (dimensional) dynamic pressure at reference conditions. Only used if `eqn_type='generic'`.

B.4.18 &shuffle_generic

This namelist enables a user to restart a generic gas solution with a different gas model. Shuffling an existing solution to restart a new solution may or may not save computer time or human time. Transients introduced by the abrupt change in the gas model are generally resolved quickly if the perturbations are minor, but may be overwhelming if the perturbations are too large. The new gas model may have a different number of species and/or a different number of energy equations and/or a different turbulence model. An important restriction on the use of shuffle is that the restarted solution must utilize the same number of processors as the original solution. This restriction derives from the formatting of metadata.

```
&shuffle_generic
  shuffle_enabled                = .false.
  shuffle_n_turb                 = -2
  shuffle_initial_trace_value    = 1.e-20
  shuffle_initial_low_vib_energy = 1.e-05
  shuffle_initial_turb_ke        = 1.e-05
  shuffle_initial_turb_diss      = 1.e-05
/
```

`shuffle_enabled = .false.`

Set this parameter to `.true.` if shuffling is requested.

`shuffle_n_turb = -2`

The option sets the new turbulence model type from the list below. The default value is not valid, which requires the user to change it to a valid input.

'`-1`' specifies an algebraic turbulence model.

'`0`' specifies laminar flow with no turbulence model.

'`1`' specifies the one-equation SA-Cataris turbulence model.

'`2`' specifies a two-equation turbulence model.

`shuffle_initial_trace_value = 1.e-20`

The initial trace concentration for newly added species.

`shuffle_initial_low_vib_energy = 1.e-05`

This is the initial low vibrational energy. It will be set to a new value by the thermodynamics routine in the flow solver.

`shuffle_initial_turb_ke = 1.e-05`

This is the initial turbulent kinetic energy.

`shuffle_initial_turb_diss = 1.e-05`

This is the initial turbulent dissipation rate ω.

B.4.19 &linear_solver_parameters

The Fun3D solution process involves constructing a linearization of the residual with appropriate time terms and then solving this linear system to compute the solution update. This namelist controls the solution process of this linear system. The linearization is grouped in to the meanflow equations and the turbulence model equations.

```
&linear_solver_parameters
  meanflow_sweeps   = 15
  turbulence_sweeps = 10
  linear_projection = .false.
  line_implicit     = 'off'
/
```

`meanflow_sweeps = 15`

This is the number of linear system relaxations at each steady iteration or time step of the meanflow equations when there is no turbulence model or a loosely-coupled turbulence model is present. For `eqn_type = 'generic'` (in which fully-coupled meanflow and turbulence relaxations are performed), this refers to all equations (meanflow and turbulence).

`turbulence_sweeps = 10`

This is the number of the linear system red-black relaxations at each steady iteration or time step of the turbulence equations, when the turbulence equations are loosely coupled. It has no effect for fully coupled meanflow and turbulence relaxation or a simulation without a turbulence model.

`linear_projection = .false.`

This options uses a Krylov projection method generalized conjugate gradient (GCR) to stabilize and improve convergence of linear system. This will execute multiple sets of red-black relaxation to form the GCR search directions, until a convergence criteria is met. The only penalty to using this option is increased execution time, which can be mitigated by reducing `meanflow_sweeps`.

`line_implicit = 'off'`

This option selects the relaxation scheme.

`'off'` uses point implicit relaxation.

`'on'` uses line implicit relaxation where lines are defined and point relaxation elsewhere. The line implicit feature requires construction of these lines prior to running Fun3D. The lines are stored in the a file named `[project_rootname].lines_fmt`, see section 4.4 for a description. The `aflr3_line_extraction` utility is distributed with FUN3D to generate these lines. `aflr3_line_extraction` is run in the user's case directory with a .`ugrid` mesh and .`mapbc` file. When running `aflr3_line_extraction`, the user will probably need to select 1 or 2 in order to change the project rootname to match the rootname of the .`ugrid` file.

B.4.20 &elasticity_gmres

This namelist controls the solution of the linear elasticity equations that are used for mesh deformation. Within FUN3D, a deforming mesh is modeled as a linear elastic medium, with the modulus of elasticity, E, typically taken as inversely proportional to either the distance from the nearest solid surface or the cell volume.

GMRES [104] is the preferred algorithm to use for solving the linear elasticity equations. The native FUN3D multicolor point-solve algorithm is used as the default preconditioner for GMRES. If the user elects not to use the GMRES solver, the multicolor point-solve algorithm is used as the linear-system solver. Using the point-solve algorithm without GMRES will generally result in slower convergence of the system of equations, especially for larger meshes. The default values indicated below are set based on the assumption that the linear solver used is GMRES; suggestions for appropriate values when not using GMRES are given where applicable.

If a few negative volumes are produced during mesh deformation, and the convergence of the elasticity equations has not stalled, decreasing the convergence tolerance may alleviate the problem. Increasing the total number of iterations allowed may be required to reach the requested convergence level. In rare cases, a different model for the modulus of elasticity may prevent negative volumes from occurring. If neither approach gets around the issue of negative volumes, the option to deform the mesh at each time step based on the mesh at the previous time step, rather than based on the initial mesh, may prove effective (see below for limitations of this option).

For coupled CFD/CSD rotorcraft simulations, and *only* for this specialized subset of simulations, a faster algebraic mesh deformation scheme may be used instead of solving the elasticity PDEs.

```
&elasticity_gmres
  algebraic_mesh_deform    = .false.
  deform_from_initial_mesh = .true.
  deformation_substeps     = 1
  use_substeps_each_step   = .false.
  elasticity               = 1
  elasticity_exponent      = 1.0
  poisson_ratio            = 0.0
  linear_solver            = 'gmres'
  linear_solver_module     = 'none'
  ileft                    = 1
  preconditioner_iters     = 5
  monitor_preconditioner   = .false.
  nsearch                  = 50
  nrestarts                = 10
  tol                      = 1.e-6
  tol_abs                  = 1.e-14
  residual_out_freq        = nsearch
  restart_deformation      = .true.
/
```

algebraic_mesh_deform = .false.

When .true., use a simple 'algebraic' mesh deformation scheme, bypassing the slower PDE solution of the linear elasticity equations. *Note: a value of .true. is only valid for coupled CFD/CSD overset rotor cases.* When .true., the remainder of the data in this namelist is ignored.

deform_from_initial_mesh = .true.

When .true., the volume mesh at each time step will be the result of a deformation of the *original* (input) mesh to fit the current surface mesh. For periodic surface meshes, this will result in a periodic volume mesh (provided the elasticity equations are well converged). When .false., the volume mesh at each time step will be the result of a deformation of the volume mesh from the *previous* time step. While this can lead to greatly improved robustness, especially for extreme deformations, it will in general *not* preserve volume mesh periodicity when the surface mesh is periodic. The degree to which the volume mesh is not periodic is a function of how extreme the surface motion is, and thus the lack of a truly periodic volume mesh may or may not be important, depending on application requirements. If the mesh cannot be successfully deformed for even the first time step (or for a one-time static deformation), where the previous mesh *is* the initial mesh, setting `deform_from_initial_mesh = .false.` alone will not be of any help. In that case try setting `deformation_substeps > 1`.

deformation_substeps = 1

This is the number of substeps to take when deforming the mesh. Often more than one substep will aid robustness at the expense of computational cost. If the first step of a moving-grid deformation or static-grid deformation cannot avoid negative volumes, setting
`deformation_substeps > 1` may circumvent the issue. It is important to note that while `deformation_substeps > 1` may yield a mesh that has positive volumes, if the initial/previous surface mesh and the target surface mesh are very different, a high level of cell distortion may also result; the user should verify the quality of the deformed mesh.

use_substeps_each_step = .false.

When .true. and `deformation_substeps > 1`, the requested number of deformation substeps will be taken for each time step in a time-dependent moving-grid case. When `.false.`, substeps are used only for the first time step.

`elasticity = 1`

This governs which grid metric is linked to the modulus of elasticity when the mesh is modeled as an elastic medium to enable mesh deformation.

`‘1’` Modulus of elasticity is based on the inverse distance from the nearest solid surface, raised to the power `elasticity_exponent`

`‘2’` Modulus of elasticity is based on the inverse of the cell volume, raised to the power `elasticity_exponent`

`elasticity_exponent = 1.0`

This governs the *inverse power* to which the grid metric selected by `elasticity` is raised to determine the modulus of elasticity. For example, if wall distance (slen) is chosen as the metric, and `elasticity_exponent = 1.0` then $E = 1/slen$. Note that `elasticity_exponent` should be positive as an inverse variation is implicitly assumed. In very rare instances, changing the default value (e.g. to 2.0) may help avoid negative volumes.

`poisson_ratio = 0.0`

This governs the Poisson ratio to use when modeling the mesh as an elastic medium. The value must be greater than -1.0 and less than 0.5.

`linear_solver = ’gmres’`

This governs which linear solver to use for solving the elasticity equations.

`‘gmres’` Use a GMRES implementation to solve the elasticity equations. This will default to SPARSKIT if it is included during configuration (see section A.13.15), otherwise, it will default to SLAT, the linear solver library distributed with FUN3D.

`‘point-multicolor’` Use the multicolor point solver to solve the elasticity equations.

`linear_solver_module = ’none’`

This governs which linear solver module plug-in to use for solving the elasticity equations.

`ileft = 1`

Switch to use left or right preconditioning with GMRES.

`‘0’` right preconditioning

`‘1’` left preconditioning

`preconditioner_iters = 5`

Number of preconditioner iterations to perform. If not using GMRES, suggest increasing this value to 500 as a starting point.

`monitor_preconditioner = .false.`

Flag to monitor the residual of the preconditioner; this is principally a developers tool.

`nsearch = 50`

Number of GMRES search directions. Memory requirements increase linearly with `nsearch`. Unused if the linear solver is not GMRES.

`nrestarts = 10`

Maximum number of restarts allowed when attempting to solve the system of equations to the specified tolerance. There is no memory penalty for increasing `nrestarts`. When using GMRES, the maximum number of search directions (iterations) performed to solve the system is `nsearch` × `nrestarts`. If not using GMRES, suggest increasing this value to 50 as a staring point, and the maximum number of iterations performed to solve the system is then `preconditioner_iters` × `nrestarts`.

`tol = 1.e-6`

Relative convergence tolerance used when solving the linear elasticity system. The linear system is considered converged when the residual ≤ `tol` × initial residual + `tol_abs`. The value of `tol` typically has the most direct influence on the convergence criterion.

`tol_abs = 1.e-14`

Absolute convergence tolerance used when solving the linear elasticity system. The linear system is considered converged when the residual ≤ `tol` × initial residual + `tol_abs`. There is generally little need to change the default value of `tol_abs`.

`residual_out_freq = nsearch`

Frequency at which the elasticity equation residual is output for convergence tracking. The first and final residuals are always output.

`restart_deformation = .true.`

When `.true.`, restart the solution of the elasticity equations for the current time step with the solution from the previous time step. This can result in significant CPU time saving for mesh deformation in time-accurate simulations. The downside is that the ability to *exactly* reproduce a time-periodic mesh may be compromised. For most practical applications this is likely not an issue.

B.4.21 &boundary_conditions

This namelist provides auxiliary information to the boundary conditions. Refer to section 3 for the definition of boundary numbers and other information.

```
&boundary_conditions
  farfield_point_vortex              = 0
  total_pressure_ratio(:)            = 1.0
  total_temperature_ratio(:)         = 1.0
  subsonic_inflow_velocity(:)        = 'normal'
  alpha_bc(:)                        = 0.0
  beta_bc(:)                         = 0.0
  theta1(:)                          = 0.0
  theta2(:)                          = 0.0
  theta3(:)                          = 0.0
  pt_amplitude(:)                    = 0.0
  pt_frequency(:)                    = 0.0
  pt_phase(:)                        = 0.0
  tt_amplitude(:)                    = 0.0
  tt_frequency(:)                    = 0.0
  tt_phase(:)                        = 0.0
  ampl(:)                            = 0.0
  freq(:)                            = 0.0
  phase(:)                           = 0.0
  random(:)                          = .false.
  ramp_constant(:)                   = 1.0
  start_time(:)                      = 0.0
  duration(:)                        = 0.0
  surge_amplitude(:)                 = 0.0
  pressure_relaxation(:)             = 1.0
  engine_reference_length            = 1.0
  engine_symmetry(:)                 = 1.0
  npr_set                            = 0.0
  static_pressure_ratio(:)           = 1.0
  static_pressure(:)                 = 0.0
  mach_bc(:)                         = 0.0
  massflow(:)                        = 0.0
  grid_units                         = 'nondim'
  q_set(:,1:5)                       = 0.0
  q_set_ramp(:)                      = 0
  profile_type(:)                    = 'radial_polynomial'
  patch_center(:,1:3)                = 0.0
  patch_scale(:)                     = 1.0
  profile_rho_coef(:,0:6)            = 0.0
  profile_u_coef(:,0:6)              = 0.0
  profile_v_coef(:,0:6)              = 0.0
```

```
profile_w_coef(:,0:6)                 = 0.0
profile_p_coef(:,0:6)                 = 0.0
profile_vib_energy_coef(:,0:6)        = 0.0
profile_tdata_id(:)                   = -1
profile_coef(:,1:3)                   = 0.0
blayer_profile_power_law_exponent(:) = 0.1666667
blayer_thickness(:)                   = 1.0
karman_constant_kappa(:)              = 0.41
log_law_constant_c(:)                 = 5.0
utau_inlet(:)                         = 0.0
vel_profile_file_name(:)              = ''
inlet_stochastic_fluctuations(:)      = .false.
fluctuations_type(:)                  = 'isotropic'
dns_data_file_name(:)                 = ''
utau_dns_ref(:)                       = 0.01
u_rms_inlet(:)                        = 0.05
wave_no_ratio(:)                      = 5
l_scale(:)                            = 0.1
t_scale_factor(:)                     = 2.0
no_of_modes(:)                        = 150
wall_velocity(:,1:3)                  = 0.0
rotation_center(:,1:3)                = 0.0
rotation_vector(:,1:3)                = 0.0
rotation_rate(:)                      = 0.0
unorm_bc(:)                           = 0.0
wall_temperature(:)                   = 1.0
farfield_turbulence(:)                = .true.
boundary_turbulence(:,1:7)            = 1.0e-12
wall_temp_flag(:)                     = .false.
wall_radeq_flag(:)                    = .false.
wall_emissivity(:)                    = 0.8
wall_emissivity_b(:)                  = 0.
wall_emissivity_c(:)                  = 0.
wall_emissivity_d(:)                  = 0.
wall_temp_relax(:)                    = 0.001
wall_catalysis_model(:)               = 'super-catalytic'
catalytic_efficiency_o(:)             = 0.
catalytic_efficiency_n(:)             = 0.
ablation_option_map(:)                = 0
ablation_recession                    = .false.
ablation_recession_freq               = 3000
start_recession                       = 0
bprime_flag_map(:)                    = 1
compute_mdot_initial_map(:)           = 1
freq_mdot_map(:)                      = 5000
```

```
  freq_wall_map(:)                    = 50
  uncoupled_ablation_flag_map(:)      = 0
  wall_blowing_model(:)               = 'none'
  virgin_density_wall(:)              = 1.
  char_density_wall(:)                = 1.
  CHONSi_frac_char_map(:,1)           = 1.
  CHONSi_frac_pyrolysis_map(:,1)      = 1.
  h_ablation_map(:,:)                 = 0.
  mdot_pressure_map(:,:)              = 0.
  t_sublimation_map(:,:)              = 0.
  plenum_t0(:)                        = 1000.
  plenum_p0(:)                        = 1000.
  plenum_id(:)                        = 0
  fixed_in_id(:)                      = 0
  fixed_in_rho(:)                     = 0.
  fixed_in_uvw(:,1:3)                 = 0.
  fixed_in_t(:)                       = 0.
  fixed_in_tv(:)                      = 0.
  fixed_in_turb(:,1:7)                = 0.
  dynamic_boundary_conditions         = .false.
  boundary_functional_name(:)         = '-1'
  field_point1(:,1:3)                 = 0.0
  field_point2(:,1:3)                 = 0.0
  specified_transition(:)             = .false.
  impose_pressure_gradient            = .false.
  pressure_gradient(:)                = 0.0
  number_of_porous_boundaries         = 0
  porous_bc(:)                        = 0
  auxiliary_bc(:)                     = 'none'
  auxiliary_condition(:)              = 'no-model'
  x_constant_boundary(:)              = .false.
  y_constant_boundary(:)              = .false.
  z_constant_boundary(:)              = .false.
  tol_const_coord                     = 1.0e-6
  number_of_engines                   = 0
  number_of_streams(:)                = 0
  inlet_bc(:)                         = 0
  core_bc(:)                          = 0
  cycle_name(:)                       = 'no_model'
  wall_function(:)                    = 'none'
  u_tau_fixed(:)                      = 0.0
  remove_phi                          = .false.
/
```

`farfield_point_vortex = 0`

This controls the point vortex correction for 2D flows. Only applicable for 2D.

'0' does not apply the point vortex correction.

'1' applies the point vortex correction based on a under relaxed total lift.

'2' applies the point vortex correction based on a frozen total lift from previous total lift in the restart file.

`total_pressure_ratio(:) = 1.0`

This is the ratio of plenum total pressure to reference pressure used by 7011 and 7015 boundary conditions. When using this boundary condition, the inflow Mach number is expected to be less than one.

`total_temperature_ratio(:) = 1.0`

This is the ratio of plenum total temperature to reference temperature used by 7011, 7015, and 7036 boundary conditions. When using this boundary condition, the inflow Mach number is expected to be less than one.

`subsonic_inflow_velocity(:) = 'normal'`

This sets the direction of the inflow velocity when using the 7011 or 7015 boundary conditions. The default is that the inflow velocity is normal to the boundary face.

'normal' for inflow normal to each element face in the patch

'alpha,beta' the angle of the inflow is specified by `alpha_bc` and `beta_bc` as shown in Fig. 4.

'interior' the angle of the inflow is specified by the Euler angles `theta1`, `theta2`, and `theta3` as shown in Fig. B7.

'offset' the angle of normal inflow to the patch boundary is rotated by the Euler angles `theta1`, `theta2`, and `theta3` as shown in Fig. B7.

'direct' the angle of flow is set directly to vector components defined by `theta1`, `theta2`, and `theta3`. This can be used on an outflow boundary.

'face_normal_outflow' for outflow normal to each element face in the patch

`alpha_bc(:) = 0.0`

When `subsonic_inflow_velocity = 'alpha,beta'`, this is the angle of attack in radians for 7011 and 7015 boundary condition inflow. This angle is with respect to the Cartesian axes, not the boundary face.

`beta_bc(:) = 0.0`

When `subsonic_inflow_velocity = 'alpha,beta'`, this is the angle of sideslip in radians for 7011 and 7015 boundary condition inflow. This angle is with respect to the Cartesian axes, not the boundary face.

`theta1(:) = 0.0`

When `subsonic_inflow_velocity = 'interior'` or `'offset'`, this is the Euler angle θ in radians for 7011 and 7015 boundary condition inflow. When `subsonic_inflow_velocity = 'direct'`, then θ directly specifies the components of a vector representing the flow direction.

`theta2(:) = 0.0`

When `subsonic_inflow_velocity = 'interior'` or `'offset'`, this is the Euler angle ϕ in radians for 7011 and 7015 boundary condition inflow. When `subsonic_inflow_velocity = 'direct'`, then θ directly specifies the components of a vector representing the flow direction.

`theta3(:) = 0.0`

When `subsonic_inflow_velocity = 'interior'` or `'offset'`, this is the Euler angle ψ in radians for 7011 and 7015 boundary condition inflow. When `subsonic_inflow_velocity = 'direct'`, then θ directly specifies the components of a vector representing the flow direction.

`pt_amplitude(:) = 0.0`

For use with the 7011 boundary condition, this sets the amplitude of pulsed total pressure. `ampl`*sin(`freq`*`simulation_time`+`phase`*pi/180).

`pt_frequency(:) = 0.0`

For use with the 7011 boundary condition, this sets the frequency of pulsed total pressure, see `ampl`.

`pt_phase(:) = 0.0`

For use with the 7011 boundary condition, this sets a phase shift (in degrees) of the inflow total pressure values, see `ampl`.

`tt_amplitude(:) = 0.0`

For use with the 7011 boundary condition, this sets the amplitude of pulsed total temperature conditions.

`tt_frequency(:) = 0.0`

For use with the 7103 boundary condition, this sets the frequency of pulsed total temperature, see `ampl`.

`tt_phase(:) = 0.0`

For use with the 7011 boundary condition, this sets a phase shift (in degrees) of the inflow total temperature, see `ampl`.

`ampl(:) = 0.0`

For use with the 7103 boundary condition, this sets the amplitude of pulsed inflow conditions. Only the velocity is varied using the equation $[u, v, w]$ = `ampl`*sin(`freq`*`simulation_time`+`phase`*pi/180).

`freq(:) = 0.0`

For use with the 7103 boundary condition, this sets the frequency of pulsed inflow velocity values, see `ampl`.

`phase(:) = 0.0`

For use with the 7103 boundary condition, this sets a phase shift (in degrees) of the inflow velocity values, see `ampl`.

`random(:) = .false.`

For use with the 7103 boundary condition, this creates random fluctuations in in the magnitude of the supersonic inflow conditions in the range [0,`ampl`] when `.true.`.

`ramp_constant(:) = 1.0`

For use with 7104 boundary condition, this ramps the velocity magnitude of supersonic inflow conditions with the exponential expression, (1 − exp(−`simulation_time`/`ramp_constant`)).

`start_time(:) = 0.0`

For use with the 5060 boundary condition, this is the start time, in terms of Fun3D `simulation_time`, of a delta function change to the subsonic outflow static pressure ratio.

`duration(:) = 0.0`

For use with the 5060 boundary condition, this sets the duration of the delta function change in units of Fun3D time, see `simulation_time`.

`surge_amplitude(:) = 0.0`

For use with the 5060 boundary condition, this sets the transient increase in the value of the outflow static pressure ratio as a percentage of the base line value. For example, a 2 percent transient applied to block 5 is input as `surge_amplitude(5) = 2.0`.

`pressure_relaxation(:) = 1.0`

For use with the 5051 or 7011 boundary conditions, this weights the user specified back pressure with the solution pressure or the specified total pressure with the solution total pressure at the boundary. For a partially nonreflecting boundary condition, set `pressure_relaxation` = 2.0. `p_applied` = `pressure_relaxation` * `static_pressure_ratio` + (1 - `pressure_relaxation`) * `pressure_internal` or in the case of total conditions, `pt_applied` = `pressure_relaxation` * `total_pressure_ratio` + (1 - `pressure_relaxation`) * `total_pressure_internal`

`engine_reference_length = 1.0`

Use this factor to scale the nondimensional time in the engine simulation package.

`engine_symmetry(:) = 1.0`

Use this factor to scale the area and massflow of an engine, e.g., the 7031 and 7036 massflow boundary conditions. This factor is used in the physical engine modeling routines, so that the actual massflow numbers can be tracked properly. For example, set to 2.0 for an inlet face on a symmetry plane or set to 360.0 for a one-degree axisymmetric wedge.

`npr_set = 0.0`

This sets the nozzle pressure ratio for nozzle component performance.

`static_pressure_ratio(:) = 1.0`

This is the ratio of specified boundary static pressure to reference (free stream) static pressure and is used by the 7012 (`subsonic_outflow_p0`) and 5051 (`back_pressure`) boundary conditions. The expected outflow Mach number must be less than one when using 7012. If there is expected to be mixed subsonic and supersonic outflow, 5051 will automatically extrapolate when the flow is supersonic. This parameter is used when `eqn_type = 'compressible'`.

`static_pressure(:) = 0.0`

This is the boundary static pressure in Pascals and is used when `eqn_type = 'generic'`.

`mach_bc(:) = 0.0`

This is Mach number on boundary face used by 5052 and 7031 boundary conditions. Outflow Mach number must be less than one.

`massflow(:) = 0.0`

This is massflow through boundary face in units of grid unit squared used by 7031 and 7036 boundary conditions—see also `engine_symmetry` and `total_temperature_ratio`. It is nondimensionalized according to $(\rho_\infty a_\infty)$ for `eqn_type = 'compressible'`.

<u>`grid_units = 'nondim'`</u>

This converts a user specified dimensional massflow to units of mesh unit squared. If massflow is input in units of mesh units squared, `grid_units` is not required.

`‘nondim’` for nondimensional input.

`‘m’` for meters.

`‘cm’` for centimeters.

`‘mm’` for millimeters.

`‘feet’` for feet.

`‘inches’` for inches.

<u>`q_set(:,1:5) = 0.0`</u>

This sets the primitive variables on boundary face used by boundary conditions 7100 and 7105.

<u>`q_set_ramp(:) = 0`</u>

When this is greater than zero, primitive variables on boundary face are ramped from zero to `q_set` over these iterations, for boundary conditions 7100 and 7105.

<u>`profile_type(:) = 'radial_polynomial'`</u>

This switches between inflow profiles,

`‘fixed’` imposes a fixed, normal velocity directed in to the computational domain.

`‘radial_polynomial’` defines a radial polynomial about `patch_center` of size `patch_scale` with `profile_rho_coef`, `profile_u_coef`, `profile_v_coef`, `profile_w_coef`, `profile_p_coef`. `profile_vib_energy_coef` and `profile_tdata_id` are also needed if `eqn_type = 'generic'`

`‘z_polynomial’` defines a z polynomial about `patch_center` of size `patch_scale` with `profile_rho_coef`, `profile_u_coef`, and `profile_p_coef`.

`‘power_law’` defines a power-law velocity profile function with `profile_coef`.

`‘boundary_layer_profile_power_law’` defines a boundary layer with a power-law velocity profile with `blayer_profile_power_law_exponent` and `blayer_thickness`.

`‘data_file’` defines a boundary layer with a profile from a given data file

`patch_center(:,1:3) = 0.0`

This is the center of a 7101 bc.

`patch_scale(:) = 1.0`

This scales the radius of a 7101 bc.

`profile_rho_coef(:,0:6) = 0.0`

This is the radius polynomial coefficients of density for 7101 bc.

`profile_u_coef(:,0:6) = 0.0`

This is the radius polynomial coefficients of u for 7101 bc.

`profile_v_coef(:,0:6) = 0.0`

This is the radius polynomial coefficients of v for 7101 bc.

`profile_w_coef(:,0:6) = 0.0`

This is the radius polynomial coefficients of w for 7101 bc.

`profile_p_coef(:,0:6) = 0.0`

This is the radius polynomial coefficients of pressure for 7101 bc.

`profile_vib_energy_coef(:,0:6) = 0.0`

This is the radius polynomial coefficients of vibration energy for 7101 bc. Only for `eqn_type = 'generic'`.

`profile_tdata_id(:) = -1`

This is the species grouping in the tdata file for 7101 bc. Only for `eqn_type = 'generic'`.

`profile_coef(:,1:3) = 0.0`

These three coefficients are wind speed reference, reference height, and power law exponent,
`profile_coef(:,1)*(z/profile_coef(:,2))**profile_coef(:,3)`.

`blayer_profile_power_law_exponent(:) = 0.1666667`

This is the exponent for power-law boundary layer profile. Use with `profile_type = 'boundary_layer_profile_power_law'`.

`blayer_thickness(:) = 1.0`

This is the thickness of the boundary layer in grid units. It is used by `profile_type = 'boundary_layer_profile_power_law'` and `profile_type = 'boundary_layer_profile_log_law'`.

`karman_constant_kappa(:) = 0.41`

This is the Kármán constant in a log-law boundary layer profile. Use with `profile_type = 'boundary_layer_profile_log_law'`.

`log_law_constant_c(:) = 5.0`

This is the constant C in a log-law boundary layer profile. Use with `profile_type = 'boundary_layer_profile_log_law'`.

`utau_inlet(:) = 0.0`

This is the frictional velocity for log-law boundary layer profile. Use with `profile_type(:) = 'boundary_layer_profile_log_law'`.

`vel_profile_file_name(:) = ''`

This is the data file containing mean velocity profile data (e.g. from DNS data), which can be applied at the inflow boundary. The ASCII file has a one-line header and then two columns of distance from the wall normalized by boundary layer thickness and mean velocity normalized by freestream speed of sound.

`inlet_stochastic_fluctuations(:) = .false.`

This is used to activate inflow stochastic turbulent fluctuations for BC 7101 when set to .true.

`fluctuations_type(:) = 'isotropic'`

This is the type of stochastic fluctuations to be generated at the inflow. They can be 'anisotropic' or 'isotropic'.

`dns_data_file_name(:) = ''`

This is the data file containing shear stress data (e.g., from DNS data). It is used to generate anisotropic fluctuations at the inlet. The ASCII file has a one-line header and then nine columns. The columns are wall distance, y^+, uu, vv, ww, uv, uw, vw, and ϵ.

`utau_dns_ref(:) = 0.01`

This is the frictional velocity used to scale the DNS data. It is used only for generating anisotropic fluctuations.

`u_rms_inlet(:) = 0.05`

This is the RMS value of the maximum amplitude of the turbulent fluctuations. It is used only for generating isotropic fluctuations.

`wave_no_ratio(:) = 5`

This is the ratio between the largest and smallest wave numbers. It is used only for generating isotropic fluctuations.

`l_scale(:) = 0.1`

This is the integral length scale of the fluctuations, which is usually about 10% of the boundary layer thickness. It is used for generating both isotropic and anisotropic fluctuations.

`t_scale_factor(:) = 2.0`

This is the scaling factor for the turbulent time scale of the turbulent fluctuations, `turbulent_timescale=(0.05*blayer_thickness/u_rms_inlet)/t_scale_factor`. It is used only for generating isotropic fluctuations.

`no_of_modes(:) = 150`

This is the number of modes used to generate the turbulent spectrum of the inlet fluctuations.

`wall_velocity(:,1:3) = 0.0`

This is the 4000 solid wall specified translational velocity. It should be tangent to the boundary to ensure a well-posed problem.

`rotation_center(:,1:3) = 0.0`

This is the 4000 solid wall specified rotational velocity center point, see `rotation_vector` and `rotation_rate`.

`rotation_vector(:,1:3) = 0.0`

This is the 4000 solid wall specified rotational vector, see `rotation_center` and `rotation_rate`. Should be unit length.

`rotation_rate(:) = 0.0`

This is the 4000 solid wall specified rotational rate in reference speed of sound per grid unit (`eqn_type = 'compressible'`) or reference speed per grid unit (`eqn_type = 'incompressible'`) ($\omega = \omega^* \frac{L^*_{ref}}{a^*_{ref} L_{ref}}$ or $\omega = \omega^* \frac{L^*_{ref}}{V^*_{ref} L_{ref}}$), see `rotation_center`, `rotation_vector`, and section 2.

`unorm_bc(:) = 0.0`

This is the specified velocity in the boundary normal direction, for 4000 solid walls.

`wall_temperature(:) = 1.0`

This is the ratio of wall temperature to reference temperature for `eqn_type = 'compressible'` or the wall temperature in Kelvins for `eqn_type = 'generic'`. If set to `-1`, then the wall temperature is computed so that there is zero heat flux, i.e., adiabatic. The `wall_temp_flag` must be set to `.true.` for this boundary condition to take effect, unless a radiative equilibrium wall temperature is desired, in which case `wall_temp_flag` must be set to `.false`. Note: for `eqn_type = 'generic'` the wall temperature must be explicitly set as `>1` or `-1`.

`farfield_turbulence(:) = .true.`

This parameter toggles the inflow of turbulence from a farfield boundary for turbulence models. The default value is `.true.`. When `farfield_turbulence(ib) = .false.`, the freestream turbulent parameters are set to `boundary_turbulence` on the specified boundary.

`boundary_turbulence(:,1:7) = 1.0e-12`

This input sets the boundary turbulence values when `farfield_turbulence` is set to false.

`wall_temp_flag(:) = .false.`

This must be `.true.` when specifying the wall temperature via the namelist variable `wall_temperature`, unless the `wall_temperature` is being used to set the initial wall temperature but a radiative equilibrium wall temperature is desired, in which case `wall_temp_flag` must be set to `.false`. The default `.false`. sets $\frac{T_{\text{wall}}}{T_{\text{ref}}} = 1 + \frac{\sqrt{P_r}(\gamma-1)M^2}{2}$ for `eqn_type = 'compressible'`, see Anderson and Bonhaus [2].

`wall_radeq_flag(:) = .false.`

Compute the wall temperature via the Stefan-Boltzmann Law. The radiative equilibrium wall temperature is computed from the heating rate q_{wall} using $q_{wall} = \epsilon\sigma T^4_{radeq}$ where the surface emissivity ϵ is entered as `wall_emissivity` and σ is the Stefan-Boltzmann constant.

`wall_emissivity(:) = 0.8`

This is ϵ_0, where emissivity is specified as a function of wall temperature with the expression $\epsilon = \epsilon_0 + T(\epsilon_b + T(\epsilon_c + T\epsilon_d))$. The other coefficients are entered via the following three variables.

`wall_emissivity_b(:) = 0.`

This is ϵ_b in the above equation.

`wall_emissivity_c(:) = 0.`

This is ϵ_c in the above equation.

`wall_emissivity_d(:) = 0.`

This is ϵ_d in the above equation.

`wall_temp_relax(:) = 0.001`

This is the relaxation factor η used for `wall_radeq_flag` wall temperature boundary condition. The wall temperature is updated as $T^{new} = T^{old} + \eta * (T_{radeq} - T^{old})$

`wall_catalysis_model(:) = 'super-catalytic'`

This defines the catalytic efficiency of the wall to promote recombination of atoms to molecules. Allowable options are:

`‘super-catalytic’` Forces the mass fraction of species at the wall to equal the mass fractions specified for the free stream in the `tdata` file.

`‘fully-catalytic’` Specifies a catalytic efficiency of 1 thereby forcing homogeneous recombination of atoms diffusing to the wall.

`‘non-catalytic’` Specifies a zero mole fraction gradient at the wall - signifying zero catalytic efficiency.

`‘equilibrium-catalytic’` Computes the equilibrium chemical composition of species at the wall temperature and pressure.

`‘constant-catalytic’` Catalytic efficiency is user specified constants

`‘Stewart-RCG’` Reaction cured glass from Stewart

`‘Zoby-RCG’` Reaction cured glass from Zoby

`‘Scott-RCG’` Reaction cured glass from Scott

`‘CSiC’` Experimental CSiC from JSC for X-38

`‘RCC-LVP’` Stewart NASA TM 112206

`‘CCAT-ACC’` Shuttle RCC from Stewart NASA TM 112206

`‘CSiC-SNECMA’` Derived from Stewart RCC

`catalytic_efficiency_o(:) = 0.`

This is the fraction of diffusion flux of atomic oxygen striking wall that is converted to molecular oxygen, when `wall_catalysis_model = 'constant-catalytic'`.

`catalytic_efficiency_n(:) = 0.`

This is the fraction of diffusion flux of atomic nitrogen striking wall that is converted to molecular nitrogen, when `wall_catalysis_model = 'constant-catalytic'`.

`ablation_option_map(:) = 0`

This is an integer that specifies whether the pyrolysis ablation rate and wall temperature are computed in addition to the char ablation rate. This option only affects cases with `bprime_flag_map` equal to 0 or 1.

`‘0’` The pyrolysis ablation rate and wall temperature are computed, in addition to the char ablation rate, assuming steady-state ablation.

`‘1’` The pyrolysis ablation rate and wall temperature are held constant (they are set to the values present in `ablation_from_laura.m`) while the char ablation rate is computed.

`ablation_recession = .false.`

This flags triggers the surface deformation and mesh movement through aeroelastic deformation. Note: this flags only applies to the internal Fun3D coupled-ablation mechanism and is not applicable to Fun3d-CHAR coupled ablation. Default: .false.

`ablation_recession_freq = 3000`

It is an integer that specifies the frequency of surface recession update. This flags only works with `ablation_recession = .true.`. Default: 3000

`start_recession = 0`

It is an integer that specifies when to start applying recession for surface deformation. Default: 0

`bprime_flag_map(:) = 1`

This is an integer defining if the b-prime approach is applied. Applicable only for blowing model `equil_char_quasi_steady = 0`.

`‘0’` Do not use bprime approach, and instead use a rigorous diffusion model. This option is consistent with the ”Fully-Coupled” approach defined in Ref. [2].

`‘1’` Use b-prime approach. This option is consistent with the ”Partially-Coupled” approach defined in Ref. [2].

`‘2’` Hold the ablation rate and wall temperature constant from the restart file, while applying the rigorous diffusion model (thus, the surface energy balance and char equilibrium constraint are not satisfied). This option is sometimes useful when transitioning from a bprime flag = 1 computation to a bprime flag = 0 computation.

`compute_mdot_initial_map(:) = 1`

This is an integer defining if the ablation rates are computed before the first flowfield iteration.

`‘0’` Applies the ablation rates and wall temperatures present in the `ablation_from_laura.m` file.

`‘1’` Computes the ablation rates and wall temperatures before the first flowfield iteration.

freq_mdot_map(:) = 5000

For bprime_flag_map = 1, this is an integer defining frequency of updating ablation rates and wall temperature.

freq_wall_map(:) = 50

For bprime_flag = 1, this is an integer defining frequency of update to ablation wall boundary conditions. For bprime_flag = 0, an integer defining frequency of update to the surface energy balance solution, which defines the wall temperature.

uncoupled_ablation_flag_map(:) = 0

This is an integer defining if an uncoupled ablation analysis is applied. The uncoupled ablation option is included to provide a baseline solution for the coupled ablation analysis.

'0' Do not apply an uncoupled ablation analysis.

'1' Apply an uncoupled ablation analysis to a converged nonablating flowfield.

wall_blowing_model(:) = 'none'

This is the blowing or ablation model.

'none' No wall blowing

'specified' blowing rate is user specified function of pressure (see also mdot_press)

'porous_chamber' Special options for simulation of buoyancy driven flow in pressurized rig for BNNT production

'quasi_steady' Compute ablation rate as function of surface energy balance and equilibrium catalytic bc

'equil_char_quasi_steady' Include equilibrium char approximation

'FIAT' Couple to material response code FIAT (not active)

virgin_density_wall(:) = 1.

This is the density (kg/m^3) of thermal protection system ablator in virgin state (prior to heating level sufficient to cause any reactions).

char_density_wall(:) = 1.

This is the density (kg/m^3) of remaining char in ablator after binding resins have pyrolized.

CHONSi_frac_char_map(:,1) = 1.

See definition below for CHONSi_frac_pyrolysis_map

`CHONSi_frac_pyrolysis_map(:,1) = 1.`

These arrays set elemental mass fraction (second index) of C, H, O, N, Si, Fe, Mg, Na, B species for char and pyrolysis gas. The fractions in each array should sum to 1.

`h_ablation_map(:,:) = 0.`

This is a vector of extent 3 used to compute the heat of ablation in MJ/kg for quasi steady blowing option as `h_ablation_0(1) + (h_ablation_0(2)) log pw + (h_ablation_0(3))(log pw)**2` where `pw` is the local pressure, in atmospheres.

`mdot_pressure_map(:,:) = 0.`

This is a vector of extent 2 is used to set the blowing or suction distribution defined as `mdot_pressure_0(1) + (mdot_pressure_0(2))*p/(rho_inf*V_inf**2)` where `p` is the local pressure, `rho_inf` is the reference density, and `V_inf` is the reference velocity. Positive value produces blowing distribution, while negative value produces suction distribution.

`t_sublimation_map(:,:) = 0.`

This is a vector of extent 3 used to compute the sublimation temperature in degrees Kelvin for quasi steady blowing option as `t_sublimation_0(1) + (t_sublimation_0(2)) log pw + (t_sublimation_0(3))(log pw)**2` where `pw` is the local pressure, in atmospheres.

`plenum_t0(:) = 1000.`

For use with the 7021 boundary condition, this is the total plenum temperature in Kelvin.

`plenum_p0(:) = 1000.`

For use with the 7021 boundary condition, The total plenum pressure in N/m^2 (Pascals) feeding this boundary.

`plenum_id(:) = 0`

For use with the 7021 boundary condition (one or more `rcs_jet` plenum bcs), the jet plenum contains this species set from the file `tdata`. For example, if an RCS jet is firing H_2 and O_2 into an air stream, the `tdata` file may look like:

```
One Temperature
N
O
N2 0.76
O2 0.24
NO
```

```
H2 0.5
O2 0.5
OH
H
O
```

In this case, if the boundary to the plenum is surface number 5 then `plenum_id(5)=2`, the second grouping of species in the `tdata` file. The numbers following the species name define the mass fraction of that species at the inflow boundary. The sum of the mass fraction in each group must equal one. Species groups are separated by blank lines and multiple RCS jets may be defined in this manner.

`fixed_in_id(:) = 0`

For use with the 70XX boundary condition (one or more supersonic inflow bcs) the supersonic inflow boundary condition contains the species set in the same way as the `plenum_id` for the `rcs_jet` plenum boundary condition described above.

`fixed_in_rho(:) = 0.`

For use with the 70XX boundary condition (one or more supersonic inflow bcs) the dimensional inflow mixture density in kg/m^3.

`fixed_in_uvw(:,1:3) = 0.`

For use with the 70XX boundary condition (one or more supersonic inflow bcs) the dimensional inflow Cartesian velocity components in m/sec.

`fixed_in_t(:) = 0.`

For use with the 70XX boundary condition (one or more supersonic inflow bcs) the dimensional inflow translational rotational temperature in Kelvin.

`fixed_in_tv(:) = 0.`

For use with the 70XX boundary condition (one or more supersonic inflow bcs) the dimensional inflow vibrational-electronic temperature in Kelvin.

`fixed_in_turb(:,1:7) = 0.`

For use with the 70XX boundary condition (one or more supersonic inflow bcs) This is for the turbulence models for a one-equation model this is the ratio of the inflow eddy viscosity to the inflow molecular viscosity, for a two-equation model this is: the inflow turbulence intensity and the ratio of the inflow eddy viscosity to the inflow molecular viscosity Full Reynolds stress models are not currently supported.

dynamic_boundary_conditions = .false.

This option creates an alternate .mapbc file, called [project]_dynamic_boundary.mapbc, that stores the most recent values of boundaries that are being dynamically managed by the code. If [project]_dynamic_boundary.mapbc is present upon code startup, the values in the dynamic mapbc file will over-ride the values specified in the fun3d.nml namelist file. The default value is .false., for no dynamic boundary condition updating using this capability.

When dynamic_boundary_conditions = .true. a Proportional-Integral-Derivative (PID) controller is engaged to update the values of specific inflow and/or outflow boundaries as determined by the user.

The following is an example of the namelist entries used to drive the outflow static pressure ratio at an outflow boundary to achieve a target Mach number at a some other point in the flow. The inflow and outflow boundaries are numbered 3 and 4, respectively. Boundaries 2 and 6 are modeled as adiabatic, no-slip walls. The back pressure is driven by the local conditions at the survey point, field_point1.

```
&boundary_conditions
grid_units = 'meters' ! grid is in meters
wall_temperature(2) = -1.      ! wall
wall_temp_flag(2) = T
wall_temperature(6) = -1.      ! wall
wall_temp_flag(6) = T
! Using (p/p_t) isentropic equation
total_pressure_ratio(3) = 1.02828
! Using (T/T_t) isentropic equation
total_temperature_ratio(3) = 1.00800
static_pressure_ratio(4) = 1.010   ! first guess

! Solution functional
! Set to true to engage the boundary condition controller
dynamic_boundary_conditions = .true.
! Name of boundary condition used by the controller
! The index 4 relates to the number of the boundary being
! modified by the controller
! Use the back pressure bc
boundary_functional_name(4)   = 'back_pressure'
field_point1(4,:) = 0.0,0.01, 0.01 ! survey point
update_frequency(4)   = 500
/
```

A namelist file named controller.nml, is required to drive the PID-controller and is included here. This file is separate from the fun3d.nml and must be included in the job execution directory.

```
&tunnel_control
number_of_controllers = 1       ! One controller to set up
controller_points(4)  = 1       ! Controls boundary 4
find_mach(4)  = .true.  ! Target is Mach number
target_mach(4)  = 0.20    ! Target value is 0.2
initial_delay(4)  = 1000    ! Delay before starting controller
kp(4)  = 0.025   ! Proportional term
ki(4)  = 0.2e-5  ! Integral term
kd(4)  = 1.0e-6  ! Derivative term
/
```

A description of the parameters used in `&tunnel_control` can be found in section B.13.1.

`boundary_functional_name(:) = '-1'`

This is the boundary condition type used when `dynamic_boundary_conditions = .true.` The type will vary depending on the form of the functional.

`'-1'` No functional boundary condition.

`'subsonic_inflow_pt'` Use the total-conditions inflow boundary condition.

`'back_pressure'` Use the static pressure with supersonic extrapolation boundary condition.

`'subsonic_outflow_p0'` Use the static pressure boundary condition not allowing reverse flow or supersonic extrapolation.

`'back_pressure_nrbc'` Use a non-reflecting, static pressure outflow boundary condition.

`field_point1(:,1:3) = 0.0`

These are the Cartesian (x,y,z) locations of the first flow survey point used by the controller software.

`field_point2(:,1:3) = 0.0`

These are the Cartesian (x,y,z) locations of the second flow survey point used by the controller software.

`specified_transition(:) = .false.`

When `.true.`, a turbulent transition point will be imposed on the solution.

`impose_pressure_gradient = .false.`

When `.true.`, a global pressure gradient in the direction of `pressure_gradient` will be imposed as a source term to the residual.

`pressure_gradient(:) = 0.0`

The nondimensional pressure gradient in Cartesian coordinates, imposed when `impose_pressure_gradient = .true.`.

`number_of_porous_boundaries = 0`

This is the number of `porous_bc` entries.

`porous_bc(:) = 0`

This is the boundary index of each porous boundary face. The default of 0 is not a valid boundary index and must be set.

`auxiliary_bc(:) = 'none'`

This is the name of the auxiliary source-term based boundary condition function.

`'fixed'` imposes a fixed, normal velocity directed out of the computational domain, for this porous boundary.

`'none'` does not apply a function to this source-term based boundary condition.

`auxiliary_condition(:) = 'no-model'`

This sets a source-term based auxiliary condition that will be applied over a `viscous_solid` boundary. For example, a fixed, normal velocity directed out of the computational domain can be specified for one porous boundary, `5`, with

```
number_of_porous_boundaries = 1
porous_bc(1)                 = 5
auxiliary_condition(5)      = 'unorm'
auxiliary_bc(5)             = 'fixed'
unorm_bc(5)                 = 0.02
```

`'no-model'` does not apply a model.

`'unorm'` imposes a fixed, normal velocity directed out of the computational domain, for this porous boundary.

`x_constant_boundary(:) = .false.`

This specifies that a boundary is an x constant face for mesh movement, constraining motion the to be tangent to face. Set automatically for x-symmetry boundaries.

y_constant_boundary(:) = .false.

This specifies that a boundary is an y constant face for mesh movement, constraining motion the to be tangent to face. Set automatically for y-symmetry boundaries.

z_constant_boundary(:) = .false.

This specifies that a boundary is an z constant face for mesh movement, constraining motion the to be tangent to face. Set automatically for z-symmetry boundaries.

tol_const_coord = 1.0e-6

This is the tolerance for verifying that a boundary surface is a planar boundary for mesh movement by restricting the minimum and maximum value in the "constant" direction.

number_of_engines = 0

number_of_engines is the number of simple turbofan engine models to be defined by the user. The mass flow rate of the core flow can be linked to the inlet fan face or the mass flow rate of the fan face can drive the core flow. The default value for the number of engines is 0.

The following is an example of the namelist entries for a single stream, single engine model where the conditions of the core flow are driven by the conditions at the fan inlet face. The engine outflow and inflow boundaries are numbered 2 and 1, respectively. The grid has one symmetry plane, so the symmetry factor is set to 2. The mass flow measured at the face face sets the mass flow rate for the engine core flow.

```
&boundary_conditions
grid_units = 'inches'
massflow(2) = 31.675 ! fan inlet face - boundary 2
number_of_engines  = 1 ! just one engine here
number_of_streams(1) = 1 ! Engine 1 has a single stream
engine_symmetry(1) = 2. ! Engine 1 grid is half plane symmetric
inlet_bc(1) = 2  ! Engine 1 fan boundary index - driver
core_bc(1) = 1  ! Engine 1 core boundary index - follower
cycle_name(1) = 'massflow' ! Engine 1 bc type for the core flow
/
```

number_of_streams(:) = 0

User input stating the number of streams for each engine input. The default value of number_of_streams is 0.

inlet_bc(:) = 0

Index of the inlet (outflow) boundary of the simple engine models. The default value of inlet_bc is 0.

`core_bc(:) = 0`

Index of core flow boundary of the simple engine models. The default value of `core_bc` is 0.

`cycle_name(:) = 'no_model'`

Name of the simple engine cycle. The default is `'no_model'`.

`‘no_model’` Default-no simple engine model.

`‘lib_engine.so’` Name of engine simulation deck shared object C-code library.

`‘massflow’` Drive the simple engine core flow using the fan inlet flow conditions.

`‘total_conditions’` Drive the simple engine fan flow using the engine core flow conditions.

`wall_function(:) = 'none'`

This is the name of the wall function model.

`‘none’` integrates to the the wall (no wall function).

`‘dlr’` uses the wall function of Knopp, Alrutz, and Schwamborn. [105] Requires a 4100 (weak) viscous boundary condition. Can be used with `turbulence_model = 'sa'`, `turbulence_model = 'sst'`, or their variations.

`u_tau_fixed(:) = 0.0`

This allows for specifying a fixed friction velocity for a wall function boundary.

`remove_phi = .false.`

Calibration coefficient in Wilcox u tau wall function model for the pressure gradient effect

B.4.22 &periodicity

This namelist contains variables that specify inputs relevant to periodic boundary conditions.

```
&periodicity
  periodic_dir(1:3) = 0
  periodic_tol      = 1.e-10
/
```

<u>`periodic_dir(1:3) = 0`</u>

This integer array specifies the Cartesian coordinate direction associated with each periodic boundary face pair in the simulation. The `periodic_dir(1)` entry is for faces marked with the 6100 boundary condition. The `periodic_dir(2)` entry is for 6101 and `periodic_dir(3)` is for 6102. For example, if a single pair of boundaries have periodicity, they should be given the 6100 boundary condition and `periodic_dir(1)` should reflect the desired Cartesian direction of periodicity.

'`0`' when this periodic boundary pair is not used.

'`1`' when this periodic boundary pair is normal to the x-Cartesian direction.

'`2`' when this periodic boundary pair is normal to the y-Cartesian direction.

'`3`' when this periodic boundary pair is normal to the z-Cartesian direction.

<u>`periodic_tol = 1.e-10`</u>

Tolerance on determining coincident points on periodic boundary pairs.

B.4.23 &two_d_trans

This namelist is used to specify a 2D transition location. If the airfoil is split into upper and lower patches, the transition location can be specified independently on each patch. If there is only a single patch, it can be split with a z value to designate the upper and lower airfoil surfaces. This transition specification is limited to specifying transition on a single-element configuration such as an airfoil or a flat plate. Only a single transition location is supported for multielement airfoils. Transition is modeled by turning off the turbulent production terms in "laminar" regions of the grid. This option is only valid for the perfect gas SA turbulence model. This is the same approach taken in CFL3D and NSU3D.Fun3D results from this approach for a DLR-F6 transonic cruise condition are shown in Lee-Rausch et al. [106]

```
&two_d_trans
  turb_transition  = .false.
  use_2d_values    = .false.
  upper_patch      = 1
  lower_patch      = 1
  use_z_value      = .false.
  z_location       = 0.0
  upper_x_location = 0.0
  lower_x_location = 0.0
/
```

`turb_transition = .false.`

This must be `.true.` to specify laminar regions of flow during a turbulent flow simulation.

`use_2d_values = .false.`

This enables 2D transition specification.

`upper_patch = 1`

This is the upper patch with specified transition.

`lower_patch = 1`

This is the lower patch with specified transition.

`use_z_value = .false.`

This allows a single patch to be split into upper and lower surfaces of an airfoil by a z plane.

`z_location = 0.0`

This is the z location to split the airfoil if `use_z_value = .true.`

upper_x_location = 0.0

This is the upper surface x transition location.

lower_x_location = 0.0

This is the lower surface x transition location.

B.4.24 &three_d_trans

This namelist is used to specify 3D boundary layer transition locations. The command line option `--turb_transition` is required to activate. If you run the flow solver without the `--turb_transition`, it will default to fully turbulent even though you have the laminar boundaries defined. Transition is modeled by turning off the turbulent production terms in "laminar" regions of the grid. This option is only valid for the perfect gas SA turbulence model. This is the same approach taken in CFL3D and NSU3D.Fun3D results from this approach for a DLR-F6 transonic cruise condition are shown in Lee-Rausch et al. [106].

This namelist provides two different means to specify transition locations on no-slip walls. If `use_3d_values` is set to `.true.`, the values of `n_transition_group`, `transition_group_patches`, `transition_x1`, `transition_y1`, `transition_x2`, and `transition_y2` will be used to determine transition locations.

If `use_file_values` is set to `.true.`, the solver will expect the user to provide a file named `project_transition.dat`. This file contains transition location information specified on no-slip boundary surfaces in a Tecplot format as follows:

```
title="arbitrary title"
variables = x y z id_l2g trans
zone t="arbitrary name", solutiontime= 0.100000E+01, strandid=0,
i=xxx, j=yyy, f=feblock
[Block of xxx values of x variable]
[Block of xxx values of y variable]
[Block of xxx values of z variable]
[Block of xxx values of id_l2g variable]
[Block of xxx values of trans variable]
[Block of 4-point sets in the zone making up yyy surface faces]
```

where `x`, `y`, and `z` are the x-, y-, and z-coordinates for each grid point in the zone; `id_l2g` is the mapping for each point in the zone into the global volume mesh; and `trans` is a transition marker provided by the user and is set to either 0 for laminar flow, or 1 for turbulent flow. As many zones may be provided as desired; not all no-slip surfaces must be included. Any points on no-slip boundaries not specified here will be assumed fully turbulent. The face connectivity data at the end of each zone may represent triangles or quadrilaterals; the third index should be repeated for triangles.

The most straightforward means to generate this file may be to use Fun3D to first generate a Tecplot boundary surface file containing the surfaces on which transition specifications will be provided. Instruct Fun3D to write only x-, y-, and z-coordinates as well as the `id_l2g` variable to this file. The user may then augment these data with one additional field containing the transition data in whatever manner is easiest.

```
&three_d_trans
  use_3d_values               = .false.
  use_file_values             = .false.
  n_transition_group          = 1
  transition_group_patches(:) = '1'
  transition_x1(:)            = 0.0
  transition_y1(:)            = 0.0
  transition_x2(:)            = 0.0
  transition_y2(:)            = 1.0
/
```

`use_3d_values = .false.`

This turns 3D transition specification on.

`use_file_values = .false.`

This turns 3D transition via file-based specification on.

`n_transition_group = 1`

This is the number of patch groups, limited to 100.

`transition_group_patches(:) = '1'`

This is the patch indexes for each group. Commas and dashes can be used to specify ranges, i.e., `'1,2,5-7'`.

`transition_x1(:) = 0.0`

This is the x value for determining the start of the transition line.

`transition_y1(:) = 0.0`

This is the y value for determining the start of the transition line.

`transition_x2(:) = 0.0`

This is the x value for determining the end of the transition line.

`transition_y2(:) = 1.0`

This is the y value for determining the end of the transition line.

B.4.25 &filters

This namelist provides auxiliary information to the porous media modeling of thick screens, honeycomb, heat exchanger, or similar, using a momentum-sink methodology. The model has an inertial term for all three Cartesian directions and a viscous loss term, as shown by the following equation. $S_i = \frac{\mu}{k}v_i + C_i\frac{1}{2}\rho|v|v_i$, where i = 2, 3, 4 are associated with the x-, y-, and z-momentum equations respectively.

```
&filters
  passive_filter_flag         = .false.
  number_of_fences            = 0
  fence_shape(:)              = 'none'
  fence_thickness(:)          = 1.0
  corners(:,:,1:3)            = 0.0
  pressure_loss_factor(:,2:4) = 0.0
  permeability(:)             = 1e+12
  tree_height(:)              = 0.0
  z_lower(:)                  = 0.0
  z_upper(:)                  = 0.0
  number_of_corners(:)        = 0
  polygon_x(:,:)              = 0.0
  polygon_y(:,:)              = 0.0
/
```

`passive_filter_flag = .false.`

This activates the source term based, porous medium model for simulating screens or the momentum loss incurred by devices like heat exchangers or honeycomb. The default is `.false.`.

`number_of_fences = 0`

The number of thick fences to be modeled. The default is 0.

`fence_shape(:) = 'none'`

The user can define fence shapes for defining thick shapes within the computational volume.

`'none'` Default-no shape defined.

`'hex'` Default-no shape defined.

`'cylinder'` Specify the endpoints and radii of a cylinder.

`'polygon'` Specify the Cartesian x-y coordinates of a polygon shaped footprint.

`fence_thickness(:) = 1.0`

This is the thickness of the fence.

`corners(:,:,1:3) = 0.0`

The coordinate of the all the corners of the base of a polyhedron.

`pressure_loss_factor(:,2:4) = 0.0`

These are the Cartesian (x,y,z) inertial loss coefficients and are equal to $C_{2,3,4}$ terms in the source term equation.

`permeability(:) = 1e+12`

This is the viscous loss contribution coefficient and is equal to the k term in source term equation.

`tree_height(:) = 0.0`

This is the height of the tree.

`z_lower(:) = 0.0`

This is the lower z coordinate of the polyhedron base.

`z_upper(:) = 0.0`

This is the upper z coordinate of the polyhedron top.

`number_of_corners(:) = 0`

This is the number of corners in the polyhedron base.

`polygon_x(:,:) = 0.0`

These are the x coordinates of the polygon base.

`polygon_y(:,:) = 0.0`

These are the y coordinates of the polygon base.

B.4.26 &flow_initialization

This namelist allows the user to initialize regions of the meanflow solution with quantities other than freestream. A maximum of 100 volumes can be defined. The volumes may overlap each other as well as domain boundaries. In the event that a grid point is contained in more than one volume, a subsequent volume in this file will supersede the volumes listed before it. Boundary conditions supersede the flow initialization.

```
 &flow_initialization
   number_of_volumes  = 0
   mks_units          = .false.
   type_of_volume(:)  = 'none'
   center(1:3,:)      = 0.0
   radius(:)          = 0.0
   point1(1:3,:)      = 0.0
   point2(1:3,:)      = 0.0
   radius1(:)         = 0.0
   radius2(:)         = 0.0
   rate(1:3,:)        = 0.0
   rho(:)             = 1.0
   c(:)               = 1.0
   u(:)               = 0.0
   v(:)               = 0.0
   w(:)               = 0.0
   mass_fraction(:,:) = 1.e-20
   temperature(:,:)   = 1000.
   import_from        = ''
/
```

number_of_volumes = 0

This is the number of initialization volumes.

mks_units = .false.

When using eqn_type='compressible' or eqn_type='generic' the initialization parameters, density and velocity, are input in normalized form. When using eqn_type='generic', setting mks_units=.true., the parameters density and velocity can be input in MKS units, i.e. kg/m^3 and m/s respectively.

type_of_volume(:) = 'none'

This is the type of initialization volume.

'box' for a box. The diagonal corners are specified by point1 and point2.

‘`sphere`’ for a sphere. The position and size is specified by `center` and `radius`.

‘`cylinder`’ for a cylinder with size `radius`. The center axis is defined between `point1` and `point2`.

‘`cone`’ for a cone or frustum. The center axis is defined between `point1` and `point2`. Two radii are required, `radius1` at `point1` and `radius2` at `point2`. A frustum is specified with two nonzero radii.

‘`rotation`’ for rotation about a point. The position and rate is specified by `center` and `rate`. Currently only valid for perfect gas simulations.

`center(1:3,:) = 0.0`

This is the center of the ’`sphere`’ volume type.

`radius(:) = 0.0`

This is the radius of the ’`sphere`’ and ’`cylinder`’ volume types.

`point1(1:3,:) = 0.0`

This is one end of the ’`cone`’ or ’`cylinder`’ volume types or one corner of a ’`box`’ volume type.

`point2(1:3,:) = 0.0`

This is the other end of the ’`cone`’ or ’`cylinder`’ volume types or the opposite corner of a ’`box`’ volume type.

`radius1(:) = 0.0`

This is the radius at `point1` of ’`cone`’ volume type.

`radius2(:) = 0.0`

This is the radius at `point2` of ’`cone`’ volume type.

`rate(1:3,:) = 0.0`

This is the angular velocity vector.

`rho(:) = 1.0`

This is the nondimensional density in the volume.

`c(:) = 1.0`

This is the nondimensional speed of sound in the volume.

`u(:) = 0.0`

This is the nondimensional x-component of velocity in the volume.

`v(:) = 0.0`

This is the nondimensional y-component of velocity in the volume.

`w(:) = 0.0`

This is the nondimensional z-component of velocity in the volume.

`mass_fraction(:,:) = 1.e-20`

This is the species mass fraction array in the volume. The maximum number of species (first index) is hardwired to 50 in this namelist. The second index is the volume index. All mass fractions are initialized to trace values (1.e-20) so it is only necessary to define non-trace species. This option only applies to `eqn_type='generic'`.

`temperature(:,:) = 1000.`

This is the translational-rotational and vibrational electronic temperature array in the volume, in units of Kelvin. First index is 1 for the translational-rotational temperature and 2 for the vibrational-electronic temperature. The second index is the volume index. This option only applies to `eqn_type='generic'`.

`import_from = ''`

Initialize the solution from this filename or use freestream when empty. Only `.solb` format is implemented. Other forms of flow initialization in this namelist are applied after this file is read. The compressible solution is expected in primitive variables $[\rho, u, v, w, p]$ and the incompressible solution is expected in $[u, v, w, p]$. Turbulence working variables are appended to these variable lists as necessary. Generic gas variables are u, v, w, turbulence working variables, temperature(s), and species densities.

B.4.27 &component_parameters

This namelist provides expanded ability to track forces, moments, and mass-flows according to user-specified groups of boundary patches that define a "component." See `component_name` for a description of the output file names. The `grid_units` in `&boundary_conditions` are used to calculate dimensional quantities.

```
&component_parameters
  number_of_components         = 0
  component_name(:)            = ''
  component_type(:)            = 'boundary'
  component_count(:)           = -1
  component_input(:)           = ''
  list_forces                  = .false.
  component_symmetry(:)        = 1.0
  area_reference_ct            = 1.0
  component_body(:)            = -1
  component_xmc(:)             = x_moment_center
  component_ymc(:)             = y_moment_center
  component_zmc(:)             = z_moment_center
  component_cref(:)            = x_moment_length
  component_bref(:)            = y_moment_length
  component_sref(:)            = reference_area
  component_yaw(:)             = 0.0
  component_pitch(:)           = 0.0
  component_roll(:)            = 0.0
  component_hinge_sweep(:)     = 0.0
  component_hinge_dihedral(:)  = 0.0
  component_alpha_driver       = 0
  calculate_cd(:)              = .false.
  calculate_thrust_ratio(:)    = .false.
  massflow_component(:)        = 0
  throat_area(:)               = 0.0
  npr(:)                       = 0.0
  ntr(:)                       = 0.0
  allow_flow_through_forces    = .false.
  calculate_arp1420_distortion = .false.
  inlet_distortion_boundary(:) = 0
  number_of_rakes(:)           = 0
  rake_clock_angle(:)          = 0.0
  rake_location(1:3,:)         = 0.0
  rake_radius(:)               = 0.0
  number_of_rings(:)           = 0
  rake_points(1:3,:,:,:)       = 0.0
  circle_center(1:3,:)         = 0.0
  circle_normal(1:3,:)         = 0.0
  circle_radius(:)             = 0.0
  corner1(1:3,:)               = 0.0
  corner2(1:3,:)               = 0.0
  corner3(1:3,:)               = 0.0
  corner4(1:3,:)               = 0.0
  box_lower_corner(1:3,:)      = 0.0
  box_upper_corner(1:3,:)      = 0.0
  sphere_center(1:3,:)         = 0.0
  sphere_radius(:)             = 0.0
  gas_properties_bc(:)         = 0
  point1(1:3,:)                = 0.0
  point2(1:3,:)                = 0.0
  point3(1:3,:)                = 0.0
  number_of_points             = 0
  points(1:3,:,:)              = 0.0
/
```

`number_of_components = 0`

This is the number of components (groups of boundary patches) to track.

`component_name(:) = ''`

This is the component output filename, `[project_rootname]_fm_[component_name].dat`. Files will be overwritten if this name is not set.

`component_type(:) = 'boundary'`

Boundaries and a selection of analytic geometries can be specified for tracking forces, moments, mass flow rate and other integrated quantities such as density, temperature, and pressure. The current choices for sampling geometries are are `'circle'`, `'sphere'`, `'quad'`, and `'box'`.

An example of the namelist entry to analyse the flow through or around a set of boundaries would look like:

```
number_of_components = 3                ! three components to track
component_count(1) = -1               ! count from list given
component_name(1) = 'wing'           ! label of component
component_input(1) = '12'             ! wing boundary number

component_count(2) = 1                ! specified count
component_name(2) = 'slat'           ! label of component
component_input(2) = '2'              ! slat boundary number

component_count(3) = 2                ! specified count
component_name(3) = 'wing_slat'      ! label of component
component_input(3) = '2,12'           ! boundary numbers
```

In this example, the outflow of a nozzle is a circle and is shown to be sampled in two ways. This namelist define a circle that spans a flow exiting a nozzle and request calculation of the discharge coefficient and thrust ratio.

```
number_of_components = 2              ! two components
component_count(1) = 1               ! just one part to this component
component_input(1) = '0'             ! not a boundary
component_type(1) = 'circle'       ! sample geometry
circle_center(1:3,1) = 0.0, 0.0, 0.0  ! center point
circle_normal(1:3,1) = 1.0, 0.0, 0.0  ! normal of the circular plane
circle_radius(1) = 5.0             ! radius of circle
component_name(1) = 'Exit'          ! label of component
component_symmetry(1) = 4.0            ! grid is quarter-plane symmetric
throat_area(1) = 0.603          ! throat area of gridded component

! apply to both components
npr(1:2) = 352.    ! input nozzle pressure ratio
ntr(1:2) = 4535.   ! input nozzle temperature ratio
calculate_cd(1:2) = T       ! request discharge coefficient
calculate_thrust_ratio(1:2) = T       ! request thrust ratio

component_count(2) = 1              ! just one part to this component
component_input(2) = '0'            ! not a boundary
component_type(2) = 'three_point' ! sample geometry
point1(:,2) = 0.0, 0.0,     5.0    ! point 1 on circumference
point2(:,2) = 0.0, 3.5355, -3.5355 ! point 2 on circumference
point3(:,2) = 0.0, 4.3301, -2.5    ! point 3 on circumference
component_name(2) = 'Outflow_3pt' ! label of component
component_symmetry(2) = 4.0           ! symmetry
```

The following is an example of using `'quad'`, `'sphere'`, and `'box'`.

```
number_of_components = 3
component_count(1) = -1
component_input(1) = '0'             ! not a boundary
component_name(1) = 'throat'
component_type(1) = 'quad'
corner1(:,1) = 1.062590 0.150 0.06342
corner2(:,1) = 1.064208 0.150 0.06342
corner3(:,1) = 1.064208 0.130 0.06342
corner4(:,1) = 1.062590 0.130 0.06342
throat_area(1) = 1.82697e-6 ! m^2
npr(1) = 800
ntr(1) = 5.86
calculate_cd(1) = T
```

```
    component_count(2) = 1
    component_input(2) = '0'            ! not a boundary
    component_name(2) = 'long_box'
    component_type(2) = 'box'
    box_lower_corner(1:3,2) = -5.0, 0.0, 0.0
    box_upper_corner(1:3,2) = 10.0, 1.0, 1.0

    component_count(3) = 1
    component_input(3) = '0'            ! not a boundary
    component_name(3) = 'big_ball'
    component_type(3) = 'sphere'
    sphere_center(1:3,3) = -5.0, 0.0, 0.0
    sphere_radius(3) = 15.0
```

The default value is `'boundary'`

`component_count(:) = -1`

This is the number of boundary patches assigned to a component. If `-1` is given, this number is computed implicitly from `component_input`.

`component_input(:) = ''`

This is the list of boundary patches to assigned to a component. Boundary indexes are separated with commas and dashes can be used to specify ranges, i.e., `'1,2,5-7'`.

`list_forces = .false.`

This option is deprecated. If `number_of_components` is greater than zero, `list_forces` is automatically set to `.true.`. Write flow quantities for each component to a file named `[project_rootname]_stream_info.dat`.

`component_symmetry(:) = 1.0`

The factor to multiply existing mesh surface by to obtain the complete area. For example, if only 1/4 revolution of an axisymmetric nozzle is modeled, `component_symmetry(ib)` should be set to 4.0.

`area_reference_ct = 1.0`

Alternate reference area for calculating a thrust coefficient instead of thrust ratio.

`component_body(:) = -1`

This is the associated body for grid motion assigned to a component. The default is to ignore this association. This body will override the moment centers.

`component_xmc(:) = x_moment_center`

This is x-coordinate of the moment center assigned to a component. The default value comes from the `&force_moment_integ_properties` force and moment namelist.

`component_ymc(:) = y_moment_center`

This is y-coordinate of the moment center assigned to a component. The default value comes from the `&force_moment_integ_properties` force and moment namelist.

`component_zmc(:) = z_moment_center`

This is z-coordinate of the moment center assigned to a component. The default value comes from the `&force_moment_integ_properties` force and moment namelist.

`component_cref(:) = x_moment_length`

This is the x-direction reference length assigned to a component used to nondimensionalize moments about y (pitching moment). The default value comes from the `&force_moment_integ_properties` force and moment namelist.

`component_bref(:) = y_moment_length`

This is the y-direction reference length assigned to a component used to nondimensionalize moments about x (rolling moment) and z (yawing moment). The default value comes from the force and moment namelist, `&force_moment_integ_properties`.

`component_sref(:) = reference_area`

This is the reference area assigned to a component used to nondimensionalize forces and moments. The default value comes from the force and moment namelist, `&force_moment_integ_properties`.

`component_yaw(:) = 0.0`

This is the yaw Euler angle (in degrees). The order of application of the Euler angles is yaw-pitch-roll; each successive transform is applied to the axis system resulting from the previous transform. The defined rotation expresses the Cartesian X-, Y-, and Z-total forces in a rotated (e.g., body axis) system as reported in the `'[project_name]_stream_info.dat'` output file.

`component_pitch(:) = 0.0`

This is the pitch Euler angle (in degrees), see `component_yaw`.

`component_roll(:) = 0.0`

This is the roll Euler angle (in degrees), see `component_yaw`.

`component_hinge_sweep(:) = 0.0`

This is the hinge sweep angle in degrees assigned to a component used to nondimensionalize moments about y (pitching moment). The default value is zero.

`component_hinge_dihedral(:) = 0.0`

This is the hinge dihedral angle in degrees assigned to a component used to nondimensionalize moments about x (rolling moment). The default value is zero.

`component_alpha_driver = 0`

This specifies which component is to be used to drive the constant lift coefficient logic. The default value is zero, i.e., use the lift coefficient from the total force calculation using all the solid wall boundaries. Activated with command line option `--cl_const [cl] [gain]`, where `cl` is lift target and `gain` is feedback gain on the integral of error.

`calculate_cd(:) = .false.`

This will request an ideal mass flow calculation for component n.

`calculate_thrust_ratio(:) = .false.`

This will request a thrust ratio calculation for component n.

`massflow_component(:) = 0`

Specifies what component to get the physical mass flow from to calculate the ideal thrust. This is typically the component assigned to the nozzle

`throat_area(:) = 0.0`

This is the throat area associated with this component. It should only include that area that is represented in the computational domain (i.e., not both halves of a symmetric domain). It can be computed with `&sampling_parameters` via the `type_of_geometry(:) = 'circle'` and `type_of_data(1) = 'integrated'`.

`npr(:) = 0.0`

This is the nozzle pressure ratio associated with the component. Typically, it has the same value as the variable `total_pressure_ratio` in the `&boundary_conditions` namelist.

`ntr(:) = 0.0`

This is the nozzle temperature ratio associated with the component. Typically, it has the same value as the variable `total_temperature_ratio` in the `&boundary_conditions` namelist.

`allow_flow_through_forces = .false.`

Pressure drag and skin friction forces are by default calculated only on boundary faces designated as solid surfaces (viscous or inviscid). To include the pressure and momentum flux forces due to nozzles or inlets `allow_flow_through_forces` should be set to `.true.`. The flow-through faces to be tracked should be listed in the `component_count` and `component_input` lists accordingly.

`calculate_arp1420_distortion = .false.`

Setting this to `.true.` requests calculation of the SAE inlet distortion standard 1420 across either a boundary face or a disk. An example of the namelist entry to request analysis of the flow through a circular disk geometry looks like:

```
calculate_arp1420_distortion(4) = .true.
inlet_distortion_boundary(4) = 0    ! survey flow
component_count(4) = 1
component_input(4) = '0'
component_type(4) = 'circle'
circle_center(1:3,4) = 0.0, 0.0, 0.0
circle_normal(1:3,4) = 1.0, 0.0, 0.0
circle_radius(4) = 5.5
component_name(4) = 'AIP'
component_symmetry(4) = 1.0
number_of_rakes(4) = 8
number_of_rings(4) = 5
rake_location(1:3,4) = 0.01, 0.0, 0.0  ! nudge off of boundary
rake_radius(4) = 3.75
rake_clock_angle(4) = 0.01 ! keep rake off exact centerline
```

An example of the namelist entry to request analysis of the flow through a the boundary face would look like:

```
calculate_arp1420_distortion(4) = .true.
inlet_distortion_boundary(4) = 2    ! survey boundary 2
component_count(4) = 1
component_input(4) = '2'
component_name(4) = 'AIP'
component_symmetry(4) = 1.0
number_of_rakes(4) = 8
number_of_rings(4) = 5
rake_location(1:3,4) = 0.01, 0.0, 0.0  ! nudge off of boundary
rake_radius(4) = 3.75
rake_clock_angle(4) = 0.01 ! keep rake off exact centerline
```

The default setting for `calculate_arp1420_distortion` is `.false.`.

<u>`inlet_distortion_boundary(:) = 0`</u>

This defines the Fun3D boundary index to use for the SAE 1420 distortion analysis.

<u>`number_of_rakes(:) = 0`</u>

This defines the number of "spokes" of the distortion rake, moving radially out from the hub to the outer edge of the inlet face, Used when `calculate_arp1420_distortion = .true.`. The current maximum number of rakes is 36.

<u>`rake_clock_angle(:) = 0.0`</u>

When `calculate_arp1420_distortion = .true.`, this defines the offset angle of the rake array in degrees for each component rake definition. The default clock angle is 0. degrees.

<u>`rake_location(1:3,:) = 0.0`</u>

This defines the `(x,y,z)` location of the component rake array and can be combined with `calculate_arp1420_distortion = .true.`. The position of the second inlet distortion rake array would be defined by `rake_location(2,1:3) = x,y,z` , for example. The default location is (0.0,0.0,0.0)

<u>`rake_radius(:) = 0.0`</u>

When `calculate_arp1420_distortion = .true.`, this defines the radius of the component rake array . The default radius is 0.

<u>`number_of_rings(:) = 0`</u>

When `calculate_arp1420_distortion = .true.`, defines the number of points on each spoke of the distortion rake. It is assumed each rake has the same number of points. The current maximum number of rings is 36.

<u>`rake_points(1:3,:,:,:) = 0.0`</u>

If `rake_location` and `rake_radius` are not defined then each individual point must be defined. This defines the `(x,y,z)` position of each rake probe. For example, `rake_points(1:3,2,1,3) = x,y,z` is the position of the first point on the third (radial) rake for the second inlet distortion calculation.

<u>`circle_center(1:3,:) = 0.0`</u>

Defines the `(x,y,z)` position of a control volume circle.

`circle_normal(1:3,:) = 0.0`

Defines the `(nx,ny,nz)` direction of a control volume circle.

`circle_radius(:) = 0.0`

Defines the radius of a control volume circle.

`corner1(1:3,:) = 0.0`

This is the coordinate of the first corner of a `'quad'`; the corners proceed clockwise.

`corner2(1:3,:) = 0.0`

The coordinate of the second corner of a `'quad'`.

`corner3(1:3,:) = 0.0`

The coordinate of the third corner of a `'quad'`.

`corner4(1:3,:) = 0.0`

The coordinate of the fourth corner of a `'quad'`.

`box_lower_corner(1:3,:) = 0.0`

This is the coordinate of the lower corner of a `'box'`.

`box_upper_corner(1:3,:) = 0.0`

This is the coordinate of the upper corner of a `'box'`.

`sphere_center(1:3,:) = 0.0`

This is the coordinate of `'sphere'` center; it fixes the location.

`sphere_radius(:) = 0.0`

This is the radius for `'sphere'`; it fixes the size.

`gas_properties_bc(:) = 0`

For generic gas simulations, identifies the boundary owning the gas properties to be used for analysis of the component.

`point1(1:3,:) = 0.0`

This defines the `(x,y,z)` location of the first of three points defining a circle in space using `'three_point'`.

`point2(1:3,:) = 0.0`

This defines the `(x,y,z)` location of the second of three points defining a circle in space using `'three_point'`.

<u>`point3(1:3,:) = 0.0`</u>

This defines the `(x,y,z)` location of the third of three points defining a circle in space using `'three_point'`.

<u>`number_of_points = 0`</u>

This is the number of points to be input for a component geometry using `component_type()='points'`. The current maximum is 64.

<u>`points(1:3,:,:) = 0.0`</u>

This defines the `(x,y,z)` location of a series of points using `component_type()='points'`.

B.4.28 &time_avg_params

This namelist controls the bookkeeping of time average and root mean square statistics for every point in the domain. Tracking these statistics enables visualization of variables like `p_tavg`, `u_trms`, etc. See the visualization output variable namelists `&volume_output_variables`, `&boundary_output_variables`, and `&sampling_output_variables` for details. When these statistics are tracked, the file `[project_rootname]_TAVG.1` maintains information across restarts. This statistics file is similar to the `[project_rootname].flow` restart file in that is not intended for the user to interact with directly.

```
&time_avg_params
  itime_avg           = 0
  prior_time_avg      = -1
  use_prior_time_avg  = 1
  tavg_header_version = 1
/
```

`itime_avg = 0`

This controls collection of temporal statistics.

`'0'` do not compute time averaging statistics.

`'1'` compute time averaging statistics required for visualization.

`prior_time_avg = -1`

This option is deprecated, see `use_prior_time_avg`.

`use_prior_time_avg = 1`

This controls if a prior execution should contribute to the temporal statistics.

`'1'` combines the prior statistics stored in `[project_rootname]_TAVG.1` with the statistics of this execution. If `[project_rootname]_TAVG.1` is not available in the current directory, only the statistics from this execution will be calculated.

`'0'` discards prior statistics. The file `[project_rootname]_TAVG.1` will be ignored, if it exists.

`tavg_header_version = 1`

This option controls the variables for which statistics are collected.

`'1'` for primitive variable averages and root mean squares.

`'2'` for primitive variable averages, primitive variable root mean squares, and the averages of `u'v',u'w',v'w',mu_t,vort_mag`.

B.4.29 &global

This namelist controls the frequency of visualization output and the logging of command line options. It also serves to control some global options otherwise set by command-line input

```
&global
  moving_grid                    = .false.
  grid_motion_only               = .false.
  grid_motion_and_dci_only       = .false.
  body_motion_only               = .false.
  boundary_animation_freq        = 0
  boundary_animation_freq_tavg   = 0
  volume_animation_freq          = 0
  visualization_freq             = 0
  slice_freq                     = 0
  record_command_lines           = .false.
  recompute_turb_dist            = .true.
  turb_dist_update_freq          = 1
  ignore_grid_velocity           = .false.
  heating_from_integral          = .true.
  isdouble                       = 0
  visualizer                     = ''
  plt_tecplot_output             = .false.
  time_timestep_loop             = .false.
  estimate_remaining_time        = .false.
  write_heating_output           = .false.
/
```

`moving_grid = .false.`

This governs whether the grid is moving or stationary

`grid_motion_only = .false.`

This turns off the flow solve during a simulation for which `moving_grid = .true.`. If the simulation involves overset grids, this command overrides `dci_on_the_fly` and DCI files are not output.

`grid_motion_and_dci_only = .false.`

This turns off the flow solve during a simulation for which `moving_grid = .true.`. If the simulation involves overset grids, this command honors `dci_on_the_fly` and DCI files are output if `dci_on_the_fly = .true.`.

`body_motion_only = .false.`

This turns off both the flow solve and the linear elasticity solve during a simulation for which `moving_grid = .true.` and `grid_motion = 'deform'` `boundary_animation_freq = 0`

This is the visualization output frequency of the domain boundaries. Zero is no output, `-1` is output at the end of run, and a positive integer is periodic output every `boundary_animation_freq` iterations. See the `&boundary_output_variables` namelist for more details.

`boundary_animation_freq_tavg = 0`

This is the visualization output frequency of time-averaged data on the domain boundaries. Zero is no output, `-1` is output at the end of run, and a positive integer is periodic output every `boundary_animation_freq_tavg` iterations.

`volume_animation_freq = 0`

This is the visualization output frequency of the domain volume. Zero is no output, `-1` is output at the end of run, and a positive integer is periodic output every `volume_animation_freq` iterations. See `&volume_output_variables` namelist for more details.

`visualization_freq = 0`

This is the visualization interface output frequency of the domain. Zero is no output and a positive integer is periodic output every `visualization_freq` iterations.

`slice_freq = 0`

This is the output frequency of boundary slices for visualization and to obtain loads. Zero is no output, `-1` is output at the end of run, and a positive integer is periodic output every `slice_freq` iterations. See `&slice_data` namelist for more details.

`record_command_lines = .false.`

This creates a file `temp.commands` that contains the command line arguments

`recompute_turb_dist = .true.`

This is a flag that allows for the distance function to be recomputed after grid deformation or when overset grids are moved.

`turb_dist_update_freq = 1`

This is the frequency with which the distance function is updated when `recompute_turb_dist = .true.`.

`ignore_grid_velocity = .false.`

If `.true.`, ignore the grid-velocity terms in a moving-grid case, e.g., static aeroelastic analysis.

`heating_from_integral = .true.`

If true, heating output is determined from integral conservation law. If false, heating output is determined from gradients to triangular or quadrilateral surface face averaged to common surface node. The false option is more robust for radiative equilibrium wall boundary conditions. The true option is thought to be a more accurate representation of the finite-volume solution.

`isdouble = 0`

This value controls the precision of binary Tecplot output files associated with volume, boundary, sampling, and schlieren outputs. A value of 0 indicates single precision; a value of 1 indicates double precision.

`visualizer = ''`

Name of optional visualization interface-conforming plug-in

`plt_tecplot_output = .false.`

When true, this will force binary tecplot output in plt instead of szplt (when code is linked with new versions of TecIO, it will be invalid if linked with TecIO-MPI), TecIO-MPI only supports szplt

`time_timestep_loop = .false.`

Time the actual timestep loop.

`estimate_remaining_time = .false.`

Estimate the remaining simulation time. This uses a running average of time per timestep and does not consider convergence criteria.

`write_heating_output = .false.`

Write integrated convective heat transfer in Watts to screen.

B.4.30 &volume_output_variables

This namelist controls volume variable output. Output frequency is controlled by `volume_animation_freq` in the `&global` namelist. The resulting volume-data files have the following naming convention for `export_to='tecplot'` output:

```
if freq > 0:
[project_rootname]_part[P]_tec_volume_timestep[T](.dat|.plt|.szplt)
if freq < 0:
[project_rootname]_Part[P]_tec_volume(.dat|.plt|.szplt)
```

where `P` = 1,2,...,`nproc` (number of processors), `T` is the iteration number, and `N` is the value of `volume_animation_freq`. The file extension is `.dat` for ASCII Tecplot™ format, `.plt` for binary Tecplot™ format (2013 and earlier), and `.szplt` for new binary Tecplot™ format. Within the files, a single zone is written, with the zone title "time 0.0000000E+00 processor 32" where the time value is the integer iteration number for steady-state cases, and the current (nondimensional) time for time-dependent cases.

When using `export_to='tecplot'`, the portion of the domain written to the resulting file(s) may be confined to subvolumes to reduce file size. Any mesh element containing at least one mesh point within the user-defined subvolumes will be included in the output. See inputs below for more information.

A request to output an unavailable variable will overruled, i.e., `turb1` will be forced to `.false` regardless of user input when there is no turbulence model in the simulation.

```
&volume_output_variables
  export_to                  = 'tecplot'
  output_initial_state       = .false.
  x                          = .true.
  y                          = .true.
  z                          = .true.
  primitive_variables        = .true.
  rho                        = .false.
  u                          = .false.
  v                          = .false.
  w                          = .false.
  p                          = .false.
  entropy                    = .false.
  mach                       = .false.
  temperature                = .false.
  iblank                     = .false.
  imesh                      = .false.
  vort_mag                   = .false.
  vort_x                     = .false.
```

```
vort_y                        = .false.
vort_z                        = .false.
q_criterion                   = .false.
div_vel                       = .false.
turbulent_fluctuations        = .false.
uuprime                       = .false.
vvprime                       = .false.
wwprime                       = .false.
uvprime                       = .false.
uwprime                       = .false.
vwprime                       = .false.
cp                            = .false.
dp_pinf                       = .false.
volume                        = .false.
residuals                     = .false.
res1                          = .false.
res2                          = .false.
res3                          = .false.
res4                          = .false.
res5                          = .false.
res_gcl                       = .false.
primitive_tavg                = .false.
rho_tavg                      = .false.
u_tavg                        = .false.
v_tavg                        = .false.
w_tavg                        = .false.
p_tavg                        = .false.
tt_tavg                       = .false.
tv_tavg                       = .false.
primitive_trms                = .false.
rho_trms                      = .false.
u_trms                        = .false.
v_trms                        = .false.
w_trms                        = .false.
p_trms                        = .false.
tt_trms                       = .false.
tv_trms                       = .false.
lambda1                       = .false.
lambda2                       = .false.
lambda3                       = .false.
lambda4                       = .false.
lambda5                       = .false.
lambda6                       = .false.
lambda7                       = .false.
htot                          = .false.
```

```
ttot                         = .false.
ptot                         = .false.
etot                         = .false.
processor_id                 = .false.
turb_ke                      = .false.
turb_diss                    = .false.
mu_t                         = .false.
turb1                        = .false.
turb2                        = .false.
turb3                        = .false.
turb4                        = .false.
turb5                        = .false.
turb6                        = .false.
turb7                        = .false.
turres1                      = .false.
turres2                      = .false.
turres3                      = .false.
turres4                      = .false.
turres5                      = .false.
turres6                      = .false.
turres7                      = .false.
slen                         = .false.
iflagslen                    = .false.
hrles_blend                  = .false.
reconstruction_limiter_phi1 = .false.
reconstruction_limiter_phi2 = .false.
reconstruction_limiter_phi3 = .false.
reconstruction_limiter_phi4 = .false.
reconstruction_limiter_phi5 = .false.
shock_sensor                 = .false.
shock_switch                 = .false.
tt                           = .false.
tv                           = .false.
sonic                        = .false.
mixture_mol_weight           = .false.
drhodx                       = .false.
drhody                       = .false.
drhodz                       = .false.
gamma                        = .false.
ev                           = .false.
rho_i(1:n_species)           = .false.
mass_fr_i(1:n_species)       = .false.
mole_fr_i(1:n_species)       = .false.
rho_i_tavg(1:n_species)      = .false.
rho_i_trms(1:n_species)      = .false.
```

```
  mu                            = .false.
  id_l2g                        = .false.
  divided_residuals             = .false.
  volume_output_precision       = 'default'
  number_of_subvolumes          = 0
  type_of_subvolume(:)          = 'none'
  point1(1:3,:)                 = 0.0
  point2(1:3,:)                 = 0.0
  radius(:)                     = 0.0
  radius1(:)                    = 0.0
  radius2(:)                    = 0.0
  center(1:3,:)                 = 0.0
/
```

<u>`export_to = 'tecplot'`</u>

file format of volume export

`'tecplot'` is Tecplot™ format (one file written for each processor)

`'cgns'` is CGNS format, requires FUN3D to be configured with a CGNS library. This format already includes `x`, `y`, and `z`. Set these variables to `.false.` to avoid duplication.

`'fvuns'` is FieldView C-binary (Fortran stream) format. This format already includes `x`, `y`, and `z`. Set these variables to `.false.` to avoid duplication.

`'fv'` is depreciated. Slated for removal, use 'fvuns'

`'vtk'` is legacy VTK format This format already includes `x`, `y`, and `z`. Set these variables to `.false.` to avoid duplication.

`'csv'` is comma separated value format

`'sol'` is INRIA Metrix sol (ASCII) format

`'solb'` is INRIA Metrix solb (binary) format

`'bin'` is binary format without header or records

`'tec'` is a single image ASCII Tecplot™ format

`'raw_ascii'` is a single image ASCII space separated format

`'native'` is the most efficient way to export the entire flow field to disk at every time step when the grid is over a billion elements, but requires specialized visualization tools to read.

<u>`output_initial_state = .false.`</u>

When `.true.`, this causes requested volume data to be written for the initial state of the solution in the current run.

`x = .true.`

X-coordinate

`y = .true.`

Y-coordinate

`z = .true.`

Z-coordinate

`primitive_variables = .true.`

Output primitive variables: `rho`, `u`, `v`, `w`, and `p` (`p` `u`, `v`, `w` for `eqn_type = 'incompressible'`)

`rho = .false.`

Density (not available for `eqn_type = 'incompressible'`)

`u = .false.`

X-component of velocity

`v = .false.`

Y-component of velocity

`w = .false.`

Z-component of velocity

`p = .false.`

Pressure

`entropy = .false.`

Entropy

`mach = .false.`

Mach number (not available for `eqn_type = 'incompressible'`)

`temperature = .false.`

Temperature (not available for `eqn_type = 'incompressible'`)

`iblank = .false.`

I-blanking variable (default becomes `.true.` for overset mesh cases)

`imesh = .false.`

For overset mesh systems, index of associated component grid where `0` indicates background grid (default becomes `.true.` for overset mesh cases) `vort_mag = .false.`

Vorticity magnitude

`vort_x = .false.`

X-component of vorticity

`vort_y = .false.`

Y-component of vorticity

`vort_z = .false.`

Z-component of vorticity

`q_criterion = .false.`

Q Criterion, the second invariant of ∇V

`div_vel = .false.`

Velocity divergence

`turbulent_fluctuations = .false.`

Activate all the following `XYprime` turbulent shear stresses normalized by u_{ref}^2; the definition of these variables depends on the turbulence model, see http://turbmodels.larc.nasa.gov/noteonrunning.html for details

`uuprime = .false.`

Turbulence fluctuation, $u'u'$

`vvprime = .false.`

Turbulence fluctuation, $v'v'$

`wwprime = .false.`

Turbulence fluctuation, $w'w'$

`uvprime = .false.`

Turbulence fluctuation, $u'v'$

`uwprime = .false.`

Turbulence fluctuation, $u'w'$

`vwprime = .false.`

Turbulence fluctuation, $v'w'$

`cp = .false.`

Pressure coefficient

`dp_pinf = .false.`

Normalized delta pressure $(p - p_\infty)/p_\infty$

`volume = .false.`

Dual-cell volume size

`residuals = .false.`

Activate all `resN` variables

`res1 = .false.`

Residual of equation 1, density

`res2 = .false.`

Residual of equation 2, x-momentum

`res3 = .false.`

Residual of equation 3, y-momentum

`res4 = .false.`

Residual of equation 4, z-momentum

`res5 = .false.`

Residual of equation 5, energy

`res_gcl = .false.`

For moving meshes, residual of grid conservation law

`primitive_tavg = .false.`

Output time-averaged primitives (requires `&time_avg_params` namelist): `rho_tavg`, `u_tavg`, `v_tavg`, `w_tavg`, and `p_tavg`

`rho_tavg = .false.`

Time-averaged density

`u_tavg = .false.`

Time-averaged x-component of velocity

`v_tavg = .false.`

Time-averaged y-component of velocity

`w_tavg = .false.`

Time-averaged z-component of velocity

`p_tavg = .false.`

Time-averaged pressure

<u>`tt_tavg = .false.`</u>

Time-averaged translational-rotational temperature (only available for `eqn_type = 'generic gas'`)

<u>`tv_tavg = .false.`</u>

Time-averaged vibrational-electronic temperature (only available for `eqn_type = 'generic gas'`)

<u>`primitive_trms = .false.`</u>

Output root mean squared primitives (requires `&time_avg_params` namelist): `rho_trms`, `u_trms`, `v_trms`, `w_trms`, and `p_trms`

<u>`rho_trms = .false.`</u>

RMS-average of density

<u>`u_trms = .false.`</u>

RMS-average of x-component of velocity

<u>`v_trms = .false.`</u>

RMS-average of y-component of velocity

<u>`w_trms = .false.`</u>

RMS-average of z-component of velocity

<u>`p_trms = .false.`</u>

RMS-average of pressure

<u>`tt_trms = .false.`</u>

RMS-average of translational-rotational temperature (only available for `eqn_type = 'generic gas'`)

<u>`tv_trms = .false.`</u>

RMS-average of vibrational-electronic temperature (only available for `eqn_type = 'generic gas'`)

<u>`lambda1 = .false.`</u>

Adjoint Lagrange multiplier for equation 1 (when running the adjoint, the primitive variables are turned off, and this is turned on)

<u>`lambda2 = .false.`</u>

Adjoint Lagrange multiplier for equation 2 (when running the adjoint, the primitive variables are turned off, and this is turned on) `lambda3 = .false.`

Adjoint Lagrange multiplier for equation 3 (when running the adjoint, the primitive variables are turned off, and this is turned on)

`lambda4 = .false.`

Adjoint Lagrange multiplier for equation 4 (when running the adjoint, the primitive variables are turned off, and this is turned on)

`lambda5 = .false.`

Adjoint Lagrange multiplier for equation 5 (when running the adjoint, the primitive variables are turned off, and this is turned on)

`lambda6 = .false.`

Adjoint Lagrange multiplier for equation 6 (when running the adjoint, the primitive variables are turned off, and this is turned on)

`lambda7 = .false.`

Adjoint Lagrange multiplier for equation 7 (when running the adjoint, the primitive variables are turned off, and this is turned on)

`htot = .false.`

Total enthalpy per unit volume (not available for `eqn_type = 'incompressible'`)

`ttot = .false.`

Total temperature (not available for `eqn_type = 'incompressible'`)

`ptot = .false.`

Total pressure

`etot = .false.`

Total energy per unit volume (not available for `eqn_type = 'incompressible'`)

`processor_id = .false.`

Processor on which a node resides

`turb_ke = .false.`

Turbulence kinetic energy

`turb_diss = .false.`

Turbulence dissipation rate

`mu_t = .false.`

Turbulent eddy viscosity

`turb1 = .false.`

Turbulence variable 1 (model dependent)

`turb2 = .false.`

Turbulence variable 2 (model dependent)

`turb3 = .false.`

Turbulence variable 3 (model dependent)

`turb4 = .false.`

Turbulence variable 4 (model dependent)

`turb5 = .false.`

Turbulence variable 5 (model dependent)

`turb6 = .false.`

Turbulence variable 6 (model dependent)

`turb7 = .false.`

Turbulence variable 7 (model dependent)

`turres1 = .false.`

Residual of 1st turbulence equation

`turres2 = .false.`

Residual of 2nd turbulence equation

`turres3 = .false.`

Residual of 3rd turbulence equation

`turres4 = .false.`

Residual of 4th turbulence equation

`turres5 = .false.`

Residual of 5th turbulence equation

`turres6 = .false.`

Residual of 6th turbulence equation

`turres7 = .false.`

Residual of 7th turbulence equation

`slen = .false.`

Length to the nearest solid wall boundary `iflagslen = .false.`

Turbulence model distance function closest boundary entity. (a negative sign indicates the node has been prescribed as laminar)

`hrles_blend = .false.`

HRLES blending function

`reconstruction_limiter_phi1 = .false.`

ϕ for the node-based reconstruction limiters (equation 1) Note: the `'minmod'`, `'vanleer'`, `'vanleer_gg'`, and `'vanalbada'` limiters are *not* node-based limiters.

`reconstruction_limiter_phi2 = .false.`

ϕ for the node-based reconstruction limiters (equation 2)

`reconstruction_limiter_phi3 = .false.`

ϕ for the node-based reconstruction limiters (equation 3)

`reconstruction_limiter_phi4 = .false.`

ϕ for the node-based reconstruction limiters (equation 4)

`reconstruction_limiter_phi5 = .false.`

ϕ for the node-based reconstruction limiters (equation 5)

`shock_sensor = .false.`

Shock sensor for `stvd` and HLLE++

`shock_switch = .false.`

Shock switch for `stvd` and HLLE++

`tt = .false.`

Translational temperature (only available for `eqn_type = 'generic'`)

`tv = .false.`

Vibrational temperature (only available for `eqn_type = 'generic'`)

`sonic = .false.`

Frozen speed of sound (only available for `eqn_type = 'generic'`)

`mixture_mol_weight = .false.`

Mixture molecular weight (only available for `eqn_type = 'generic'`)

`drhodx = .false.`

Gradient of density with respect to the x-direction `drhody = .false.`

Gradient of density with respect to the y-direction

`drhodz = .false.`

Gradient of density with respect to the z-direction

`gamma = .false.`

Ratio of specific heats

`ev = .false.`

Vibrational energy (only available for `eqn_type = 'generic'`)

`rho_i(1:n_species) = .false.`

Species density (only available for `eqn_type = 'generic'`)

`mass_fr_i(1:n_species) = .false.`

Species mass fractions (only available for `eqn_type = 'generic'`)

`mole_fr_i(1:n_species) = .false.`

Species mole fractions (only available for `eqn_type = 'generic'`)

`rho_i_tavg(1:n_species) = .false.`

Time averaged species density (only available for `eqn_type = 'generic'`)

`rho_i_trms(1:n_species) = .false.`

RMS-average of species density (only available for `eqn_type = 'generic'`)

`mu = .false.`

Total viscosity

`id_l2g = .false.`

Local-to-global node map

`divided_residuals = .false.`

adds a `_vol` suffix to the residual output variable names and divide by volume

`volume_output_precision = 'default'`

Sets the storage size (FP32/FP64) of volume output variables, possible values are 'default', 'single', and 'double'

`number_of_subvolumes = 0`

Indicates the number of subvolumes desired for Tecplot output `type_of_subvolume(:) = 'none'`

This is the type of each subvolume desired; types may include `'box'`, `'sphere'`, `'cone'`, or `'cylinder'`.

`point1(1:3,:) = 0.0`

This is one end of the `'cone'` or `'cylinder'` subvolume types or one corner of a `'box'` subvolume type.

`point2(1:3,:) = 0.0`

This is the other end of the `'cone'` or `'cylinder'` subvolume types or the opposite corner of a `'box'` subvolume type.

`radius(:) = 0.0`

This is the radius of the `'sphere'` and `'cylinder'` subvolume types.

`radius1(:) = 0.0`

This is the radius at `point1` of the `'cone'` subvolume type.

`radius2(:) = 0.0`

This is the radius at `point2` of the `'cone'` subvolume type.

`center(1:3,:) = 0.0`

This is the center of the `'sphere'` subvolume type.

B.4.31 &boundary_output_variables

This namelist controls the boundary variable output. Output frequency is controlled by `boundary_animation_freq` in the `&global` namelist. By default, the output of solution data for all solid surfaces in 3D and on one y-constant symmetry plane in 2D is included unless `boundary_list` is specified.

Each time boundary data output is triggered, all output boundaries are written to one file with the following naming convention:

```
[project_rootname]_tec_boundary_timestep[T](.dat|.plt|.szplt)  if N > 0
[project_rootname]_tec_boundary(.dat|.plt|.szplt)              if N < 0
```

where `T` is the iteration number and `N` is the value of `boundary_animation_freq`. The file extension is `.dat` for ASCII Tecplot™ format, `.plt` for binary Tecplot™ format (2013 and earlier), and `.szplt` for new binary Tecplot™ format. Within the files, each boundary is written as a separate zone. The zones are identified with the title "time 0.0000000E+00 boundary 5" where the time value is the integer iteration number for steady-state cases, and the current (nondimensional) time for time-dependent cases.

By default, output is in the inertial reference frame. For moving body problems, the `&observer_motion` namelist can be used to change the visualization reference system to a body reference system or a reference system with arbitrary motion.

A request to output an unavailable variable will overruled, i.e., `turb1` will be forced to `.false` regardless of user input when there is no turbulence model in the simulation.

```
&boundary_output_variables
  number_of_boundaries    = 0
  boundary_list           = ''
  output_initial_state    = .false.
  x                       = .true.
  y                       = .true.
  z                       = .true.
  primitive_variables     = .true.
  rho                     = .false.
  u                       = .false.
  v                       = .false.
  w                       = .false.
  p                       = .false.
  entropy                 = .false.
  mach                    = .false.
  temperature             = .false.
  iblank                  = .false.
  imesh                   = .false.
  vort_mag                = .false.
```

```
vort_x                   = .false.
vort_y                   = .false.
vort_z                   = .false.
q_criterion              = .false.
div_vel                  = .false.
turbulent_fluctuations   = .false.
uuprime                  = .false.
vvprime                  = .false.
wwprime                  = .false.
uvprime                  = .false.
uwprime                  = .false.
vwprime                  = .false.
cp                       = .false.
dp_pinf                  = .false.
volume                   = .false.
residuals                = .false.
res1                     = .false.
res2                     = .false.
res3                     = .false.
res4                     = .false.
res5                     = .false.
res_gcl                  = .false.
primitive_tavg           = .false.
rho_tavg                 = .false.
u_tavg                   = .false.
v_tavg                   = .false.
w_tavg                   = .false.
p_tavg                   = .false.
rho_i_tavg(1:n_species) = .false.
tt_tavg                  = .false.
tv_tavg                  = .false.
primitive_trms           = .false.
rho_trms                 = .false.
u_trms                   = .false.
v_trms                   = .false.
w_trms                   = .false.
p_trms                   = .false.
rho_i_trms(1:n_species) = .false.
tt_trms                  = .false.
tv_trms                  = .false.
lambda1                  = .false.
lambda2                  = .false.
lambda3                  = .false.
lambda4                  = .false.
lambda5                  = .false.
```

```
lambda6                  = .false.
lambda7                  = .false.
htot                     = .false.
ttot                     = .false.
ptot                     = .false.
etot                     = .false.
processor_id             = .false.
turb_ke                  = .false.
turb_diss                = .false.
mu_t                     = .false.
turb1                    = .false.
turb2                    = .false.
turb3                    = .false.
turb4                    = .false.
turb5                    = .false.
turb6                    = .false.
turb7                    = .false.
turres1                  = .false.
turres2                  = .false.
turres3                  = .false.
turres4                  = .false.
turres5                  = .false.
turres6                  = .false.
turres7                  = .false.
de_turb1                 = .false.
slen                     = .false.
tt                       = .false.
tv                       = .false.
sonic                    = .false.
mixture_mol_weight       = .false.
ev                       = .false.
rho_i(1:n_species)       = .false.
mass_fr_i(1:n_species)   = .false.
mole_fr_i(1:n_species)   = .false.
mu                       = .false.
id_l2g                   = .false.
yplus                    = .false.
recovery_temperature     = .false.
turb_mach                = .false.
turbindex                = .false.
average_velocity         = .false.
uavg                     = .false.
vavg                     = .false.
wavg                     = .false.
surface_normals          = .false.
```

```
  cf_x                      = .false.
  cf_y                      = .false.
  cf_z                      = .false.
  skinfr                    = .false.
  cq                        = .false.
  shear_x                   = .false.
  shear_y                   = .false.
  shear_z                   = .false.
  heating                   = .false.
  mdot                      = .false.
  utau_wf                   = .false.
  phi_wf                    = .false.
  mu_t_wf                   = .false.
  k_wallfunction_bc         = .false.
  omega_wallfunction_bc     = .false.
  re_cell                   = .false.
/
```

`number_of_boundaries = 0`

Number of boundary patches given in `boundary_list` (if `-1` is given, this number is computed from `boundary_list`)

`boundary_list = ''`

List of boundary patch numbers. Commas and dashes can be used to specify ranges, i.e., `'1,2,5-7'`. If nothing is specified, then all but flow-through boundaries are output for 3D or a single symmetry plane in 2D.

`output_initial_state = .false.`

When `.true.`, this causes requested boundary data to be written for the initial state of the solution in the current run.

`x = .true.`

X-coordinate

`y = .true.`

Y-coordinate

`z = .true.`

Z-coordinate

`primitive_variables = .true.`

Output primitive variables: `rho`, `u`, `v`, `w`, and `p` (`p` `u`, `v`, `w` for `eqn_type = 'incompressible'`)

`rho = .false.`

Density (not available for `eqn_type = 'incompressible'`) `u = .false.`

X-component of velocity

`v = .false.`

Y-component of velocity

`w = .false.`

Z-component of velocity

`p = .false.`

Pressure

`entropy = .false.`

Entropy

`mach = .false.`

Mach number (not available for `eqn_type = 'incompressible'`)

`temperature = .false.`

Temperature (not available for `eqn_type = 'generic gas', 'incompressible'`)

`iblank = .false.`

I-blanking variable (default becomes `.true.` for overset mesh cases)

`imesh = .false.`

For overset mesh systems, index of associated component grid where `0` indicates background grid (default becomes `.true.` for overset mesh cases)

`vort_mag = .false.`

Vorticity magnitude

`vort_x = .false.`

X-component of vorticity

`vort_y = .false.`

Y-component of vorticity

`vort_z = .false.`

Z-component of vorticity

`q_criterion = .false.`

Q Criterion, the second invariant of ∇V

`div_vel = .false.`

Velocity divergence

`turbulent_fluctuations = .false.`

Activate all the following `XYprime` turbulent shear stresses normalized by u_{ref}^2; the definition of these variables depends on the turbulence model, see `http://turbmodels.larc.nasa.gov/noteonrunning.html` for details

`uuprime = .false.`

Turbulence fluctuation, $u'u'$

`vvprime = .false.`

Turbulence fluctuation, $v'v'$

`wwprime = .false.`

Turbulence fluctuation, $w'w'$

`uvprime = .false.`

Turbulence fluctuation, $u'v'$

`uwprime = .false.`

Turbulence fluctuation, $u'w'$

`vwprime = .false.`

Turbulence fluctuation, $v'w'$

`cp = .false.`

Pressure coefficient

`dp_pinf = .false.`

Normalized delta pressure $(p - p_\infty)/p_\infty$

`volume = .false.`

Dual-cell volume size

`residuals = .false.`

Activate all `resN` variables.

`res1 = .false.`

Residual of equation 1, density (pressure residual for `eqn_type = 'incompressible'`)

`res2 = .false.`

Residual of equation 2, x-momentum

`res3 = .false.`

Residual of equation 3, y-momentum

`res4 = .false.`

Residual of equation 4, z-momentum

`res5 = .false.`

Residual of equation 5, energy (not available for `eqn_type = 'incompressible'`)

`res_gcl = .false.`

For moving meshes, residual of grid conservation law

`primitive_tavg = .false.`

Output time-averaged primitives (requires `&time_avg_params` namelist): `rho_tavg`, `u_tavg`, `v_tavg`, `w_tavg`, and `p_tavg`

`rho_tavg = .false.`

Time-averaged density

`u_tavg = .false.`

Time-averaged x-component of velocity

`v_tavg = .false.`

Time-averaged y-component of velocity

`w_tavg = .false.`

Time-averaged z-component of velocity

`p_tavg = .false.`

Time-averaged pressure

`rho_i_tavg(1:n_species) = .false.`

Time-averaged species density (only available for `eqn_type = 'generic'`)

`tt_tavg = .false.`

Time-averaged translational-rotational temperature (only available for `eqn_type = 'generic'`)

`tv_tavg = .false.`

Time-averaged vibrational-electronic temperature (only available for `eqn_type = 'generic'`)

`primitive_trms = .false.`

Output root mean squared primitives (requires `&time_avg_params` namelist): `rho_trms`, `u_trms`, `v_trms`, `w_trms`, and `p_trms` `rho_trms = .false.`

RMS-average of density

`u_trms = .false.`

RMS-average of x-component of velocity

`v_trms = .false.`

RMS-average of y-component of velocity

`w_trms = .false.`

RMS-average of z-component of velocity

`p_trms = .false.`

RMS-average of pressure

`rho_i_trms(1:n_species) = .false.`

RMS-average of species density (only available for `eqn_type = 'generic'`)

`tt_trms = .false.`

RMS-average of translational-rotational temperature (only available for `eqn_type = 'generic'`)

`tv_trms = .false.`

RMS-average of vibrational-electronic temperature (only available for `eqn_type = 'generic'`)

`lambda1 = .false.`

Adjoint Lagrange multiplier for equation 1 (when running the adjoint, the primitive variables are turned off, and this is turned on)

`lambda2 = .false.`

Adjoint Lagrange multiplier for equation 2 (when running the adjoint, the primitive variables are turned off, and this is turned on)

`lambda3 = .false.`

Adjoint Lagrange multiplier for equation 3 (when running the adjoint, the primitive variables are turned off, and this is turned on)

`lambda4 = .false.`

Adjoint Lagrange multiplier for equation 4 (when running the adjoint, the primitive variables are turned off, and this is turned on)

`lambda5 = .false.`

Adjoint Lagrange multiplier for equation 5 (when running the adjoint, the primitive variables are turned off, and this is turned on) `lambda6 = .false.`

Adjoint Lagrange multiplier for equation 6 (when running the adjoint, the primitive variables are turned off, and this is turned on)

`lambda7 = .false.`

Adjoint Lagrange multiplier for equation 7 (when running the adjoint, the primitive variables are turned off, and this is turned on)

`htot = .false.`

Total enthalpy per unit volume (not available for `eqn_type = 'incompressible'`)

`ttot = .false.`

Total temperature (not available for `eqn_type = 'incompressible'`)

`ptot = .false.`

Total pressure

`etot = .false.`

Total energy per unit volume (not available for `eqn_type = 'incompressible'`)

`processor_id = .false.`

Processor on which a node resides

`turb_ke = .false.`

Turbulence kinetic energy

`turb_diss = .false.`

Turbulence dissipation rate

`mu_t = .false.`

Turbulent eddy viscosity

`turb1 = .false.`

Turbulence variable 1 (model dependent)

`turb2 = .false.`

Turbulence variable 2 (model dependent)

`turb3 = .false.`

Turbulence variable 3 (model dependent)

`turb4 = .false.`

Turbulence variable 4 (model dependent)

`turb5 = .false.`

Turbulence variable 5 (model dependent)

`turb6 = .false.`

Turbulence variable 6 (model dependent)

`turb7 = .false.`

Turbulence variable 7 (model dependent)

`turres1 = .false.`

Residual of 1st turbulence equation

`turres2 = .false.`

Residual of 2nd turbulence equation

`turres3 = .false.`

Residual of 3rd turbulence equation

`turres4 = .false.`

Residual of 4th turbulence equation

`turres5 = .false.`

Residual of 5th turbulence equation

`turres6 = .false.`

Residual of 6th turbulence equation

`turres7 = .false.`

Residual of 7th turbulence equation

`de_turb1 = .false.`

Discretization error of 1st turbulence equation

`slen = .false.`

Length to the nearest solid wall boundary

`tt = .false.`

Translational temperature, (only available for `eqn_type = 'generic'`)

`tv = .false.`

Vibrational temperature, (only available for `eqn_type = 'generic'`)

`sonic = .false.`

Frozen speed of sound, (only available for `eqn_type = 'generic'`) `mixture_mol_weight = .false.`

Mixture molecular weight, (only available for `eqn_type = 'generic'`)

`ev = .false.`

Vibrational energy, (only available for `eqn_type = 'generic'`)

`rho_i(1:n_species) = .false.`

Species concentration, (only available for `eqn_type = 'generic'`)

`mass_fr_i(1:n_species) = .false.`

Species mass fractions, (only available for `eqn_type = 'generic'`)

`mole_fr_i(1:n_species) = .false.`

Species mole fractions, (only available for `eqn_type = 'generic'`)

`mu = .false.`

Total viscosity (only available for `eqn_type = 'generic'`)

`id_l2g = .false.`

Local-to-global node map

`yplus = .false.`

Dimensionless wall distance, y^+

`recovery_temperature = .false.`

Recovery temperature (only available for `eqn_type = 'generic'`, a Prandtl number of 0.72 is assumed)

`turb_mach = .false.`

Turbulent Mach number (only available for `eqn_type = 'compressible'`)

`turbindex = .false.`

Turbulent index. When zero, the `turbindex` indicates laminar flow; when 1 it indicates turbulent flow. When in-between, it is indeterminate (may be laminar, turbulent, or transitional). The `turbindex` should be used cautiously, solely as a rough guide concerning the state of the boundary layer. Other measures can also give an indication, such as the maximum nondimensional eddy viscosity level within the boundary layer "above" the wall (when greater than approximately 1, the boundary layer is typically considered turbulent). However, the peak μ_t is a difficult measure to find in 3-D, as it must be searched along lines normal to walls. The turbulence index at walls uses equation (10) from Spalart and Allmaras, [51] which involves the wall-normal gradient of the Spalart-Allmaras (SA) turbulence variable $\tilde{\nu}$ at the wall. For other turbulence models, eddy viscosity is translated into SA variable $\tilde{\nu}$ form, in order to make use of the SA formula. Therefore, this formula is formally appropriate for SA, but is only an approximate (crude) indicator for other models.

`average_velocity = .false.`

Turns on `uavg`, `vavg`, `wavg`

`uavg = .false.`

X-component of average velocity near a wall (used to plot surface streamlines)

`vavg = .false.`

Y-component of average velocity near a wall (used to plot surface streamlines)

`wavg = .false.`

Z-component of average velocity near a wall (used to plot surface streamlines)

`surface_normals = .false.`

Turns on surface normals: `xnormal`, `ynormal`, `znormal`; direction of the normal points into the computational domain. Note that individual normal components are not selectable. These are area-weighted normals, not unit normals. The magnitude of any particular normal is the area of the dual-grid face centered about the surface node at the base of the normal.

`cf_x = .false.`

X-component of skin friction

`cf_y = .false.`

Y-component of skin friction

`cf_z = .false.`

Z-component of skin friction

`skinfr = .false.`

Skin friction magnitude with sign determined by the inner product of the skin friction vector and the freestream velocity vector

`cq = .false.`

Temperature gradient normal to the wall, with "dimensions" of nondimensional temperature per unit grid length for `eqn_type = 'compressible'`. Heating rate nondimensionalized by $\rho_\infty V_\infty^3$ for `eqn_type = 'generic'`.

`shear_x = .false.`

X-component of shear on the boundary, in MKS units

`shear_y = .false.`

Y-component of shear on the boundary, in MKS units

`shear_z = .false.`

Z-component of shear on the boundary, in MKS units

`heating = .false.`

Heating on the boundary in Watts per centimeter squared (for `eqn_type = 'compressible'`, make sure the grid is in meters)

`mdot = .false.`

Dimensionless blowing rate nondimensionalized by $\rho_\infty V_\infty$

`utau_wf = .false.`

Friction velocity calculated from a wall function model.

`phi_wf = .false.`

Pressure gradient term from a wall function model.

`mu_t_wf = .false.`

Wall function turbulent eddy viscosity at the wall.

`k_wallfunction_bc = .false.`

Turbulent kinetic energy wall function boundary condition.

`omega_wallfunction_bc = .false.`

Omega wall function boundary condition.

`re_cell = .false.`

Cell Reynolds number based on speed of sound and distance to nearest node off boundary

B.4.32 &sampling_output_variables

This namelist controls output of variables from user defined regions of the computational domain. To use sampling, the `&sampling_parameters` namelist must be used to define the sampling geometries and the `sampling_frequency(:)` set for each geometry.

The resulting sampling data files will have the following naming convention:

```
if N > 0:
[project_rootname]_tec_sampling_geom[G]_timestep[T](.dat|.plt|.szplt)
if N < 0:
[project_rootname]_tec_sampling_geom[G](.dat|.plt|.szplt)
```

where `G` = 1,2,...,`number_of_geometries`, `T` is the iteration number, and `N` is the value of `sampling_frequency(G)`. The file extension is `.dat` for ASCII Tecplot™ format, `.plt` for binary Tecplot™ format (2013 and earlier), and `.szplt` for new binary Tecplot™ format. A global image of the sampling surface is output with the zone title "time 0.0000000E+00 geom 3" where the time value is the integer iteration number for steady-state cases, and the current (nondimensional) time for time-dependent cases.

A request to output an unavailable variable will overruled, i.e., `turb1` will be forced to `.false` regardless of user input when there is no turbulence model in the simulation.

```
&sampling_output_variables
  x                            = .true.
  y                            = .true.
  z                            = .true.
  primitive_variables          = .true.
  rho                          = .false.
  rho_i(:)                     = .false.
  mass_fr_i(:)                 = .false.
  mole_fr_i(:)                 = .false.
  u                            = .false.
  v                            = .false.
  w                            = .false.
  p                            = .false.
  entropy                      = .false.
  mach                         = .false.
  temperature                  = .false.
  tt                           = .false.
  sonic                        = .false.
  mixture_mol_weight           = .false.
  drhodx                       = .false.
  drhody                       = .false.
  drhodz                       = .false.
```

```
dudx                          = .false.
dudy                          = .false.
dudz                          = .false.
dvdx                          = .false.
dvdy                          = .false.
dvdz                          = .false.
dwdx                          = .false.
dwdy                          = .false.
dwdz                          = .false.
dpdx                          = .false.
dpdy                          = .false.
dpdz                          = .false.
tv                            = .false.
mu                            = .false.
iblank                        = .false.
imesh                         = .false.
vort_mag                      = .false.
vort_x                        = .false.
vort_y                        = .false.
vort_z                        = .false.
q_criterion                   = .false.
div_vel                       = .false.
turbulent_fluctuations        = .false.
uuprime                       = .false.
vvprime                       = .false.
wwprime                       = .false.
uvprime                       = .false.
uwprime                       = .false.
vwprime                       = .false.
cp                            = .false.
dp_pinf                       = .false.
volume                        = .false.
residuals                     = .false.
res1                          = .false.
res2                          = .false.
res3                          = .false.
res4                          = .false.
res5                          = .false.
res_gcl                       = .false.
primitive_tavg                = .false.
rho_tavg                      = .false.
u_tavg                        = .false.
v_tavg                        = .false.
w_tavg                        = .false.
p_tavg                        = .false.
```

```
rho_i_tavg(:)                = .false.
tt_tavg                      = .false.
tv_tavg                      = .false.
gamma_tavg                   = .false.
mu_t_tavg                    = .false.
vort_mag_tavg                = .false.
vort_x_tavg                  = .false.
vort_y_tavg                  = .false.
vort_z_tavg                  = .false.
primitive_trms               = .false.
rho_trms                     = .false.
u_trms                       = .false.
v_trms                       = .false.
w_trms                       = .false.
p_trms                       = .false.
rho_i_trms(:)                = .false.
tt_trms                      = .false.
tv_trms                      = .false.
lambda1                      = .false.
lambda2                      = .false.
lambda3                      = .false.
lambda4                      = .false.
lambda5                      = .false.
lambda6                      = .false.
lambda7                      = .false.
htot                         = .false.
ttot                         = .false.
ptot                         = .false.
etot                         = .false.
processor_id                 = .false.
turb_ke                      = .false.
turb_diss                    = .false.
mu_t_wf                      = .false.
mu_t                         = .false.
turb1                        = .false.
turb2                        = .false.
turb3                        = .false.
turb4                        = .false.
turb5                        = .false.
turb6                        = .false.
turb7                        = .false.
turres1                      = .false.
turres2                      = .false.
turres3                      = .false.
turres4                      = .false.
```

```
turres5                      = .false.
turres6                      = .false.
turres7                      = .false.
gamma                        = .false.
slen                         = .false.
iflagslen                    = .false.
hrles_blend                  = .false.
cdesdelta_des_length         = .false.
dtilde_des_length            = .false.
rd_ddes_length               = .false.
fd_ddes_blend                = .false.
dtilde_ddes_length           = .false.
vort_x_rms                   = .false.
vort_y_rms                   = .false.
vort_z_rms                   = .false.
vort_mag_rms                 = .false.
yplus                        = .false.
cmu_star                     = .false.
mu_t_ratio                   = .false.
iib                          = .false.
iiib                         = .false.
uplus                        = .false.
kplus                        = .false.
wplus                        = .false.
yplusretau                   = .false.
t11plus                      = .false.
t12plus                      = .false.
t13plus                      = .false.
t22plus                      = .false.
t23plus                      = .false.
t33plus                      = .false.
t11_res                      = .false.
t12_res                      = .false.
t13_res                      = .false.
t22_res                      = .false.
t23_res                      = .false.
t33_res                      = .false.
tke_modeled                  = .false.
bird_breakdown               = .false.
gllk_breakdown               = .false.
vgradrho                     = .false.
f_r1                         = .false.
xi_k                         = .false.
reconstruction_limiter_phi1 = .false.
reconstruction_limiter_phi2 = .false.
```

```
  reconstruction_limiter_phi3 = .false.
  reconstruction_limiter_phi4 = .false.
  reconstruction_limiter_phi5 = .false.
  shock_sensor                = .false.
  shock_switch                = .false.
  s_modulus                   = .false.
  w_modulus                   = .false.
  vorticity_shell             = .false.
/
```

<u>`x = .true.`</u>

X-coordinate

<u>`y = .true.`</u>

Y-coordinate

<u>`z = .true.`</u>

Z-coordinate

<u>`primitive_variables = .true.`</u>

Output primitive variables: `rho`, `u`, `v`, `w`, and `p` (`p` `u`, `v`, `w` for `eqn_type = 'incompressible'`)

<u>`rho = .false.`</u>

Density (not available for `eqn_type = 'incompressible'`)

<u>`rho_i(:) = .false.`</u>

Species densities (not available for `eqn_type = 'compressible', 'incompressible'`)

<u>`mass_fr_i(:) = .false.`</u>

Species mass fractions (not available for `eqn_type = 'compressible', 'incompressible'`)

<u>`mole_fr_i(:) = .false.`</u>

Species mole fractions (not available for `eqn_type = 'compressible', 'incompressible'`)

<u>`u = .false.`</u>

X-component of velocity

<u>`v = .false.`</u>

Y-component of velocity

<u>`w = .false.`</u>

Z-component of velocity

p = .false.

Pressure

entropy = .false.

Entropy

mach = .false.

Mach number (not available for `eqn_type = 'incompressible'`)

temperature = .false.

Temperature (not available for `eqn_type = 'incompressible'`)

tt = .false.

Translational-rotational temperature in the generic gas path (only available for `eqn_type = 'generic gas'`)

sonic = .false.

Frozen speed of sound (only available for `eqn_type = 'generic'`)

mixture_mol_weight = .false.

Mixture molecular weight (only available for `eqn_type = 'generic'`)

drhodx = .false.

Gradient of density with respect to the x-direction

drhody = .false.

Gradient of density with respect to the y-direction

drhodz = .false.

Gradient of density with respect to the z-direction

dudx = .false.

Gradient of u-velocity with respect to the x-direction

dudy = .false.

Gradient of u-velocity with respect to the y-direction

dudz = .false.

Gradient of v-velocity with respect to the z-direction

dvdx = .false.

Gradient of v-velocity with respect to the x-direction

dvdy = .false.

Gradient of v-velocity with respect to the y-direction `dvdz = .false.`

Gradient of v-velocity with respect to the z-direction

`dwdx = .false.`

Gradient of w-velocity with respect to the x-direction

`dwdy = .false.`

Gradient of w-velocity with respect to the y-direction

`dwdz = .false.`

Gradient of w-velocity with respect to the z-direction

`dpdx = .false.`

Gradient of pressure with respect to the x-direction

`dpdy = .false.`

Gradient of pressure with respect to the y-direction

`dpdz = .false.`

Gradient of pressure with respect to the z-direction

`tv = .false.`

Vibrational-electronic temperature in the generic gas path (only available for `eqn_type = 'generic gas'`)

`mu = .false.`

Molecular viscosity

`iblank = .false.`

I-blanking variable (default becomes `.true.` for overset mesh cases)

`imesh = .false.`

For overset mesh systems, index of associated component grid where `0` indicates background grid (default becomes `.true.` for overset mesh cases)

`vort_mag = .false.`

Vorticity magnitude

`vort_x = .false.`

X-component of vorticity

`vort_y = .false.`

Y-component of vorticity

`vort_z = .false.`

Z-component of vorticity

`q_criterion = .false.`

Q Criterion, the second invariant of ∇V

`div_vel = .false.`

Velocity divergence

`turbulent_fluctuations = .false.`

Activate all the following `XYprime` turbulent shear stresses normalized by u_{ref}^2; the definition of these variables depends on the turbulence model, see http://turbmodels.larc.nasa.gov/noteonrunning.html for details

`uuprime = .false.`

Turbulence fluctuation, $u'u'$

`vvprime = .false.`

Turbulence fluctuation, $v'v'$

`wwprime = .false.`

Turbulence fluctuation, $w'w'$

`uvprime = .false.`

Turbulence fluctuation, $u'v'$

`uwprime = .false.`

Turbulence fluctuation, $u'w'$

`vwprime = .false.`

Turbulence fluctuation, $v'w'$

`cp = .false.`

Pressure coefficient

`dp_pinf = .false.`

Normalized delta pressure $(p - p_\infty)/p_\infty$

`volume = .false.`

Dual-cell volume size

`residuals = .false.`

Activate all `resN` variables

`res1 = .false.`

Residual of equation 1, density (pressure for `eqn_type = 'incompressible'`)

`res2 = .false.`

Residual of equation 2, x-momentum

`res3 = .false.`

Residual of equation 3, y-momentum

`res4 = .false.`

Residual of equation 4, z-momentum

`res5 = .false.`

Residual of equation 5, energy (not available for `eqn_type = 'incompressible'`)

`res_gcl = .false.`

For moving meshes, residual of grid conservation law

`primitive_tavg = .false.`

Output time-averaged primitives (requires `&time_avg_params` namelist):
`rho_tavg`, `u_tavg`, `v_tavg`, `w_tavg`, and `p_tavg`

`rho_tavg = .false.`

Time-averaged density

`u_tavg = .false.`

Time-averaged x-component of velocity

`v_tavg = .false.`

Time-averaged y-component of velocity

`w_tavg = .false.`

Time-averaged z-component of velocity

`p_tavg = .false.`

Time-averaged pressure

`rho_i_tavg(:) = .false.`

Time-averaged species densities (not available for `eqn_type = 'compressible', 'incompressible'`)

`tt_tavg = .false.`

Time-averaged translational-rotational temperature (not available for `eqn_type = 'compressible', 'incompressible'`) tv_tavg = .false.

Time-averaged vibrational-electronic temperature (not available for eqn_type = 'compressible', 'incompressible')

gamma_tavg = .false.

Time-averaged ratio of specific heats (not available for eqn_type = 'compressible', 'incompressible')

mu_t_tavg = .false.

Time-averaged turbulent eddy viscosity (not available for eqn_type = 'generic gas')

vort_mag_tavg = .false.

Time-averaged vorticity magnitude

vort_x_tavg = .false.

Time-averaged x-component vorticity

vort_y_tavg = .false.

Time-averaged y-component vorticity

vort_z_tavg = .false.

Time-averaged z-component vorticity

primitive_trms = .false.

Output root mean squared primitives (requires &time_avg_params namelist): rho_trms, u_trms, v_trms, w_trms, and p_trms

rho_trms = .false.

RMS-average of density

u_trms = .false.

RMS-average of x-component of velocity

v_trms = .false.

RMS-average of y-component of velocity

w_trms = .false.

RMS-average of z-component of velocity

p_trms = .false.

RMS-average of pressure

rho_i_trms(:) = .false.

RMS-average of species densities (not available for `eqn_type = 'compressible' , 'incompressible'`)

tt_trms = .false.

RMS-average of translational-rotational temperature (not available for `eqn_type = 'compressible', 'incompressible'`)

tv_trms = .false.

RMS-average of vibrational-electronic temperature (not available for `eqn_type = 'compressible', 'incompressible'`)

lambda1 = .false.

Adjoint Lagrange multiplier for equation 1 (when running the adjoint, the primitive variables are turned off, and this is turned on)

lambda2 = .false.

Adjoint Lagrange multiplier for equation 2 (when running the adjoint, the primitive variables are turned off, and this is turned on)

lambda3 = .false.

Adjoint Lagrange multiplier for equation 3 (when running the adjoint, the primitive variables are turned off, and this is turned on)

lambda4 = .false.

Adjoint Lagrange multiplier for equation 4 (when running the adjoint, the primitive variables are turned off, and this is turned on)

lambda5 = .false.

Adjoint Lagrange multiplier for equation 5 (when running the adjoint, the primitive variables are turned off, and this is turned on)

lambda6 = .false.

Adjoint Lagrange multiplier for equation 6 (when running the adjoint, the primitive variables are turned off, and this is turned on)

lambda7 = .false.

Adjoint Lagrange multiplier for equation 7 (when running the adjoint, the primitive variables are turned off, and this is turned on)

htot = .false.

Total enthalpy per unit volume (not available for `eqn_type = 'incompressible'`) `ttot = .false.`

Total temperature (not available for `eqn_type = 'incompressible'`)

`ptot = .false.`

Total pressure

`etot = .false.`

Total energy per unit volume (not available for `eqn_type = 'incompressible'`)

`processor_id = .false.`

Processor on which a node resides

`turb_ke = .false.`

Turbulence kinetic energy

`turb_diss = .false.`

Turbulence dissipation rate

`mu_t_wf = .false.`

Turbulent eddy viscosity at the wall from a wall function model.

`mu_t = .false.`

Turbulent eddy viscosity

`turb1 = .false.`

Turbulence variable 1 (model dependent)

`turb2 = .false.`

Turbulence variable 2 (model dependent)

`turb3 = .false.`

Turbulence variable 3 (model dependent)

`turb4 = .false.`

Turbulence variable 4 (model dependent)

`turb5 = .false.`

Turbulence variable 5 (model dependent)

`turb6 = .false.`

Turbulence variable 6 (model dependent)

`turb7 = .false.`

Turbulence variable 7 (model dependent)

`turres1 = .false.`

Residual of 1st turbulence equation

`turres2 = .false.`

Residual of 2nd turbulence equation

`turres3 = .false.`

Residual of 3rd turbulence equation

`turres4 = .false.`

Residual of 4th turbulence equation

`turres5 = .false.`

Residual of 5th turbulence equation

`turres6 = .false.`

Residual of 6th turbulence equation

`turres7 = .false.`

Residual of 7th turbulence equation

`gamma = .false.`

Ratio of specific heats

`slen = .false.`

Length to the nearest solid wall boundary

`iflagslen = .false.`

Turbulence model distance function closest boundary entity. (a negative sign indicates the node has been prescribed as laminar)

`hrles_blend = .false.`

HRLES blending function

`cdesdelta_des_length = .false.`

The maximum edge length incident to a vertex, $C_{DES}\Delta$ of Spalart et al. [75] A purely geometric LES length scale.

`dtilde_des_length = .false.`

DES length scale, $\tilde{d} = \min(d, C_{DES}\Delta)$ of Spalart et al. [75]

`rd_ddes_length = .false.`

An argument of the delayed DES blending function, $r_d = \frac{\tilde{\nu}}{\sqrt{U_{i,j}U_{i,j}}\kappa^2 d^2}$, which is an alternate form of equation (2.1) in Spalart et al. [92] suggested by the reference.

`fd_ddes_blend = .false.`

Delayed DES blending function, $f_d = 1 - \tanh\left([8r_d]^3\right)$, equation (2.2) of Spalart et al. [92]

`dtilde_ddes_length = .false.`

DDES length scale, $\tilde{d} = d - f_d \max(0, d - C_{DES}\Delta)$, equation (2.3) of Spalart et al. [92]

`vort_x_rms = .false.`

RMS-average of x-component of vorticity

`vort_y_rms = .false.`

RMS-average of y-component of vorticity

`vort_z_rms = .false.`

RMS-average of z-component of vorticity

`vort_mag_rms = .false.`

RMS-average of vorticity magnitude

`yplus = .false.`

Dimensionless wall distance, y^+

`cmu_star = .false.`

$k - \epsilon$ model turbulent length scale parameter

`mu_t_ratio = .false.`

Ratio of turbulent eddy viscosity to laminar (bulk) viscosity

`iib = .false.`

$-\text{trace}(B_{ij} * B_{ij})/2$

`iiib = .false.`

$\text{trace}(B_{ij} * B_{ij} * B_{ij})/3$

`uplus = .false.`

Dimensionless velocity, u^+

`kplus = .false.`

$\frac{k}{u_\tau^2}$

`wplus = .false.`

$\frac{\omega\nu}{u_\tau^2}$

yplusretau = .false.

$\frac{u^+}{re_\tau}$

t11plus = .false.

τ_{11}^+

t12plus = .false.

τ_{12}^+

t13plus = .false.

τ_{13}^+

t22plus = .false.

τ_{22}^+

t23plus = .false.

τ_{23}^+

t33plus = .false.

τ_{33}^+

t11_res = .false.

$u\bar{'u}'$ Reynolds stress from the subgrid scale model. `tavg_header_version = 2` and `itime_avg /= 0` are required. (not available for `eqn_type = 'generic', 'incompressible'`)

t12_res = .false.

$u\bar{'v}'$ Reynolds stress from the subgrid scale model. `tavg_header_version = 2` and `itime_avg /= 0` are required. (not available for `eqn_type = 'generic_gas', 'incompressible'`)

t13_res = .false.

$u\bar{'w}'$ Reynolds stress from the subgrid scale model. `tavg_header_version = 2` and `itime_avg /= 0` are required. (not available for `eqn_type = 'generic_gas', 'incompressible'`)

t22_res = .false.

$v\bar{'w}'$ Reynolds stress from the subgrid scale model. `tavg_header_version = 2` and `itime_avg /= 0` are required. (not available for `eqn_type = 'generic_gas', 'incompressible'`)

t23_res = .false.

$v\bar{'w}'$ Reynolds stress from the subgrid scale model. `tavg_header_version = 2` and `itime_avg /= 0` are required. (not available for `eqn_type = 'generic_gas', 'incompressible'`) t33_res = .false.

$\bar{w'w'}$ Reynolds stress from the subgrid scale model. tavg_header_version = 2 and itime_avg /= 0 are required. (not available for eqn_type = 'generic_gas', 'incompressible')

tke_modeled = .false.

Pseudoturbulent kinetic energy term for use in plotting the normal stresses generated by the QCR Reynolds stress models. (Available only for turbulence_model = 'sa', 'sa-neg, 'des', 'des-neg)

bird_breakdown = .false.

Bird breakdown parameter sampling is no longer supported. For continuum flow breakdown analysis, use gllk_breakdown.

gllk_breakdown = .false.

Gradient Length Local Knudsen Number in Ref. [107]

vgradrho = .false.

$[u, v, w] \cdot \nabla\rho$

f_r1 = .false.

Curvature correction model function

xi_k = .false.

Cross-diffusion term for Wilcox k-ω 1998

reconstruction_limiter_phi1 = .false.

ϕ for the node-based reconstruction limiters (equation 1) Note: the 'minmod', 'vanleer', 'vanleer_gg', and 'vanalbada' limiters are *not* node-based limiters.

reconstruction_limiter_phi2 = .false.

ϕ for the node-based reconstruction limiters (equation 2)

reconstruction_limiter_phi3 = .false.

ϕ for the node-based reconstruction limiters (equation 3)

reconstruction_limiter_phi4 = .false.

ϕ for the node-based reconstruction limiters (equation 4)

reconstruction_limiter_phi5 = .false.

ϕ for the node-based reconstruction limiters (equation 5)

shock_sensor = .false.

Shock sensor for stvd and HLLE++

`shock_switch = .false.`

Shock switch for `stvd` and HLLE++

`s_modulus = .false.`

$|S| = \sqrt{2S_{ij}S_{ji}}$

`w_modulus = .false.`

$|W| = \sqrt{2\,|W_{ij}W_{ji}|}$

`vorticity_shell = .false.`

$\mu_t max(0, |S| - 2|W|)$

B.4.33 &sampling_parameters

This namelist specifies the types and frequency of sampling data to be exported for visualization. The output variables themselves are specified in the `&sampling_output_variables` namelist. The last dimension of each array except for `points` references the geometry index, which is one to `number_of_geometries`.

```
&sampling_parameters
  number_of_geometries            = 0
  sampling_frequency(:)           = 0
  label(:)                        = ''
  type_of_geometry(:)             = 'none'
  export_to(:)                    = 'tec'
  crinkle                         = .false.
  nodal                           = .false.
  output_initial_state            = .false.
  plot(:)                         = 'tecplot'
  patch_list_count(:)             = 0
  patch_list(:)                   = ''
  type_of_data(:)                 = 'volume'
  move_with_body(:)               = ''
  boundary_list                   = ''
  default_boundary                = .true.
  plane_center(1:3,:)             = 0.0
  plane_normal(1:3,:)             = 0.0
  box_lower_corner(1:3,:)         = -1e+12
  box_upper_corner(1:3,:)         = 1e+12
  sphere_center(1:3,:)            = 0.0
  sphere_radius(:)                = 0.0
  circle_center(1:3,:)            = 0.0
  circle_normal(1:3,:)            = 0.0
  circle_radius(:)                = 0.0
  cylinder_face1(1:3,:)           = 0.0
  cylinder_face2(1:3,:)           = 0.0
  cylinder_radius(:)              = 0.0
  cone_face1(1:3,:)               = 0.0
  cone_face2(1:3,:)               = 0.0
  cone_radius1(:)                 = 0.0
  cone_radius2(:)                 = 0.0
  corner1(1:3,:)                  = 0.0
  corner2(1:3,:)                  = 0.0
  corner3(1:3,:)                  = 0.0
  corner4(1:3,:)                  = 0.0
  patch_symmetry(:)               = 1.0
  number_of_points(:)             = 0
  points(1:3,:,:)                   = 0.0
  p1_line(1:3,:)                    = 0.0
  p2_line(1:3,:)                    = 0.0
  schlieren_aspect                  = ''
  window_normal(1:3,:)              = 0.0
  window_height(:)                  = 0.0
  window_width(:)                   = 0.0
  window_depth(:)                   = 1.0e+6
  window_center(1:3,:)              = 0.0
  number_of_rows(:)                 = 0
  number_of_columns(:)              = 0
  model_center(1:3,:)               = 0.0
  plot_lines(:)                     = .false.
  make_shadow                       = .false.
  blanking_list_count(:)            = 0
  blanking_list(:)                  = ''
  isosurf_variable(:)               = 'p'
  isosurf_value(:)                  = 0.0
  isosurf_box(:)                    = .false.
  x_range_lower(:)                  = -1.0
  x_range_upper(:)                  = 1.0
  y_range_lower(:)                  = -1.0
  y_range_upper(:)                  = 1.0
  z_range_lower(:)                  = -1.0
  z_range_upper(:)                  = 1.0
  isosurf_dist_threshold(:)         = 0.0
  variable_list(:)                  = ''
  snap_output_xyz                   = .true.
  dist_tolerance                    = 1.0e-3
  fwh_formatted                     = .false.
  append_history(:)                 = .false.
  asynchronous_fwh                  = .false.
  boundary_point(1:3,:)             = 0.0
  boundary(:)                       = 0
  boundary_layer_edge_parameter(:) = 'enthalpy'
  boundary_layer_edge_value(:)     = 0.995
  boundary_layer_search_extent(:)  = 1.0e+6
  need_wall_data(:)                 = .false.
  reference_length                  = 0.0
/
```

`number_of_geometries = 0`

This is the total number of sampling geometries.

`sampling_frequency(:) = 0`

This specifies the iteration interval at which sampling is performed. The special value of `-1` means to only perform sampling at the end of a successful run.

<u>`label(:) = ''`</u>

This customizes the filename of sampling output. When it is blank, the file will be `[project_rootname]_tec_sampling_geomN.(dat,plt)` where `N` is the sampling geometry number, `.dat` is ASCII format, and `.plt` is binary format.

<u>`type_of_geometry(:) = 'none'`</u>

This is the type of sampling geometry,

`'streamsurface'` is a stream surface, requires `number_of_points` and `points`.

`'boundary_points'` for boundary point sampling, requires `number_of_points` and `points`, modified by `snap_output_xyz` and `dist_tolerance`.

`'volume_points'` for point sampling in the domain, requires `number_of_points` and `points`.

`'schlieren'` is a schlieren image via an integral of the refractive index field, requires `number_of_rows`, `number_of_columns`, `window_height`, `window_width`, `window_center`, and `schlieren_aspect`. It is controlled by `make_shadow` and `plot_lines`.

`'isosurface'` is an isosurface that requires `isosurf_variable` and `isosurf_value`. It is controlled by `*_range_lower` and `*_range_upper`.

`'box'` samples a the surface of a box. It requires `box_lower_corner` and `box_upper_corner`.

`'sphere'` samples a spherical surface. It requires `sphere_center` and `sphere_radius`.

`'cylinder'` samples a cylindrical surface. It requires `cylinder_face1`, `cylinder_face2`, and `cylinder_radius`.

`'cone'` samples a conic surface. It requires `cone_face1`, `cone_face2`, `cone_radius1`, and `cone_radius2`.

`'plane'` samples a plane. It requires `plane_center` and `plane_normal`.

`'quad'` samples a quadrilateral. It requires `corner1`, `corner2`, `corner3`, `corner4`, and `window_normal`.

`'circle'` samples a circle. It requires `circle_center`, `circle_normal`, and `circle_radius`.

`'line'` is line sampling, which requires `p1_line` and `p2_line`. The line sample will return data at each element center intersected by the line.

`'analyse_line'` traces a velocity profile from the boundary point in the normal direction off of the boundary surface. `analyse_line` additionally requires `boundary_layer_edge_parameter`, `boundary_layer_edge_value`, and `boundary_layer_search_extent`. A sample input for requesting a boundary layer thickness parameters on boundary 6 at the surface point (0.589, -0.500, 0.258) using the criterion of 0.99 velocity edge and limiting the search to 5 grid units normal to the surface is given below.

```
type_of_geometry(1) = 'analyse_line'
boundary(1) = 6
boundary_point(:,1) = 0.589, -0.500, 0.258
boundary_layer_edge_parameter(1) = 'velocity'
boundary_layer_edge_value(1) = 0.99
boundary_layer_search_extent(1) = 5.0
sampling_frequency(1) = -1
```

`'analyse_boundary'` creates a full surface map (scatter plot) of boundary layer parameters. It requires `boundary`, `boundary_layer_edge_parameter`, `boundary_layer_edge_value`, and `boundary_layer_search_extent`. A sample input for requesting a boundary layer thickness parameters on boundary 2 using the criterion of 0.001 vorticity edge and limiting the search to 5 grid units normal to the surface is given below.

```
type_of_geometry(4) = 'analyse_boundary'
boundary(4) = 2
label(4) = 'vorticity_0z001'
boundary_layer_edge_parameter(4) = 'vorticity'
boundary_layer_edge_value(4) = 1.0e-3
boundary_layer_search_extent(4) = 5.0
sampling_frequency(4) = -1
```

`'mesh'` is requesting analysis of mesh quality. Requires in addition `type_of_data(geom_number)='mesh_statistics'`.

`'partition'` creates a full volume output. Sample inputs for requesting a volume output are given below. Sample 1 returns an ASCII volume file with label 'volume1' containing 3 Cartesian coordinates, the 5 primitive variables, total pressure and total temperature. Sample 2 returns an INRIA binary volume file every other iteration containing Mach number.

```
&sampling_parameters
number_of_geometries = 2
type_of_geometry(1) = 'partition'
export_to(1) = 'tec'
label(1) = 'volume1'
```

```
    sampling_frequency(1) = -1
    variable_list(1) = 'x,y,z,rho,u,v,w,p,ptot,ttot'
    type_of_geometry(2) = 'partition'
    export_to(2) = 'solb'
    sampling_frequency(2) = 2
    variable_list(2) = 'mach'
    /
```

'`grid_var`' is multiple line sampling.

<u>`export_to(:) = 'tec'`</u>

This option is active when using `type_of_geometry(ivol)='partition'` for output of entire volume of requested data specified in either the `variable_list(ivol)` namelist parameter or the output specified in the `&sampling_output` namelist. The options are `export_to{ivol}='tec'` for a single image ASCII Tecplot™ output file or `export_to{ivol}= 'solb'` for an INRIA solb (binary) output file.

<u>`crinkle = .false.`</u>

This snaps the sampling surface to nearest grid faces instead of using linear interpolation.

<u>`nodal = .false.`</u>

This uses the nearest nodal values instead of interpolating.

<u>`output_initial_state = .false.`</u>

When `.true.`, this causes requested sampling data to be written for the initial state of the solution in the current run.

<u>`plot(:) = 'tecplot'`</u>

This option is active when using a `type_of_geometry` of `'streamsurface'`, `'boundary_points'`, `'volume_points'`, `'isosurface'`, `'isocrinkle'`, `'box'`, `'filledbox'`, `'sphere'`, `'cylinder'`, `'cone'`, `'plane'`, `'quad'`, `'circle'`, `'line'`, `'analyse_line'`, `'analyse_boundary'`, or `'grid_var'` The options are `plot{ivol}='tecplot'` for an ASCII Tecplot™ output file,

```
  &sampling_parameters
  number_of_geometries = 9
  ...
  type_of_geometry(9) = 'line'
  plot(9) = 'tecplot'
  label(9) = 'profile'
  p1_line(1:3,9) =  0.01 0.01 -10.0
  p2_line(1:3,9) =  0.01 0.01  10.0
```

```
      variable_list(9) = 'x,y,z,rho,u,v,w,p,ptot,ttot,htot,mach'
      sampling_frequency(9) = -1
      /
```

`plot{ivol}='fwh'` for a Ffowcs Williams-Hawkings output,

```
      &sampling_parameters
      number_of_geometries = 1
      type_of_geometry(1) = 'quad'
      sampling_frequency(1) = 1
      plot(1) = 'fwh'
      corner1(:,1) =-2.0,-5.0, 6.0,
      corner2(:,1) =-2.0,-5.0,-6.0,
      corner3(:,1) =-2.0 ,5.0,-6.0,
      corner4(:,1) =-2.0, 5.0, 6.0,
      variable_list(1) = 'x,y,z,nx,ny,nz,rho,u,v,w,p_gage'
      /
```

or `plot{ivol}='serial_history'` for an iterative or time history of a series of sampled points.

```
      &sampling_parameters
      number_of_geometries = 1
      type_of_geometry(1) = 'volume_points'
      number_of_points(1) = 40
      sampling_frequency(1) = 1
      variable_list(1) = 'ptot,ttot'
      plot(1) = 'serial_history'
      /
```

In summary, the options are:

`'tecplot'` Tecplot™ format.

`'fwh'` format for Ffowcs Williams-Hawkings analysis.

`'serial_history'` custom low-overhead point sampling format where all locations listed once at the top and then just the requested values per `sampling_frequency`.

`patch_list_count(:) = 0`

This is the number of patches in `patch_list`.

`patch_list(:) = ''`

A string list of patch face IDs to limit boundary survey to a subset of the boundary faces. Commas and dashes can be used to specify ranges, i.e., `'1,2,5-7'`.

`type_of_data(:) = 'volume'`

The source of data for extracting the requested sampling variables for each `type_of_geometry`.

`'volume'` extract data from the computational volume.

`'boundary'` extract data from a boundary.

`'integrated'` extract data from the computational volume and integrate over defined geometry.

`move_with_body(:) = ''`

Move the sampling geometry with the body if body is in motion. Use the fixed inertial reference frame when blank.

`boundary_list = ''`

List of patches to include when sampling boundaries; Commas and dashes can be used to specify ranges, i.e., `'1,2,5-7'`.

`default_boundary = .true.`

Use FUN3D default solid-wall-only boundary patches when sampling boundary points, i.e., ignore symmetry, slip, and flow-through boundaries.

`plane_center(1:3,:) = 0.0`

This is a point on a requested sampling `'plane'`; it fixes the location.

`plane_normal(1:3,:) = 0.0`

This is a normal vector of sampling `'plane'`; it fixes the orientation.

`box_lower_corner(1:3,:) = -1e+12`

This is the coordinate of the lower corner of a `'box'`.

`box_upper_corner(1:3,:) = 1e+12`

This is the coordinate of the upper corner of a `'box'`.

`sphere_center(1:3,:) = 0.0`

This is the coordinate of `'sphere'` center; it fixes the location.

`sphere_radius(:) = 0.0`

This is the radius for `'sphere'`; it fixes the size.

`circle_center(1:3,:) = 0.0`

This is the coordinate of center of a `'circle'`; it fixes the location.

circle_normal(1:3,:) = 0.0

This is the normal vector for a 'circle'; it fixes the orientation.

circle_radius(:) = 0.0

This is the radius for a 'circle'; it fixes the size.

cylinder_face1(1:3,:) = 0.0

This is the coordinate for the center of the first face of a 'cylinder'.

cylinder_face2(1:3,:) = 0.0

This is the coordinate for center of the second face of a 'cylinder'.

cylinder_radius(:) = 0.0

This is the radius of a 'cylinder'.

cone_face1(1:3,:) = 0.0

This is the coordinate for center of the first face of a 'cone'.

cone_face2(1:3,:) = 0.0

This is the coordinate for center of the second face of a 'cone'.

cone_radius1(:) = 0.0

This is the radius of the first face of a 'cone'.

cone_radius2(:) = 0.0

This is the radius of the second face of a 'cone'.

corner1(1:3,:) = 0.0

This is the coordinate of the first corner of a 'quad'; the corners proceed clockwise.

corner2(1:3,:) = 0.0

The coordinate of the second corner of a 'quad'.

corner3(1:3,:) = 0.0

The coordinate of the third corner of a 'quad'.

corner4(1:3,:) = 0.0

The coordinate of the fourth corner of a 'quad'.

patch_symmetry(:) = 1.0

When type_of_data(:)= 'integrated', this option multiplies the computed integrand on a partial domain to represent the entire configuration.

For example, 2.0 should be used to represent the whole configuration when simulating a model with half symmetry and 4.0 should be used to represent the whole configuration when simulating a model with quarter symmetry.

`number_of_points(:) = 0`

This is the number of points to be sampled by `'boundary_point'` or `'volume_point'`.

`points(1:3,:,:) = 0.0`

These are the coordinates of `boundary_point` and `volume_point` sampling. The first index is the Cartesian direction, the second index is the geometry, and the last index is the point in this geometry.

`p1_line(1:3,:) = 0.0`

This is the first end point of a line in `line` sampling.

`p2_line(1:3,:) = 0.0`

This is the second end point of a line in `line` sampling.

`schlieren_aspect = ''`

This is the Cartesian direction for `'schlieren'` view,

`'y'` Schlieren viewing along y axis.

`'z'` Schlieren viewing along z axis.

`' '` Schlieren viewing along `window_normal`.

`window_normal(1:3,:) = 0.0`

This is the window normal vector for `'schlieren'`; it fixes orientation.

`window_height(:) = 0.0`

This is the window height for `'schlieren'`.

`window_width(:) = 0.0`

This is the window width for `'schlieren'`.

`window_depth(:) = 1.0e+6`

This is the window depth or “thickness” for `'schlieren'`.

`window_center(1:3,:) = 0.0`

This is the window center for `'schlieren'`.

`number_of_rows(:) = 0`

This is the vertical number of pixels in the `'schlieren'` window.

<u>`number_of_columns(:) = 0`</u>

This is the horizontal number of pixels in the `'schlieren'` window.

<u>`model_center(1:3,:) = 0.0`</u>

This is the model center for `'schlieren'`.

<u>`plot_lines(:) = .false.`</u>

This plots lines for `'schlieren'`.

<u>`make_shadow = .false.`</u>

The boundary will cast a shadow in `schlieren` output.

<u>`blanking_list_count(:) = 0`</u>

This is the number of boundaries to search for `'schlieren'` boundary shadow.

<u>`blanking_list(:) = ''`</u>

This is a list of boundaries to search for `'schlieren'` shadow. Commas and dashes can be used to specify ranges, i.e., `'1,2,5-7'`.

<u>`isosurf_variable(:) = 'p'`</u>

This is the variable used to define the geometry of an `'isosurface'` and `isocrinkle`.

`‘p’` Pressure.

`‘rho’` Density.

`‘u’` X-component of velocity.

`‘v’` Y-component of velocity.

`‘w’` Z-component of velocity.

`‘entropy’` Entropy.

`‘vort_x’` X-component of vorticity.

`‘vort_y’` Y-component of vorticity.

`‘vort_z’` Z-component of vorticity.

`‘vort_mag’` Total magnitude of vorticity vector.

`‘vort_mag_avg’` Average total magnitude of vorticity vector.

`‘vort_mag_rms’` RMS total magnitude of vorticity vector.

`‘q_criterion’` Q-criterion.

`‘mach’` Mach number.

`‘temperature’` Temperature.

`‘p_tavg’` Time average pressure.

`‘rho_tavg’` Time average density.

`‘u_tavg’` Time average x-component of velocity.

`‘v_tavg’` Time average y-component of velocity.

`‘w_tavg’` Time average z-component of velocity.

`‘p_trms’` RMS of pressure.

`‘rho_trms’` RMS of density.

`‘u_trms’` RMS of the x-component of velocity.

`‘v_trms’` RMS of the y-component of velocity.

`‘w_trms’` RMS of the z-component of velocity.

`‘critical_d’` critical d

`‘s1a’` other option

`‘s1b’` other option

`‘s1’` other option

`‘s2’` other option

`‘lambda1’` Adjoint variable for the 1st governing equation.

`‘lambda2’` Adjoint variable for the 2nd governing equation.

`‘lambda3’` Adjoint variable for the 3rd governing equation.

`‘lambda4’` Adjoint variable for the 4th governing equation.

`‘lambda5’` Adjoint variable for the 5th governing equation.

`‘lambda6’` Adjoint variable for the 6th governing equation.

`‘lambda7’` Adjoint variable for the 7th governing equation.

`‘processor_id’` The assigned processor ID.

`‘bird_breakdown’` Bird breakdown factor.

`‘gllk_breakdown’` Gradient Length Local Knudsen breakdown factor.

`‘tt’` Translational-rotational temperature in the generic gas path

`‘tv’` Vibrational-electronic temperature in the generic gas path

`‘ptot’` Total pressure

`‘ttot’` Total temperature

`‘species_????’` Species density where ???? is the species name in file tdata

`‘mass_fr_????’` Species mass fraction where ???? is the species name in file tdata `‘mole_fr_????’` Species mole fraction where ???? is the species name in file tdata

<u>`isosurf_value(:) = 0.0`</u>

This is the value of `isosurf_variable(:)` to create the `’isosurface’` and `isocrinkle` geometry.

<u>`isosurf_box(:) = .false.`</u>

This clips the sampling geometry to be inside a box sized by `*_range_*` within `isosurf_dist_threshold`.

<u>`x_range_lower(:) = -1.0`</u>

This limits `isosurface` or `isocrinkle` when `isosurf_box(:) = .true.`

<u>`x_range_upper(:) = 1.0`</u>

This limits `isosurface` or `isocrinkle` when `isosurf_box(:) = .true.`

<u>`y_range_lower(:) = -1.0`</u>

This limits `isosurface` or `isocrinkle` when `isosurf_box(:) = .true.`

<u>`y_range_upper(:) = 1.0`</u>

This limits `isosurface` or `isocrinkle` when `isosurf_box(:) = .true.`

<u>`z_range_lower(:) = -1.0`</u>

This limits `isosurface` or `isocrinkle` when `isosurf_box(:) = .true.`

<u>`z_range_upper(:) = 1.0`</u>

This limits `isosurface` or `isocrinkle` when `isosurf_box(:) = .true.`

<u>`isosurf_dist_threshold(:) = 0.0`</u>

This trims portions of an `isosurface` or `isocrinkle` that have a distance to the surface less then this threshold. It requires `isosurf_box(:) = .true.`

<u>`variable_list(:) = ’’`</u>

The variables to be sampled. These variables override the variables specified in `&sampling_output_variables` for this sampling object.

<u>`snap_output_xyz = .true.`</u>

This snaps the requested points to the nearest surface.

<u>`dist_tolerance = 1.0e-3`</u>

This is the tolerance used when `snap_output_xyz` is engaged.

`fwh_formatted = .false.`

Write Ffowcs Williams-Hawkings in Fortran unformatted format. The default is Fortran stream (C-binary).

`append_history(:) = .false.`

This option removes the step number from the filename and opens it with append.

`asynchronous_fwh = .false.`

This uses asynchronous I/O for permeable FWH output.

`boundary_point(1:3,:) = 0.0`

This defines the location on a boundary where the friction velocity is to be calculated for use in plotting data in wall units. Additional input that is required is `boundary(ib)`.

`boundary(:) = 0`

This defines the boundary on which `boundary_point(1:3,ib)` will be found. Also used with `type_of_geometry()='analyse_boundary'` to calculate the displacement and momentum thickness of the surface boundary layer.

`boundary_layer_edge_parameter(:) = 'enthalpy'`

This defines the physical parameter used to determine the edge of the boundary layer analysis. It is used in conjunction with `type_of_geometry()= 'analyse_boundary'`.

`'enthalpy'` Enthalpy.

`'velocity'` Velocity.

`'vorticity'` Vorticity.

`boundary_layer_edge_value(:) = 0.995`

This defines the value of the boundary layer edge parameter used for determining the boundary layer edge. It is used in conjunction with `type_of_geometry()='analyse_boundary'` in terms of the quantity specified by `boundary_layer_edge_parameter(ib)`.

`boundary_layer_search_extent(:) = 1.0e+6`

This defines the maximum boundary layer integration extent from the `boundary(:)` definition in grid units.

`need_wall_data(:) = .false.`

This option activates the calculation of the friction velocity at a point on a wall for plotting of data in wall units. Other required input would be `boundary(ib)` and `boundary_point(:,ib)`. A sample asking for a boundary layer profile in wall units on boundary number 2 at the point (4., 0.05,0.) could have the following form:

```
&sampling_parameters
number_of_geometries = 1
type_of_geometry(1)  = 'line'
p1_line(:,1)         = 4.0,0.05,-10.
p2_line(:,1)         = 4.0,0.05, 10.
boundary(1)          = 2
boundary_point(:,1)  = 4.0,0.05,0.
need_wall_data(1)    = T
variable_list(1)='uplus,yplus,kplus,wplus'
sampling_frequency(1) = -1
/
```

`reference_length = 0.0`

This is the reference length for Re_τ used in `&sampling_output_variables`.

B.4.34 &spatial_avg

This feature is used to spatially average a time-accurate solution (i.e., from a DES simulation) of flow configurations with homogeneous directions. This option produces profiles of turbulence quantities and other desired output variables versus the wall normal coordinate. The averaging can be performed along one or two homogeneous directions. An underlying structured grid convention is used to specify details of the averaging location(s), where $\hat{k}$ is the spanwise, $\hat{i}$ is the streamwise, and $\hat{j}$ is the wall normal direction. The structured grid used as the averaging domain can be a Cartesian grid defined by the bounds in the namelist or a structured grid provided with a name to a file with a single-block grid in ASCII PLOT3D format.

```
&spatial_avg
  req_spatial_avg            = .false.
  domain_type                = 'box'
  struct_grid_file           = ''
  box_min_bound(1:3)         = 0.0
  box_max_bound(1:3)         = 0.0
  span_avg                   = .false.
  stream_avg                 = .false.
  spatial_avg_variable_list  = ''
  no_of_spanwise_locations   = -1
  no_of_streamwise_locations = -1
  streamwise_location(:)     = 0.0
  spanwise_location(:)       = 0.0
  ref_wall_bdry_index        = 0
/
```

`req_spatial_avg = .false.`

This enables FUN3D to perform spatial averaging of the results in one or two homogeneous directions.

`domain_type = 'box'`

This describes the type of the domain over which the averaging has to be performed. The in-built domain is a `'box'`. A structured `'grid'` file in PLOT3D format can be read for averaging over arbitrary planes and domain shapes with a file name of `struct_grid_file`.

`struct_grid_file = ''`

This is the name of the structured grid file stored in PLOT3D format, with `i` as the streamwise direction, `j` as the wall normal direction, and `k` as the spanwise direction. This file name is only used only when `domain_type = 'grid'`.

`box_min_bound(1:3) = 0.0`

This is the minimum value of each Cartesian coordinate of the bounding box of the spatial averaging domain. It is also used to control bounds on the grid when averaging is performed with `domain_type = 'grid'`.

`box_max_bound(1:3) = 0.0`

This is the maximum value of each Cartesian coordinate of the bounding box of the spatial averaging domain. It is also be used to control bounds on the grid when averaging is performed with `domain_type = 'grid'`.

`span_avg = .false.`

This option enables spatial averaging in the spanwise direction.

`stream_avg = .false.`

This option enables spatial averaging in the streamwise direction.

`spatial_avg_variable_list = ''`

This parameter is used to specify the variables which are required to be averaged. All the variables in the `&sampling_output_variables` namelist are also available here.

`no_of_spanwise_locations = -1`

This parameter is used to set the number of locations while performing spanwise averaging. The averaging is performed at equally spaced intervals between the `box_min_bound` and `box_min_bound`. When set to -1, it uses the grid lines as the basis for performing the averaging. When only streamwise averaging is required, the spanwise locations can be specified as well.

`no_of_streamwise_locations = -1`

This parameter is used to set the number of locations while performing streamwise averaging. The averaging is performed at equally spaced intervals between the `box_min_bound` and `box_min_bound`. When set to -1, it uses the grid lines as the basis for performing the averaging. When only spanwise averaging is required, the stream locations can be specified as well.

`streamwise_location(:) = 0.0`

This parameter is used to specify the coordinate of the streamwise locations at which spanwise averages are required. It is used when only spanwise averages are required at certain locations in streamwise direction.

`spanwise_location(:) = 0.0`

This parameter is used to specify the coordinate of the spanwise locations at which streamwise averages are required. It is used when only streamwise averages are required at certain locations in the spanwise direction.

`ref_wall_bdry_index = 0`

This is the boundary index of the wall which is to be used as reference, while computing average wall related quantities such as `u+`, `y+`. The average value of `u_tau` is also printed in the output.

B.4.35 &slice_data

This namelist specifies boundary slices for visualization and to obtain loads. Output frequency is controlled by `slice_freq` in the `&global` namelist, where zero for no output, `-1` for output at the end of run, and a positive integer for periodic output.

This is a limited ability to take slices through **boundary surfaces**. For example, spanwise cuts along a wing may be extracted, and then the resulting pressure and skin friction data may be plotted at each station.

Slices can only be specified (input) as Cartesian planes (e.g., constant y). In some cases, slices may be desired on planes that are not Cartesian planes. For example, slices may be desired normal to the quarter chord of a swept wing. To accommodate this type of situation, the user may input a 'custom transform' matrix that takes the grid boundaries on which slices are desired, and maps them into a coordinate system in which slice planes *are* Cartesian planes. For moving-body cases, slices are specified as Cartesian planes in the grid at $t = 0$, which define a set of body-fixed slices. A 'custom transform' may be supplied in the case where the desired slices on the grid at $t = 0$ do not align with a Cartesian plane. As the body moves in time these slices move with the body. Each slice will be written as one or more zones (loops) in an ASCII formatted Tecplot™ file with the naming convention:

```
[project_rootname]_slice.dat
```

The variables output to this file are: `x`, `y`, `z`, `cp`, `cfx`, `cfy`, and `cfz` at each output time step. These slicing output variables are not customizable by the user.

Slicing occurs in the inertial frame, unless an alternate reference frame is specified. For stationary geometries, the inertial frame is the only option. For moving body cases, either the frame of one of the moving bodies or an observer frame may specified.

When slicing boundary surfaces, a file called `slice.info` is output that echos much of the input data. When the slicing is successful, the file will also contain information about the number of points in the slice.

Below, namelist variables are defined. See section B.4.35 for some important considerations when using this capability. Note that the concepts of chord, leading edge, trailing edge, etc are only applicable to geometries that are topologically similar to wings and airfoils. Slicing can still be performed on other types of geometries in order to extract sectional data, but the resulting integrated sectional forces projected onto the chord, normal and span directions will likely be meaningless.

```
&slice_data
  nslices                    = 0
  replicate_all_bodies       = .false.
  output_initial_state      = .false.
  slice_append_on_restart   = .true.
  slice_x(:)                = .false.
  slice_y(:)                = .true.
  slice_z(:)                = .false.
  slice_location(:)         = 0.0
  slice_increment           = 0.0
  coordinate_shift          = epsilon(1.0)
  debug_xyz_to_slice        = .false.
  debug_slice_to_xyz        = .false.
  debug_le_locations        = .false.
  debug_te_locations        = .false.
  xx_box_max(:)             = huge(1.0)
  yy_box_max(:)             = huge(1.0)
  zz_box_max(:)             = huge(1.0)
  xx_box_min(:)             = -huge(1.0)
  yy_box_min(:)             = -huge(1.0)
  zz_box_min(:)             = -huge(1.0)
  slice_xmc(:)              = huge(1.0)
  slice_ymc(:)              = huge(1.0)
  slice_zmc(:)              = huge(1.0)
  n_bndrys_to_slice(:)      = 0
  bndrys_to_slice(:,:)      = 0
  slice_frame(:)            = ''
  slice_group(:)            = 1
  chord_dir(:)              = 1
  te_def(:)                 = 1
  le_def(:)                 = 30
  corner_angle(:)           = 120.0
  use_local_chord           = .true.
  tecplot_slice_output      = .true.
  slice_output_frame        = ''
  output_sectional_forces   = .true.
  slice_initial_coords      = .false.
  pitch_plane_normal(1,1:3) = 0.0, 0.0, 1.0
  compute_solidity          = .false.
  nblades                   = -1
  rotor_radius              = -1.0
  custom_transform(1,1,1:4) = 1.0, 0.0, 0.0, 0.0
  custom_transform(1,2,1:4) = 0.0, 1.0, 0.0, 0.0
  custom_transform(1,3,1:4) = 0.0, 0.0, 1.0, 0.0
  custom_transform(1,4,1:4) = 0.0, 0.0, 0.0, 1.0
  output_in_slice_coords(:) = .false.
/
```

`nslices = 0`

This is the number of slices to create. If negative, then data for only one slice station need be input, along with `slice_increment`, and all the data specified for the first station will be applied to subsequent stations, with the exception of the slice location, which will be set using the slice increment between stations.

`replicate_all_bodies = .false.`

This will set similar slice stations on multiple bodies with minimal input beyond that required for slicing the first body. This is particularly useful for rotorcraft applications where multiple blades are to be sliced. This variable duplicates the input slice info for all moving bodies, with the exception of the `slice_frame` and the `bndrys_to_slice`.

`output_initial_state = .false.`

When `.true.`, this causes requested slice data to be written for the initial state of the solution in the current run.

`slice_append_on_restart = .true.`

When `.true.`, this appends to the existing slicing-related output files when restarting; when `.false.`, this removes the existing slicing-related output files at startup and new ones are created for the current execution. Slicing-related output files are the `[project_rootname]_slice.dat` file and the `[project_rootname].sectional_forces` file.

`slice_x(:) = .false.`

This extracts the slice at $x =$ `slice_location` in the specified reference frame.

`slice_y(:) = .true.`

This extracts the slice at $y =$ `slice_location` in the specified reference frame.

`slice_z(:) = .false.`

This extracts the slice at $z =$ `slice_location` in the specified reference frame.

`slice_location(:) = 0.0`

This is the coordinate value at which slice is taken.

`slice_increment = 0.0`

When `nslices` is negative, this is the increment in slice coordinate between consecutive slice stations. Unused if `nslices` is positive.

`coordinate_shift = epsilon(1.0)`

This is the surface point coordinate shift used if the `slice_location` happens to fall within machine zero of a surface point. Note that the surface point is temporarily shifted by −`coordinate_shift`, so in some instances a negative value of `coordinate_shift` may be needed if the problem point is on the inboard side of a geometry feature.

`debug_xyz_to_slice = .false.`

This triggers Tecplot™ output of the boundary surfaces selected for slicing in both the original, input grid position, as well in the transformed position corresponding to any 'custom transforms'. This output can be useful for verifying that a user-supplied custom transform is correct. Output files are named as `slice_surface_n.dat` for boundary surface `n` in the original position, and `slice_surface_100+n.dat` for boundary surface `n` in the transformed position. Intended primarily for developer use.

`debug_slice_to_xyz = .false.`

This triggers Tecplot™ output of the boundary surfaces selected for slicing in both the original, input grid position, as well as the the corresponding surfaces after they are transformed back from the slicing frame to current grid position. Output files are named as `slice_surface_200+n.dat` for boundary surface `n` in the original grid position, and `slice_surface_300+n.dat` for boundary surface `n` after it is transformed back to the original orientation. Intended primarily for developer use.

`debug_le_locations = .false.`

This triggers Tecplot™ output that may be useful in sorting out issues with the determination of the leading edge. Two files are output - the first contains the collection of points that will be examined to determine the leading edge location, and is named `debug_le_candidates.dat`. The second file contains the computed location of the leading edge, and is named `debug_le_locations.dat`. In both files there is a separate zone for each slice location. It is recommended that `debug_le_locations` be used together with `debug_xyz_to_slice`, so that the computed leading edge locations can be plotted as scatter points on top of the surface mesh(es) that result from `debug_xyz_to_slice`. The allows the user to assess whether the trailing edge points were correctly determined. The candidate points may also be plotted as scatter points on the surface meshes; obviously if the candidate points are incorrect or inadequate, then the leading edge point cannot be correctly computed. Finally, if used with `debug_xyz_to_slice`, note that two series of files are output from that option; for purposes of leading edge assessment, use the set of files whose names contain numbers greater than 100.

`debug_te_locations = .false.`

This triggers Tecplot™ output that may be useful in sorting out issues with the determination of the trailing edge. Use this option in the same manner as described for `debug_le_locations.dat`.

`xx_box_max(:) = huge(1.0)`

This is the maximum x-coordinate used to define a bounding box to constrain the slicing to filter unwanted intersections.

`yy_box_max(:) = huge(1.0)`

This is the maximum y-coordinate used to define a bounding box to constrain the slicing to filter unwanted intersections.

`zz_box_max(:) = huge(1.0)`

This is the maximum z-coordinate used to define a bounding box to constrain the slicing to filter unwanted intersections.

`xx_box_min(:) = -huge(1.0)`

This is the minimum x-coordinate used to define a bounding box to constrain the slicing to filter unwanted intersections.

`yy_box_min(:) = -huge(1.0)`

This is the minimum y-coordinate used to define a bounding box to constrain the slicing to filter unwanted intersections.

`zz_box_min(:) = -huge(1.0)`

This is the minimum z-coordinate used to define a bounding box to constrain the slicing to filter unwanted intersections.

`slice_xmc(:) = huge(1.0)`

This is the x-coordinate of the moment center, in the specified reference frame, for aerodynamic moments acting on the slice. The default value will result in the moment center being taken as the computed quarter chord of the slice.

`slice_ymc(:) = huge(1.0)`

This is the y-coordinate of the moment center, in the specified reference frame, for aerodynamic moments acting on the slice. The default value will result in the moment center being taken as the computed quarter chord of the slice.

`slice_zmc(:) = huge(1.0)`

This is the z-coordinate of the moment center, in the specified reference frame, for aerodynamic moments acting on the slice. The default value will result in the moment center being taken as the computed quarter chord of the slice.

`n_bndrys_to_slice(:) = 0`

This is the number of candidate boundaries to search while computing slice-plane intersections. The index is the slice. By default, all solid boundaries will be searched. Specifying which boundaries are candidates for slicing may speed up the slicing process and can be used to filter out unwanted intersections or to slice nonsolid boundaries.

`bndrys_to_slice(:,:) = 0`

This is the list of `n_bndrys_to_slice` boundaries, when the variable `n_bndrys_to_slice` is greater than zero. The first index is the slice and the second index is the boundary.

`slice_frame(:) = ''`

This is the name of a reference frame to associate with the slice. Blank indicates the inertial frame, and is the only valid choice for slicing stationary boundaries. For slicing boundaries of moving bodies, the slice frame should be specified as the `body_name` entered in the `&body_definitions` namelist.

`slice_group(:) = 1`

This assigns this slice to a particular group number. Within a group, slice locations must be given in ascending order.

`chord_dir(:) = 1`

This is the direction of local chord relative to the direction from leading edge to trailing edge, in the slice plane. The value 1 indicates local chord in direction from leading edge to trailing edge. The value -1 indicates local chord in direction from trailing edge to leading edge. Determination of the leading and trailing edges is described below.

`te_def(:) = 1`

This is the number of points or line segments to consider when defining the trailing edge of the slice (see Fig. B4). A value of 1 defines the trailing edge as the aft-most point. This is best for sharp trailing edges. A value of `te_def` > 1 initiates a search over the aft-most `te_def` segments for corners, after which the trailing edge is taken as the average coordinate over all the detected corners. Two corners are assumed to be the desired number, and warnings are output if only one or more than two are found. The value of `te_def` must be chosen judiciously. It should be large enough to allow both corners to be found but not so large as to cause excessive searching or for any non-trailing edge corners to be found. A value greater than one is best for, and recommended only for, squared-off trailing edges. A negative number indicates a parabolic fit of the aft-most abs(`te_def`) points, which may be best for rounded or blunted trailing edges. However, the parabolic-fit option entails extra work, which may be a consideration for time-dependent cases with many slices, such as coupled CFD/CSD rotorcraft cases

`le_def(:) = 30`

This is the number of points to consider when defining the leading edge of the slice (see Fig. B5). A value of 1 defines the leading edge as the forward-most point. Use this if nothing else works or for special cases. A value of `le_def` > 1 indicates a search over the forward-most `le_def` points for the one that has the maximum distance from the previously determined trailing edge. A value of `le_def` > 1 is generally the best choice provided that the trailing edge can be accurately located. A negative number indicates a parabolic fit over the forward-most abs(`le_def`) points. This option may be useful for poorly defined leading edges. However, the parabolic-fit option entails extra work, which may be a consideration for time-dependent cases with many slices, such as coupled CFD/CSD rotorcraft cases

`corner_angle(:) = 120.0`

This is used in conjunction with a `te_def` greater than 1. Angles between adjacent sliced segments that are less than `corner_angle` degrees will be considered a corner between the two segments. For squared-off trailing edges, two and only two corners should be detected; warnings are output if only one or more than two are found.

`use_local_chord = .true.`

Use the computed local (sectional) chord based on the computed leading edge and trailing edge locations to normalize the sectional force and moment data. When `.false.`, the value of `x_moment_length` in `&force_moment_integ_properties` will be used instead of the locally computed chord.

`tecplot_slice_output = .true.`

This outputs the sliced data to a formatted Tecplot™ file that is named `[project_rootname]_slice.dat`. This file can become very large for unsteady flows with frequently written data at many slice locations.

`slice_output_frame = ''`

This is the reference frame in which slice data is output when Tecplot™ output is requested (replaces obsolete `output_in_slice_coords`).

`' '` slice data output in the inertial frame

`'slice_frame'` slice data output the frame that was associated with the slice.

`'blade_frame'` slice data output in the rotor blade frame *valid only if* `overset_rotor = .true.`

`'custom_frame'` slice data output in the frame specified via the optional `custom_transform`

`'observer'` slice data output in the observer frame (only for `moving_grid = .true.`)

`output_sectional_forces = .true.`

This outputs detailed force and moment data for each slice to a formatted file, `[project_rootname].sectional_forces`. This file contains force and moment data, like that in the `[project_rootname].forces` file, for each and every slice. In addition, it contains geometrical data for each slice (leading and trailing edge coordinates, moment center, etc.) This file can become very large for unsteady flows with frequently written data at many slice locations. The data in the file, especially the geometry data, can be useful to assess whether the slicing is working as expected.

`slice_initial_coords = .false.`

Obsolete; current slicing algorithm evaluates interpolation coefficients only once at the start of the computation based on the initial mesh. It is included for backwards compatibility with fun3d.nml files, but may be removed in a future version.

`pitch_plane_normal(1,1:3) = 0.0, 0.0, 1.0`

This is the unit normal to the plane from which the geometric pitch angle is measured. For articulated rotor cases (`overset_rotor = .true.`), `pitch_plane_normal` is automatically taken as a unit vector along the rotor shaft.

`compute_solidity = .false.`

This compute the (thrust weighted) rotor solidity from the sliced geometry.

`nblades = -1`

This is the number of blades to use when computing rotor solidity; note: a positive number should be used only when `overset_rotor = .false.` When `overset_rotor = .true.`, `blades` is set automatically based on rotor-specific input from `rotor.input`.

`rotor_radius = -1.0`

This is the rotor radius to use when computing rotor solidity; note: a positive number should be used only when `overset_rotor = .false.` When `overset_rotor = .true.`, `rotor_radius` is set automatically based on rotor-specific input from `rotor.input`.

`custom_transform(1,1,1:4) = 1.0, 0.0, 0.0, 0.0`

This is a user-specified 4x4 transform matrix to allow slicing in a custom coordinate system in which slice planes are Cartesian planes. Note: `custom_transform` is compatible only with nonmoving grids at present. The transform matrix is specified as a mapping that takes coordinates *from* the input grid and maps them *to* the "slice" coordinate system in which slicing is performed on a Cartesian plane.

`output_in_slice_coords(:) = .false.`

Obsolete - specify `slice_output_frame` instead It is included for backwards compatibility with fun3d.nml files, but may be removed in a future version.

Important Considerations for Determination of Leading And Trailing Edges Determining the locations of airfoil leading and trailing edges is especially important for rotorcraft applications where airloads are usually examined (and provided to a CSD code, if applicable), in an airfoil section-aligned coordinate system. The leading and trailing edge points determine the orientation of this section aligned coordinate system. In the section-aligned system, the local x coordinate is aligned with the local chord, positive in the direction from the leading edge to the trailing edge. The local span direction is defined by the moment centers at the `slice_location` points, positive in the direction of increasing `slice_location`. The local normal direction is defined as the cross product of the local chord vector and local span vector. When slicing boundary data, the computed forces are computed in both the selected frame of reference (see `slice_frame`) and in an airfoil section aligned system. If the data in the section-aligned system is irrelevant to you, then you do not need to worry about choosing the detection parameters carefully; the default values should be reasonable. However, if resolution of forces and moments into a section-aligned system is important to you, then there are a number of things that should be considered:

1. Make sure the chord direction `chord_dir` is correct; the default is that going from the leading edge to the trailing edge is the same as traveling in the positive "chordwise" coordinate direction. For most applications this is the usual situation; however, the convention for rotorcraft applications is the opposite, requiring `chord_dir = -1`.

2. Since the best option for determining the leading edge (`le_def` > 1) uses the trailing edge location, care should be taken to get the trailing edge correct. For *sharp* trailing edges, this is very simple since the default of `te_def = 1` (i.e., use the aft-most point) is the best option. However, smoothly blunted or squared-off trailing edges are more difficult. When the boundary surface of an unstructured mesh is sliced, the resulting section will be comprised of line segments determined by the intersection of the specified plane and the edges of the surface triangles. These segments and the points that make up the segments will not usually be the same as the surface points; typically there are more segments and points arising from intersected triangles, as illustrated in Fig. B3. This greater point count should influence the selection of `te_def` and `le_def` values. You will need enough segments (`te_def` and `le_def`) to ensure that both corners are detected, but not so many that other, non trailing-edge corners (if present) are detected. Another parameter that may be of use to aid in the detection of corners is the `corner_angle`; corners with angles larger than `corner_angle` between adjacent segments will require a larger value of `corner_angle` for detection.

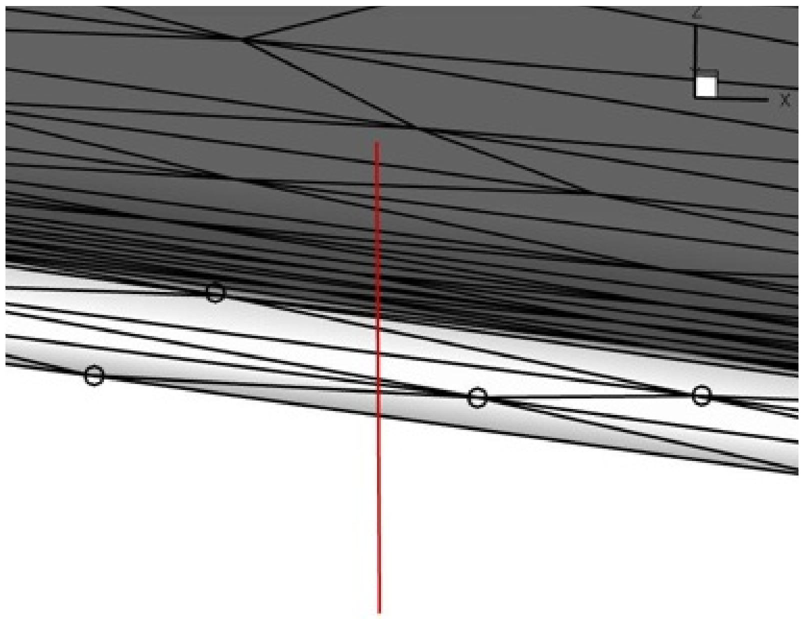


Figure B3: View looking upstream from the trailing edge of a rotor blade mesh; the light-colored region is the squared-off trailing edge; the red line shows the location where an x=constant slice will be taken; black circles indicate surface grid points on the trailing edge.

The resulting section corresponding to the slice depicted in Fig. B3 is shown in Fig. B4, where the view is zoomed in to the trailing edge region. The aft-most 8 segments (of the approximately 30 segments in this view) are shown in red. The computed trailing edge locations using two different `te_def` values are shown. The minimum `te_def` value at this particular station to pick up both corners would be 8, but a value of 20 was used in case another slice required more segments. If the blade was pitched downward rather than upward, then the point chosen by `te_def = 1` would be the lower corner, rather than the upper corner as shown.

3. Smoothly-blunted (rounded) trailing edges should be done with either `te_def = 1` (aft-most point) or via a parabolic fit of the aft-most abs(`te_def`) points; the latter option is probably better in general but will require some experimentation for the particular case at hand to choose the optimal number of points over which to fit the parabola.

4. The leading edge is typically easier to determine, if a good trailing edge position has already been found. The default value of `le_def = 30` (search the 30 forward-most points for the one with the greatest distance from the trailing edge location) should do a decent job for most cases.

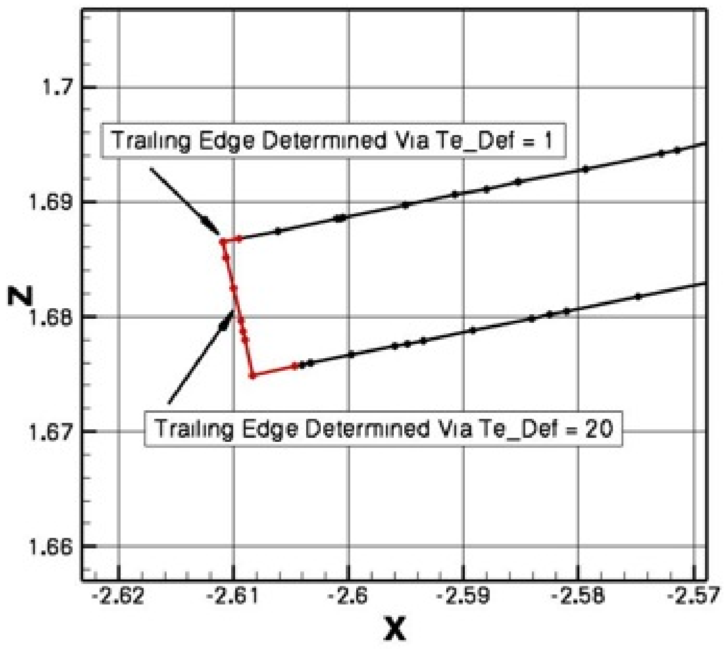


Figure B4: Sliced section corresponding to Fig. B3; zoomed in to the trailing edge region.

Figure B5 shows a sliced section, zoomed in to the leading edge region. The forward-most 20 segments (of the approximately 30 segments in this view) are shown in red. The computed leading edge locations using two different `le_def` values are shown. In this case, both results are fairly close but `le_def = 30` has picked out the true leading edge (as judged from the leading edge geometry at zero pitch angle).

5. The leading edge and trailing edge detection schemes can be somewhat sensitive to the input choices. For cases that rely on accurate resolution of forces and moments into section-aligned coordinates (e.g., rotorcraft), it is strongly recommended to spend some time up front to make sure that things are coming out as expected. To do this, inspect the `[project_rootname].sectional_forces` file for a particular slice station; at each station the computed leading and trailing edge coordinates will be output. Plot the corresponding station from the `[project_rootname]_slice.dat`, as done above, and make sure the computed coordinates are the correct ones. If many stations are sliced, it is impractical to inspect all of them in this manner, but it is good practice to spot check at least a few stations. For moving-geometry cases, try first running the case with `body_motion_only = .true.` in the `&global` namelist. This will allow output of the `[project_rootname].sectional_forces`

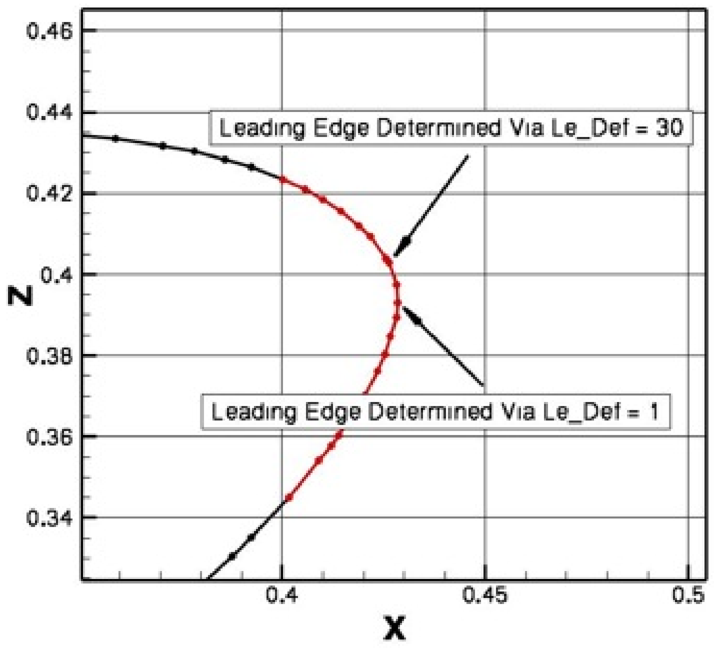


Figure B5: A sliced section, zoomed in to the leading edge region.

and [project_rootname]_slice.dat files without the expense of a flow solve or mesh deformation; for spot checking you may want to have the slicing done infrequently, perhaps using fewer stations than ultimately desired, as these output files can be huge.

6. While the [project_rootname].sectional_forces can be useful for spot checking, the data in the file is not in a format that is amenable to plotting. The FUN3D distribution utils/Rotorcraft directory contains a utility code that will read in both the files slice.info and [project_rootname].sectional_forces to output Tecplot™ files, for each slice group, containing force and moment data in the section-aligned coordinate system, as well as geometry data (leading edge, trailing edge, quarter-chord coordinates, and pitch angle).

7. After making sure that the leading edge and trailing edge positions are being computed correctly, you may want to turn off one or both of the [project_rootname].sectional_forces and [project_rootname] _slice.dat files unless needed. For instance, in rotorcraft applications with coupling to external CSD codes, although the blade boundary surfaces must be sliced to generate the aerodynamic loads data for the CSD code, this information is actually passed to the CSD code by another file; the [project_rootname].sectional_forces and [project_rootname] _slice.dat files are not used.

8. Although the slicing process will work for multielement airfoils, at this time the computation of the leading edge and trailing edge is only done for the entire section, not each element individually.

B.4.36 &overset_data

This namelist specifies information for overset grid simulations. The overset option may be used with either static or dynamic meshes. In either case, an initial overset connectivity file, corresponding to the configuration at t=0, must be provided. Connectivity files must be in the SUGGAR++ 'dci' format, or, if using the `dci_io` option (see below), in the FUN3D 'dcif' format. A conversion utility (`utils/dci_to_dcif`) is provided to convert from the dci format to the dcif format. The initial overset connectivity file must be named `[project].dci` (or `[project].dcif` if using the `dci_io` option). For dynamic meshes, the user may elect to provide precomputed connectivity files for each time step, in which case the files must be named `[project]1.dci` for time step 1 at t=Δt, `[project]2.dci` for time step 2 at t=2Δt, and so on. The naming convention is the same when using the `dci_io` option, except the file extension is `.dcif` rather than `.dci`.

```
&overset_data
  overset_flag         = .false.
  assembler            = 'suggar++'
  n_component_grids    = 0
  component_nnodes(:)  = 0
  component_mesh_id(:) = 0
  dci_on_the_fly       = .false.
  dci_period           = huge(1)
  reset_dci_period     = .false.
  dci_freq             = 1
  dci_dir              = '.'
  reuse_existing_dci   = .false.
  skip_dci_output      = .false.
  dci_io               = .false.
  dci_io_nproc         = 1
  input_xml_file       = ''
  input_imesh_file     = ''
  designbody2imesh(:)  = -1
/
```

`overset_flag = .false.`

When `.true.`, overset mesh capability is enabled.

`assembler = 'suggar++'`

This specifies the overset mesh system assembly/overset connectivity tool.

`'suggar++'` uses the SUGGAR++/DiRTlib assembler.

`'yoga'` uses the YOGA assembler.

`n_component_grids = 0`

This specifies the number of component grids in the composite mesh. Needed only when `assembler = 'yoga'`.

`component_nnodes(:) = 0`

For each of the `n_component_grids`, this specifies the number of nodes (grid points) in the component mesh. Needed only when `assembler = 'yoga'`.

`component_mesh_id(:) = 0`

For each of the `n_component_grids`, this specifies the `mesh_id` of the component mesh. Needed only when `assembler = 'yoga'`. A component grid that serves as a background grid and does not move in time should have a `component_mesh_id=0`. For dynamic (moving) components, the value of `component_mesh_id` should match the value of `mesh_id` specified for the moving body associated with this mesh in the `&body_definition` namelist (in the `moving_body.input` file). For statically transformed components, the value of `component_mesh_id` should match he value of `mesh_id` specified for the static transform associated with this mesh in the `&grid_transform` namelist. A combination of static transform (applied once at the beginning of the simulation) and dynamic transforms (applied each step of a time-accurate simulation) can be applied to the same component mesh provided the `mesh_id` values are consistent across the namelists.

Note: the data order in the `component_mesh_id` and `component_nnodes` arrays must be consistent with the ordering of the component grids in the composite mesh. For example, if the composite mesh was generated by concatenating a wing grid followed by a flap grid, then the value of `component_nnodes(1)` should reflect the number of nodes in the to the wing grid and `component_nnodes(2)` should reflect the number of nodes in the flap grid.

`dci_on_the_fly = .false.`

This controls whether overset connectivity is computed as the grid moves, or whether overset connectivity has been pre-computed for each grid position and is available to read in. Ignored if `overset_flag = .false.` and `overset_rotor = .false.` in the `&rotor_data` namelist.

Note: if new overset connectivity is required each time step (e.g. for grid motion), then set `dci_on_the_fly = .true.`, regardless of the choice for `assembler`. All other `'dci'` type variables are ignored if `assembler = 'yoga'`.

dci_period = huge(1)

This controls the period (in terms of timesteps) at which the dci counter is reset. At time step dci_period+1, the flow solver will read overset data from the dci file [project]1.dci, corresponding to time step 1 at t=Δt. (Note: if dci_io = .true., the corresponding file name would be [project]1.dcif). For example, if the simulation involves rotation through 360 degrees with a time step corresponding to 1 degree per step, set dci_period = 360, and, if the user is providing precomputed dci files, the final one provided would be named [project]360.dci and would correspond to the 360 degree position. If instead the same 360 degree rotation was run with 0.5 degrees per step, set dci_period = 720 and the final dci file would be [project]720.dci, again corresponding to the 360 degree position. When using the dci_on_the_fly option, connectivity files are computed and written for the first dci_period timesteps, after which the code begins reading and reusing the connectivity files. The default value implies that the motion is not periodic and therefore the dci counter is never reset. Ignored if overset_flag = .false. and overset_rotor = .false. in the &rotor_data namelist.

reset_dci_period = .false.

When .true., allows dci_period to be reset to a different value for restarting with a different time step.

dci_freq = 1

This controls how frequently the dci data is updated, either by computation within the flow solver, or by reading a new dci file. Dci data is updated every dci_freq time steps.

dci_dir = '.'

This is the directory where dci files are located. Note: A trailing forward slash (/) is automatically added and should not be included in the directory name.

reuse_existing_dci = .false.

When .true., allows the computation of dci data to be skipped if a dci file for the current time step already exists. This option is typically used in conjunction with dci_period, so that dci files are computed on the fly for the first dci_period time steps, and then files are reused for all subsequent periods of grid motion, without having to change dci_on_the_fly in between. Ignored if dci_on_the_fly = .false..

skip_dci_output = .false.

When .true., the solver will not save the dci data to a file. Ignored if dci_on_the_fly = .false..

`dci_io = .false.`

When `.true.`, dci files are read from disk with a dedicated rank (processor) to help mask communication with computation.

`dci_io_nproc = 1`

When `dci_io = .true.`, this specifies the number of ranks to use for loading of dci files.

`input_xml_file = ''`

This is the file containing the XML commands for SUGGAR++. Specify the same `Input.xml` file that was used to generate the initial composite grid with the "stand-alone" SUGGAR++ code. Needed only when `assembler = 'suggar++'`.

`input_imesh_file = ''`

This is the file containing imesh data specified by the T-Infinity composite mesh builder. Needed only when `assembler = 'yoga'` and when not specified manually via namelist parameters.

`designbody2imesh(:) = -1`

When using a user-specified parameterization that parameterizes an entire component grid rather than just surface points on a body in that component grid, this array of integers is used to map the body index to the imesh index for that component grid and must be provided by the user.

B.4.37 &rotor_data

This namelist controls high-level rotor simulation settings. Eventually, this namelist may subsume `rotor.input`.

```
&rotor_data
  comprehensive_rotor_coupling = 'none'
  niters_cfd(1,1:2)            = 360, 180
  overset_rotor                = .false.
  solidity_evaluation          = 'rectangular_blade'
  solidity(:)                  = -1.0
  output_rotor_performance     = .false.
  output_comprehensive_loads   = .false.
/
```

<u>`comprehensive_rotor_coupling = 'none'`</u>

This controls whether the code is to be coupled to a rotorcraft comprehensive code, and if so, which one. Note: with one exception, coupling requires that additional data be set in the `&slice_data` namelist to define and extract the sectional aerodynamic loads that are passed to the comprehensive code.

`'none'` not coupled.

`'camrad'` loosely coupled to CAMRAD-II.

`'helios'` coupled into the CREATE-AV/Helios rotorcraft simulation framework as a near-body solver; `&slice_data` input not required but may be set if desired to extract sectional airloads for postprocessing.

`'rcas'` loosely coupled to RCAS.

`'rcas_tight'` tightly coupled to RCAS.

`'dymore'` loosely coupled to DYMORE.

`'dymore_tight'` tightly coupled to DYMORE.

<u>`niters_cfd(1,1:2) = 360, 180`</u>

This flag pertains only to loose coupling with RCAS and DYMORE. `niters_cfd(rotor_index,1)` controls the number of CFD time steps to perform before the first loose coupling data interchange with the comprehensive code for `rotor_index`; `niters_cfd(rotor_index,2)` controls the number of CFD time steps taken between subsequent data interchanges for `rotor_index`. Each rotor may have different interchange controls, with the first index of the `niters_cfd` array indicating the rotor number. The number of steps should be multiples of 360 divided by the number of blades on the particular rotor.

`overset_rotor = .false.`

This controls whether overset meshes are used for moving rotor simulations. When `.true.`, the rotor motion is governed by the `rotor.input` file.

`solidity_evaluation = 'rectangular_blade'`

This controls how the rotor solidity, σ, is evaluated. Solidity is used for the rotor performance measures that are output.

`'specified'` User specifies the value of solidity for each rotor.

`'compute'` Solidity is evaluated via 'slicing' of the blade surface into segments and integrating over the segments. The same segments defined in the `&slice_data` namelist for sectional loads extraction are used for the evaluation of solidity. The method computes the thrust-weighted solidity using an effective blade chord; $\sigma = \frac{nblades\ R\ C_{eff}}{\pi\ R^2}$, where C_{eff} is the (thrust-weighted) effective blade chord computed by integrating the surface slices, *nblades* is the number of blades, and R is the rotor radius, the last two as specified in the `rotor.input` file.

`'rectangular_blade'` Solidity is evaluated by assuming the rotor blades have a rectangular planform; $\sigma = \frac{nblades\ R\ chord}{\pi\ R^2}$, where *nblades* is the number of blades, R is the rotor radius, and *chord* is the blade chord, all as specified in the `rotor.input` file.

`solidity(:) = -1.0`

This is the user-supplied value for rotor solidity; only used when `solidity_evaluation = 'specified'`.

`output_rotor_performance = .false.`

This flag turns on evaluation and output of rotor performance metrics such and thrust, torque, power (dimensional and coefficient). This flag will automatically be set to true when `overset_rotor = .true.` Requires the case be run with moving grids, not with actuator disc or noninertial options.

`output_comprehensive_loads = .false.`

This flag turns on evaluation and output of rotor blade sectional airloads, primarily to enable coupling with a rotorcraft comprehensive code. However, even without comprehensive code coupling, this data may be of interest for rigid-blade analysis. This flag will automatically be set to true when `overset_rotor = .true.` This option requires the specification of `&slice_data` to define stations at which airloads are extracted.

B.4.38 &adapt_metric_construction

This namelist controls how the metric is formed for metric-based mesh adaptation. More details on grid adaptation can be obtained in section 8. The metric is either feature-based or output-based and is constructed through a common framework.

For a feature-based metric, `adapt_feature_scalar_key` specifies the scalar field S. See the description of `adapt_feature_scalar_key` for a list of available options. The intensity I at each node is constructed from S at each node with the `adapt_feature_scalar_form` method applied across each incident edge. See the description of `adapt_feature_scalar_form` for a list of available edge operators and how they are gathered to the nodes. The feature-based isotropic scaling of the grid is h/h_0,

$$\frac{h}{h_0} = \min\left(\left(\frac{I}{T}\right)^{-\omega}, g\right), \tag{B2}$$

where T is the `adapt_output_tolerance`, ω is the `adapt_exponent`, and g is the `adapt_max_edge_growth`. For feature-based adaptation, the setting `adapt_output_tolerance` is chosen to scale I, and its value will depend of the choice of `adapt_feature_scalar_key` and `adapt_feature_scalar_form`. Trial and error is required to determine an `adapt_output_tolerance` (T) setting that scales I to be greater than one in regions that have the features that require grid refinement. Therefore, it is case specific.

For an output-based metric, the intensity is given as

$$I = \frac{1}{2}\sum_{i=1}^{5}\left\{\left|[R^{\lambda}(\hat{\lambda})]_{i,\kappa}[\hat{Q}-\bar{Q}]_{i,\kappa}\right| + \left|[\hat{\lambda}-\bar{\lambda}]_{i,\kappa}[R(\hat{Q})]_{i,\kappa}\right|\right\}, \tag{B3}$$

where R is the flow residual, R^{λ} is the adjoint residual, Q is the flow solution, λ is the adjoint solution, the $\hat{}$ accent is a high-order reconstruction or interpolation, and the $\bar{}$ accent is a linear reconstruction or interpolation. Reconstruction or interpolation is selected with `adapt_error_estimation`. See Park [70] for details. The output-based isotropic scaling of the grid is h/h_0,

$$\frac{h}{h_0} = \max\left(\min\left(\left(\frac{I}{T}\right)^{-\omega}, g\right), 1/g\right), \tag{B4}$$

where ω is the `adapt_exponent` and g is the `adapt_max_edge_growth`. T is the `adapt_output_tolerance` divided by the number of nodes when it is positive. When `adapt_output_tolerance` is negative, T is `-adapt_output_tolerance` times the average or median I based on `adapt_statistics`.

The anisotropic feature-based and output-based metrics are constructed from h/h_0 in the same manner. The `adapt_hessian_method` is applied to the `adapt_hessian_key` scalar field to compute the Hessian H. The symmetric H is decomposed into eigenvalues Λ and eigenvectors X. The eigenvectors of H form the eigenvectors of the metric M, which are also the principle directions of M. The eigenvalues of M are related to the spacing in each of the principle directions by $\Lambda = 1/h^2$, which start as the absolute value of the eigenvalues of H,

$$M = X \begin{bmatrix} |\Lambda_1| & & \\ & |\Lambda_2| & \\ & & |\Lambda_3| \end{bmatrix} X^T = X \begin{bmatrix} \left(\frac{1}{h_1}\right)^2 & & \\ & \left(\frac{1}{h_2}\right)^2 & \\ & & \left(\frac{1}{h_3}\right)^2 \end{bmatrix} X^T. \quad \text{(B5)}$$

The aspect ratio of M is limited by controlling the two smallest eigenvalues to be $(1/$`adapt_max_anisotropy`$)^2$ times the largest. The eigenvalues of M are scaled so that the largest eigenvalue is $(h_o^*(h/h_o))^{-2}$, where h_o^* is an estimate of the current isotropic grid size given by `adapt_current_h_method`. The gradation of the M is limited with `adapt_gradation` if `adapt_gradation` is positive. If `adapt_complexity` is positive, the complexity (an integral measure of the adapted grid size) of M is scaled to match the requested `adapt_complexity`. Finally, if `adapt_min_edge_length` or `adapt_max_edge_length` are positive, the eigenvalues are limited to bound the spacing of the metric.

```
&adapt_metric_construction
  adapt_hessian_key               = 'mach'
  adapt_hessian_method            = 'lsq'
  adapt_max_anisotropy            = 1.0e6
  adapt_max_edge_growth           = 2.0
  adapt_max_edge_length           = -1.0
  adapt_min_edge_length           = -1.0
  adapt_output_tolerance          = -0.5
  adapt_complexity                = -1.0
  adapt_gradation                 = -1.0
  adapt_error_estimation          = 'embed'
  adapt_statistics                = 'median'
  adapt_exponent                  = 0.2
  adapt_feature_scalar_key        = 'density'
  adapt_feature_scalar_form       = 'delta'
  adapt_feature_length_exp        = 0.5
  adapt_intersect_metric_in_time  = .false.
  adapt_metric_from_file          = ''
  adapt_export_metric             = .false.
  adapt_twod                      = .false.
  adapt_verbose                   = .false.
  adapt_export_feature_scalar_key = 'none'
  adapt_visualize_metric          = 'none'
  adapt_current_h_method          = 'edge'
  adapt_current_h_gradation       = 1.5
/
```

`adapt_hessian_key = 'mach'`

This variable is used to define anisotropic Hessian,

`'mach'` is Mach number.

`'pressure'` is pressure.

`'entropy'` is entropy.

`'temp'` is temperature.

`'density'` is density.

`'vorticity-magnitude'` is the magnitude of the vorticity vector.

`adapt_hessian_method = 'lsq'`

This is the mathematical method used to recover the Hessian,

`'lsq'` applies a least-squares gradient calculation twice. First it computes gradients via least-squares. Then the Hessian is computed by a second application of least-squares to the reconstructed gradient.

`'green'` use a Green variational approach, see Alauzet and Loseille [108] for details.

`'kexact'` reconstructs the Hessian with a k-exact approach. See Barth [109] for details.

`'grad'` is volume-averaged element-based gradients, applied twice.

`'mesh'` implies the metric of the current grid for use in testing grid adaptation mechanics or maintaining the current anisotropy.

`adapt_max_anisotropy = 1.0e6`

This is the upper limit of the largest to smallest spacing in the metric.

`adapt_max_edge_growth = 2.0`

This is a limit on the change of isotropic grid size of the metric, where `adapt_max_edge_growth` is g and the change in isotropic grid size between the next and current grids is h/h_0. The feature-based metric is limited to $h/h_0 < g$ and the output-based metric is limited to $1/g < h/h_0 < g$. A value of 1 would constrain feature-based metric to refinement only. This setting limits the change in grid size for subsequent output-adapted grids and can be increased for more benign simulations or near output-adapted grid convergence.

`adapt_max_edge_length = -1.0`

This sets a maximum allowable spacing of the metric. It is a grid- and problem-dependent value and should be expressed in grid units. A negative value is unlimited.

`adapt_min_edge_length = -1.0`

This sets a minimum allowable spacing of the metric. It is a grid/problem dependent value and should be expressed in grid units. A negative value is unlimited.

`adapt_output_tolerance = -0.5`

This is the error request for output-based adaptation and the scaling of the scalar term for feature-based adaptation. Feature-based adaptation requires a positive number. Output-based adaptation can be negative to indicate a relative error reduction or positive to indicate an absolute error request. It is difficult to choose a good value for this tolerance, see `adapt_complexity` for a more intuitive way to request the adapted grid size.

`adapt_complexity = -1.0`

This is the target complexity for the metric. The complexity of a metric (C) is an integral measure of the density of the metric field,

$$C = \int_{\Omega} \sqrt{\det(M)}\, \mathrm{d}\Omega, \tag{B6}$$

where Ω is the computational domain. This option is intended to allow a user specification of the number of nodes in the adapted grid by uniformly scaling the metric to set its C to `adapt_complexity`. There is a difference between the requested complexity and the number of nodes in the adapted grid, which is examined by Park et al. [110] This is because the requested complexity is a continuous measure, but the metric is discrete. Also, the adaptation mechanics produce a grid that is near, but does not exactly match, the metric. Adjust the requested complexity manually to obtain the desired grid size if the grid is smaller or larger than expected. As detailed in reference [110], the ratio of nodes to complexity approaches two for 3D grids and a half for 2D grids.

`adapt_gradation = -1.0`

This is the allowable gradation of spacing between adjacent metric tensors. [111] A positive value activates gradation control, which should have parameter in the range $[1.1, 2.0]$. A negative value deactivates this option. A smaller value produces a more gradual spatial variation of the spacing request, e.g., a value of 1.0 indicates no variation and would result in a uniform grid.

`adapt_error_estimation = 'embed'`

This selects the method used for error estimation for output-based adaptation,

`'embed'` uses a uniformly refined grid and interpolated solution to estimate the output error. [70, 112] Warning: This option requires a large amount of memory to construct the embedded, uniformly refined grid.

`'single'` uses the current grid and reconstructed solution to estimate the output error. It requires much less memory than `'embed'` but does not provide an improved estimate of the functional. [70]

`'opt-goal'` is the optimal goal-oriented metric. [113] It has the same memory requirements as `'single'` and requires the namelist option `adapt_complexity` to be set. It is only recommended for steady Euler flow.

`adapt_statistics = 'median'`

This selects the method used for determining the intensity I level to equidistribute,

`'median'` uses the median of the error estimate intensity I to convert a relative error tolerance of a negative `adapt_output_tolerance` into an absolute tolerance. This is useful when there are a few, extremely large error estimates that corrupt the average.

`'average'` uses the average of the error estimate intensity I to convert a relative error tolerance of a negative `adapt_output_tolerance` into an absolute tolerance. This is provides better equidistribution when the error estimate is well behaved.

`adapt_exponent = 0.2`

This is the exponent on error estimate to map local error to a change in grid spacing. It is based on an a priori spatial error convergence estimate. [112]

`adapt_feature_scalar_key = 'density'`

This is the "key" flow variable (feature) on which to adapt for feature-based adaptation. It forms the scalar field S at each node based on,

`'mach'` is Mach number.

`'pressure'` is pressure.

`'entropy'` is entropy.

`'temp'` is temperature.

`'density'` is density.

`'vorticity-magnitude'` is the magnitude of the vorticity vector.

`adapt_feature_scalar_form = 'delta'`

This is the method to calculate feature-based refinement indicator from the `adapt_feature_scalar_key` scalar field S. The following terms are computed for each edge in the grid I_e and the nodal adaptation intensity I_n is the maximum edge value for all edges incident to a node,

$$I_n = \max_{e \in n} (I_e) \,, \tag{B7}$$

where e denotes each edge incident to node n. The S subscripts 1 and 2 denote the value at each end point of an edge. The edge terms are,

`'delta'` is the S jump across the edge, $I_e = |S_2 - S_1|$.

`'delta-l'` is the S jump across the edge times the edge length l to the `adapt_feature_length_exp` power p, $I_e = l^p|S_2 - S_1|$.

`'average-l'` is the average S of the two nodes of an edge times the edge length l, $I_e = \frac{l}{2}(S_2 + S_1)$.

`'ratio'` is the ratio of the largest to the smallest S at the edge nodes, $I_e = \max(|S_2/S_1|, |S_1/S_2|)$.

`'max'` is the largest S at the nodes of the edge, $I_e = \max(|S_2|, |S_1|)$.

`'none'` will not use scalar term. It uses a local Hessian normalization term accounting for the sensitivity of the L_p norm, where $p = 2$. [108] This option requires `adapt_complexity` to be set and `adapt_output_tolerance` is ignored.

`adapt_feature_length_exp = 0.5`

This is the exponent for use with `adapt_feature_scalar_form = 'delta-l'`.

`adapt_intersect_metric_in_time = .false.`

This will export a metric intersected over a window that includes each time step of the current run. It is used for fixed-point adaptation of time-accurate simulations. [113]

`adapt_metric_from_file = ''`

This reads the metric from this file instead of computing it when it is blank.

`adapt_export_metric = .false.`

This exports the metric for external grid adaptation tools.

`adapt_twod = .false.`

When `.true.`, compute a 2D metric from a one cell wide 3D grid containing a single layer of spanwise prism elements between symmetry planes. This is required when a 2D adaptation method is selected but the grid is actually a one cell wide 3D grid, because the adjoint does not have a 2D specific mode.

`adapt_verbose = .false.`

When `.true.`, this option reports more information during the error estimation process. This can be helpful for finding the source of NaNs.

`adapt_export_feature_scalar_key = 'none'`

This is the export format for the feature scalar key, intensity I, and new isotropic size request h/h_0. It can be used for visualization or compiling statistics. It may be helpful for gaining insight into a setting for the `adapt_output_tolerance` T that targets a specific feature. The file name root is `[project_rootname]_key`. The formats available are,

`'none'` will not export.

`'cgns'` is CGNS format, requires FUN3D to be configured with a CGNS library.

`'fvuns'` is FieldView C-binary (Fortran stream) format.

`'VTK'` is legacy VTK format.

`'csv'` is a comma separated value format.

`'tec'` is a single image ASCII tecplot format.

`'raw_ascii'` is a single image raw ASCII space separated format

`adapt_visualize_metric = 'none'`

This is the format to export the metric for visualization,

`'none'` will not export.

`'cgns'` is CGNS format, requires FUN3D to be configured with a CGNS library.

`'fvuns'` is FieldView C-binary (Fortran stream) format.

`'VTK'` is legacy VTK format.

`'csv'` is a comma separated value format.

`'raw_ascii'` is a single image raw ASCII space separated format.

`adapt_current_h_method = 'edge'`

This is the method to estimate the current spacing of the grid,

`'edge'` will use the shortest incident edge at a node.

`'implied'` will use the largest eigenvalue of adjacent element implied metrics.

`adapt_current_h_gradation = 1.5`

This limits the gradation of the current spacing estimate by requiring it to be larger than this ratio of its neighbor's spacing estimate.

B.4.39 &adapt_mechanics

This namelist contains variables that control how grid adaptation is performed. More details on general metric-based grid adaptation can be obtained in section 8. This namelist also contains variables to control specialized line adaptation `adapt_library = 'line'` and shock fitting line adaptation `adapt_library = 'sfline'`.

Variables with the `ladapt_` prefix control line adaptation and variables with a `sfline_` prefix control shock fitting line adaptation. These specialized 1D adaptation methods originated in the LAURA code and have a number of requirements that are described in the LAURA User's Manual. [114] The grid origin must be structured and all nodes assigned to a unique line. All lines must have the same number of nodes. If running in parallel, the `partition_lines = .true.` option in the `&partitioning` namelist must be active. The outer boundary (opposite solid walls) can be moved in or out to align with a developing bow shock and the distribution of points across the boundary layer can be adjusted to recover a target cell Reynolds number. If the grid has prisms grown off a solid surfaces then the distribution of prism heights can be adjusted to recover a target cell Reynolds number at the wall while retaining the the original spacing at the top of the prism stack.

Variable names beginning with `sfline_` control how shock fitting meshes are adapted. Currently the shock fitting is only available with line adaptation which is engaged by specifying `adapt_library = 'sfline'`. The variables `ladapt_re_cell`, `ladapt_ep0_grd`, `ladapt_fstr`, and `ladapt_g_limiter` are also active with shock fitting.

```
&adapt_mechanics
  adapt_library              = 'refine/one'
  adapt_project              = ''
  adapt_freezebl             = -1.0
  adapt_cycles               = 2
  adapt_bamg_command         = 'bamg'
  adapt_bamg_geometry_format = 'amdba'
  ladapt_fsh                 = 0.8
  ladapt_fstr                = 0.75
  ladapt_fctrjmp             = 1.05
  ladapt_re_cell             = 1.
  ladapt_beta_grd            = 0.
  ladapt_ep0_grd             = 0.
  ladapt_max_distance        = 1.e+06
  ladapt_jumpflag            = 2
  ladapt_freq                = 0
  ladapt_max                 = 1000
  ladapt_g_limiter           = 0.
  sfadapt_fsbuffr            = 3
  sfadapt_ceqinc              = 0.5
  sfadapt_shkdtct             = 1.0e-01
  sfadapt_fsfrac0             = 1.0e+00
  sfadapt_fsfraci             = 1.0e-01
/
```

adapt_library = 'refine/one'

This is the adaptation library to call from FUN3D. The options are,

`'refine/one'` represents a deprecated workflow where the refine tetrahedral metric-based adaptation is coupled to the flow and adjoint solvers. See Park [70] for a detailed description. This library lacks a 2D specific mode.

`'refine/two'` represents a deprecated workflow where the refine tetrahedral metric-based adaptation is coupled to the flow and adjoint solvers. It is based on `refine/one` with some ideas from Michal and Krakos. [115] This library emulates a 2D capability by adapting a 3D grid with one layer of spanwise prism elements between symmetry planes.

`'meshsim'` is the Simmetrix MeshSim™ adaptation library.

`'bamg'` is the BAMG [4] 2D metric-based adaptation library. The 2D metric and solution files in BAMG format will be exported from a 3D grid with one layer of spanwise prism elements between symmetry planes. The BAMG executable will be run in the ../`Flow` directory by using its native .`ambda` or .`msh` formats, see `adapt_bamg_geometry_format`.

`'line'` is line-based adaptation [114] for structured grids. The user must set `partition_lines = `.`true`. in the `&partitioning` namelist when running in parallel.

`'sfline'` is shock-fitting line-based adaptation for structured grids. The user must set `partition_lines = `.`true`. in the `&partitioning` namelist when running in parallel.

`'interpolate'` will linearly interpolate the `project_rootname` solution to an existing `adapt_project` grid without adaptation by using the approach of Shenoy. [116] This option reuses some of the feature-based adaptation mechanics, which requires `adapt_tolerance_output = 1.0` to be set in the `&adapt_metric_reconstruction` namelist.

adapt_project = ''

This is the project name for exporting the adapted grid and solution. An empty string appends `_R` to the `project_rootname` from the `&project` namelist.

`adapt_freezebl = -1.0`

This prevents modification of the grid within this distance of solid wall boundaries. It is used to preserve an existing boundary layer grid structure and is specified in grid units. A negative value will disable freezing. A distance that equates to a y^+ of approximately 300, the middle of the log-layer is recommended. To reduce size inconsistencies between the frozen and adapted zones, chose a `adapt_freezebl` where boundary layer element aspect ratio matches the `adapt_max_anisotropy` in the `&adapt_metric_construction` namelist. For more discussion on this technique, see Park and Carlson. [10] This option is only used by `adapt_library = 'refine/one'`.

`adapt_cycles = 2`

This is the number of adaptation passes. Choosing more cycles will produce a grid that better matches the metric, but can increase the time required for adaptation. This option is only used by `adapt_library = 'refine/one'`.

`adapt_bamg_command = 'bamg'`

This is the system command to execute BAMG. It may include the full path or command line arguments.

`adapt_bamg_geometry_format = 'amdba'`

BAMG geometry file format

`'amdba'` specifies `-b [project_rootname].ambda` as the BAMG geometry source. This will spline current boundary nodes to form the geometry of the domain boundary. It is approximate, but less likely to fail than `.msh` file boundary reconstruction.

`'msh'` specifies `-b [project_rootname].msh` as the BAMG geometry source. This will access the original geometry `.msh` file to define the domain boundary, but BAMG may have problems with boundary reconstruction.

`ladapt_fsh = 0.8`

This is the fraction of the distance between the body and the opposing boundary along a line of nodes where the captured shock is situated.

`ladapt_fstr = 0.75`

This is the fraction of edges along a line that are intended to resolve the boundary layer.

`ladapt_fctrjmp = 1.05`

This is the property ratio used to detect the shock when marching from the freestream toward the body. It is assumed the flow above the shock is uniform and the property ratios across edges along the line remain equal to one until the shock is encountered.

`ladapt_re_cell = 1.`

This is the target cell Reynolds number based on the speed of sound used to define the edge length Δn of the first edge leaving the wall. $re_{cell} = \rho \Delta n c / \mu$.

`ladapt_beta_grd = 0.`

This is an exponential grid distribution parameter. Any value greater than 1 will override adaptation. If it is used to override adaptation to local flow, the recommended value is 1.15.

`ladapt_ep0_grd = 0.`

This is a grid clustering factor to pull nodes into the captured shock. A minimum value of 0 produces no clustering. A maximum value of 6.25 produces greatest clustering. Large values can produce negative volumes. If a negative volume is reported, then reduce the magnitude of this parameter.

`ladapt_max_distance = 1.e+06`

This is the maximum distance in grid units the outer boundary can be moved away from the body. This parameter is useful when adapting to the shock in the wake, where the adapting grid may become excessively skewed. This value then effectively defines the maximum length of the wake domain.

`ladapt_jumpflag = 2`

This is an integer flag used to select the method of shock detection,

`'0'` is no movement of outer boundary. Resolution in the boundary layer is adjusted to recover target `ladapt_re_cell`.

`'1'` uses pressure as sensing parameter.

`'2'` uses density as sensing parameter.

`'3'` uses temperature as sensing parameter.

`'4'` scales all edges along the line by a factor equal to `ladapt_fctrjmp`.

`ladapt_freq = 0`

This is the number of relaxation steps between calls to line adaptation. The value 0 prevents line adaptation.

`ladapt_max = 1000`

This is the maximum number of calls to line adaptation permitted.

`ladapt_g_limiter = 0.`

This parameter insures a minimum mesh size does not get too big on a line and cause local skewing. It must be a positive number to engage.

`sfadapt_fsbuffr = 3`

This is the number of buffer nodes between the freestream boundary and the fitted shock. Zero buffer nodes make the freestream boundary the shock fitting surface, three buffer nodes moves the shock fitting surface three nodes into the interior of the computational domain relative to the freestream boundary.

`sfadapt_ceqinc = 0.5`

This is the shock fitting compatibility equation influence coefficient. A value of 0.0 means that shock fitting is controlled by the continuity equal to the compatibility equation, a value of 1.0 means that the shock fitting is controlled by momentum equation compatibility equal to the compatibility equation, and a value of 0.5 means that shock fitting is equally controlled by the continuity and momentum compatibility equations.

`sfadapt_shkdtct = 1.0e-01`

This is the shock fitting shock-boundary interaction detector coefficient. This parameter controls when the shock is considered to be interacting with the shock fitting boundary nodes and determines when the boundary begins to be fitted to the shock. The value is one minus the local relative density jump below which the shock is not considered to be interacting with the boundary. Increasing this parameter decreases the sensitivity of the sensor and decreasing this parameter increases the sensitivity of the sensor.

`sfadapt_fsfrac0 = 1.0e+00`

This is the shock fitting initial freestream velocity boundary velocity fraction. When the bow shock has not yet reached the freestream boundary, the shock fitting equations are not valid. However, the code allows the freestream boundary to initially move towards the body at some fraction of the freestream boundary velocity. A value of 0.0 freezes the freestream boundary until the shock reaches it, a value of 1.0 moves the freestream boundary at the freestream velocity until the shock reaches it, and a value in the range $(0.0, 1.0)$ moves the freestream boundary towards the body at freestream total velocity*`sfadapt_fsfrac0`.

`sfadapt_fsfraci = 1.0e-01`

This is the shock fitting interaction freestream velocity boundary velocity fraction. When the bow shock has been determined to be interacting with the freestream boundary, the shock fitting equations are not valid all along the shock. However, the code allows the initial interaction speed of the shock with the freestream boundary to be scaled back at some fraction of the freestream boundary and/or shock velocity. A value of 0.1 constrains the freestream boundary to move at a fraction of the freestream velocity and a value in the range $(0.0, 1.0)$ moves the freestream boundary towards/away from the body at freestream total velocity*`sfadapt_fsfraci`.

B.4.40 &massoud_output

This namelist controls the output of files for interaction with the MDO packages (e.g., design, aeroelastics). In a design setting, these files contain the information necessary to parameterize the surface grid(s). The command-line option `--write_aero_loads_to_file` is required to output the aeroelastics file `[project_rootname]_ddfdrive_bodyN.dat` and the command line option `--write_massoud_file` is required to output the design parameterization file `[project_rootname]_massoud_bodyN.dat` for each of the `N` body groups present.

```
&massoud_output
  n_bodies                      = 0
  nbndry(:)                     = 0
  boundary_list(:)              = ''
  output_initial_state          = .false.
  massoud_output_freq           = -1
  massoud_file_format           = 'ascii'
  massoud_use_initial_coords    = .false.
  aero_loads_output_freq        = -1
  aero_loads_file_format        = 'ascii'
  include_time_info             = .true.
  aero_loads_use_initial_coords = .false.
  aero_loads_dynamic_pressure   = 1.0
  output_transform(1,1:4)       = 1.0, 0.0, 0.0, 0.0
  output_transform(2,1:4)       = 0.0, 1.0, 0.0, 0.0
  output_transform(3,1:4)       = 0.0, 0.0, 1.0, 0.0
  output_transform(4,1:4)       = 0.0, 0.0, 0.0, 1.0
  output_scale_factor           = 1.0
  input_transform(1,1:4)        = 1.0, 0.0, 0.0, 0.0
  input_transform(2,1:4)        = 0.0, 1.0, 0.0, 0.0
  input_transform(3,1:4)        = 0.0, 0.0, 1.0, 0.0
  input_transform(4,1:4)        = 0.0, 0.0, 0.0, 1.0
  input_scale_factor            = 1.0
/
```

`n_bodies = 0`

This is the number of user-defined bodies. For moving-grid cases, these bodies are typically the same as those defined as moving bodies, but that need not be the case.

`nbndry(:) = 0`

This is the number of boundary patches listed for a given body.

`boundary_list(:) = ''`

This is a list of boundary patch numbers for a given body. Commas and dashes can be used to specify ranges, i.e., `'1,2,5-7'`.

`output_initial_state = .false.`

When `.true.`, this causes requested massoud or aero loads data to be written for the initial state of the solution in the current run.

`massoud_output_freq = -1`

This is the iteration frequency of massoud output, where the special value `-1` corresponds to once at the end of a successful run.

`massoud_file_format = 'ascii'`

This is the format of the massoud file; the alternate choice is `'stream'` (C binary).

`'ascii'` is ASCII file format

`'stream'` is Fortran stream (C binary) format

`massoud_use_initial_coords = .false.`

Write the massoud file for the x, y, z surface coordinates at t=0. Otherwise, use current x, y, z surface coordinates.

`aero_loads_output_freq = -1`

This is the iteration frequency of aerodynamic loads output, where the special value `-1` corresponds to once at the end of a successful run.

`aero_loads_file_format = 'ascii'`

This is the format of the aerodynamic loads file; the alternate choice is `'stream'` (C binary).

`'ascii'` is ASCII file format

`'stream'` is Fortran stream (C binary) format

`include_time_info = .true.`

Write simulation time and strand info to ASCII Tecplot™ file(s). Including time info in the files makes animation within Tecplot™ very simple.

`aero_loads_use_initial_coords = .false.`

Write the current aerodynamic loads mapped to the x, y, z surface coordinates at t=0. Otherwise, use current x, y, z surface coordinates. This option is only relevant if the grid is moved or changed during the solution process.

`aero_loads_dynamic_pressure = 1.0`

The dynamic pressure used to convert force coefficients into forces; the default value leaves the output in coefficient form. Note that the input variable `output_scale_factor` separately handles scaling of coordinate values, so care must be exercised to insure appropriate dimensional forces if both are used in combination.

`output_transform(1,1:4) = 1.0, 0.0, 0.0, 0.0`

This is a user-specified transform matrix to allow output of aero loads in a custom coordinate system; typically used to output aero loads in an FEM/CSD coordinate system that differs from the CFD coordinate system. Note, at this point in time, this transform is applied to ALL bodies that are defined in this namelist. Note: the transform matrix should NOT include a scaling factor (e.g. inches to meters); any required scale factor is input separately.

`output_scale_factor = 1.0`

Allows a scaling of the output x,y,z coordinates (e.g. meters to inches). Scaling is applied as a multiplicative factor.

`input_transform(1,1:4) = 1.0, 0.0, 0.0, 0.0`

This is a user-specified transform matrix to allow input of a new surface mesh that is defined in a custom coordinate system; typically used to read in a displaced surface defined in an FEM/CSD coordinate for use in the CFD coordinate system. Note, at this point in time, this transform is applied to ALL bodies that are defined in this namelist. Note: the transform matrix should NOT include a scaling factor (e.g. inches to meters); any required scale factor is input separately.

`input_scale_factor = 1.0`

Allows a scaling of the input x,y,z coordinates (e.g. inches to meters). Scaling is applied as a multiplicative factor.

B.4.41 &sonic_boom

This namelist specifies how near-field pressures are extracted from FUN3D for comparison to wind tunnel measurements, atmospheric propagation to the ground by another code, or for boom related adjoint cost functions. When `nsignals` is greater than zero, the pressure signature is output as a Tecplot™ file. The number of points in this output file is determined by the number of element faces intersected by each user-specified ray, and will span the x-extent of the entire mesh at that ray location.

The rays are rotated about (`x_cor`,`z_cor`) by `angle_of_attack` when the variable `rotate_ray_by_angle_of_attack` is true.

This namelist is also used with the sonic boom adjoint cost functions `boom_targ` (section 9.2.7) and `sboom` (section 9.2.8). To form the cost function the ../`rubber.data` file is required for the flow and adjoint solvers. See section 6.3 for details on the minimum inputs required for specifying the adjoint cost function. To compute the value of the objective function in the flow solver, the `--design_run` command line option is required.

```
&sonic_boom
  nsignals                      = 0
  y_ray(:)                      = 0.001
  z_ray(:)                      = 0.0
  x_cor                         = 0.0
  z_cor                         = 0.0
  rotate_ray_by_angle_of_attack = .true.
  npoints                       = 1000
  ray_x_limit_method            = 'local'
  x_lower_bound                 = -1.e20
  x_upper_bound                 = 1.e20
  dp_pinf                       = .true.
  p_pinf                        = .false.
  weight                        = .false.
/
```

`nsignals = 0`

This is the total number of signal rays.

`y_ray(:) = 0.001`

This is the y value of each ray before `angle_of_attack` rotation. It is dimensioned 1 to `nsignals`.

`z_ray(:) = 0.0`

This is the z value of each ray before `angle_of_attack` rotation. It is dimensioned 1 to `nsignals`.

`x_cor = 0.0`

This is the x center of `angle_of_attack` rotation.

`z_cor = 0.0`

This is the z center of `angle_of_attack` rotation

`rotate_ray_by_angle_of_attack = .true.`

When `.true.`, this will rotate the rays by `angle_of_attack` about `x_cor` and `z_cor`.

`npoints = 1000`

This is the nominal number of points in each ray that is used to construct the cost function. Any points lying outside the domain will be ignored. The points are linearly spaced between the lower and upper bound of x.

`ray_x_limit_method = 'local'`

This is the method used to determine the x-direction start and end of the ray when forming the objective function. If `x_lower_bound` and/or `x_upper_bound` is specified, then `ray_x_limit_method` must be set to 'explicit'.

`‘local’` sets each ray limit independently by computing the x min and x max of all grid cells intersected by the ray.

`‘explicit’` explicitly sets the x min and x max of all rays with the `x_lower_bound` and `x_upper_bound` namelist variables. Only valid for adjoint cost function `boom_targ`.

`x_lower_bound = -1.e20`

This is the explicit x lower bound for `ray_x_limit_method='explicit'` in the cost function definition.

`x_upper_bound = 1.e20`

This is the explicit x upper bound for `ray_x_limit_method='explicit'` in the cost function definition.

`dp_pinf = .true.`

When `.true.`, this will include normalized delta pressure $(p - p_\infty)/p_\infty$ in tecplot output.

`p_pinf = .false.`

When `.true.`, this will include normalized pressure $(p)/p_\infty$ in tecplot output.

`weight = .false.`

When `.true.`, this will include a weight of one in tecplot output for use in setting up a near-body target pressure design.

B.4.42 &sboom

This namelist contains variables that specify the inputs required to execute the sBOOM library. See section 9.2.8 for details.

```
&sboom
  alt                   = 45000.0
  hg                    = 0.0
  headangle             = 0.0
  climbangle            = 0.0
  dmdt                  = 0.0
  turnrate              = 0.0
  climbrate             = 0.0
  rs                    = 500.0
  signum                = 10000
  zeronum               = 1200
  tol2                  = 1.e-6
  nonlinear             = 1
  thermoviscous         = 1
  relaxation            = 1
  initialtimestep       = 0.01
  refl                  = 1.9
  outflag               = 0
  numouts               = 1000
  tol                   = 0.005
  inputininches         = 0
  adjmode               = 1
  runmode               = 1
  objmode               = 1
  nazimuths             = 1
  phi(:)                = 0.0
  targetdbas(:)         = 56.0
  trajinfo(:)           = 0.0
  createtarget          = 0
  lowerbound            = -1000.0
  upperbound            = -1000.0
  lappass               = 500
  target_numpts(:)      = 0
  target_xx(:,:)        = 0.0
  target_dpress(:,:)    = 0.0
  tflag                 = 0
  ntalt                 = 0
  ztalt(:)              = 0.0
  talt(:)               = 0.0
  windflag              = 0
  nwindx                = 0
  zwindx(:)          = 0.0
  windx(:)           = 0.0
  nwindy             = 0
  zwindy(:)          = 0.0
  windy(:)           = 0.0
  rflag              = 0
  nhalt              = 0
  zrh(:)             = 0.0
  rh(:)              = 0.0
  bodylen            = 127.0
  reg                = 1.e-4
  regr               = 1.e-4
  lbd                = 0.99
  ubd                = 1.0
  robust_var_list    = ''
/
```

`alt = 45000.0`

This is the cruise altitude of the full-scale vehicle in feet.

`hg = 0.0`

This is the height of the ground in feet.

`headangle = 0.0`

This is the vehicle heading angle in degrees. A 180 heading would mean away from the x-axis, a 90 heading would mean away from the y-axis. This option is only relevant when winds are specified.

`climbangle = 0.0`

The is the vehicle climb angle in degrees.

`dmdt = 0.0`

This is the acceleration of the vehicle in 1/second (Mach number per second).

`turnrate = 0.0`

This is the turning rate of the vehicle in degrees per second.

`climbrate = 0.0`

This is the climb rate of the vehicle in degrees per second.

`rs = 500.0`

This is the off-body distance in full-scale feet. This full-scale distance should match the location in grid units used to define `y_ray` and `z_ray` in `&sonic_boom`.

`signum = 10000`

This is the number of points representing the off-body waveform for propagation. Increasing the number points reduces the discretization error of the Burgers equation and increase the execution time of sBOOM.

`zeronum = 1200`

This in the number of points used to zero pad the front part of the signature. These points are required to prevent the initial shock from propagating to the front of the Burgers equation domain and causing a numerical instability. The actual waveform will be sampled using (`signum-zeronum`) points. Typically, 10–20% of `signum` is required.

`tol2 = 1.e-6`

This is the leading zero tolerance of the input waveform. It is used to truncate the initial portion of the off-body waveform before zero padding, where dp/p value are less than this number.

`nonlinear = 1`

This controls solution nonlinearity,

`‘1’` uses cumulative nonlinearity.

`‘0’` does not use cumulative nonlinearity.

`thermoviscous = 1`

This controls the modeling of thermoviscous absorption,

`‘1’` uses cumulative thermoviscous absorption.

`‘0’` does not use thermoviscous absorption.

`relaxation = 1`

This controls the modeling of molecular relaxation,

`‘1’` uses cumulative molecular relaxation.

`‘0’` does not use molecular relaxation.

`initialtimestep = 0.01`

Nondimensional initial step size for propagation. A smaller number prevents a multivalued function, but increases execution time. If you receive a discontinuity error, reduce this value.

`refl = 1.9`

This is the ground reflection factor used to scale ground signatures.

`outflag = 0`

This determines the format of the output, `‘0’` outputs delta pressure in psf as a function of time in ms.

`‘1’` outputs delta pressure divided by freestream pressure as a function of x in feet.

`numouts = 1000`

This is the number of points requested in the ground signature.

`tol = 0.005`

This is the slope tolerance needed for removing zero paddings from the ground signatures. This allows signatures at different azimuthal angles to have the same time axis starting with the initial shock.

`inputininches = 0`

This is the units of the FUN3D grid and geometry to scale the near field signature x for propagation,

`‘0’` for FUN3D grid units in feet.

`‘1’` for FUN3D grid units in inches.

`‘2’` for FUN3D grid units in meters.

`adjmode = 1`

This is the sBOOM simulation mode,

`‘1’` for a primal and adjoint simulation.

`‘0’` for primal simulation only.

`‘2’` for a primal and adjoint simulation with additional atmospheric sensitivities

`runmode = 1`

This is the type of propagation and the class of cost function,

`‘1’` propagates near field dp/p_∞ to ground and adjoint sensitivities of ground based metrics defined by `objmode`. The target pressure is specified with `target_numpts`, `target_dpress`, and `target_xx`.

`‘0’` reverse propagates near field dp/p_∞ to compute equivalent area (when `rs`$<$ (`alt`$-$`hg`)) and directly converts off-body pressures to equivalent area (when `rs`$>$ (`alt`$-$`hg`)). No ground signature or ground-based cost function is computed. See `bodylen`, `reg`, `regr`, `lbd`, and `ubd`. The target area distribution is specified with `target_numpts`, `target_dpress`, and `target_xx`.

`objmode = 1`

This is the cost function definition. The value of `runmode` changes its behavior as follows, '1' for an A-weighted loudness target of $I = (dBA - dBA_t)^2$ when `runmode=1` and an equivalent area target $I = \sum_{i=1}^{N} \frac{1}{2}[Ae(i) - Ae_{target}(i)]^2$ when `runmode=0`.

'2' for an inverse pressure design objective of $I = \sum_{i=1}^{N}[p(i) - p(t,i)]^2$ when `runmode=1` and an equivalent area sum $I = \sum_{i=1}^{N} Ae(i)^2$ when `runmode=0`.

'3' for combined A-weighted loudness and inverse pressure design objective of $I = (dBA - 56.0)^2 + \sum_{i=1}^{N}[p(i) - p(t,i)]^2$ when `runmode=1`.

'4' for an A-weighted loudness objective of $I = dBA$ when `runmode=1`.

`nazimuths = 1`

This is the number of azimuths to propagate. The `nazimuths` must match `nsignals` in `&sonic_boom`.

`phi(:) = 0.0`

This is a `nazimuths` length vector of azimuthal locations in degrees.

`targetdbas(:) = 56.0`

This is the target (dBA_t) at each azimuthal location when `objmode=1`.

`trajinfo(:) = 0.0`

This is the trajectory information input to sBOOM

`createtarget = 0`

This is the source of a ground target signature,

'0' for no ground target signature.

'1' to internally create a ground target by Laplace smoothing. The smoothing is controlled by `lowerbound`, `upperbound`, and `lappass`.

'2' for a user specified target. The target is defined by the `target_numpts`, `target_xx`, and `target_dpress` variables.

`lowerbound = -1000.0`

When `createtarget=1`, this is the lower bound in time (milliseconds) after which the user wants to Laplace smooth the computed signature to form the target. When `runmode=0`, this defines the start of locations in X (feet) where any difference in equivalent area between actual and target equivalent areas will contribute to the cost functional. Outside these bounds, even if the equivalent area does not match the target, it does not contribute to the cost functional.

`upperbound = -1000.0`

When `createtarget=1`, this is the upper bound in time (milliseconds) before which the user wants to Laplace smooth the computed signature to form the target. When `runmode=0`, this defines the end of locations in X (feet) where any difference in equivalent area between actual and target equivalent areas will contribute to the cost functional. Outside these bounds, even if the equivalent area does not match the target, it does not contribute to the cost functional.

`lappass = 500`

The number of Laplace smoothing passes to generate a target ground signature. A higher number increases the smoothness of the target. It is only used for `createtarget=1`.

`target_numpts(:) = 0`

This is the number of points in the `target_xx` and `target_dpress` at each azimuth. It is used for `runmode=1` with `objmode=2,3` or `runmode=0` with `objmode=1`.

`target_xx(:,:) = 0.0`

This is the time (in milliseconds) of the target signature at each azimuth for `runmode=1` or x in feet for `runmode=0`. It is only used for some objective functions, see `objmode`. The first index is azimuthal location and the second index is the target signature point.

`target_dpress(:,:) = 0.0`

This is the delta pressure (in psf) of the target signature at each azimuth for `runmode=1` or the equivalent area target for `runmode=0`. It is only used for some objective functions, see `objmode`. The first index is azimuthal location and the second index is the target signature point.

`tflag = 0`

This controls the source for the atmospheric temperature distribution,

'`0`' for the 1976 U.S. Standard Atmosphere temperature profile.

'`1`' for a temperature profile specified by `ntalt`, `ztalt`, and `talt`.

`ntalt = 0`

This is the number of `ztalt` altitude and `talt` temperature pairs to define the temperature profile.

`ztalt(:) = 0.0`

This is `ntalt` length vector of altitudes (in meters) to specify an atmospheric temperature distribution.

`talt(:) = 0.0`

This is `ntalt` length vector of temperature (in Fahrenheit) to specify an atmospheric temperature distribution.

`windflag = 0`

This controls the source for winds,

'`0`' for no winds.

'`1`' for a wind profile specified by `nwindx`, `windx`, `zwindx`, `nwindy`, `windy`, and `zwindy`.

`nwindx = 0`

This is the number of `zwindx` altitude and `windx` x-wind pairs that define the wind profile.

`zwindx(:) = 0.0`

This is a vector of length `nwindx` of altitude (in meters) to specify a wind profile.

`windx(:) = 0.0`

This is a vector of length `nwindx` of x-wind speeds (in meters/sec) to specify a wind profile.

`nwindy = 0`

This is the number of `zwindy` altitude and `windy` y-wind pairs that define the wind profile.

`zwindy(:) = 0.0`

This is a vector of length `nwindy` of altitude (in meters) to specify a wind profile.

`windy(:) = 0.0`

This is a vector of length `nwindy` of y-wind speeds (in meters/sec) to specify a wind profile.

`rflag = 0`

This controls the source for the atmospheric humidity distribution,

'`0`' for the 1976 U.S. Standard Atmosphere relative humidity profile.

'`1`' for a relative humidity profile specified by `nhalt`, `zrh`, and `rh`.

`nhalt = 0`

This is the number of `zrh` altitude and `rh` relative humidity pairs.

`zrh(:) = 0.0`

This is a vector of length `nhalt` of altitude (in meters) to specify a relative humidity profile.

`rh(:) = 0.0`

This is a vector of length `nhalt` of relative humidity (in percent) to specify a relative humidity profile.

`bodylen = 127.0`

This is the aircraft body length in feet. It is only used for the adjoint of equivalent area matching, `runmode=0` and `adjmode=1`.

`reg = 1.e-4`

This is the thermoviscous absorption regularization parameter. A smaller value could lead to an ill-posed reverse diffusion problem. A higher value increases error. Applicable only when `rs`$<$ (`alt`$-$`hg`). It is only used for equivalent area, `runmode=0`.

`regr = 1.e-4`

This is the molecular relaxation regularization parameter. A smaller value could lead to an ill-posed reverse diffusion problem. A higher value increases error. Applicable only when `rs`$<$ (`alt`$-$`hg`). It is only used for equivalent area, `runmode=0`.

`lbd = 0.99`

This is the under-deviation parameter for reversed equivalent area matching. Equivalent area deviations below a target are generally favorable to deviations above a target. Equivalent area matching cost functions and their sensitivities are only computed when the equivalent area is within the `lbd` and `ubd` limits. It is only used for the adjoint of equivalent area matching, `runmode=0` and `adjmode=1`.

`ubd = 1.0`

This is the over-deviation parameter for reversed equivalent area matching. Equivalent area deviations below a target are generally favorable to deviations above a target. Equivalent area matching cost functions and their sensitivities are only computed when the equivalent area is within the `lbd` and `ubd` limits. It is only used for the adjoint of equivalent area matching, `runmode=0` and `adjmode=1`.

`robust_var_list = ''`

This is a list of uncertain variables to be used during robust optimization. Commas and dashes can be used to specify ranges, i.e., `'1,2,5-7'`.

B.4.43 &equivalent_area

This namelist contains variables that specify the inputs associated with equivalent area-based sonic boom cost functions, which is described in section 9.2.9. The number and order of these inputs should match the equivalent area (Ae) functions appearing in `../rubber.data`.

```
&equivalent_area
  nfunctions                = 0
  nplane(:)                 = 0
  global_scaling_factor(:) = 1.0
  lift_scaling_factor(:)   = 1.0
  off_track_angle(:)       = 0.0
/
```

`nfunctions = 0`

This is the total number of Ae functions, including functions used as objectives and constraints.

`nplane(:) = 0`

This is the total number of cutting planes along x-axis for each Ae function.

`global_scaling_factor(:) = 1.0`

This is the Ae(x) scaling factor for each Ae function.

`lift_scaling_factor(:) = 1.0`

This is the Lift L(x) scaling factor for each Ae function.

`off_track_angle(:) = 0.0`

This is the off-track angle in degrees for each Ae function.

B.4.44 &press_box_function

This namelist contains variables that are required for the `press_box` function, which is described in section 9.2.10. The namelist requires the ../`rubber.data` file for the flow and adjoint solvers. See section 6.3 for details on the minimum inputs required for specifying the adjoint cost function. To compute the functional in the flow solver, the `--design_run` command line option is required.

```
 &press_box_function
   integrand_type = 0
   gid            = 1
   xmin           = -huge(1.0)
   xmax           = huge(1.0)
   ymin           = -huge(1.0)
   ymax           = huge(1.0)
   zmin           = -huge(1.0)
   zmax           = huge(1.0)
/
```

`integrand_type = 0`

This integer indicates the functional form.

'`0`' is the volume integral of pressure squared inside the defined box.

'`1`' is the volume integral of w-momentum inside the defined box.

'`2`' is the volume integral of u-momentum inside the defined box.

'`3`' is the density at the global node with index `gid`. Not admissible for `eqn_type = 'incompressible'`.

'`4`' is the time derivative of pressure at the global node with index `gid`.

'`5`' is the time derivative of density at the global node with index `gid`. Not admissible for `eqn_type = 'incompressible'`.

`gid = 1`

This integer is a global grid point index to be used with `integrand_type` = 3-5.

`xmin = -huge(1.0)`

This real value defines the lower bound in the x-direction for the box that encloses the volume integral.

`xmax = huge(1.0)`

This real value defines the upper bound in the x-direction for the box that encloses the volume integral.

`ymin = -huge(1.0)`

This real value defines the lower bound in the y-direction for the box that encloses the volume integral.

`ymax = huge(1.0)`

This real value defines the upper bound in the y-direction for the box that encloses the volume integral.

`zmin = -huge(1.0)`

This real value defines the lower bound in the z-direction for the box that encloses the volume integral.

`zmax = huge(1.0)`

This real value defines the upper bound in the z-direction for the box that encloses the volume integral.

B.4.45 &pstag_function

This namelist contains variables that are required for the `pstag` function, which is described in section 9.2.5. This namelist requires the `../rubber.data` file for the flow and adjoint solvers. See section 6.3 for details on the minimum inputs required for specifying the adjoint cost function. To compute the functional in the flow solver, the `--design_run` command line option is required.

```
&pstag_function
  slice_orientation = 1
  disk_radius       = 1.0
  x_disk_origin     = 0.0
  y_disk_origin     = 0.0
  z_disk_origin     = 0.0
/
```

`slice_orientation = 1`

This integer represents the orientation of the cutting plane. The acceptable values are 1 (x-plane), 2 (y-plane), and 3 (z-plane).

`disk_radius = 1.0`

This real value is the radius of the disk over which the function is to be evaluated.

`x_disk_origin = 0.0`

This real value is the x-coordinate of the origin of the disk.

`y_disk_origin = 0.0`

This real value is the y-coordinate of the origin of the disk.

`z_disk_origin = 0.0`

This real value is the z-coordinate of the origin of the disk.

B.4.46 &fan_distortion

This namelist specifies input parameters associated with the fan distortion objective function.

```
&fan_distortion
  sector_angle     = 60.0
  slice_angle      = 10.0
  weighting_factor = 'area'
/
```

<u>`sector_angle = 60.0`</u>

This real-valued scalar is the sector angle in degrees.

<u>`slice_angle = 10.0`</u>

This real-valued scalar is the sector increment angle in degrees.

<u>`weighting_factor = 'area'`</u>

This string specifies how to weight contributions to the fan-face averages appearing in the objective function. Valid values are:

`‘area’` use area as weighting factor.

`‘massflow’` use massflow as weighting factor.

B.4.47 &special_parameters

This namelist specifies changes to the discretization to handle elements with large face angles.

```
&special_parameters
  large_angle_fix        = 'off'
  override_bc_limitation = .false.
  distance_chunk_size    = 20000000
  distance_from_file     = ''
  write_slen             = .false.
  read_slen              = .false.
/
```

large_angle_fix = 'off'

Grids with with elements that have adjacent face angles that approach 180 degrees may result in a sudden onset of not a number (NaN). Grids produced by VGRID may contain these elements.

`'off'` uses all elements in viscous flux evaluation. This is a consistent viscous discretization.

`'on'` neglects viscous fluxes in cells containing angles between adjacent faces of 178 degrees or greater. This is an inconsistent discretization, but may allow the calculation of a solution on a grid that is not suitable for the consistent viscous discretization.

override_bc_limitation = .false.

Allow sensitivity analysis for cases with element-based boundary conditions. Users should contact `Fun3D-Support@lists.nasa.gov` to determine if this option can be used for their simulation.

distance_chunk_size = 20000000

Size of the buffer surface boundary for use in the minimum distance determination required for turbulence models.

distance_from_file = ''

When set to a file name, read turbulence model wall distance from this file. Only a single-image `.solb` format is implemented. Values of the wall distance on solid walls will be overwritten internally to an estimate of adjacent volume cell normal spacing.

write_slen = .false.

This option writes the wall distance information to disk, which reduces initialization time when used with `read_slen` to increase throughput for database creation or multiple restarts. The mesh and partitions must be kept constant (one file is written for each partition).

read_slen = .false.

This option reads the wall distance information from disk, see restrictions of `write_slen`.

B.4.48 &partitioning

This namelist controls options related to the domain decomposition required for parallel execution. The load balancing of each partition execution time can be adjusted with the relative weighting (and cost model) controls to improve parallel scaling.

```
&partitioning
  partitioner                = 'parmetis'
  lb_method                  = 'graph'
  topology                   = ''
  partition_lines            = .false.
  override_partition_vector = .false.
  write_partition_vector     = .false.
  metis_numbering_entry      = 17
  imbalance_tol              = 1.01
  edge_weighting             = 1
  imesh_weighting            = 1
  element_weighting          = 0
  tet_cost                   = 10
  pyr_cost                   = 68
  prz_cost                   = 75
  hex_cost                   = 138
  read_partitions            = .false.
  write_partitions           = .false.
  simultaneous_partition_io = 100
/
```

`partitioner = 'parmetis'`

This determines which partitioning package to use. FUN3D must be built against the corresponding external library to enable an option. The available tools are,

`'parmetis'` is the ParMETIS partitioning tool, see section A.13.2 for installation details.

`'metis'` is the METIS partitioning tool distributed with ParMETIS, see section A.13.2 for installation details. METIS can provide improved load balancing versus ParMETIS; however, serial METIS execution can increase memory requirements on the master node and grid processing time. See the related `metis_numbering_entry` variable description.

`'zoltan'` is the Zoltan partitioning tool, see section A.13.3 for installation details.

`lb_method = 'graph'`

This is the Zoltan load-balancing algorithm, requires `partitioner='zoltan'`.

‘graph’ graph partitioning.

‘hypergraph’ hypergraph partitioning. It is not valid for twod_mode = .true..

‘hier’ hierarchical graph partitioning for multicore architectures. This option requires topology to be defined.

‘rcb’ recursive coordinate bisection.

‘rib’ recursive inertial bisection.

‘hsfc’ Hilbert space-filling curve partitioning.

topology = ’’

From the Zoltan documentation: “This comma-separated list of integers describes the topology of the multicore node. For example: ’2,8’ may refer to a dual-socket processor where the socket has 8 cores. ’2,4,6’ may refer to a dual-socket, 4-die, 6-core node. ’16’ would refer to a 16-core node. Zoltan assumes each node (processor) of the multicore machine has the same architecture, and that the MPI process ranks are consecutive on the multicore nodes. In particular, if your parameters imply there are 4 MPI processes on each multicore node, Zoltan will assume that processes 0, 1, 2, and 3 are on the same node. Your system administrator should be able to show you how to ensure that your processes are loaded in this order.” This string has no default value, only applies when partitioner=’zoltan’, and must be set if lb_method=’hier’.

partition_lines = .false.

When .true., this option accounts for implicit lines during partitioning to minimize intersections of lines with partition boundaries. See section 4.4 for more details on implicit lines and how they are specified. Only relevant when partitioner=’parmetis’.

override_partition_vector = .false.

Override the node- (element-) based partition vector with a vector specified from disk. This file should be named [project]_partition.data. The data should be written as Fortran stream. The first entry in the file should be a 4-byte integer stating the number of intended partitions, followed by an 8-byte integer specifying the total number of grid points (elements). The remainder of the file should contain 4-byte integers representing the 1-based partition index for each of the grid points (elements) in the mesh. If a node-based partition is provided, it should be ordered according to the global points in the mesh file being used. If an element-based partition vector is provided, the list of elements corresponds to all of the elements contained in element family 1, followed by all of the elements contained in element family 2, and so forth, up through element family `n` (max of 4 families). The element families are numbered according to the order in which the first element type in each family was encountered while loading the mesh file.

<u>`write_partition_vector = .false.`</u>

After performing domain decomposition, write partition vector to disk.

<u>`metis_numbering_entry = 17`</u>

When `partitioner='metis'`, this integer indicates the location in the `METIS_OPTION_NUMBERING` metis.h header file enumerated type entry used for the 'options' array. The C enumerated type is **0-based**. This value is dependent on the METIS version being used; the default is appropriate for the METIS library that is bundled with ParMETIS v4.0.3.

<u>`imbalance_tol = 1.01`</u>

This specifies the maximum acceptable load imbalance. For example, 1 percent imbalance or less is requested with `imbalance_tol = 1.01`.

<u>`edge_weighting = 1`</u>

This weight expresses the relative importance of the number of adjacent edges to a vertex on the cost of that vertex. It is a relative quantity as compared to `element_weighting` and `imesh_weighting`. Any of these three weights can be zero to deactivate that style of weighting. The cost of the nearest neighbor linear solution scheme of a vertex is proportional to the number of adjacent vertices (number of nonzeros in a matrix row).

<u>`imesh_weighting = 1`</u>

This is the relative importance of component grid (imesh) weighting for a composite overset grid system as compared to the other partitioning weights, `element_weighting` and `edge_weighting`. Any of these three weights can be zero to deactivate that style of weighting. This weight is ignored unless dynamic overset grids are utilized in the simulation. The intent is to load-balance the elasticity solver across the component meshes, which solves linear elasticity equations to determine volume grid deformation.

<u>`element_weighting = 0`</u>

This weight expresses the relative importance of the number of adjacent elements to a vertex on the cost of that vertex. It is a relative quantity as compared to `edge_weighting` and `imesh_weighting`. Any of these three weights can be zero to deactivate that style of weighting. The vertex cost of forming the viscous residual and linearization for the mixed-element implicit scheme scales with the number and relative cost of adjacent elements. A model for the relative cost of element types is constructed with `tet_cost`, `pyr_cost`, `prz_cost`, and `hex_cost`.

<u>`tet_cost = 10`</u>

This is the relative expense of computing fluxes and jacobians on tetrahedral elements that is used to construct an `element_weighting` cost model. The cost is a relative measure as compared to `pyr_cost`, `prz_cost`, and `hex_cost`.

<u>`pyr_cost = 68`</u>

This is the relative expense of computing fluxes and jacobians on pyramid elements that is used to construct an `element_weighting` cost model. The cost is a relative measure as compared to `tet_cost`, `prz_cost`, and `hex_cost`.

<u>`prz_cost = 75`</u>

This is the relative expense of computing fluxes and jacobians on prismatic elements that is used to construct an `element_weighting` cost model. The cost is a relative measure as compared to `tet_cost`, `pyr_cost`, and `hex_cost`.

<u>`hex_cost = 138`</u>

This is the relative expense of computing fluxes and jacobians on hexahedral elements that is used to construct an `element_weighting` cost model. The cost is a relative measure as compared to `tet_cost`, `pyr_cost`, and `prz_cost`.

<u>`read_partitions = .false.`</u>

Load a previously-computed set of grid partitions in from disk, rather than performing conventional partitioning of grid files. This increases throughput for database creation or multiple restarts. The mesh and partitions must be kept constant (one file is written for each partition).

<u>`write_partitions = .false.`</u>

Write partitioned grid data to disk.

<u>`simultaneous_partition_io = 100`</u>

Maximum number of MPI ranks that may perform simultaneous I/O if `read_partitions` and/or `write_partitions` is enabled.

B.4.49 &fwh_acoustic_data

This namelist controls the output of files for interaction with Ffowcs Williams and Hawkings (FWH) [117] acoustic propagation packages such as PSU-WOPWOP [118] or ANOPP2 [119].

```
&fwh_acoustic_data
  fwh_data_freq          = 0
  append_to_prior_data   = .true.
  output_initial_state   = .false.
  n_fwh_bndry            = 0
  fwh_bndry_list         = ''
  geom_time_variation(:) = 'constant'
  data_time_variation(:) = 'aperiodic'
  steps_per_period(:)    = 360
  face_center_data       = .false.
/
```

`fwh_data_freq = 0`

This is the iteration frequency of FWH output, where the special value `-1` corresponds to once at the end of a successful run. This frequency applies to all FWH surfaces.

`append_to_prior_data = .true.`

When `.true.`, this causes FWH data from the current run to be appended to existing FWH files from previous runs. If `.false.` and there are existing output files for the surfaces in the current `fwh_bndry_list`, the data in those files will be discarded and overwritten. This option applies to all FWH surfaces.

`output_initial_state = .false.`

When `.true.`, this causes FWH data to be written for the initial state of the solution in the current run. This option is typically used for periodic cases so that if the solution is run for a multiple of the period, FWH data is written for both the starting and ending points of the period.

`n_fwh_bndry = 0`

This is the number of mesh boundaries for which FWH surface data will be written, and is the number of boundary patches to be specified in `fwh_bndry_list`. When `-1` is given, the number of mesh boundaries are inferred from `fwh_bndry_list`. It can be helpful to set this number explicitly to ensure consistency with the variables `geom_time_variation`, `geom_time_variation`, and `steps_per_period`, where `fwh_bndry_list` is used to size their maximum dimension.

`fwh_bndry_list = ''`

This is a list of boundary patch numbers for which FWH surface data will be written. Commas and dashes can be used to specify ranges, i.e., `'1,2,5-7'`.

`geom_time_variation(:) = 'constant'`

This categorizes the time variation of the *geometry* on which FWH data is output. The most common use case, where all FWH geometries have the same type of time variation, is facilitated by setting the first entry in the array with a string that includes an `_all` suffix as described below. Its maximum dimension is set by `fwh_bndry_list`.

`'constant'` when this FWH surface geometry does not vary with time.

`'periodic'` when this FWH surface geometry is periodic in time

`'aperiodic'` when this FWH surface geometry varies with time, but it is *not* periodic in time.

`'constant_all'` assigns `'constant'` to all FWH surface geometries, when `geom_time_variation(1) = 'constant_all'`.

`'periodic_all'` assigns `'periodic'` to all FWH surface geometries, when `geom_time_variation(1) = 'periodic_all'`.

`'aperiodic_all'` assigns `'aperiodic'` to all FWH surface geometries, when `geom_time_variation(1) = 'aperiodic_all'`.

`data_time_variation(:) = 'aperiodic'`

This categorizes the time variation of the FWH *data* (surface pressure) that is output. The most common use case, where all FWH data have the same type of time variation, is facilitated by setting the first entry in the array with a string that includes an `_all` suffix as described below. Its maximum dimension is set by `fwh_bndry_list`.

`'constant'` when this FWH surface data does not vary with time.

`'periodic'` when this FWH surface data is periodic in time

`'aperiodic'` when this FWH surface data varies with time, but it is *not* periodic in time.

`'constant_all'` assigns `'constant'` to all FWH surface data, when `data_time_variation(1) = 'constant_all'`.

`'periodic_all'` assigns `'periodic'` to all FWH surface data, when `data_time_variation(1) = 'periodic_all'`.

`'aperiodic_all'` assigns `'aperiodic'` to all FWH surface data, when `data_time_variation(1) = 'aperiodic_all'`.

`steps_per_period(:) = 360`

When `geom_time_variation = 'periodic'` or `data_time_variation = 'periodic'`, this indicates the number of time steps in one period. When both `geom_time_variation` and `data_time_variation` are set to `'periodic'` for a particular surface, both geometry and data are assumed to have the same period. The most common use case, where all periodic FWH surfaces have the same period, is facilitated by setting the period of the first FWH surface to a negative value. For example, `steps_per_period(1) = -(steps_per_period for all)`. Its maximum dimension is also set by `fwh_bndry_list`.

`face_center_data = .false.`

When `.true.`, this option provides data at face centers. When `.false.`, the acoustic data is provided at surface grid nodes. Nodal output is the preferred, since it results in smaller files and provides connectivity data that may be required by the FWH code. The option for data at face centers is provided for backwards compatibility with older processes. This option applies affects all output FWH surfaces.

B.4.50 &vortex_generator

This namelist is used to specify parameters related to source terms designed to simulate the effects of vortex generators.

```
&vortex_generator
  number_of_vgs           = 0
  configuration(:)        = 'area_and_height'
  boundary_patch1(:)      = 0
  boundary_patch2(:)      = 0
  planform_area(:)        = 0.0
  height(:)               = 0.0
  calibration_constant(:) = 0.0
  point1_xcoord(:)        = 0.0
  point1_ycoord(:)        = 0.0
  point1_zcoord(:)        = 0.0
  point2_xcoord(:)        = 0.0
  point2_ycoord(:)        = 0.0
  point2_zcoord(:)        = 0.0
  point3_xcoord(:)        = 0.0
  point3_ycoord(:)        = 0.0
  point3_zcoord(:)        = 0.0
  reverse_t(:)            = .false.
  reverse_n(:)            = .false.
/
```

`number_of_vgs = 0`

Specifies the number of user-defined vortex generators (max 1000).

`configuration(:) = 'area_and_height'`

Specifies the input configuration for each vortex generator. If configuration(i) is 'area_and_height', then the ith vortex generator planform will be a symmetric trapezoid with the endpoints of its base defined by the point1 and point2 coordinate inputs in the namelist and its area and height defined by the planform_area(i)and height(i) values. If configuration(i) is 'three_points', then the ith vortex generator will have a triangular planform where the user must supply the coordinates of a third point defining the off-surface tip of the vortex generator geometry. In this case, the planform area and height are derived quantities based on the three input point locations.

`boundary_patch1(:) = 0`

Specifies the boundary patch index for point1 of the ith vortex generator. This patch will be used for projection of the point1 coordinates specified.

`boundary_patch2(:) = 0`

Specifies the boundary patch index for point2 of the ith vortex generator. This patch will be used for projection of the point2 coordinates specified.

`planform_area(:) = 0.0`

Specifies the intended planform area of the ith vortex generator. This input only required if configuration = 'area_and_height'.

`height(:) = 0.0`

Specifies the height of the ith vortex generator. This input only required if configuration = 'area_and_height'.

`calibration_constant(:) = 0.0`

Specifies the calibration constant of the ith vortex generator.

`point1_xcoord(:) = 0.0`

Specifies the x-coordinate of point 1 defining the ith vortex generator location. These coordinates need not be precise; the input coordinates will be projected onto boundary_patch1(i).

`point1_ycoord(:) = 0.0`

Specifies the y-coordinate of point 1 defining the ith vortex generator location. These coordinates need not be precise; the input coordinates will be projected onto boundary_patch1(i).

`point1_zcoord(:) = 0.0`

Specifies the z-coordinate of point 1 defining the ith vortex generator location. These coordinates need not be precise; the input coordinates will be projected onto boundary_patch1(i).

`point2_xcoord(:) = 0.0`

Specifies the x-coordinate of point 2 defining the ith vortex generator location. These coordinates need not be precise; the input coordinates will be projected onto boundary_patch2(i).

`point2_ycoord(:) = 0.0`

Specifies the y-coordinate of point 2 defining the ith vortex generator location. These coordinates need not be precise; the input coordinates will be projected onto boundary_patch2(i).

`point2_zcoord(:) = 0.0`

Specifies the z-coordinate of point 2 defining the ith vortex generator location. These coordinates need not be precise; the input coordinates will be projected onto boundary_patch2(i).

`point3_xcoord(:) = 0.0`

Specifies the x-coordinate of point 3 defining the ith vortex generator location. This input only required if configuration = 'three_points'.

`point3_ycoord(:) = 0.0`

Specifies the y-coordinate of point 3 defining the ith vortex generator location. This input only required if configuration = 'three_points'.

`point3_zcoord(:) = 0.0`

Specifies the z-coordinate of point 3 defining the ith vortex generator location. This input only required if configuration = 'three_points'.

`reverse_t(:) = .false.`

Reverse the assumed direction of the unit vector $\hat{t}$ for the ith vortex generator.

`reverse_n(:) = .false.`

Reverse the assumed direction of the unit vector $\hat{n}$ for the ith vortex generator.

Additional information on the use of vortex generator source terms The implementation of these source terms in FUN3D is based on the heuristic model described in Ref. [120] and previous references cited therein. The approach avoids the need for resolving geometric details of vortex generator devices during mesh generation. Sufficient grid resolution may still be required to convect the simulated effects of the vortex generator downstream as desired. The user must also provide a calibration constant for each simulated vortex generator. This value should be chosen carefully to produce the desired impact on the local flowfield. Vortex generator source terms currently may only be applied to static grid simulations. The source terms are treated fully implicit during the solution procedure.

After developing the desired set of namelist inputs, it is useful to run the solver for a single iteration, requesting boundary output for (at least) the boundaries on which vortex generators are to be placed. The geometry for each vortex generator as determined by FUN3D based on the namelist inputs will be provided in the Tecplot™ file `[project_rootname]_vg_geometry.dat`. The user should visualize the placement of each vortex generator in relation to the boundary patches of the grid to ensure the desired placement.

The user will also be provided with a Tecplot™ file `[project_rootname]_vg_vectors.dat`. This file contains the unit vectors $\hat{b}$, $\hat{t}$, and $\hat{n}$ according to the notation described in Ref. [120]. The user should visualize these vectors to ensure that they are oriented appropriately. The vector $\hat{b}$ is defined uniquely by the local boundary orientation; however, FUN3D attempts to infer the directions of the vectors $\hat{t}$ and $\hat{n}$ based on the freestream direction. If the local flow direction is expected to be substantially different, these vectors may need to be manually reversed using the appropriate namelist inputs.

If desired, the user may also plot the points at which the actual source terms will be computed by 'scatter plotting' the data contained in the Tecplot™ file `[project_rootname]_vg_source_locations.dat`. These locations are determined by the intersections of the vortex generator geometries with edges in the grid. Source terms are computed at each of these locations then scattered to the residual values at either end of the intersecting edge.

An example of the geometry features described above and the local flowfield in the vicinity of simulated trapezoidal and triangular vortex generators near the leading edge of a wing is shown here.

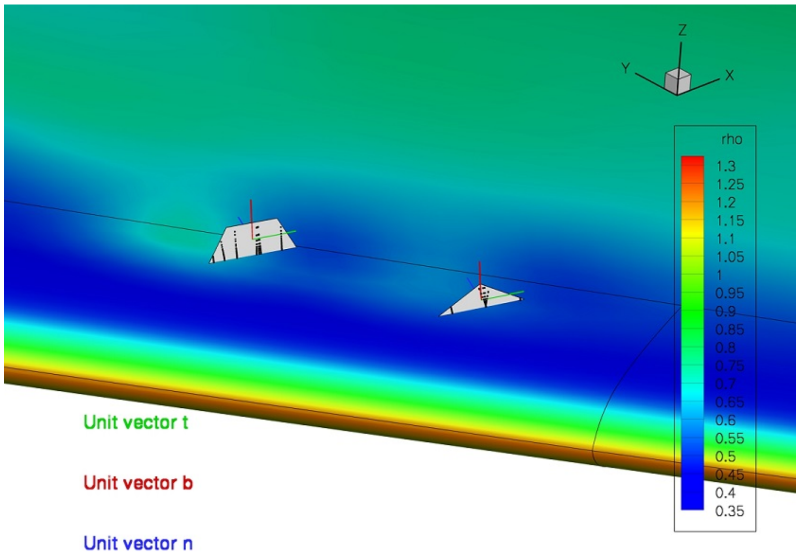


Figure B6: View of trapezoidal and triangular vortex generators placed near the leading edge of a wing geometry. The unit vectors $\hat{b}$, $\hat{t}$, and $\hat{n}$ are shown, as well as the points where the actual source terms will be computed.

B.4.51 &gust_data

This namelist controls the construction of gust profiles for the gust model. The gust functionality introduces an arbitrary perturbation velocity via the field velocity method (FVM) in a stationary grid by prescribing the grid velocity at either all or some of the field grid points without moving the grid. In the current formulation, the gust velocities are oriented with the grid axes. The gusts convect in the x-direction. See Reference [121] for more details about the model.

Several discrete gust profiles are available in the Fun3D code: Gaussian profile, one-minus-cosine profile, sine profile, cosine profile. To define the Gaussian profile, the reference time defines the center of the Gaussian profile. The length parameter defines the half-width. The one-minus-cosine profile is a half-cycle and then a ramp-hold function. This enables the simulation of sharp-edged gusts, or a full-cycle one-minus-cosine if two profiles of opposite sign are placed successively. The sine and cosine profiles allow a continuous gust oscillation through the duration of the solution. The starting location of the gust perturbation defines the point at which perturbation velocities begin, and ahead of which there is no perturbation. This allows one to place the gust origin and the time-to-impact the vehicle arbitrarily.

```
&gust_data
  x0_gust          = 0.0
  ngust_omc        = 0
  u_gust_omc(:)    = 0.0
  v_gust_omc(:)    = 0.0
  w_gust_omc(:)    = 0.0
  tref_gust_omc(:) = 0.0
  l_gust_omc(:)    = 1.0
  ngust_sin        = 0
  u_gust_sin(:)    = 0.0
  v_gust_sin(:)    = 0.0
  w_gust_sin(:)    = 0.0
  tref_gust_sin(:) = 0.0
  l_gust_sin(:)    = 1.0
  ngust_cos        = 0
  u_gust_cos(:)    = 0.0
  v_gust_cos(:)    = 0.0
  w_gust_cos(:)    = 0.0
  tref_gust_cos(:) = 0.0
  l_gust_cos(:)    = 1.0
  ngust_exp        = 0
  u_gust_exp(:)    = 0.0
  v_gust_exp(:)    = 0.0
  w_gust_exp(:)    = 0.0
  tref_gust_exp(:) = 0.0
  l_gust_exp(:)    = 1.0
/
```

`x0_gust = 0.0`

x starting location of all gust perturbations, grid units.

`ngust_omc = 0`

Number of one-minus-cosine profiles.

`u_gust_omc(:) = 0.0`

x velocity magnitude of one-minus-cosine profile, nondimensional; the array index is the one-minus-cosine profile number.

`v_gust_omc(:) = 0.0`

y velocity magnitude of one-minus-cosine profile, nondimensional; the array index is the one-minus-cosine profile number.

`w_gust_omc(:) = 0.0`

z velocity magnitude of one-minus-cosine profile, nondimensional; the array index is the one-minus-cosine profile number.

`tref_gust_omc(:) = 0.0`

Reference time for start of one-minus-cosine profile, nondimensional; the array index is the one-minus-cosine profile number.

`l_gust_omc(:) = 1.0`

Length of one-minus-cosine profile, grid units; the array index is the one-minus-cosine profile number.

`ngust_sin = 0`

Number of sine profiles.

`u_gust_sin(:) = 0.0`

x velocity magnitude of sine profile, nondimensional; the array index is the sine profile number.

`v_gust_sin(:) = 0.0`

y velocity magnitude of sine profile, nondimensional; the array index is the sine profile number.

`w_gust_sin(:) = 0.0`

z velocity magnitude of sine profile, nondimensional; the array index is the sine profile number.

`tref_gust_sin(:) = 0.0`

Reference time for start of sine profile, nondimensional; the array index is the sine profile number.

`l_gust_sin(:) = 1.0`

Wave length of sine profile, grid units; the array index is the sine profile number.

`ngust_cos = 0`

Number of cosine profiles.

`u_gust_cos(:) = 0.0`

x velocity magnitude of cosine profile, nondimensional; the array index is the cosine profile number.

`v_gust_cos(:) = 0.0`

y velocity magnitude of cosine profile, nondimensional; the array index is the cosine profile number.

`w_gust_cos(:) = 0.0`

z velocity magnitude of cosine profile, nondimensional; the array index is the cosine profile number.

`tref_gust_cos(:) = 0.0`

Reference time for start of cosine profile, nondimensional; the array index is the cosine profile number.

`l_gust_cos(:) = 1.0`

Wave length of cosine profile, grid units; the array index is the cosine profile number.

`ngust_exp = 0`

Number of Gaussian profiles.

`u_gust_exp(:) = 0.0`

x velocity magnitude of Gaussian profile, nondimensional; the array index is the Gaussian profile number.

`v_gust_exp(:) = 0.0`

y velocity magnitude of Gaussian profile, nondimensional; the array index is the Gaussian profile number.

`w_gust_exp(:) = 0.0`

z velocity magnitude of Gaussian profile, nondimensional; the array index is the Gaussian profile number.

`tref_gust_exp(:) = 0.0`

Reference time for start of Gaussian profile, nondimensional; the array index is the Gaussian profile number.

`l_gust_exp(:) = 1.0`

Half-height length of Gaussian profile, grid units; the array index is the Gaussian profile number.

B.4.52 &gpu_support

This namelist contains variables that specify inputs relevant to GPU processing. See section 10 for GPU instructions which further explain the use of these options. For entries which are said to "control" a feature, the feature will be enabled if the entry is `.true.`.

```
&gpu_support
  use_cuda                     = .false.
  use_cuda_mpi                 = .false.
  gpus_per_node                = 1
  use_setdevice                = .false.
  use_half_precision           = .false.
  use_cuda_pin                 = .true.
  use_auto_device_select       = .true.
  print_crude_dev_mem          = .false.
  device_list                  = ''
  cuda_start_mps               = .false.
  cuda_mps_verbose             = .false.
  cuda_mps_pipedir             = 'env'
  cuda_mps_logdir              = 'env'
  ignore_cuda_visible_devices = .false.
/
```

`use_cuda = .false.`

If `.true.`, FUN3D will use CUDA kernels where available. If the compiled executable does not provide CUDA support, `use_cuda` will be forced to `.false.`. If the specified options do not have all necessary CUDA kernels available (see section 10.4), the simulation results will not be valid.

`use_cuda_mpi = .false.`

If `.true.`, FUN3D will use CUDA-aware MPI calls where they have been defined (see section 10.9). CUDA-aware MPI must be supported by the MPI implementation being used.

`gpus_per_node = 1`

This specifies the number of GPUs available per node on the current system and is only used in conjunction with `use_setdevice` (see section 10.7). If the compute nodes are not homogeneous with respect to GPUs, this variable and thus `use_setdevice` should not be used.

`use_setdevice = .false.`

If `.true.`, this assigns an MPI rank to local GPU (*rank % gpus_per_node*), where % is the `modulo` operator (see section 10.7).

`use_half_precision = .false.`

If `.true.`, FUN3D will use the IEEE 754 FP16 format to store the LHS matrix for compressible perfect gas simulations (see section 10.13). This is an experimental option which may negatively impact convergence, so use at your own risk. It may speed up the main timestep loop by 1.1-1.2×, depending on the case.

`use_cuda_pin = .true.`

This controls the pinning (page-locking) of host buffers used for host/device transfers by CUDA. If disabled (set `.false.`), this will also disable `use_cuda_unpin`. Disabling this will negatively impact performance, but may be necessary for diagnosing certain memory issues, especially those involving `cuda-memcheck`. See section 10.12 for possible reasons to use this option.

`use_auto_device_select = .true.`

This controls automatic device selection and the starting of the CUDA MPS daemon (if enabled). See sections 10.7 and 10.10.

`print_crude_dev_mem = .false.`

If `.true.`, this prints used and free device memory after each device initialization phase. See section 10.12.

`device_list = ''`

If using automatic device selection, this will instruct FUN3D MPI ranks to select a GPU based on this comma-separated list (see section 10.7). The list uses the same format as `CUDA_VISIBLE_DEVICES`, with the exception that only device numbers are accepted (0, 1, etc). The maximum length is 255 characters.

`cuda_start_mps = .false.`

If using automatic device selection, FUN3D will attempt to start the NVIDIA® CUDA MPS daemon on each node. If `.true.`, this will also kill the daemon when FUN3D shuts down if it was started by the current instance of FUN3D. See sections See sections 10.7 and 10.10.

`cuda_mps_verbose = .false.`

This controls the printing of extra information about CUDA MPS. See section 10.10.

`cuda_mps_pipedir = 'env'`

This sets the environment variable `CUDA_MPS_PIPE_DIRECTORY`. The default ('env') will instruct FUN3D to use the current environment value. If empty (''), the variable will be unset. See section 10.10.

`cuda_mps_logdir = 'env'`

This sets the environment variable `CUDA_MPS_LOG_DIRECTORY`. The default ('env') will instruct FUN3D to use the current environment value. If empty (''), the variable will be unset. See section 10.10.

`ignore_cuda_visible_devices = .false.`

If `.true.`, FUN3D will unset the environment variable `CUDA_VISIBLE_DEVICES` before device initialization. See section 10.7.

B.5 `moving_body.input`

This namelist file is only used for time-dependent, moving grid cases to specify grid motion as a function of time. This file must be used in conjunction with input variable `moving_grid = .true.` in the `&global` namelist of the `fun3d.nml` input file. See section 7 for an overview of moving grid capabilities.

The grid-motion options in FUN3D are fairly generally in order to handle a wide variety of applications. The basic approach is for the user to define one or more boundaries in the grid to be a 'body'. Multiple bodies may be defined. Basic setup for these bodies is established using the `&body_definitions` namelist. A hierarchical relation may be established between multiple bodies to allow the motion of one body (a 'child') to follow the motion of another body (the 'parent'). The top level of this hierarchy is the inertial reference frame, and all motion is ultimately referenced back to this inertial frame. Each body has its own reference frame, and the reference frames of all bodies are assumed to be coincident with the inertial reference frame at t=0.

Having established the basic body definition(s), the user specifies a general descriptor (`motion_driver`) for the mechanism that will drive the motion of the body, and specifies how the mesh is to be moved - by rigid motion or by deformation (or both) - in response to the body motion. Note that mesh deformation requires more CPU time than rigid mesh motion, and is less robust. Mesh deformation may lead to negative cell volumes, at which point the solution is terminated, while rigid motion will preserve positive cell volumes. Thus, rigid mesh motion should be favored over mesh deformation whenever possible. However, there are certain situations where only a deforming mesh is appropriate. For example if the body is aeroelastic, then the mesh must be deformed to fit the deformed body surface. In some situations the potential for a deforming mesh to encounter negative cell volumes can be mitigated by combining deformation with rigid motion. An example of this is the motion of elastic rotor blades, wherein the overall rotational motion of the blades is handled via rigid rotation, but the relatively smaller elastic deflection of the blades is handled via deformation.

The `motion_driver` specification simply provides a notional mechanism for how the body is to be moved; details of this mechanism are then provided by one or more additional namelists. For example, if `motion_driver = 'forced'` then details of how to move the body, perhaps by rotation with a given frequency and amplitude, are specified via the `&forced_motion` namelist. Other options for `motion_driver` - and the required auxiliary namelists to specify the details - are given in the following sections.

By default, boundary output from FUN3D for visualization purposes (see section 5.3.1) is provided in the inertial frame. It is sometimes useful to have this output in a different reference frame, an 'observer' frame. For example, the observer frame might be one attached to a moving body. Specification of an alternate observer frame is handled via the `&observer_motion` namelist.

See the following sections for descriptions of the namelists in this file.

B.5.1 &body_definitions

This namelist specifies which mesh surfaces define the moving bodies. In general, each body may have a different motion. However, there are some fundamental constraints. For example, in a mesh with multiple bodies undergoing different motions, either overset meshes or a deforming mesh would be required. Note that a deforming mesh might well support only small relative motions between the bodies before the mesh becomes invalid (negative cell volumes). For a single rigid mesh, all bodies within the mesh would need to have the same motion.

Body motions are referenced to an inertial-axis system that is aligned with the CFD grid-axis system and remains aligned with the CFD grid-axis system for all time. At $t = 0$, the native body-axis system used by FUN3D is aligned with the grid-axis / inertial system (see Fig. 3). For $t > 0$, these two axis systems will depart as the body moves. Within FUN3D, all motion operations are done between the native body-axis and inertial-axis systems. However, for *output* purposes, the user may define a different body-axis system. For example, trajectory analysis often defines a body-axis system that has the x-axis running in the forward direction, the z-axis running downward, and the y-axis in the span direction. Such a body axis would be related to the native FUN3D body axis by a 180 degree rotation about the y axis. Three methods are available to define this user-specified body-axis system: via a rotation angle and direction; via a set of Euler angles; or via a 4×4 transform matrix. Only one method can be used for any given body.

Additional output files are generated for moving bodies. `PositionBody_N.dat`, `VelocityBody_N.dat`, and `AeroForceMomentBody_N.dat` (with `N` the body number), contain the position (linear and angular), velocity (linear and angular), and aerodynamic forces/moments, respectively, all referenced to the inertial system, as functions of time. The files `PositionBody_N_body_frame.dat`, `VelocityBody_N_body_frame.dat`, and `AeroForceMomentBody_N_body_frame.dat` contain position, velocity and forces/moments relative to the body frame. The velocity and force/moment data in these files are relative to the instantaneous body-axis system, while the position data is relative to the $t = 0$ body-axis system (since the positions relative to the instantaneous body axis system remain unchanged for all time and are not particularly useful). If the user does not specify a body-axis system, output of the body-frame data is relative to the native FUN3D body-axis system. Note that the specification of a user-defined body-axis system is only used for *output* purposes; all *input* data is relative to the native FUN3D body-axis system.

A few words of caution about the Euler angles (yaw, pitch, roll) reported in the `PositionBody_N` files. Euler angles in general are subject to nonuniqueness and singularities, and thus are poor quantities for position description (despite their widespread use). Furthermore, for all but 6DOF simulations, the Euler angles reported in these files are derived from the 4×4 transform matrix that is used to move the body (for 6DOF the Euler angles are provided directly by the 6DOF library). Under the best of circumstances (i.e., ignoring nonuniqueness / singularities), Euler angles can be correctly derived from the transform matrix *if* Euler angles (and only Euler angles) were used to construct the transform in the first place *and* the order of Euler angle application is known. In general the transform matrix used to move a body in FUN3D is generally *not* based on Euler angles, and furthermore, an order of `'ypr'` is assumed when the angles are extracted from the transform. Use discretion when relying on the Euler angles reported in the `PositionBody_N` files.

```
&body_definitions
  n_moving_bodies              = 0
  output_transform             = .false.
  relative_velocity_scaling    = .false.
  relative_vel_twall_adiabatic = .false.
  extra_digits_body_state      = .false.
  dimensional_output           = .false.
  ref_velocity                 = 1117.0
  ref_density                  = 0.002378
  ref_length                   = 1.0
  body_name(:)                 = ''
  parent_name(:)               = ''
  mesh_id(:)                   = 0
  n_defining_bndry(:)          = 0
  defining_bndry(:,:)          = 0
  motion_driver(:)             = 'none'
  mesh_movement(:)             = 'static'
  x_mc(:)                      = xmc
  y_mc(:)                      = ymc
  z_mc(:)                      = zmc
  s_ref(:)                     = sref
  c_ref(:)                     = cref
  b_ref(:)                     = bref
  move_mc(:)                   = 1
  trim_control(:)              = 'none'
  baseline_psi(:)              = 0.0
  steps_per_period(:)          = 0
  body_axes_theta(:)           = 0.0
  body_axes_tx(:)              = 1.0
  body_axes_ty(:)              = 0.0
  body_axes_tz(:)              = 0.0
  body_axes_x0(:)              = xmc
  body_axes_y0(:)              = ymc
  body_axes_z0(:)              = zmc
  euler_angle_order(:)         = 'ypr'
  body_axes_yaw(:)             = 0.0
  body_axes_pitch(:)           = 0.0
  body_axes_roll(:)            = 0.0
  echo_body_axes_transform     = .false.
  body_axes_transform(1,1:4,1) = 1.0, 0.0, 0.0, 0.0
  body_axes_transform(2,1:4,1) = 0.0, 1.0, 0.0, 0.0
  body_axes_transform(3,1:4,1) = 0.0, 0.0, 1.0, 0.0
/
```

`n_moving_bodies = 0`

This is the number of bodies in motion.

`output_transform = .false.`

This outputs the transform matrix to `TransformMatrixBody_N.hst` for body `N` when `.true.`.

`relative_velocity_scaling = .false.`

When `.true.`, the force and moment coefficients are scaled by the velocity of the body relative to the air, rather than scaling with a fixed freestream velocity. Note: this alternate scaling is applied only to the forces and moments in the `AeroForceMomentBody_N` files associated with a moving body, not the standard `[project].forces` and `[project]_hist.dat` files.

`relative_vel_twall_adiabatic = .false.`

When `.true.`, and the wall temperature is set as the (flat plate) adiabatic wall temperature (the default for the compressible path), this will use the *relative* Mach number of the body to compute the temperature, rather than the freestream (reference) Mach. Note that in the unusual case of multiple moving bodies, only the relative velocity of the first body is used to compute the adiabatic wall temperature applied to all bodies. By way of example, if one obtained a solution for a stationary body in a uniform Mach 0.8 flow, and then a solution for a body moving at Mach 0.4 against a uniform Mach 0.4 flow, the solutions will only be identical if the `relative_vel_twall_adiabatic = .true.`. If not `.true.`, there will be differences related to the differing adiabatic wall temperatures imposed. In addition, note that while `relative_vel_twall_adiabatic` affects the solution itself, if one is comparing the reported forces and moments in the `AeroForceMomentBody_N` files, then in addition, `relative_velocity_scaling` must also be set to `.true.`.

`extra_digits_body_state = .false.`

This outputs the body state data using 15 decimal digits rather than 5.

`dimensional_output = .false.`

This outputs the body state data (displacements, velocities, and aero forces) in dimensional form for forced or 6-DOF motions. Use with `ref_velocity`, `ref_density`, and `ref_length` to produce body state output with the desired units.

`ref_velocity = 1117.0`

This is the reference velocity to make aerodynamic forces dimensional, when `dimensional_output = .true`. Note that this should correspond to the sound speed if compressible. The default corresponds to standard sea level speed of sound, in ft/sec.

`ref_density = 0.002378`

This is the reference density to make aerodynamic forces dimensional, when `dimensional_output = .true`. The default corresponds to standard sea level density, in slugs/ft^3.

`ref_length = 1.0`

This is the reference length to make aerodynamic forces dimensional, when `dimensional_output = .true`. The value is set to the ratio of the physical length to the corresponding grid units of the computational mesh. For example, for a body with a characteristic length of 1 meter and the corresponding body length is represented in 1 grid unit, `ref_length = 1.0`.

`body_name(:) = ''`

This is the name used to identify the body; the array index corresponds to body number.

`parent_name(:) = ''`

This is the name of the parent body; the array index corresponds to body number. The motion of a body follows (i.e., is added to) the motion of the parent. When naming the parent, `''` signifies that the parent is the inertial frame. For single or independently-moving bodies, the default `parent_name` should be used.

`mesh_id(:) = 0`

This selects the *component* mesh within the *composite* mesh to associate with this body. For example, if body 2 has `mesh_id = 1`, then motion of component mesh 1 will be governed by the motion of body 2. For non-overset cases, there is only one component, and the terms 'component' and 'composite' are synonymous.

`n_defining_bndry(:) = 0`

This is the number of boundaries that define the body; the array index corresponds to body number.

`defining_bndry(:,:) = 0`

This is a list of `n_defining_bndry` boundaries that define the body; the array index 1 corresponds to the boundary (from 1 to `n_defining_bndry`) defining the body; the array index 2 corresponds to the body number. If boundary lumping is used (section B.4.2), the boundaries must correspond to *lumped* boundaries.

`motion_driver(:) = 'none'`

This is the body motion mechanism; the array index corresponds to body number:

`'none'` is for no motion.

`'forced'` uses forced motion prescribed in `&forced_motion`.

`'surface_file'` uses surface motion prescribed in `&surface_motion_from_file`.

`'motion_file'` uses motion prescribed in `&motion_from_file`.

`'6dof'` computes motion via the "libmo" 6DOF library, which is governed by `&sixdof_motion`.

`'6dof_external'` computes motion via an external 6DOF library

`'aeroelastic'` computes motion via `&aeroelastic_modal_data`, or by coupling with an external FEM.

`'funtofem'` couples the motion with an external source via the FUNtoFEM Python interface [122].

`'custom'` uses `custom_kinematics`; the user supplies a custom transform matrix as a function of time and design variables.

`mesh_movement(:) = 'static'`

This is the type of grid movement associated with body motion; the array index corresponds to body number:

`'static'` no mesh movement.

`'rigid'` rigid mesh movement; all nodes of the mesh rotate/translate in unison with the body.

`'deform'` deforms the mesh locally to accommodate the motion of the solid body.

`'rigid+deform'` mesh undergoes both rigid and deforming motions x_mc(:) = xmc

The is the x-coordinate of moment center at $t = 0$; the array index corresponds to body number.

y_mc(:) = ymc

This is the y-coordinate of moment center at $t = 0$; the array index corresponds to body number.

z_mc(:) = zmc

This is the z-coordinate of moment center at $t = 0$; the array index corresponds to body number.

s_ref(:) = sref

This is the nondimensional reference area for force and moment normalization; the array index corresponds to body number. If not specifically set, the value is set to `area_reference` from the `&force_moment_integ_properties` namelist.

c_ref(:) = cref

This is the nondimensional reference chord length for force and moment normalization; the array index corresponds to body number. If not specifically set, the value is set to `x_moment_length` from the `&force_moment_integ_properties` namelist.

b_ref(:) = bref

This is the nondimensional reference span length for force and moment normalization; the array index corresponds to body number. If not specifically set, the value is set to `y_moment_length` from the `&force_moment_integ_properties` namelist.

move_mc(:) = 1

This controls the movement of the moment center; the array index corresponds to body number.

'0' leave moment center fixed in space

'1:' move moment center following the body number indicated by the value of `move_mc(body)` (applicable only for rigid-body motion of body `move_mc(body)`)

trim_control(:) = 'none'

This controls whether trim as applied to the body; the array index corresponds to body number.

'none' no trim control applied;

'design' trim control applied via design variables;

`baseline_psi(:) = 0.0`

This is the starting azimuth for trim when trim is used as a design variable; the array index corresponds to body number.

`steps_per_period(:) = 0`

This is the number of steps to define a trim period when trim is used as a design variable; the array index corresponds to body number.

`body_axes_theta(:) = 0.0`

This is the body axes rotation angle (in degrees). A positive rotation is applied by the right hand rule with the thumb pointing in direction of the rotation axis. Used in conjunction with `body_axes_tx, body_axes_ty, body_axes_tz`

`body_axes_tx(:) = 1.0`

This is the x-component of the body axes rotation unit vector. Used in conjunction with `body_axes_theta`

`body_axes_ty(:) = 0.0`

This is the y-component of the body axes rotation unit vector. Used in conjunction with `body_axes_theta`

`body_axes_tz(:) = 0.0`

This is the z-component of the body axes rotation unit vector. Used in conjunction with `body_axes_theta`

`body_axes_x0(:) = xmc`

This is the x-component of the body axes center

`body_axes_y0(:) = ymc`

This is the y-component of the body axes center

`body_axes_z0(:) = zmc`

This is the z-component of the body axes center

`euler_angle_order(:) = 'ypr'`

This is the order of application of the Euler angles; each successive transform is applied to the axis system resulting from the previous transform. There are six possibilities:

`'ypr'` first yaw about the z-axis, then pitch about the y'-axis, then roll about the x" axis (primes denote new axes after previous rotation(s)). Right hand rule applies to rotations.

`‘rpy’` first roll about the x-axis, then pitch about the y’-axis, then yaw about the z” axis

`‘pyr’` first pitch about the y-axis, then yaw about the z’-axis, then roll about the x” axis

`‘ryp’` first roll about the x-axis, then yaw about the z’-axis, then pitch about the y” axis

`‘pry’` first pitch about the y-axis, then roll about the x’-axis, then yaw about the z” axis

`‘yrp’` first yaw about the z-axis, then roll about the x’-axis, then pitch about the y” axis

`body_axes_yaw(:) = 0.0`

This is the yaw angle of the body axes (in degrees).

`body_axes_pitch(:) = 0.0`

This is the pitch angle of the body axes(in degrees).

`body_axes_roll(:) = 0.0`

This is the roll angle of the body axes (in degrees).

`echo_body_axes_transform = .false.`

This is a flag to write the final transform(s) (and inverse) to the screen. Primarily a developer tool, but potentially useful for setup debugging.

`body_axes_transform(1,1:4,1) = 1.0, 0.0, 0.0, 0.0`

This is a 4×4 transform matrix; user must ensure the transform provides rotations about the moment center specified via this namelist.

B.5.2 &forced_motion

```
&forced_motion
  rotate(:)                = 0
  rotation_rate(:)         = 0.0
  rotation_freq(:)         = 0.0
  rotation_phase(:)        = 0.0
  rotation_tphase(:)       = 0.0
  rotation_amplitude(:)    = 0.0
  rotation_origin_x(:)     = 0.0
  rotation_origin_y(:)     = 0.0
  rotation_origin_z(:)     = 0.0
  rotation_vector_x(:)     = 0.0
  rotation_vector_y(:)     = 1.0
  rotation_vector_z(:)     = 0.0
  rotation_start(:)        = 0.0
  rotation_duration(:)     = 1.0e99
  translate(:)             = 0
  translation_rate(:)      = 0.0
  translation_freq(:)      = 0.0
  translation_phase(:)     = 0.0
  translation_tphase(:)    = 0.0
  translation_amplitude(:) = 0.0
  translation_vector_x(:)  = 0.0
  translation_vector_y(:)  = 0.0
  translation_vector_z(:)  = 1.0
  translation_start(:)     = 0.0
  translation_duration(:)  = 1.0e99
/
```

`rotate(:) = 0`

This is the type of rotational motion; the array index corresponds to body number:

'`:-1`' rotate to match Cl target

'`0`' for no rotation.

'`1`' for constant rotation rate, `rotation_rate`.

'`2`' is sinusoidal rotation where $\theta =$ `rotation_amplitude` $\sin(2\pi$ `rotation_freq` $t +$ `rotation_phase` $\pi/180) +$ `rotation_tphase` $\pi/180$ and t is nondimensional.

'`3`' is a square-wave doublet in rotation $\theta =$ `rotation_amplitude` for the first half of the specified `rotation_duration` and $\theta =$ `-rotation_amplitude` for the second half of the specified `rotation_duration` `rotation_rate(:) = 0.0`

This is the nondimensional rotation rate associated with `rotate=1`; the array index corresponds to body number. For `eqn_type = 'compressible'`, the nondimensional rate is $\omega = \omega^* \frac{L^*_{ref}}{a^*_{ref} L_{ref}}$, where ω^* is the dimensional rotation rate, in rad/sec. For `eqn_type = 'incompressible'`, the nondimensional rate is $\omega = \omega^* \frac{L^*_{ref}}{V^*_{ref} L_{ref}}$; see also section 2.

`rotation_freq(:) = 0.0`

This is the nondimensional rotation reduced frequency associated with `rotate=2`; the array index corresponds to body number. For `eqn_type = 'compressible'`, the nondimensional frequency is $f = f^* \frac{L^*_{ref}}{a^*_{ref} L_{ref}}$, where f^* is the dimensional frequency, in Hertz (cycles/sec). For `eqn_type = 'incompressible'`, the nondimensional frequency is $f = f^* \frac{L^*_{ref}}{V^*_{ref} L_{ref}}$; see also section 2.

`rotation_phase(:) = 0.0`

This is the rotation phase shift (in degrees) associated with `rotate=2`; the array index corresponds to body number.

`rotation_tphase(:) = 0.0`

This is the rotation offset (in degrees) associated with `rotate=2 or 3`; the array index corresponds to body number.

`rotation_amplitude(:) = 0.0`

This is the rotation amplitude (in degrees) associated with `rotate=2 or 3`; the array index corresponds to body number.

`rotation_origin_x(:) = 0.0`

This is the x-coordinate of rotation center; the array index corresponds to body number.

`rotation_origin_y(:) = 0.0`

This is the y-coordinate of rotation center; the array index corresponds to body number.

`rotation_origin_z(:) = 0.0`

This is the z-coordinate of rotation center; the array index corresponds to body number.

`rotation_vector_x(:) = 0.0`

This is the x-component of a unit vector along the rotation axis; the array index corresponds to body number.

rotation_vector_y(:) = 1.0

This is the y-component of a unit vector along the rotation axis; the array index corresponds to body number.

rotation_vector_z(:) = 0.0

This is the z-component of a unit vector along the rotation axis; the array index corresponds to body number.

rotation_start(:) = 0.0

This is the nondimensional time at which the rotational motion begins the array index corresponds to body number.

rotation_duration(:) = 1.0e99

This is the nondimensional time over which the rotational motion is active; the array index corresponds to body number. After this time the rotational motion is zeroed out.

translate(:) = 0

This is the type of translational motion; the array index corresponds to body number:

'0' for no translation.

'1' for a constant translation rate, translation_rate.

'2' is sinusoidal translation where displacement = translation_amplitude $\sin(2\pi$ translation_freq $t +$ translation_phase $\pi/180) +$ translation_tphase $\pi/180$ and t is nondimensional.

translation_rate(:) = 0.0

This is the nondimensional translation rate associated with translate= 1; the array index corresponds to body number. For eqn_type = 'compressible', the nondimensional translation rate is $V = V^*/a^*{}_{ref}$, where V^* is the dimensional translation rate (for example, in ft/sec). For eqn_type = 'incompressible', the nondimensional translation rate is $V = V^*/V^*{}_{ref}$; see also section 2.

translation_freq(:) = 0.0

This is the nondimensional translation reduced frequency associated with translate=2; the array index corresponds to body number. For eqn_type = 'compressible', the nondimensional frequency is $f = f^* \frac{L^*{}_{ref}}{a^*{}_{ref} L_{ref}}$, where f^* is the dimensional frequency, in Hertz (cycles/sec). For eqn_type = 'incompressible', the nondimensional frequency is $f = f^* \frac{L^*{}_{ref}}{V^*{}_{ref} L_{ref}}$; see also section 2.

`translation_phase(:) = 0.0`

This is the translation phase shift (in degrees) associated with `translate=` `2`; the array index corresponds to body number.

`translation_tphase(:) = 0.0`

This is the translation offset (in grid units) associated with `translate=2`; the array index corresponds to body number.

`translation_amplitude(:) = 0.0`

This is the translation amplitude (in grid units) associated with `translate=` `2`; the array index corresponds to body number.

`translation_vector_x(:) = 0.0`

This is the x-component of a unit vector along the translation axis; the array index corresponds to body number.

`translation_vector_y(:) = 0.0`

This is the y-component of a unit vector along the translation axis; the array index corresponds to body number.

`translation_vector_z(:) = 1.0`

This is the z-component of a unit vector along the translation axis; the array index corresponds to body number.

`translation_start(:) = 0.0`

This is the nondimensional start time of the translational motion; the array index corresponds to body number.

`translation_duration(:) = 1.0e99`

This is the nondimensional duration of the translational motion; the array index corresponds to body number.

B.5.3 &observer_motion

This namelist specifies motion of an observer as a function of time for boundary animation purposes (see the `&boundary_output_variables` namelist for more details); the animation is output from the point of view of the observer.

```
&observer_motion
  ob_parent_name           = ''
  ob_rotate                = 0
  ob_rotation_rate         = 0.0
  ob_rotation_freq         = 0.0
  ob_rotation_phase        = 0.0
  ob_rotation_tphase       = 0.0
  ob_rotation_amplitude    = 0.0
  ob_rotation_origin_x     = 0.0
  ob_rotation_origin_y     = 0.0
  ob_rotation_origin_z     = 0.0
  ob_rotation_vector_x     = 0.0
  ob_rotation_vector_y     = 1.0
  ob_rotation_vector_z     = 0.0
  ob_translate             = 0
  ob_translation_rate      = 0.0
  ob_translation_freq      = 0.0
  ob_translation_phase     = 0.0
  ob_translation_tphase    = 0.0
  ob_translation_amplitude = 0.0
  ob_translation_vector_x  = 0.0
  ob_translation_vector_y  = 0.0
  ob_translation_vector_z  = 1.0
/
```

`ob_parent_name = ''`

This is the observer parent reference frame. The default indicates the inertial reference frame - the same as when observer motion is not explicitly specified. `ob_parent_name` can alternatively be set to any moving body defined in the `&body_definitions` namelist. For articulated, moving-blade rotorcraft cases, for which `overset_rotor=.true.`, special observer reference frames, fixed to the motion of the rotor hub are available. These special observer frames are activated by setting `ob_parent_name='rotor_hub_N'`, where N is the rotor number.

`ob_rotate = 0`

This is the type of rotational motion of the observer fame, *to be applied relative to the* `ob_parent_name` *reference frame*:

'`0`' for no rotation.

'1' for a constant rotation rate, `ob_rotation_rate`.

'2' is sinusoidal rotation where $\theta =$ `ob_rotation_amplitude` $\sin(2\pi$ `ob_rotation_freq` $t +$ `ob_rotation_phase` $\pi/180)$ and t is nondimensional.

<u>`ob_rotation_rate = 0.0`</u>

This is the nondimensional rotation rate associated with `rotate=1`. For `eqn_type = 'compressible'`, the nondimensional rate is $\omega = \omega^* \frac{L^*{}_{ref}}{a^*{}_{ref} L_{ref}}$, where ω^* is the dimensional rotation rate, in rad/sec. For `eqn_type = 'incompressible'`, the nondimensional rate is $\omega = \omega^* \frac{L^*{}_{ref}}{V^*{}_{ref} L_{ref}}$; see also section 2.

<u>`ob_rotation_freq = 0.0`</u>

This is the nondimensional rotation reduced frequency associated with `rotate=2`.

<u>`ob_rotation_phase = 0.0`</u>

This is the rotation phase shift (in degrees) associated with `rotate=2`.

<u>`ob_rotation_tphase = 0.0`</u>

This is the rotation phase shift (in degrees) applied to transform matrix.

<u>`ob_rotation_amplitude = 0.0`</u>

This is the rotation amplitude (in degrees) associated with `rotate=2`.

<u>`ob_rotation_origin_x = 0.0`</u>

This is the x-coordinate of rotation center.

<u>`ob_rotation_origin_y = 0.0`</u>

This is the y-coordinate of rotation center.

<u>`ob_rotation_origin_z = 0.0`</u>

This is the z-coordinate of rotation center.

<u>`ob_rotation_vector_x = 0.0`</u>

This is the x-component of a unit vector along the rotation axis.

<u>`ob_rotation_vector_y = 1.0`</u>

This is the y-component of a unit vector along the rotation axis.

<u>`ob_rotation_vector_z = 0.0`</u>

This is the z-component of a unit vector along the rotation axis.

<u>`ob_translate = 0`</u>

This is the type of translational motion:

'`0`' for no translation

'`1`' for constant translation rate, `ob_translate_rate`.

'`2`' is sinusoidal translation where displacement = `ob_translation_amplitude` $\sin(2\pi$ `ob_translation_freq` $t +$ `ob_translation_phase` $pi/180)$ and t is nondimensional.

<u>`ob_translation_rate = 0.0`</u>

This is the nondimensional translation rate associated with `translate=1`. For `eqn_type = 'compressible'`, the nondimensional translation rate is $V = V^*/a^*_{ref}$, where V^* is the dimensional translation rate (for example, in ft/sec). For `eqn_type = 'incompressible'`, the nondimensional translation rate is $V = V^*/V^*_{ref}$; see also section 2.

<u>`ob_translation_freq = 0.0`</u>

This is the nondimensional translation reduced frequency associated with `translate=2`. For `eqn_type = 'compressible'`, the nondimensional frequency is $f = f^* \frac{L^*_{ref}}{a^*_{ref} L_{ref}}$, where f^* is the dimensional frequency, in Hertz (cycles/sec). For `eqn_type = 'incompressible'`, the nondimensional frequency is $f = f^* \frac{L^*_{ref}}{V^*_{ref} L_{ref}}$; see also section 2.

<u>`ob_translation_phase = 0.0`</u>

This is the translation phase shift (in degrees) associated with `translate=2`.

<u>`ob_translation_tphase = 0.0`</u>

This is the translation phase shift (in degrees) applied to transform matrix.

<u>`ob_translation_amplitude = 0.0`</u>

This is the translation amplitude (in grid units) associated with `translate=2`.

<u>`ob_translation_vector_x = 0.0`</u>

This is the x-component of a unit vector along the translation axis.

<u>`ob_translation_vector_y = 0.0`</u>

This is the y-component of a unit vector along the translation axis.

<u>`ob_translation_vector_z = 1.0`</u>

This is the z-component of a unit vector along the translation axis.

B.5.4 &motion_from_file

This namelist specifies *rigid* body (and grid) motion via a file containing a 4×4 transform matrix as a function of time. It allows user-defined motion that is more general than the motions available in the `&forced_motion` namelist.

```
&motion_from_file
  n_time_slices_file(:)           = 0
  repeat_time_file(:)             = 1.0e+99
  motion_file(:)                  = ''
  motion_file_type(:)             = 'transform_matrix'
  motion_file_has_static_xform(:) = .false.
/
```

`n_time_slices_file(:) = 0`

This is the number of transforms (at selected points in time) defining the motion of the body; the array index corresponds to body number. All the transforms for a particular body are contained in a single file.

`repeat_time_file(:) = 1.0e+99`

This is the nondimensional time when the motion will repeat; the array index corresponds to body number.

`motion_file(:) = ''`

This is the name of the ASCII file that contains the transform matrices or Euler angles used to set the grid position and orientation of the body for each of the specified instants in time; the array index corresponds to body number. The file format and content differs depending on whether the file contains transform data or Euler-angle data.

Example 1: The following pseudocode illustrates how such a motion file might be created if it contains transform matrices:

```
 do i=1,9
write() text of your choice
end do
loop over time steps
write() simulation_time
write() xcg, ycg, zcg
do i=1,4
write() transform_matrix(i,j), j=1,4)
end do
end time step loop
```

where `simulation_time` is the nondimensional time, and where `xcg`,`ycg`,`zcg` are the coordinates of the center of gravity of the body at that time (which is also the rotation center) in grid units.

Note: the file must have 9 (nine) text lines (can be blank) before the data for the first time step. The initial value of `simulation_time` should be zero; if restarting from a time-accurate solution on a nonmoving grid, or from a steady-state solution on a nonmoving grid, the `restart_read` parameter in the `&code_run_control` namelist should be set to `"on_nohistorykept"`. The initial transform matrix should be the identity matrix.

If you need an example of how this file should appear, run a moving-grid case using forced motion with `output_transform = .true.` in the `&body_definitions` namelist. The resulting transform output files are exactly the same format as the input files associated with `motion_file`.

Example 2: The following pseudocode illustrates how such a motion file might be created if it contains Euler angles:

```
 do i=1,6
write() text of your choice
end do
write() euler_angle_order
loop over time steps
write() simulation_time, xcg, ycg, zcg, yaw, pitch, roll
end time step loop
```

where `yaw`, `pitch`, `roll` are the Euler angles in degrees. `euler_angle_order` denotes the order in which the Euler angles in the file are *applied*; the values in the file are always ordered by `yaw`, `pitch`, `roll` at each nondimensional time step.

Note: the file must have 6 (six) text lines (can be blank) before the character string indicating the euler angle order. The first value of `simulation_time` should be zero, the the initial Euler angles should also be zero. The comment about `restart_read` in Example 1 above also applies here. At each step the grid will be rotated using the *current* Euler angles about the *initial* CG location and then translated by the *difference* between the current CG location and the initial CG location.

`motion_file_type(:) = 'transform_matrix'`

This is the type of motion data specified; the array index corresponds to body number:

`'transform_matrix'` a 4 × 4 matrix that specifies the transform from the body position at time t=0 (i.e., the position in the input grid file) to the current position at time t=T.

`'inverse_transform'` a 4 × 4 matrix that specifies the transform from the current body position (in motion) at time t=T to the body position at time t=0 (the position in the input grid file).

`‘euler_angles’` a set of three Euler angles that specify the rotations that transform the orientation at time t=0 (i.e., the orientation in the input grid file) to the current orientation at time t=T.

`motion_file_has_static_xform(:) = .false.`

This flag is used to indicate the that transforms in the motion file include a static transform as part of the transform at each time step. Not used if `motion_file_type = ’euler_angles’`.

B.5.5 &surface_motion_from_file

This namelist allows the motion of surface grids to be defined in files. Since only the surface is specified, `mesh_movement = 'deform'` must be used to deform the volume mesh to conform to the specified surface. These files must be named `[project_rootname]_bodyN_timestepM.dat`, where N is the body number and M is the time slice index (1 to `n_time_slices`). The files are ASCII Tecplot files with a zone title line that contains "TIME `simulation_time`", where `simulation_time` is nondimensional time. The variables in the file are the values of `x`, `y`, `z`, as well as `id`, where `id` is the global grid number of the surface point.

```
&surface_motion_from_file
  n_time_slices(:) = 0
  repeat_time(:)   = 1.0e+99
/
```

`n_time_slices(:) = 0`

This is the number of equally spaced instants in time (and files describing the shape at those times) defining the motion of the body; the array index corresponds to body number. Each file contains the surface shape at a point in time.

`repeat_time(:) = 1.0e+99`

This is the nondimensional time when the motion will repeat; the array index corresponds to body number.

B.5.6 &sixdof_motion

This namelist provides details of 6-DOF motion simulations. It requires linking to the 6-DOF library, see section A.13.6. NOTE: most data in this namelist is input as *dimensional* data. For 6-DOF motion, the variables `ref_velocity`, `ref_density` and `ref_length` in the namelist `&body_definitions` must also be set, in units consistent with those used in `&sixdof_motion`. Note (important): data are assumed to be in FUN3D body coordinates, which are the FUN3D coordinates at t=0. Note that the assumption of FUN3D coordinates applies to the moments of inertia.

```
&sixdof_motion
  mass(:)                      = 1.0
  cg_x(:)                      = 0.0
  cg_y(:)                      = 0.0
  cg_z(:)                      = 0.0
  i_xx(:)                      = 1.0
  i_yy(:)                      = 1.0
  i_zz(:)                      = 1.0
  i_xy(:)                      = 0.0
  i_xz(:)                      = 0.0
  i_yz(:)                      = 0.0
  body_lin_vel(:,:)            = 0.0
  body_ang_vel(:,:)            = 0.0
  euler_ang(:,:)               = 0.0
  gravity_dir(1:3)             = 0.0, 0.0, -1.0
  gravity_mag                  = 32.2
  n_extforce(:)                = 0
  n_extmoment(:)               = 0
  file_extforce(:,:)           = ''
  file_extmoment(:,:)          = ''
  ignore_x_aeroforce(:)        = .false.
  ignore_y_aeroforce(:)        = .false.
  ignore_z_aeroforce(:)        = .false.
  ignore_x_aeromoment(:)       = .false.
  ignore_y_aeromoment(:)       = .false.
  ignore_z_aeromoment(:)       = .false.
  print_sixdof_summary         = .false.
  use_specified_aero_data      = .false.
  use_restart_mass_properties = .false.
/
```

`mass(:) = 1.0`

This is the mass of the body; the array index corresponds to the body number.

`cg_x(:) = 0.0`

This is the x-coordinate of the center of gravity; the array index corresponds to the body number.

`cg_y(:) = 0.0`

This is the y-coordinate of the center of gravity; the array index corresponds to the body number.

`cg_z(:) = 0.0`

This is the z-coordinate of the center of gravity; the array index corresponds to the body number.

`i_xx(:) = 1.0`

This is the moment of inertia about the x axis as the body rotates about the x axis; the array index corresponds to the body number.

`i_yy(:) = 1.0`

This is the moment of inertia about the y axis as the body rotates about the y axis; the array index corresponds to the body number.

`i_zz(:) = 1.0`

This is the moment of inertia about the z axis as the body rotates about the z axis; the array index corresponds to the body number.

`i_xy(:) = 0.0`

This is the moment of inertia about the x axis as the body rotates about the y axis; the array index corresponds to the body number.

`i_xz(:) = 0.0`

This is the moment of inertia about the x axis as the body rotates about the z axis; the array index corresponds to the body number.

`i_yz(:) = 0.0`

This is the moment of inertia about the y axis as the body rotates about the z axis; the array index corresponds to the body number.

`body_lin_vel(:,:) = 0.0`

These are the components of linear velocity; The first array index (ranging from 1 to 3) corresponds to the x, y, and z components; The second array index corresponds to the body number.

`body_ang_vel(:,:) = 0.0`

These are the components of angular velocity (degrees/sec); The first array index (ranging from 1 to 3) corresponds to the x, y, and z components; The second array index corresponds to the body number.

`euler_ang(:,:) = 0.0`

These are the Euler angles (degrees); The first array (ranging from 1 to 3) corresponds to the rotation angle components; the second array index corresponds to the body number.

`gravity_dir(1:3) = 0.0, 0.0, -1.0`

This is a unit length gravity vector.

`gravity_mag = 32.2`

This is the magnitude of the gravity vector (default units: ft/sec^2).

`n_extforce(:) = 0`

This is the number of imposed external forces, excluding gravity. The array index corresponds to the body number.

`n_extmoment(:) = 0`

This is the number of imposed external moments; the array index corresponds to the body number.

`file_extforce(:,:) = ''`

This file specifies the external forces; the first array (ranging from 1 to `n_extforce`) corresponds to the external force number. The second array index corresponds to the body number.

`file_extmoment(:,:) = ''`

This file specifies the external moments; the first array (ranging from 1 to `n_extforce`) corresponds to the external force number. The second array index corresponds to the body number.

`ignore_x_aeroforce(:) = .false.`

Flag to ignore the influence of the x-component of the aerodynamic force on the 6-DOF motion of the body; the array index corresponds to the body number.

`ignore_y_aeroforce(:) = .false.`

Flag to ignore the influence of the y-component of the aerodynamic force on the 6-DOF motion of the body; the array index corresponds to the body number.

`ignore_z_aeroforce(:) = .false.`

Flag to ignore the influence of the z-component of the aerodynamic force on the 6-DOF motion of the body; the array index corresponds to the body number.

`ignore_x_aeromoment(:) = .false.`

Flag to ignore the influence of the x-component of the aerodynamic moment on the 6-DOF motion of the body; the array index corresponds to the body number.

`ignore_y_aeromoment(:) = .false.`

Flag to ignore the influence of the y-component of the aerodynamic moment on the 6-DOF motion of the body; the array index corresponds to the body number.

`ignore_z_aeromoment(:) = .false.`

Flag to ignore the influence of the z-component of the aerodynamic moment on the 6-DOF motion of the body; the array index corresponds to the body number.

`print_sixdof_summary = .false.`

Print out a summary of the 6-DOF data for each body at each time step, from within the 6-DOF solver itself; useful for verifying that the 6-DOF library is getting the correct data from FUN3D

`use_specified_aero_data = .false.`

Use aero forces and moments read from file(s) instead of the computed values (used for validating 6-DOF library implementation). When this option is used, files must be specified for ALL bodies moving under 6-DOF. For body N, the file name must be named `specified_aero_data_bodyN.dat`

`use_restart_mass_properties = .false.`

Use mass and moments of inertia from the FUN3D restart file instead of those in this namelist; only needed FUN3D has modified the mass and inertia during the simulation.

B.5.7 &aeroelastic_modal_data

This namelist specifies data for modal static and dynamic aeroelastic analysis via time integration of the structural dynamics equations (see for example [123]).

```
&aeroelastic_modal_data
  nfsi_subiters                = 1
  grefl                        = 1.0
  uinf                         = 0.0
  qinf                         = 0.0
  fsi_coord_tol                = 1.e-8
  genforce_include_shear       = .false.
  time_integration_scheme      = 'PC_2ndorder'
  nmode(:)                     = 0
  plot_modes                   = .false.
  gdisp0(:,:)                  = 0.0
  gvel0(:,:)                   = 0.0
  gforce0(:,:)                 = 0.0
  gforce_static(:,:)           = 0.0
  gmass(:,:)                   = 0.0
  freq(:,:)                    = 0.0
  damp(:,:)                    = 0.0
  moddfl(:,:)                  = 0
  moddfl_amp(:,:)              = 0.0
  moddfl_freq(:,:)             = 0.0
  moddfl_t0(:,:)               = 0.0
  moddfl_offset(:,:)           = 0.0
  moddfl_add(:,:)              = 0
  modal_ae_complex_to_perturb = 0
  modal_ae_complex_epsilon     = 1.e-20
  modal_ae_mode_to_perturb     = 1
  modal_ae_body_to_perturb     = 1
  use_modal_deform             = .false.
  modal_ref_amp(:,:)           = 1.0
  write_volume_modes           = .false.
  read_volume_modes            = .false.
  lfd_nfreq                    = 0
  lfd_freq(:)                  = 0.0
  lfd_nmodes_perturbed         = -1
  lfd_modes_perturbed(:)       = 0
  lfd_write_mode_files         = .true.
  lfd_use_existing_mode_files = .false.
/
```

`nfsi_subiters = 1`

This sets the number of fluid-structure subiterations to perform at each time step. Mesh deformation is required for each fluid-structure subiteration, which makes these subiterations expensive in terms of execution time.

`grefl = 1.0`

This is the scaling factor between CFD grid units and the structural dynamics equation units. Note: `grefl` is now a scalar value; versions prior to 13.1 required an array value.

`uinf = 0.0`

This is the freestream velocity, in structural dynamics equation units. Note: `uinf` is now a scalar value; versions prior to 13.1 required an array value.

`qinf = 0.0`

This is the freestream dynamic pressure, in structural dynamics equation units. Note: `qinf` is now a scalar value; versions prior to 13.1 required an array value.

`fsi_coord_tol = 1.e-8`

This specifies the tolerance used to determine whether a point on the fluid side of the FSI interface (defined by one or more boundaries in the grid) has a matching point on the structure side of the interface (defined in the mode-shape files). There must be a 1-1 correspondence, to within this tolerance. If the code stops with an error 'failed to find unique mapping...', try increasing this tolerance.

`genforce_include_shear = .false.`

This includes the shear-stress contribution in the generalized force computation. By default, only the pressure contribution is included in the generalized forces, which is historically common practice.

`time_integration_scheme = 'PC_2ndorder'`

This prescribes the temporal integration scheme for the linear structural dynamics equations.

`'PC_2ndorder'` Uses the traditional second-order accurate predictor-corrector scheme described in Ref. [123] to advance the linear structural dynamics equations. Two 'subiterations' are used, the first being the predictor step and the second the corrector step.

`'BDF_1storder'` Uses a first-order backward differencing scheme (backward Euler) to advance the linear structural dynamics equations. This scheme and all the other BDF schemes are subject to the same the number of flow `subiterations` as used for the flow solver.

`‘BDF_2ndorder’` Uses a second-order backward differencing scheme to advance the linear structural dynamics equations.

`‘BDF_2ndorderOPT’` Uses an optimized second-order backward differencing scheme (BDF2opt in Ref. [95]) to advance the linear structural dynamics equations. This scheme is second-order accurate in time but has an order-of-magnitude lower leading coefficient than standard BDF2.

`‘BDF_3rdorder’` Uses a third-order backward differencing scheme to advance the linear structural dynamics equations.

`nmode(:) = 0`

This is the number of aeroelastic modes used to represent the structural deformation; the array index indicates the body number.

`plot_modes = .false.`

This generates tecplot files of each mode shape added to the body surface. These can be inspected these to insure the validity of input modal surface data.

`gdisp0(:,:) = 0.0`

This is the generalized displacement of a specified mode at the starting time step. It is used to perturb a mode to excite a dynamic response. The first array index indicates the mode number and the second array index indicates the body number.

`gvel0(:,:) = 0.0`

This is the generalized velocity of a specified mode at the starting time step. It is used to perturb a mode to excite a dynamic response. The first array index indicates the mode number and the second array index indicates the body number.

`gforce0(:,:) = 0.0`

This is the generalized force of a specified mode at the starting time step. It is used to perturb a mode to excite a dynamic response. The first array index indicates the mode number and the second array index indicates the body number.

`gforce_static(:,:) = 0.0`

This is the static offset for the generalized force of a specified mode. It is used to add a constant load, such as a gravitational load, to the mode of the structure. The first array index indicates the mode number and the second array index indicates the body number.

gmass(:,:) = 0.0

This is the generalized mass of a specified mode. The first array index indicates the mode number and the second array index indicates the body number.

freq(:,:) = 0.0

This is the frequency of specified mode, in rad/sec. The first array index indicates the mode number and the second array index indicates the body number.

damp(:,:) = 0.0

This is the critical damping ratio (z) of the specified mode. The first array index indicates the mode number and the second array index indicates the body number.

moddfl(:,:) = 0

This is the type of time-varying mode perturbation. The first array index indicates the mode number and the second array index indicates the body number.

'-1' is the modal displacement held fixed at the initial value specified by gdisp0.

'0' is no modal perturbation.

'1' is a harmonic modal oscillation.

'2' is a Gaussian pulse modal deflection.

'3' is a step pulse modal deflection.

'5' specifies simultaneous inputs for reduced order model.

'6' specify modal displacement from python interface.

moddfl_amp(:,:) = 0.0

This is the amplitude of mode perturbation. The first array index indicates the mode number and the second array index indicates the body number.

moddfl_freq(:,:) = 0.0

This is the frequency of mode perturbation. If moddfl=2, it is the Gaussian pulse half-life. The first array index indicates the mode number and the second array index indicates the body number.

moddfl_t0(:,:) = 0.0

If moddfl=1, this is the dimensional time at which the sinusoidal perturbation starts. If moddfl=2, this is the dimensional time of the center of the Gaussian pulse. If `moddfl=3`, this is the start time of a step pulse. The first array index indicates the mode number and the second array index indicates the body number.

`moddfl_offset(:,:) = 0.0`

If `moddfl=1`, this is the mean or offset of the harmonic oscillation. The first array index indicates the mode number and the second array index indicates the body number.

`moddfl_add(:,:) = 0`

This determines how the modal perturbation is applied. The first array index indicates the mode number and the second array index indicates the body number.

'0' replaces the aeroelastic solution with the perturbation.

'1' adds the perturbation to aeroelastic solution

`modal_ae_complex_to_perturb = 0`

This value selects from a list of available modal-aeroelastic related complex perturbations for generating sensitivities. Note: depending on the selected value of `modal_ae_complex_to_perturb`, one may also need to specify a particular mode to perturb and a particular body to perturb via `modal_ae_mode_to_perturb` and `modal_ae_body_to_perturb`, respectively.

'0' No complex-valued perturbation applied

'1' Perturb the freestream dynamic pressure, qinf

'2' Perturb the freestream velocity, uinf

'3' Perturb the structural damping of mode `modal_ae_mode_to_perturb` for body number `modal_ae_body_to_perturb`

'4' Perturb the frequency of mode `modal_ae_mode_to_perturb` for body number `modal_ae_body_to_perturb`

'5' Perturb the generalized mass of mode `modal_ae_mode_to_perturb` for body number `modal_ae_body_to_perturb`

'6' Perturb the generalized velocity of mode `modal_ae_mode_to_perturb` for body number `modal_ae_body_to_perturb`

'7' Perturb the generalized displacement of mode `modal_ae_mode_to_perturb` for body number `modal_ae_body_to_perturb`

`modal_ae_complex_epsilon = 1.e-20`

This is the size of the complex perturbation to apply.

`modal_ae_mode_to_perturb = 1`

This is the mode to be perturbed.

`modal_ae_body_to_perturb = 1`

This is the body with the mode that is to be perturbed.

`use_modal_deform = .false.`

This uses a modal decomposition of the volume mesh deformation. The mesh deformation problem is solved once per mode during the initialization phase of the code to compute volume mesh mode shapes. At each time step the new volume mesh is computed as the sum of the product of modal displacements and volume mode shapes. Note: this option requires additional memory to store the volume mode shapes.

`modal_ref_amp(:,:) = 1.0`

This is the amplitude during initial mesh deformation step of modal decomposition of the volume mesh deformation. The first array index indicates the mode number and the second array index indicates the body number.

`write_volume_modes = .false.`

Write the partitioned volume mode shapes from the modal mesh deformation. Used in conjunction with `read_volume_modes` to avoid recomputing the volume mode shapes if performing multiple analyses with the same mesh and number of mpi ranks.

`read_volume_modes = .false.`

Read the partitioned volume mode shapes instead of computing them. Used in conjunction with `write_volume_modes` to avoid recomputing the volume mode shapes if performing multiple analyses with the same mesh and number of mpi ranks.

`lfd_nfreq = 0`

Number of input frequencies for the LFD solver (SFE only).

`lfd_freq(:) = 0.0`

Input frequencies for the LFD solver, in rad/sec (SFE only).

`lfd_nmodes_perturbed = -1`

Number of numbers to perturb for the right hand side of the LFD problem. The default value of -1 will perturb each mode.

`lfd_modes_perturbed(:) = 0`

The list of mode numbers to perturb for the right hand side of the LFD problem.

`lfd_write_mode_files = .true.`

When active, the volume mode shapes for each perturbed mode will be saved to disk during the first frequency and reloaded on subsequent frequencies.

`lfd_use_existing_mode_files = .false.`

When active, reuse lfd mode shape files from a previous run. The previous run must have used the same number of MPI ranks.

B.5.8 &composite_overset_mesh

This namelist provides the input for SUGGAR++ (see the documentation supplied with SUGGAR++ for details). This namelist is deprecated, and has been subsumed by the `&overset_data` namelist in the `fun3d.nml` file.

```
&composite_overset_mesh
  input_xml_file = ''
/
```

<u>`input_xml_file = ''`</u>

This is the file containing the XML commands for SUGGAR++. Specify the same `Input.xml` file that was used to generate the initial composite grid with the "stand-alone" SUGGAR++ code. Deprecated - this file should now be specified in the `&overset_data` namelist in the `fun3d.nml` file.

B.6 `rotor.input`

FUN3D is capable of modeling a rotating blade system using different levels of approximation. In order of increasing complexity/fidelity/cost, rotor systems may be analyzed using either a time-averaged actuator disk, or via first principles modeling of the moving, articulated, rotor blades using overset, moving grids. The actuator method utilizes momentum/energy source terms to represent the influence of the rotating blade system. Use of the source terms simplifies grid generation, since the actuator surfaces do not need to be built into the computational grid. However, the computational grid should have some refinement in the vicinity of the actuator surfaces to obtain accurate results. The steady-state actuator disk capability was originally implemented by Dave O'Brien, at the time a PhD candidate at Georgia Tech. [124] O'Brien also initiated the overset capability in FUN3D, which was later extended and coupled to a rotorcraft comprehensive code by Biedron et al. [125]

The rotor.input file is used primarily for specifying input quantities related to an actuator surface model for rotor/propeller combinations. When using overset, moving grids and/or coupling FUN3D to a rotorcraft comprehensive code for a more detailed simulation, a limited number of the input fields in the `rotor.input` file are also required. The fields required for coupled rotorcraft simulations include *(required for coupled simulation)* in the variable description. The command line option `--rotor` is required for both types of analysis.

The two parameters used to set the flight condition and force/moment coefficient normalization in compressible rotorcraft simulations are `mach_number` in `fun3d.nml` and `Vinf_Ratio` in `rotor.input`. To nondimensionalize the forces with the rotor tip velocity, set `mach_number` to the tip mach number and `Vinf_Ratio` to the ratio of freestream velocity to rotor tip velocity. When `mach_number` is the tip mach number then `reynolds_number` should be set to the corresponding tip Reynolds number. To nondimensionalize the forces with the freestream velocity, set `mach_number` to the freestream mach number and `Vinf_Ratio` to one. The `Vinf_Ratio` will still affect the force nondimensionalization as described above, for incompressible solutions.

A sample `rotor.input` file is shown below for a conventional main rotor and tail rotor helicopter.

```
  # Rotors  Vinf_Ratio  Write Soln   Force_ref  Moment_ref
         2         0.1          50         1.0         1.0
=== Main Rotor =========================================================
Rotor Type   Load Type    # Radial    # Normal  Tip Weight
         1           0          50         720         0.0
  X0_rotor    Y0_rotor    Z0_rotor        phi1        phi2        phi3
      0.00        0.00        0.00        0.00       -5.00        0.00
  Vt_Ratio  ThrustCoff  TorqueCoff        psi0  PitchHinge      DirRot
      1.00       0.005        0.00        0.00         0.0           0
  # Blades   TipRadius  RootRadius  BladeChord   FlapHinge    LagHinge
         4        1.00        0.00        0.05        0.00        0.00
 LiftSlope  alpha, L=0         cd0         cd1         cd2
```

```
      6.28        0.00       0.002        0.00        0.00
    CL_max      CL_min      CD_max      CD_min       Swirl
      1.50       -1.50        1.50       -1.50           0
    Theta0  ThetaTwist     Theta1s     Theta1c  Pitch-Flap
      5.00       -2.00        0.00        0.00        0.00
  # FlapHar       Beta0      Beta1s      Beta1c
         0        0.00        0.00        0.00
    Beta2s      Beta2c      Beta3s      Beta3c
      0.00        0.00        0.00        0.00
   # LagHar      Delta0     Delta1s     Delta1c
         0        0.00        0.00        0.00
   Delta2s     Delta2c     Delta3s     Delta3c
      0.00        0.00        0.00        0.00
=== Tail Rotor =======================================================
Rotor Type   Load Type    # Radial    # Normal  Tip Weight
         1           0         100         720         0.0
  X0_rotor    Y0_rotor    Z0_rotor        phi1        phi2        phi3
      1.00        0.00        0.00       90.00        0.00        0.00
  Vt_Ratio  ThrustCoff  TorqueCoff        psi0  PitchHinge      DirRot
      1.25       0.001        0.00        0.00         0.0           0
  # Blades   TipRadius  RootRadius  BladeChord   FlapHinge    LagHinge
         3        0.20        0.00        0.01        0.00        0.00
 LiftSlope  alpha, L=0         cd0         cd1         cd2
      6.28        0.00       0.002        0.00        0.00
    CL_max      CL_min      CD_max      CD_min       Swirl
      1.50       -1.50        1.50       -1.50           1
    Theta0  ThetaTwist     Theta1s     Theta1c  Pitch-Flap
      8.00        0.00        0.00        0.00        0.00
  # FlapHar       Beta0      Beta1s      Beta1c
         0        0.00        0.00        0.00
    Beta2s      Beta2c      Beta3s      Beta3c
      0.00        0.00        0.00        0.00
   # LagHar      Delta0     Delta1s     Delta1c
         0        0.00        0.00        0.00
   Delta2s     Delta2c     Delta3s     Delta3c
      0.00        0.00        0.00        0.00
```

The header line is where the user specifies the number of rotors, the freestream velocity ratio, and how often to output the plot3d loading file. The remainder of the file is in a block structure, where each block represents the inputs for one rotor. The first line of each block is a text line that can be edited to keep the rotors organized for the user. The input values do not have to be in a fixed format (spaces and number of decimal points do not matter), but the input values do have to be in the correct order as noted by the header lines for the individual input parameters.

B.6.1 Header

`# Rotors` *(required for coupled simulation)*

This is the number of actuator surfaces to create. The number of rotor input blocks in this file must match the number of rotors specified.

`Vinf_Ratio` *(required for coupled simulation)*

This is the ratio of the freestream velocity to the the velocity used for force normalization. The force normalization velocity is typically the tip velocity for rotorcraft applications.

Write Soln

This is the frequency (in iterations) of Plot3D rotor loading data output, which is pairs of `source_grid_00000.p3d` and `source_data_00000.p3d` files. To write once, set `Write Soln` to `steps`. If a nonzero value is specified, a pair of the Plot3D files will be output at the end of the computation corresponding to the final solution state. For `Load type = 3` or `7`, a total of 14 variables are output to the data files for each point on the disk mesh. The output variables are, respectively, the local disk-force components in the inertial frame, disk area, effective angle of attack, lift and drag coefficients, blade pitch angle (combined collective pitch and twist), local velocity components, density, static pressure, and total pressure. For other load types, a total of 4 variables are output to the data files per disk source point; and these are the local disk-force components in the inertial frame and disk area.

Force_ref

This is the conversion factor to obtain forces in alternate units,

1.0 will output the standard Fun3D nondimensionalization

$(L_{ref}^2 a_{ref}^2)/(\pi R_{rotor}^2 V_{tip}^2)$ will output standard rotorcraft nondimensionalization

${\rho^*}_{ref} {a^*}_{ref}^2 L_{ref}^2$ will output dimensional units

Moment_ref

This is the conversion factor to obtain moments in alternate units.

1.0 will output the standard Fun3D nondimensionalization

$(L_{ref}^3 a_{ref}^2)/(\pi R_{rotor}^3 V_{tip}^2)$ will output standard rotorcraft nondimensionalization

${\rho^*}_{ref} {a^*}_{ref}^2 L_{ref}^3$ will output dimensional units

In the above expressions used in `Force_ref` and `Moment_ref`, L_{ref} is the ratio of the physical characteristic length to the corresponding grid units; ${\rho^*}_{ref}$ and ${a^*}_{ref}$ are the dimensional air density and dimensional speed of sound, respectively, at the reference state.

B.6.2 Actuator Surface Model

Rotor Type

Type of rotor model to apply,

1 models the rotor as an actuator disk.

2 models the rotor as actuator blades.

`Load Type`

Type of loading to apply to the rotor model.

1 is a pressure jump based on `ThrustCoff` that is constant over the disk. This loading does not model swirl. Adjoint-based sensitivity analysis and rotor optimization can be performed with this load type, see section 9.12.

2 is a pressure jump based on `ThrustCoff` that increases linearly with radius. This loading does not model swirl.

3 is blade-element-based loading based defined by the blade element parameters defined in section B.6.6 and section B.6.7. This loading can model swirl if `Swirl=1` in `rotor.input`.

4 is user-specified source geometry and strength. Not recommended unless you have experience in actuator disk modeling. See the subroutine `read_user_source2` in `LibF90/rotors.f90` for input format.

5 is user specified thrust and torque radial distributions in the file `propeller_propertiesN.dat`, where `N` is the rotor index. The first line of the file is the number of radial stations. The following lines have three numbers per station, with r/R, $dC_T/d(r/R)$, $dC_Q/d(r/R)$. This file sets the thrust and torque loading directly, so `ThrustCoff` and `TorqueCoff` are ignored. This loading can model swirl if `Swirl=1` in `rotor.input`.

6 is a body force based on on the optimal distribution of Goldstein [126] implemented as described by Stern, Kim, and Patel. [127] Use `ThrustCoff` to set the thrust and `TorqueCoff` to produce swirl (with `Swirl=1` in `rotor.input`).

7 is blade-element based loading similar to `Load Type=3`. However, instead of constant blade element parameters and linear twist applied from the blade root to tip, this load type supports radial distributions of blade chord length, twist, as well as blade element properties (section B.6.6 and section B.6.7) specified at different airfoil sections. Moreover, this load type can also use C81 airfoil performance tables (e.g., Ref. [128]) to account for flow compressibility effects. In this case, the local Mach number and effective angle of attack computed from the CFD solution are used to determine airfoil lift, drag, and pitching moment coefficients. An additional input file, `bladegeom_rotor.dat`, is required for all rotors that are specified with `Load Type=7`. The file format and the required input for enabling the use of C81 tables are given in section B.6.10. This loading can model swirl if `Swirl=1` in `rotor.input`. Adjoint-based sensitivity analysis and rotor optimization can also be performed with this load type; see section 9.12.

`# Radial`

This is the number of sources to distribute along the blade radius. This should be set to approximately match the resolution of the volume grid, otherwise a suggested value is 100.

`# Normal`

This is the number of sources to distribute along the circumferential direction. This should be set to approximately match the resolution of the volume grid. For `Rotor Type=1`, 720 is suggested for a source every 0.5 degrees. For `Rotor Type=2`, 20 points in the chord direction is suggested.

`Tip Weight`

This is the hyperbolic weighting factor for distributing sources along the blade radius. A suggested value is 0.0, which yields uniform distribution. A value larger than 2.0 is not advised because it places too many sources at the blade tip.

B.6.3 Rotor Reference System

`X0_rotor` *(required for coupled simulation)*

This is the x coordinate of the hub center of rotation, in grid units.

`Y0_rotor` *(required for coupled simulation)*

This is the y coordinate of the hub center of rotation, in grid units.

`Z0_rotor` *(required for coupled simulation)*

This is the z coordinate of the hub center of rotation, in grid units.

`phi1`

This is the first Euler angle describing a rotation about the x axis, in degrees. For a propeller oriented in the x-positive direction, this should be 0.

`phi2`

This is the second Euler angle describing a rotation about the a_2 axis, in degrees. For a propeller oriented in the x-positive direction, this should be -90.

`phi3`

This is the third Euler angle describing a rotation about the b_3 axis, in degrees. For a propeller oriented in the x-positive direction, this should be 0.

The Euler angles must be input correctly to obtain the correct orientation of the source based actuator disk. The following example illustrates how to determine these angles. Figure B7 depicts the rotations `phi1` $= 10$, `phi2` $= -15$, and `phi3` $= 15$. Initially, the thrust is assumed to be in the z direction and the disk is located in the $x - y$ plane. The first rotation of `phi1` about the x axis takes the $x - y - z$ system to the $a_1 - a_2 - a_3$ system shown in red. The second rotation of `phi2` about the a_2 axis takes the $a_1 - a_2 - a_3$ system to the $b_1 - b_2 - b_3$ system shown in green. The final rotation of `phi3` about the b_3 axis takes the $b_1 - b_2 - b_3$ system to the rotor reference system shown in blue. The black circle represents the initial disk orientation and the blue circle represents the final disk orientation. In general `phi1` and `phi2` are sufficient to define the thrust orientation. The variable `phi3` only changes the location of the zero azimuth angle definition for the rotor.

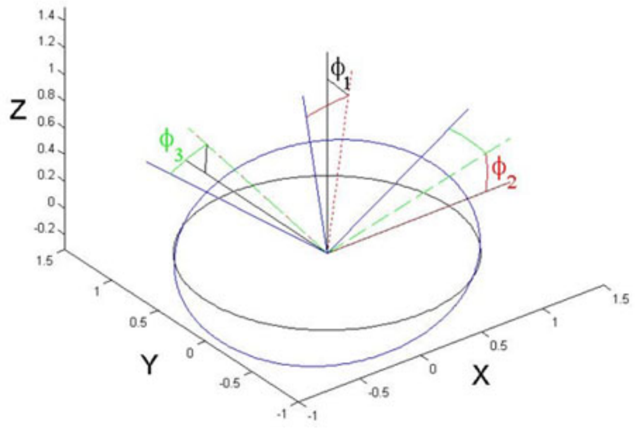


Figure B7: Rotor disk Euler angles.

B.6.4 Rotor Loading

`Vt_Ratio` *(required for coupled simulation)*

This is the ratio of the tip speed to the velocity used for force normalization, which is `mach_number` for compressible simulations. For `Load Type = 3`, `Load Type = 5`, and `Load Type = 6` a negative value will reverse the rotation direction. The propeller convention is $J = \frac{V_a}{nD}$, where V_a is speed of advance (true airspeed), n is revolutions per unit time, and D is diameter, i.e, `Vt_Ratio`$= \frac{\pi}{J}$.

`ThrustCoff`

This is the rotor thrust coefficient defined as, $C_T = \text{Thrust} / (\pi \rho_{ref} R^2 (\Omega_{Dim} R)^2)$, when `Load Type`=1, `Load Type`=2, or `Load Type`=6. The blade-element model does not trim to specified thrust coefficient. The propeller convention is K_T = Thrust / $(\rho_{ref} n^2 D^4)$, where n is revolutions per unit time, and D is diameter, i.e, `ThrustCoff`= $\frac{4}{\pi^3} K_T$.

<u>`TorqueCoff`</u>

This is the rotor torque coefficient defined as, C_Q = Torque / $(\pi \rho_{ref} R^3 (\Omega_{Dim} R)^2)$, when `Load Type`=6. The blade-element model is not effected by specified torque coefficient. The propeller convention is K_Q = Torque / $(\rho_{ref} n^2 D^5)$, where n is revolutions per unit time, and D is diameter, i.e, `TorqueCoff`= $\frac{8}{\pi^3} K_Q$.

B.6.5 Blade Parameters

<u>`psi0` *(required for coupled simulation)*</u>

This is the initial azimuthal position of blade one, in degrees; the azimuth position is defined as zero when the blade is oriented along the x-axis with the tip at the most positive x location.

<u>`PitchHinge` *(required for coupled simulation)*</u>

This is the radial position of the blade pitch hinge normalized by tip radius.

<u>`DirRot` *(required for coupled simulation)*</u>

This is the direction of rotor rotation. Zero is counter-clockwise rotation and one is clockwise rotation. This option only applies to coupled simulation, not actuator models.

<u>`# Blades` *(required for coupled simulation)*</u>

This is the number of rotor blades. It is only used for `Load Type`=3 and overset rotor simulations.

<u>`TipRadius` *(required for coupled simulation)*</u>

This is the radius of the blade, in grid units.

<u>`RootRadius` *(required for coupled simulation)*</u>

This is the radius of the blade root, in grid units. It accounts for the cutout region immediately surrounding the hub.

<u>`BladeChord` *(required for coupled simulation)*</u>

This is the chord length of the blade, in grid units. It can only handle rectangular blade planforms and is only valid for `Load Type`=3.

<u>`FlapHinge` *(required for coupled simulation)*</u>

This is the radial position of the blade flap hinge normalized by tip radius.

LagHinge *(required for coupled simulation)*

This is the radial position of the blade lag hinge normalized by tip radius.

B.6.6 Blade Element Parameters for Load Type=3

These inputs are used to set the blade element lift and drag curves according to:

$$C_L = \texttt{LiftSlope}(\alpha - \alpha_{L=0}) \tag{B8}$$

and

$$C_D = \texttt{cd0} + \texttt{cd1}\,\alpha + \texttt{cd2}\,\alpha^2 \tag{B9}$$

LiftSlope

This is the lift curve slope per radian.

alpha, L=0

This is the zero lift angle of attack, in degrees.

cd0, cd1, and cd2

These are the quadratic drag polar coefficients; where cd1 is per radian and cd2 is per radian squared.

CL_max and CL_min

These limiters to control the lift coefficient beyond the linear region.

CD_max and CD_min

These limiters to control the drag coefficient.

Swirl

0 neglects the sources terms that create rotor swirl.

1 the swirl inducing source terms.

B.6.7 Pitch Control Parameters for Load Type=3

These inputs are used to specify the pitch/flap controls according to:

$$\theta = \texttt{Theta0} + \texttt{ThetaTwist}\,(r/R) + \texttt{Theta1c}\,\cos(\psi) + \texttt{Theta1s}\,\sin(\psi) \tag{B10}$$

Theta0

This is the collective pitch defined at r/R=0, in degrees.

ThetaTwist

This is the total amount of linear blade twist from the origin to the blade tip, in degrees.

`Theta1s`

This is the longitudinal cyclic pitch input, in degrees.

`Theta1c`

This is the lateral cyclic pitch input, in degrees.

`Pitch-Flap`

Pitch-Flap coupling parameter [not implemented].

B.6.8 Prescribed Flap Parameters

These inputs are used to specify the flap harmonics according to:

$$\begin{aligned}\beta = \texttt{Beta0} + \texttt{Beta1s}\,\sin(\psi) + \texttt{Beta1c}\,\cos(\psi) \\ +\texttt{Beta2s}\,\sin(2\psi) + \texttt{Beta2c}\,\cos(2\psi) \\ +\texttt{Beta3s}\,\sin(3\psi) + \texttt{Beta3c}\,\cos(3\psi)\end{aligned} \tag{B11}$$

`# FlapHar`

This is the number of flap harmonics to include. The valid input range is zero to three.

`Beta0`

This is the coning angle, in degrees.

`Beta1s` and `Beta1c`

This is the first flap harmonics, in degrees.

`Beta2s` and `Beta2c`

This is the second flap harmonics, in degrees.

`Beta3s` and `Beta3c`

This is the third flap harmonics, in degrees.

B.6.9 Prescribed Lag Parameters

These inputs are used to specify the lag harmonics according to:

$$\begin{aligned}\delta = \texttt{Delta0} + \texttt{Delta1s}\,\sin(\psi) + \texttt{Delta1c}\,\cos(\psi) \\ +\texttt{Delta2s}\,\sin(2\psi) + \texttt{Delta2c}\,\cos(2\psi) \\ +\texttt{Delta3s}\,\sin(3\psi) + \texttt{Delta3c}\,\cos(3\psi)\end{aligned} \tag{B12}$$

`# LagHar`

This is the number of lag harmonics to include. The valid input range is zero to three.

Delta0

This is the coning angle, in degrees.

Delta1s and Delta1c

This is the first lag harmonics, in degrees.

Delta2s and Delta2c

This is the second lag harmonics, in degrees.

Delta3s and Delta3c

This is the third lag harmonics, in degrees.

B.6.10 bladegeom_rotor.dat

The blade-element based actuator disk model with `Load Type=7` requires an additional input file, `bladegeom_rotor.dat`, to be placed in the flow solver execution directory. This input file provides detailed blade element and geometry specifications for all rotors with `Load Type=7`. The blade properties are given at a number of stations and the distributions between two consecutive stations are determined through linear interpolations. If a C81 table (e.g., [128] is specified, the inputs for blade element parameters are still required but will not be used for loading calculations. A sample file is shown below for an aircraft configuration with two propulsors, one located on each wing.

```
===================================== Rotor Index ==========================================
1
Nstations  C81 Table
3          0
r/Rtip Twist(deg) Chord/Rtip LiftSlope a,L=0 cd0 cd1 cd2 CL_max CL_min CD_max CD_min C81 File
0.1504  0.0    0.0605    6.28    0.0   0.013   0.0   0.004   1.5   -0.5   1.5   0.0   none
0.5280  0.0    0.0605    6.28    0.0   0.013   0.0   0.004   1.5   -0.5   1.5   0.0   none
1.0000  0.0    0.0605    6.28    0.0   0.030   0.0   0.020   1.5   -0.5   1.8   0.0   none
===================================== Rotor Index ==========================================
2
Nstations  C81 Table
3          0
r/Rtip Twist(deg) Chord/Rtip LiftSlope a,L=0 cd0 cd1 cd2 CL_max CL_min CD_max CD_min C81 File
0.1504  0.0    0.0605    6.28    0.0   0.013   0.0   0.004   1.5   -0.5   1.5   0.0   none
0.5280  0.0    0.0605    6.28    0.0   0.013   0.0   0.004   1.5   -0.5   1.5   0.0   none
1.0000  0.0    0.0605    6.28    0.0   0.030   0.0   0.020   1.5   -0.5   1.8   0.0   none
```

This input file assumes a block structure where each block contains the geometry and blade element specifications for various airfoil sections. In each block, the first line is a text line separating inputs for individual rotors. The integer value on the second line is the rotor index specified with `Load Type=7` in `rotor.input`. The third and fifth lines are text lines showing the order of the required data, as detailed in the following list.

`Nstations`

This is the number of airfoil stations associated with specific geometry and aerodynamic properties.

`C81 Table`

0 uses blade element definitions (section B.6.6 and section B.6.7).

1 uses C81 airfoil performance tables to determine blade aerodynamics. If this option is selected, the user should provide the lookup table and include the filename in the `C81 File` input (last entry).

`r/Rtip`

The radial station at which blade parameters are given. Note that this input requires the radial station normalized by the blade radius.

`Twist(deg)`

The blade twist at the current station, in degrees.

`Chord/Rtip`

The ratio of blade chord length at the current station to the blade tip radius.

`LiftSlope` and `a,L=0`

The lift slope and zero lift angle of attack (in degrees) used in the blade element lift curve (section B.6.6) at the current station.

`cd0`, `cd1`, and `cd2`

The parameters used in the quadratic drag profile (section B.6.6) at the current station.

`CL_max`, `CL_min`, `CD_max`, and `CD_min`

These are limiters to control lift and drag coefficients (section B.6.6) at the current station.

`C81 File`

This is the filename of the C81 airfoil performance table at the current station. For example, if the desired table is contained in a file named `naca23012.dat`, then this input should be specified as `naca23012.dat`, and the file should be placed in the same directory as `bladegeom_rotor.dat`. Any radial locations between the current station and the nearest inboard station will use this lookup table to determine blade loading. If `C81 Table=0`, set this input to `none` as shown in the sample file; no lookup tables will be used for this rotor.

B.7 `tdata`

This file defines the gas model when `eqn_type = 'generic'`. A keyword is required on the first line of `tdata`. Many of these models require additional information as detailed in each keyword section.

Some keywords require a list the species. For these keywords, additional groups of species can be specified for boundary conditions after a blank line. If new species are introduced in subsequent instances their mass fractions are automatically initialized to zero at any previous inflow boundary. All the species entries in this file are available as reactants throughout the entire flow field.

B.7.1 `perfect_gas` Keyword

A perfect gas can be modeled with the `perfect_gas` keyword. The parameters can be explicitly defined in `tdata` by the namelist `&species_properties`. The namelist in `tdata` has different variables than the `&species_properties` in `species_thermo_data`. Here is an example of the namelist with defaults that are all given in SI units,

```
perfect_gas
&species_properties
 gamma   =   1.4
 mol_wt  =  28.8
 suther1 =   0.1458205E-05
 suther2 = 110.333333
 prand   =   0.72
/
```

Where `gamma` is the gas specific heat ratio, `mol_wt` is the gas molecular weight, `prand` is the gas Prandtl number, and `suther1` and `suther2` are the first and second Sutherland's viscosity coefficients, s_1 and s_2, in

$$\mu = s_1 \frac{T^{3/2}}{T + s_2} \tag{B13}$$

These defaults are used if the `&species_properties` namelist or any of its variables are omitted.

B.7.2 `equilibrium_air_t` Keyword

The `equilibrium_air_t` keyword engages the Tannehill curve fits for thermodynamic and transport properties of equilibrium air. [129] This keyword does not require additional lines.

```
equilibrium_air_t
```

B.7.3 `equilibrium_air_r` Keyword

The `equilibrium_air_r` keyword engages the Tannehill curve fits for transport properties and a table look-up for equilibrium gases [130], This keyword does not require additional lines.

```
equilibrium_air_r
```

B.7.4 `one` Keyword

This one-temperature (1-T) model assumes that all the species are thermally in equilibrium state; the translational temperature T and vibrational temperature T_v are equal. This is a mixture of thermally perfect gases and multispecies transport. In this example, only molecular oxygen and nitrogen are present on the inflow boundary, but atomic nitrogen and oxygen and nitric oxide may be produced elsewhere in the flow field due to chemical reactions. The inflow boundary mass fraction of molecular oxygen and nitrogen is given next to their symbols. Mass fractions should sum to one.

```
one
N2  .767
N
O2  .233
O
NO
```

B.7.5 `two` Keyword

This two-temperature (2-T) model assumes that energy distribution in the translational and rotational modes of heavy particles (not electrons) are equilibrated at translational temperature T and all other energy modes (vibrational, electronic, electron translational) are equilibrated at vibrational temperature T_v. In this example, the gas is assumed to be a mixture of 11 thermally perfect gases. The inflow boundary mass fraction of molecular oxygen and nitrogen is given next to their symbols. Mass fractions should sum to one. Other products are the results of chemical reactions flow field.

```
two
N2  .767
N
O2  .233
O
NO
O2+
O+
NO+
e-
```

B.7.6 `FEM` Keyword

This Free-Energy Minimization (FEM) model causes the species continuity equations to be replaced with elemental continuity equations and equilibrium relations for remaining species. In this example, the gas is assumed to be a mixture of 11 thermally perfect gases. The inflow boundary mass fraction of molecular oxygen and nitrogen is given next to their symbols. Mass fractions should sum to one. Other products are the results of chemical reactions flow field.

```
FEM
N2  .767
N
O2  .233
O
NO
O2+
O+
NO+
e-
```

B.8 `species_thermo_data`

The `species_thermo_data` file is the master file for species thermodynamic data. The majority of simulations do not require changes to this file. Investigating other sources of thermodynamic data is the only reason to edit this file. If the file is not found in the local run directory, it is assumed to be located in the `[install-prefix]/share/physics_modules` directory. See section A.3 for `[install-prefix]`.

Each species record consists of the species name, a `&species_properties` namelist, the number of thermodynamic property curve fit ranges, and the curve fit coefficients for each range. [131] No blank line is allowed in this file. This `&species_properties` namelist has different variables than the `&species_properties` in `tdata`. The elements of the `&species_properties` namelist are:

`mol_wt`

This sets the molecular weight of the particle. It is always required.

`elec_impct_ion = -1._dp`

This sets the energy for neutrals `ion=.false.` that is required to liberate an electron when the neutral impact a free electron, in units of electron volts (eV).

`siga(:) = -1._dp`

This is an array of three real numbers, which correspond to the curve fit coefficients for electron-neutron energy exchange. The cross section is defined as

$$\sigma_{en} = a + bT + cT^2 \tag{B14}$$

where σ_{en} is the electron-neutron energy exchange collision cross section in m^2. The variables a, b, and c are the curve fit coefficients and T is the gas temperature. [132, 133] For example, `siga=7.5e-20, 0, 0`.

`disoc_ener = 0._dp`

This is the dissociation energy of molecule in electron volts (eV).

`alantel = 0._dp`

This is the Landau-Teller constant to compute vibrational relaxation time for molecule. [134, 135]. It is required when `molecule=.true.`.

`cprt0 = 0._dp`

This nondimensional real number defines translational-rotational heat capacity. It is normalized by the gas constant and is equal to

$$cprt() = \frac{f+2}{2} \tag{B15}$$

where f is the number of degrees of freedom in translation and rotation. The defaults for atoms and diatomic molecules are 2.5 and 3.5, respectively.

A portion of the `species_thermo_data` that provides thermodynamic properties of carbon is shown below.

```
C
&species_properties
elec_impct_ion = 11.264
siga = 7.5e-20, 5.5e-24, -1.e-28
mol_wt = 12.01070
/
3
 0.64950315E+03 -0.96490109E+00  0.25046755E+01 -0.12814480E-04
 0.19801337E-07 -0.16061440E-10  0.53144834E-14  0.00000000E+00
 0.85457631E+05  0.47479243E+01   200.000  1000.000
-0.12891365E+06  0.17195286E+03  0.26460444E+01 -0.33530690E-03
 0.17420927E-06 -0.29028178E-10  0.16421824E-14  0.00000000E+00
 0.84105978E+05  0.41300474E+01  1000.000  6000.000
 0.44325280E+09 -0.28860184E+06  0.77371083E+02 -0.97152819E-02
 0.66495953E-06 -0.22300788E-10  0.28993887E-15  0.00000000E+00
 0.23552734E+07 -0.64051232E+03  6000.000 20000.000
```

The species name is defined in line `1`. Between lines `2` and `9` species properties are defined. Line `10` shows that there are three thermodynamic property curve fits for temperature ranges of 200 K $< T <$ 1,000 K, 1,000 K $< T <$ 6,000 K, and 6,000 K $< T <$ 20,000 K. Each data range consists of 12 real numbers. Four real numbers must be given on each line. The first 10 real numbers are the thermodynamic curve fit coefficients, and the last two real numbers identify the temperature range for the given curve fit coefficients. These coefficients are used to calculate the following thermodynamic properties

$$c_p(T)/R = a_1T^{-2} + a_2T^{-1} + a_3 + a_4T + a_5T^2 + a_6T^3 + a_7T^4 \tag{B16}$$

$$h(T)/RT = -a_1T^{-2} + a_2T^{-1}ln\,T + a_3 + a_4\frac{T}{2} + a_5\frac{T^2}{3} + a_6\frac{T^3}{4} + a_7\frac{T^4}{5} + \frac{a_9}{T} \tag{B17}$$

$$s(T)/R = -a_1\frac{T^{-2}}{2} - a_2T^{-1} + a_3ln\,T + a_4T + a_5\frac{T^2}{2} + a_6\frac{T^3}{3} + a_7\frac{T^4}{4} + a_{10} \tag{B18}$$

where T is the gas temperature, R is the universal gas constant, c_p, h, and s are the species specific heat, enthalpy and entropy, respectively, and a_i are the provided curve fit coefficients in `species_thermo_data`.

The following corrections will be applied if the gas temperature is out of the given range for the given curve fit coefficients:

$$c_p(T) = c_p(T^*) \tag{B19}$$

$$h(T) = h(T^*) + (T - T^*)c_p(T^*) \tag{B20}$$

$$s(T) = s(T^*) + ln\,\frac{T}{T^*}c_p(T^*) \tag{B21}$$

where

$$T^* = \begin{cases} T_{lower} & \text{for } T < T_{lower} \\ T_{upper} & \text{for } T > T_{upper} \end{cases} \tag{B22}$$

B.9 `kinetic_data`

The `kinetic_data` file defines possible chemical reactions and is optional. If the file is not found in the local run directory, it is assumed to be located in the `[install-prefix]/share/physics_modules` directory. See section A.3 for `[install-prefix]`. Reactants and products can be any species defined in the `species_thermo_data` described in section B.8. A sample entry looks like

```
O2 + M    <=>   2O + M
2.000e+21  -1.50  5.936e+04
teff1 = 2
exp1 = 0.7
t_eff_min = 1000.
t_eff_max = 50000.
```

```
C = 5.0
O = 5.0
N = 5.0
H = 5.0
Si = 5.0
e- = 0.
```

The first line specifies the reaction while line 2 provides three coefficients of an Arrhenius-like equation,

$$K_f = c_f T_{eff}^{\eta} e^{-\epsilon_0/kT_{eff}} \tag{B23}$$

where c_f is the preexponential factor, η is the power of temperature dependence on the preexponential factor, ϵ_0 is the Arrhenius activation energy, and k is the Boltzmann constant. The arrowheads in line 1 signify the allowed directionality of the reaction. The symbol `=>` denotes forward reaction only while `<=>` denotes forward and backward rates are computed. The coefficients in line 2 correspond to c_f, η, and ϵ_0/k, respectively. For reactions with a generic collision partner, `M`, such as this one, these coefficients correspond to Argon; and other collision partners and their efficiencies (multipliers of c_f) are specified on lines following line 5 and 6, which give the valid temperature range for the reaction. The effective temperature, T_{eff}, is defined according to a given integer number in line 3.

`teff1 = 1`,`teff2 = 1`

This defines the formula to compute the effective temperature T_{eff} for the forward rate and backward rate, respectively. It is engaged for the case of thermal nonequilibrium. Options for `teff` are:

1: $T_{eff} = T_{tr}$

2: $T_{eff} = T_{tr}^{exp1} T_v^{1-exp1}$

3: $T_{eff} = T_v$

where T_{tr} and T_v are translational and vibrational temperatures, respectively.

`exp1 = 0.7`

This is the exponent used to define the effective temperature when `teff1= 2` (forward rate) or `teff2 = 2` (backward rate).

`rf_max = 1.e+20`

This is the upper limit on forward reaction rate in cgs units when `augment_kinetics_limiting = .true`. For unlimited rates as function of temperature, see the output file `kinetic_diagnostics_output`.

`rf_min = 1.e-30`

This is the lower limit on forward reaction rate in cgs units when `augment_kinetics_limiting= .true`. For unlimited rates as function of temperature, see the output file `kinetic_diagnostics_output`.

`rb_max = 1.e+30`

This is the upper limit on backward reaction rate in cgs units when `augment_kinetics_limiting = .true`. For unlimited rates as function of temperature, see the output file `kinetic_diagnostics_output`.

`rb_min = 1.e-30`

This is the lower limit on backward reaction rate in cgs units when `augment_kinetics_limiting = .true`. For unlimited rates as function of temperature, see the output file `kinetic_diagnostics_output`.

`t_eff_min = 1000.`

This is the minimum temperature for T_{eff}. This may circumvent stiff source terms when computing reaction rates.

`t_eff_max = 50000.`

The maximum temperature for T_{eff}. This may circumvent stiff source terms when computing reaction rates.

B.10 species_transp_data

The `species_transp_data` file contains log-linear curve fit coefficients for species collision cross sections that are defined based on temperature range of 2,000–4,000 K. [133] This temperature range spans boundary-layer temperatures for typical hypersonic entry. The curve fit for the given coefficients is poor at temperatures below 1,000 K. Higher order curve fit data is available in `species_transp_data_0`. The file should not be changed by the user unless there is a need to investigate other sources of collision cross-section data. If the file is not found in the local run directory, it is assumed to be located in the `[install-prefix]/share/physics_modules` directory. See section A.3 for `[install-prefix]`. An example of the entries in the file is

```
 Ar Ar
-14.6017 -14.6502 -14.5501 -14.6028 ! trr132+kestin et al
Ar+ Ar+
-11.48 -12.08 -11.50 -12.10
Ar N2
-14.5995 -14.6475 -14.5480 -14.5981 ! kestin et al
Ar CO
-14.5975 -14.6455 -14.5459 -14.5964 ! kestin et al
```

B.11 species_transp_data_0

The file species_transp_data_0 provides collision cross section coefficients [136,137] for a higher order curve fit data than those that are in the species_transp_data. The data in species_transp_data_0 supersedes the data in species_transp_data The file should not be changed by the user unless there is a need to investigate other sources of collision cross-section data. If the file is not found in the local run directory, it is assumed to be located in the [install-prefix]/share/physics_modules directory. See section A.3 for [install-prefix]. An example of the entries in the file is

```
O2 N      1 1 1      (c)
        -1.1453028E-03  1.2654140E-02 -2.2435218E-01  7.7201588E+01
        -1.0608832E-03  1.1782595E-02 -2.1246301E-01  8.4561598E+01
         1.4943783E-04 -2.0389247E-03  1.8536165E-02  1.0476552E+00

NO N      1 1 1      (c)
        -1.5770918E-03  1.9578381E-02 -2.7873624E-01  9.9547944E+01
        -1.4719259E-03  1.8446968E-02 -2.6460411E-01  1.0911124E+02
         2.1014557E-04 -3.0420763E-03  2.5736958E-02  1.0359598E+00

NO O      1 1 1      (c)
        -1.0885815E-03  1.1883688E-02 -2.1844909E-01  7.5512560E+01
        -1.0066279E-03  1.1029264E-02 -2.0671266E-01  8.2644384E+01
         1.4145458E-04 -1.9249271E-03  1.7785767E-02  1.0482162E+00

END END   1 1 0
            0. 0. 0. 0.
            0. 0. 0. 0.
```

B.12 hara_namelist_data

The hara_namelist_data file controls the radiation models used by the Hara radiation module. [138, 139] It is optional for coupled radiation simulations. If it is not present, then the code automatically chooses the radiative mechanisms associated with species present in the flowfield that have number densities greater than 1000 particles/cm^2) Other options are set to the defaults. For users not experienced in shock-layer radiation, the recommended default options should be used. this hara_namelist_data A default hara_namelist_data is available in the PHYSICS_MODULES directory of the Fun3D distribution.

specifying radiation mechanisms for atomic species: The treatment of radiation resulting from atomic lines, atomic bound-free, and free-free photoionization (referred to here as atomic continuum), and negative ion continuum is available for atomic carbon, hydrogen, oxygen, and nitrogen. These mechanisms are specified through the following binary flags (on=1/off=0). If any of these flags are not present in `hara_namelist_data`, then that flag is set to true only if the number density of the associated atomic species is greater than 1000 particles/cm^2 somewhere in the flowfield.

`treat_[?]_lines`

A binary flag to enable the treatment of atomic lines for species `[?]`, where `[?]` can be c, h, n, and o, for atomic carbon, hydrogen, nitrogen and oxygen, respectively.

`treat_[?]_cont`

A binary flag to enable the treatment of atomic bound-free and free-free continuum for species `[?]`, where `[?]` can be c, h, n, and o, for atomic carbon, hydrogen, nitrogen and oxygen, respectively.

`treat_[?]_other`

A binary flag to enable the treatment of the negative-ion continuum for species `[?]`, where `[?]` can be c, h, n, and o, for atomic carbon, hydrogen, nitrogen and oxygen, respectively.

specifying radiation mechanisms for molecular species: The treatment of radiation resulting from numerous molecular band systems is available through the following flags (0 = off, 1 = SRB, 2 = LBL). The smeared rotational band (SRB) approach applies a simplified and efficient treatment of each molecular band system, which is accurate for optically thin conditions. The line-by-line (LBL) approach is a detailed but highly inefficient treatment of each molecular band system. The LBL option is not recommended for coupled radiation-flowfield computations, except for possibly the CO 4+ system, which may be optically thick for Mars entry conditions. If any of these flags are not present in `hara_namelist_data`, then that flag is set to the SRB option only if the number density of the associated molecular specie is greater than 1000 particles/cm^2 somewhere in the flowfield. Additional band systems are listed in the following paragraph. These additional band systems are generally considered negligible relative to those listed in this section. Therefore, for computational efficiency, they are not engaged by default. Definitions of each band system and the modeling data applied are discussed in Refs. [138, 140].

`treat_band_c2_swan`

A flag activating the C_2 Swan band system.

`treat_band_c2h`

A flag activating the C_2H band system.

`treat_band_c3`

A flag activating the C_3 and vacuum ultra-violet (VUV) band systems.

`treat_band_cn_red`

A flag activating the CN red band system.

`treat_band_cn_violet`

A flag activating the CN violet band system.

`treat_band_co4p`

A flag activating the CO 4+ band system.

`treat_band_co_bx`

A flag activating the CO B-X band system.

`treat_band_co_cx`

A flag activating the CO C-X band system.

`treat_band_co_ex`

A flag activating the CO E-X band system.

`treat_band_co_ir`

A flag activating the CO X-X band system.

`treat_band_h2_lyman`

A flag activating the H_2 Lyman band system.

`treat_band_h2_werner`

A flag activating the H_2 Werner band system.

`treat_band_n2fp`

A flag activating the N_2 1+ band system.

`treat_band_n2sp`

A flag activating the N_2 2+ band system.

`treat_band_n2pfn`

A flag activating the N_2^+ first-negative band system.

`treat_band_n2_bh1`

A flag activating the N_2 Birge-Hopfield I band system.

`treat_band_n2_bh2`

A flag activating the N_2 Birge-Hopfield II band system.

`treat_band_no_beta`

A flag activating the NO beta band system.

`treat_band_no_delta`

A flag activating the NO delta band system.

`treat_band_no_epsilon`

A flag activating the NO epsilon band system.

additional molecular band systems: This paragraph lists molecular band systems available in addition to those listed in the paragraph above. The band systems listed here are generally weak emitters and absorbers, and are therefore not engaged as a default. Therefore, for these band systems to be engaged, the following flags (0 = off, 1 = SRB, 2 = LBL) must be present in the `hara_namelist_data` file. The LBL treatment of these bands is not recommended.

`treat_band_c2_br`

A flag activating the C_2 Ballik-Ramsay band system.

`treat_band_c2_da`

A flag activating the C_2 Deslandres-d'Azambuja band system.

`treat_band_c2_fh`

A flag activating the C_2 Fox-Herzberg band system.

`treat_band_c2_mulliken`

A flag activating the C_2 Mulliken band system.

`treat_band_c2_philip`

A flag activating the C_2 Philips band system.

`treat_band_co3p`

A flag activating the CO 3+ band system.

`treat_band_co_angstrom`

A flag activating the CO angstrom band system.

`treat_band_co_asundi`

A flag activating the CO Asundi band system.

`treat_band_co_triplet`

A flag activating the CO triplet band system.

`treat_band_co2`

A flag activating the CO_2 band system. A value of two activates an approximate nonequilibrium model for UV emission, while a value of one assumes Boltzmann emission. The LBL treatment of this band is not available.

`treat_band_n2_cy`

A flag activating the N_2 Carrol-Yoshino band system.

`treat_band_n2_wj`

A flag activating the N_2 Worley-Jenkins band system.

`treat_band_n2_worley`

A flag activating the N_2 Worley band system.

`treat_band_no_gamma`

A flag activating the NO gamma band system.

`treat_band_no_betap`

A flag activating the NO beta-prime band system.

`treat_band_no_gammap`

A flag activating the NO gamma-prime band system.

`treat_band_o2_sr`

A flag activating the O_2 Schumann-Runge band system.

`treat_[?]_photo_dis`

A binary flag activating the molecular photo-dissociation mechanism [141] for `[?]` specie, where `[?]` can be O2 or N2. This mechanism is not technically a molecular band system.

`treat_[?]_photo_ion`

A binary flag activating the molecular photo-ionization mechanism [141] for `[?]` specie, where `[?]` can be O2 or N2. This mechanism is not technically a molecular band system.

`treat_no_photo`

A binary flag activating the molecular photo-ionization mechanism [141] for NO.

atomic line models: There are various models available for atomic line radiation, one of which must be chosen for each specie that engages atomic line radiation (as specified using `treat_[?]_lines`). This choice of atomic line model is made using the following flags. The listed defaults are applied if the individual flag is not present in `hara_namelist_data`, or if `hara_namelist_data` is not present in the working directory. All model types in this category must be surrounded by a quotation marks.

`c_atomic_line_model`, `h_atomic_line_model`

A character identifier for selecting the atomic line model for atomic carbon or hydrogen. Presently, the only available option is the model compiled in Ref. [140], which is referred to here as the Complete Line Model (CLM). Default : `'clm'`

`n_atomic_line_model`, `o_atomic_line_model`

A character identifier for selecting the atomic line model for atomic nitrogen or oxygen. The available models are compiled and compared in Ref. [138], which is referred to here as the Complete Line Model (CLM). Default : `'clm'` Available models are:

`'all multiplets'`

This model treats all lines as grouped multiplets. This significantly reduces the number of lines treated as well as the computational expense. However, this grouped multiplet approximation will lead to errors for non-optically-thin conditions.

`'clm'`

This model, which stands for Complete Line Model, applies the individual treatment of strong atomic lines while applying multiplet averages for weak lines. This is the recommended model.

electronic state population models: These flags specify the model applied for predicting the electronic state populations of atoms and molecules. The listed defaults are applied if the individual flag is not present in `hara_namelist_data`, or if `hara_namelist_data` is not present in the working directory. All model types in this category must be surrounded by a quotation marks, e.g. ‘ ’.

atomic electronic states

The electronic state populations for atoms are required for computing atomic line and photoionization emission and absorption. The compilation and comparison of the available models are presented in Ref. [139].

`c_electronic_state`, `h_electronic_state`

A character identifier for selecting the electronic state model for atomic carbon and hydrogen. Available models are (default : 'boltzmann'):

`'boltzmann'`

Applies Boltzmann population of electronic states.

`'Gally_1st_order_LTNE'`

Applies the Gally first-order local thermodynamic nonequilibrium method [142], which approximately accounts for the non-Boltzmann population of atomic states.

`n_electronic_state`, `o_electronic_state`

A character identifier for selecting the electronic state model for atomic nitrogen and oxygen. Available models are (default : 'CR'):

`'boltzmann'`

Same as for `c_electronic_state`

`'Gally_1st_order_LTNE'`

Same as for `c_electronic_state`

`'CR'`

Applies the detailed collisional radiative (CR) model developed in Ref. [139].

`'AARC'`

Applies the approximate atomic collisional radiative (AARC) model developed in Ref. [139]. This model is essentially a curve-fit based approximation of the CR model, which allows for improved computational efficiency with a slight loss in accuracy.

molecular electronic states

The electronic state populations for molecules are required for computing molecular band emission and absorption. The compilation and comparison of the available models are presented in Refs. [139, 143].

`molecular_electronic_state`

A character identifier for selecting molecular electronic state for all molecular band systems. Available models are (default : 'CR'):

`'boltzmann'`

Applies Boltzmann population of electronic states.

`'CR'`

Applies a detailed collisional radiative model considering both heavy-particle and electron impact transitions. Some molecular states are still assumed Boltzmann with this model because no data is presently available for the CR model.

other flags:

<u>`use_triangles`</u>

A logical flag specifying whether optically-thin atomic lines are modeled as triangles to reduce computational time. This option has shown to result in a negligible loss of accuracy while greatly reducing the computational time, [138] and is therefore recommended. Default : .true. Note: This flag is automatically set to `.true.` when `n_` or `o_atomic_line_model='clm'` — see **atomic line models** earlier in this section.

<u>`use_edge_shift`</u>

A logical flag to engage the photoionization edge shift [138] for atomic bound-free radiation. (on=1/off=0). Default : .true.

B.13 `controller.nml`

This file contains parameters for using a Proportional-Integral-Derivative (PID) controller for updating boundary condition values during code execution. Several parameters must be defined to activate the controller capability.

B.13.1 &tunnel_control

This namelist provides input parameters for a PID controller.

```
&tunnel_control
  number_of_controllers         = 0
  controller_points(:)          = -999
  initial_delay(:)              = -1
  kp(:)                         = 1.0
  ki(:)                         = 1.0
  kd(:)                         = 1.0
  find_mach(:)                  = .false.
  find_q(:)                     = .false.
  target_mach(:)                = 1.0
  target_q(:)                   = 1.0
  velocity_method(:)            = 'incompressible_bernoulli'
  ntf_calibration(:)            = .false.
  controller_mode(:)            = 'entrance_cone'
  settling_chamber(:)           = 'total'
  number_of_calibration_points = 0
  q(:)                          = 0.0
  cprime(:)                     = 0.0
/
```

`number_of_controllers = 0`

Number of different boundaries to be updated by the PID controller.

`controller_points(:) = -999`

This selects the boundary condition driver routine. Originally indicated the number of points used to calculate flow conditions.

'`1`' A single survey point, `field_point1` defined in section B.4.21, is used to determine the simulation flow conditions.

'`2`' Two survey points, `field_point1` and `field_point2` defined in section B.4.21, are used to determine the simulation flow conditions.

'`-999`' No controller set.

`initial_delay(:) = -1`

Number of iterations to wait before starting the controller. The delay is to allow initial transients to settle down to keep the controller from injecting spurious commands in to the system. Typically used when new simulations are started from scratch.

`kp(:) = 1.0`

The coefficient multiplying the proportional term.

`ki(:) = 1.0`

The coefficient multiplying the integral term.

`kd(:) = 1.0`

The coefficient multiplying the derivative term.

`find_mach(:) = .false.`

Set to `.true.` to use Mach number as the targeted condition.

`find_q(:) = .false.`

Set to `.true.` to use dynamic pressure as the targeted condition.

`target_mach(:) = 1.0`

Used in conjunction with `find_mach` logical parameter. This specifies the value of the Mach number desired in the simulation.

`target_q(:) = 1.0`

Used in conjunction with the `find_q` logical parameter. This specifies the dynamic pressure desired in the simulation in units of Pascals.

`velocity_method(:) = 'incompressible_bernoulli'`

This parameter specifies which thermodynamic method is used to calculate the velocity of the flow when using the two-point controller.

`'compressible_bernoulli'` Use the compressible Bernoulli equation to calculate flow conditions.

`'incompressible_bernoulli'` Use the incompressible Bernoulli equation to calculate flow conditions.

`ntf_calibration(:) = .false.`

This parameter specifies whether to add the NTF specific Mach number calibration to the two-point plenum based controller logic.

`controller_mode(:) = 'entrance_cone'`

Determines the location of the second survey point used by the two-point controller.

`'plenum'` The static pressure survey at `field_point2` is located in plenum chamber surrounding the test section.

`'entrance_cone'` The static pressure survey at `field_point2` is located upstream of the tunnel test section in the entrance cone.

`settling_chamber(:) = 'total'`

Either the total pressure or the static pressure at the first survey point can be used for determining the flow conditions when using the two-point controller.

`'total'` Use the total pressure from the settling chamber survey point.

`'static'` Use the static pressure from the settling chamber survey point.

`number_of_calibration_points = 0`

Number of calibration points.

`q(:) = 0.0`

List of dynamic pressures associated with a calibration coefficient, cprime(i).

`cprime(:) = 0.0`

List of calibration coefficients.

B.14 `fun3d.nml` inputs for SFE

To use SFE, set the `flow_solver` in the `fun3d.nml`:

```
&governing_equations
flow_solver = 'sfe'
/
```

Options for the finite-volume discretization or algorithm settings do not affect SFE, e.g., `&inviscid_flux_method` and `linear_solver_parameters`. Typical options used with SFE:

```
&project, &raw_grid, &sampling namelists
&code_run_control - steps, restart_read
&governing_equations - viscous_terms
&reference_physical_properties - mach_number, reynolds_number,
                                  temperature, angle_of_attack
```

If you have a question about what options are compatible with SFE, please reach out to `Fun3D-Support@lists.nasa.gov`.

B.15 `sfe.cfg`

The `sfe.cfg` file controls SFE-specific parameters like smoother or nonlinear controller options. The options are described in detail below with defaults listed before the descriptions. Only those variables that are different from the defaults need to be specified. Note that the array inputs like `smoother_type` are zero-based indexing unlike the namelist files.

B.15.1 Nonlinear Controller

Psuedotime stepping toward a steady-state solution is controlled by the Courant-Friedrichs-Lewy (CFL) controller. An overview of the controller logic is included in Figure 7, which depicts the major operations and control flow logic within a nonlinear step. Realizability constraints on density and temperature are enforced by reducing the maximum extent of the line search, $\omega_{realizable} <= 1$. The line search is performed using two root-means-square (RMS) measures of residuals; `p_rms` and `p_rms_check`. `p_rms` is used to determine the location (ω) of the minimum residual based on the curve fit. If this residual meets the targeted reduction value, the `p_rms_check` residual is also evaluated at this same ω. Whereas the `p_rms` residual may be indicating that it is safe to increase the CFL number, `p_rms_check` is consulted and must indicate that a reduction in the `p_rms_check` residual is achieved before the CFL number is actually increased. The default values check an unweighted residual to determine the location of the minimum residual, whereas an inverse volume weighted residual helps to ensure that residuals in cells close to the boundary are not increasing. The combined use of `p_rms` and `p_rms_check` helps in obtaining reasonably fast and reliable convergence.

```
cfl_init               =  1.0
cfl_min                =  0.10
cfl_max                =  1.0e6
cfl_multiplier         =  1.25
cfl_pause_factor       =  0.8
cfl_divisor            =  0.1
round_off_termination  =  1
round_off_tolerance    =  1.0e-12
p_rms                  =  4
p_rms_check            =  2
```

<u>`cfl_init = 1.0`</u>

Initial CFL number.

<u>`cfl_min = 0.10`</u>

Minimum CFL number.

<u>`cfl_max = 1.0e6`</u>

Maximum CFL number.

<u>`cfl_multiplier = 1.25`</u>

Multiplicative increase in CFL number applied when a full nonlinear step is taken.

<u>`cfl_pause_factor = 0.8`</u>

Multiplicative factor applied to the CFL number when a partial nonlinear step is taken.

<u>`cfl_divisor = 0.1`</u>

Multiplicative decrease in CFL number applied when a nonlinear step is rejected.

<u>`round_off_termination = 1`</u>

`'0'` no round off termination.

`'1'` nonlinear solve ends successfully when the ratio of a nonlinear step's RMS and the state's RMS is below tolerance $||\Delta Q||_{RMS}/||Q||_{RMS} <$ `round_off_tolerance`.

`round_off_tolerance = 1.0e-12`

Minimum ratio of nonlinear step RMS to state vector RMS.

`p_rms = 4`

Selects primary residual weighting for CFL controller,

where, R = nodal residual, V = nodal volume, $\Delta\tau$ = pseudo-time.

`'1'` $||R/V||_1$.

`'2'` $||R/V||_2$.

`'3'` $||R/V||_4$.

`'4'` $||R||_2$.

`'5'` $||R/(V^{1/3})||_2$.

`'6'` $||R/\Delta\tau||_2$.

`p_rms_check = 2`

Selects secondary residual weighting for CFL controller,

where, R = nodal residual, V = nodal volume, $\Delta\tau$ = pseudo-time.

`'1'` $||R/V||_1$.

`'2'` $||R/V||_2$.

`'3'` $||R/V||_4$.

`'4'` $||R||_2$.

`'5'` $||R/(V^{1/3})||_2$.

`'6'` $||R/\Delta\tau||_2$.

B.15.2 Linear algebra

The linear algebra parameters control the methods used to solve linear subproblems within steady-state calculations and the those for adjoint and LFD calculations. Two linear solver algorithms are supported, Generalized Minimum Residual (GMRES) and Flexible GMRES (FGMRES) [58,59]. GMRES is often sufficient when a stable and moderately accurate preconditoner can be formed. FMGRES is recommended for ill-conditioned problems where the range of values available from double precision arithmetic may not be sufficient to span the Krylov subspace. `krylov_dimension` sets the maximum number of search directions that will be used in the GMRES or FGMRES algorithms. The memory footprint of GMRES is $\sim$1.5$\times$ the number of mesh points $\times$ the number of equations $\times$ the number of Kyrlov dimensions $\times$ 8 bytes for real valued linear systems or 16 bytes for complex-valued linear systems. The memory footprint of FGMRES is $\sim$5/3$\times$ greater than GMRES because each preconditioned search direction is stored and used to reconstruct the solution. The creation of each search direction in (F)GMRES requires a matrix-vector product operation to be preformed. `max_matvecs` sets the limit of how many matrix-vector product operations will be preformed for each linear system. When `max_matvecs` > `krylov_dimension`, the (F)GMRES algorithms will restart (`max_matvecs` / `krylov_dimension`) - 1 times. `max_matvecs` is typically set to an integer multiple of `krylov_dimension`, but this is not a requirement.

Two preconditioning algorithms are supported, Incomplete LU factorization with level of fill (ILU(k)) and Level-set ILU(k) (LS-ILU(k)) [58, 59]. ILU(k) is a sequential algorithm that is appropriate to use with an all MPI parallelization strategy. LS-ILU(k) is a parallel algorithm implemented using OpenMP threads that is appropriate to use with a hybrid MPI+OpenMP parallelization strategy. Increasing the `level_of_fill` may improve the approximation of the Jacobian matrix's inverse provided by the preconditioner's factorization and application operations. Increasing the `level_of_fill` in combination with some matrix orderings can also lead to numeric instabilities in the preconditioner that degrade the convergence of (F)GMRES.

Two reordering algorithms are supported Cuthill-McKee (CMK) [144] and k-ordering [60]. Both of these orderings can be reversed. Q-ordering [60] can optionally be as a second step applied after the initial reordering (CMK or k-ordering) to create localized randomization within the ordering to improve stability of the preconditioner. Q-ordering is recommended for difficult linear problems, e.g., transonic LFD. `prune_width` controls the size of regions where the matrix sparsity pattern is randomized. Using smaller values of `prune_width` can stabilize a preconditioner but can increase the matrix band-width and fill-ratio of the preconditioner, both of which can increase the computational cost of a linear solve.

```
linear_solver                       =  gmres
krylov_dimension                    =  300
max_matvecs                         =  600
linear_report_interval              =  10
preconditioner                      =  iluk
level_of_fill                       =  2
reorder                             =  k-ordering
reverse                             =  .false.
q_ordering                          =  0
prune_width                         =  12
relative_linear_residual_tolerance  =  1.0e-8
absolute_linear_residual_tolerance  =  1.0e-15
```

`linear_solver = gmres`

Linear solver algorithm.

`gmres` Generalized Minimum Residual.

`fgmres` Flexible Generalized Minimum Residual.

`krylov_dimension = 300`

Dimension of Krylov search space used by linear solver.

`max_matvecs = 600`

Maximum number of matrix-vector applied by linear solver for each linear system.

`linear_report_interval = 10`

Number of linear solver iterations between convergence reports.

`preconditioner = iluk`

Preconditioning algorithm.

`iluk` Incomplete LU factorization with level of symbolic fill, k.

`lsiluk` Level-Set ILU(K).

`level_of_fill = 2`

Level of symbolic fill used in the preconditioner.

`reorder = k-ordering`

Reorder algorithm.

`cmk` Cuthill-McKee [144].

`k-ordering` Single-step chain length reduction [60].

`reverse = .false.`

Toggles reversing of ordering.

`q_ordering = 0`

Localized randomization of matrix ordeirng for chain length reduction [60].

`'0'` do not apply q-ordering.

`'1'` apply q-ordering after selected reordering algorithm.

`prune_width = 12`

Number of subdivisions of a partition's matrix's band-width used in q-ordering algorithm.

relative_linear_residual_tolerance = 1.0e-8

Maximum relative linear residual reduction, the linear solve will stop once the linear residual is reduced below this tolerance or the `absolute_linear_residual_tolerance`.

absolute_linear_residual_tolerance = 1.0e-15

Maximum absolute linear residual reduction, the linear solve will stop once the linear residual is reduced below this tolerance or the `relative_linear_residual_tolerance`.

B.15.3 Dynamic reordering

This first development in automated dynamic reordering is based on Q-ordering. The efficacy of each partition's preconditioner is assessed by monitoring the growth in the L2 norm of the partition's portion of the first Krylov vector due to the application of the preconditioner. Partitions where the `dynamic_reordering_growth_trigger` is exceeded are reordered with `prune_width` $\times=$ `dynamic_reordering_prune_factor`. Only reordering the partitions with unstable preconditioners adds robustness to the linear solve without incurring unnecessary computational costs.

```
dynamic_reordering                      =  2
dynamic_reordering_growth_trigger       =  1.0e+10
dynamic_reordering_write_linear_system  =  0
dynamic_reordering_prune_factor         =  0.75
dynamic_reordering_min_prune_width      =  1.0e-6
```

dynamic_reordering = 2

`'0'` off.

`'1'` reorder before Krylov solve if growth trigger exceeded.

`'2'` reorder if Krylov solve did not reach residual reduction target and the growth trigger is exceeded.

dynamic_reordering_growth_trigger = 1.0e+10

The threshold of acceptable growth in the L2 norm of the first Krylov vector due to the application of the preconditioner, typical value is 1.0e+10.

dynamic_reordering_write_linear_system = 0

`'0'` do not write the Jacobian matrix and residual vector of each partition where the growth trigger is exceeded to a file for offline analysis.

`'1'` write the Jacobian matrix and residual vector of each partition where the growth trigger is exceeded to a file for offline analysis.

`dynamic_reordering_prune_factor = 0.75`

Multiplicative adjustment to the size of the groups of rows and columns in the matrix that are reordered when growth trigger is exceeded, values < 1 will result in larger reordering groups, typical values are 0.5 and 0.75.

`dynamic_reordering_min_prune_width = 1.0e-6`

Minimum prune width.

B.15.4 Thermodynamics

`viscosity_model` is exposed to enable benchmark cases that require constant viscosity. `thermal_conductivity_model` is exposed to enable compatibility with the conservative-viscous metric [54] used in goal-oriented adaptation within REFINE [55].

```
viscosity_model              =  1
thermal_conductivity_model   =  0
```

`viscosity_model = 1`

This controls whether the viscosity is a function of temperature or is held constant.

`'0'` constant value.

`'1'` governed by Sutherland's Law.

`thermal_conductivity_model = 0`

This controls whether the thermal conductivity is a function of temperature or is held constant.

`'0'` constant value.

`'1'` computed from Prandtl number and Sutherland's Law.

B.15.5 Boundary conditions

Weak enforcement is typically preferred due to better convergence. Underresolved meshes that may occur early in a mesh adaptation process may lead to poor enforcement of no-slip boundary through residual penalization. Therefore, underresolved meshes may benefit from strong boundary conditions.

The flat-plate adiabatic wall recovery temperature is enforced at solid wall boundaries. This matches the default temperature enforced by the finite volume solver.

```
weak_bc = 2
```

`weak_bc = 2`

This parameter controls how no-slip viscous wall boundary conditions (BC 4000) are enforced.

`'0'` strong enforcement of no-slip through Dirichlet boundary conditions.

`'2'` weak enforcement through residual penalization

B.15.6 Turbulence model parameters

For turbulent simulations, SFE utilizes the Negative Continuation Spalart-Allmaras [61] model. The Quadratic Constitutive Relation (QCR), 2000 version [145] is the only supported augmentation. QCR is typically used for simulations where the prediction of secondary vortices common in corner flows is of importance.

```
qcr = .false.
```

`qcr = .false.`

Toggles use of Quadratic Constitutive Relation, 2000 version [145].

B.15.7 Smoothing

Smoothers locally add dissipation by augmenting the local viscosity to capture under-resolved flow features. SFE has several smoother types that use different sensors used to detect where to add dissipation: `uniform`, `uniform_ramped`, `pressure`, `entropy`, `metric_pressure`, and `metric_entropy`. Each of these smoothers can be individually adjusted through a common set of input parameters: `smoother_type(:)`, `smoother_coef(:)`, `smoother_clip(:)`, and `smoother_beta(:)`, where (:) denotes the nth smoother using 0-based indexing.

When `smoothing` = .true., default settings for the smoothing parameters are activated: `number_of_smoothers` = 1 and `smoother_type(0)` = `metric_pressure`. The `metric_pressure` smoother is appropriate for flows where local supersonic conditions may occur. The `uniform` smoother is appropriate for the initialization of high-speed flows but should be turned off for the final solution. The `uniformed_ramped` provides uniform dissipation that decreases to zero as `CFL` increases. This can improves initial convergence of difficult flows. The `pressure`, `entropy`, and `metric_entroy` smoothers are alternative formulations that are suitable for flows where local supersonic flows occur.

```
smoothing            =  .false.
number_of_smoothers  =  0
smoother_type(:)     =  metric_pressure
smoother_coef(:)     =  1.0
smoother_clip(:)     =  2.0
smoother_beta(:)     =  10.0
```

`smoothing = .false.`

Toggles smoothing.

`number_of_smoothers = 0`

Number of active smoothers.

`smoother_type(:) = metric_pressure`

Name of the nth smoother.

`'metric_pressure'` uses a feature sensor based on an element's topology and spatial pressure gradients similar to Ref [52]. Default parameter values for this smoother are `smoother_coef(0)` = 1.0, `smoother_clip(0)` = 2.0, and `smoother_beta(0)` = 10.0.

`'uniform'` does not use a sensor to detect features and determine active regions. Rather, dissipation is added uniformly throughout the volume and the amount of dissipation is set by `smoother_coef(:)` directly.

`'uniform_ramped'` reduces the amount of uniform dissipation added as the CFL increases. `smoother_coef(:)` sets the maximum amount of dissipation added.

`'pressure'` uses a feature sensor based on the local velocity vector and spatial pressure gradients. Typically, `smoother_clip(:)` = 0.05 is sufficient.

`'entropy'` uses a feature sensor based on the local velocity vector and spatial entropy gradients. Typically, `smoother_clip(:)` = 0.1 is sufficient.

`'metric_entropy'` uses a feature sensor based on an element's topology and spatial entropy gradients similar to Ref [52]. Typical parameter values for this smoother are `smoother_coef(0)` = 1.0, `smoother_clip(0)` = 2.0, and `smoother_beta(0)` = 10.0.

`smoother_coef(:) = 1.0`

Controls the magnitude of the dissipation added by the nth smoother, a typical value for this sensor is 1.0, larger values will increase the amount of dissipation added.

`smoother_clip(:) = 2.0`

Controls where the smoother is active, the default value is 2.0, larger values will reduce the regions of activity and moderately diminish the magnitude of dissipation added.

`smoother_beta(:) = 10.0`

Controls the blending between active and inactive regions of the `metric_pressure` and `metric_entropy` smoothers, the default value is 10.0, larger values will make the transition more abrupt.

B.15.8 Residual smoothing

Locally adds consistently linearized dissipation based on change in state, $\Delta\mathbf{Q}$, during the line search when `residual_smoothing` = .true.. The procedure used here is similar to that described by Kamenetskiy et al. [146] except that the residual-smoothing coefficient can be regularly toggled between two values, denoted as the primary and secondary values. This procedure is useful for avoiding stalled convergence due to local minimums during the line search procedure.

```
residual_smoothing                          =  .true.
residual_smoothing_coefficient              =  10.0
residual_smoothing_secondary_coefficient    =  50.0
residual_smoothing_switch_interval          =  5
```

`residual_smoothing = .true.`

Toggles residual smoothing.

`residual_smoothing_coefficient = 10.0`

Primary residual smoothing coefficient.

`residual_smoothing_secondary_coefficient = 50.0`

Secondary residual smoothing coefficient.

`residual_smoothing_switch_interval = 5`

Number of nonlinear steps that each residual smoothing coefficient is applied before switching to the other coefficient.

B.15.9 Code Run Control

```
time_accuracy       =  {read from fun3d.nml}
ignore_restart_cfg  =  .false.
```

`time_accuracy = {read from fun3d.nml}`

Controls the time stepping mode of SFE. By default is read from `fun3d.nml`. In general, the user should not set this variable directly in `sfe.cfg` to avoid a mismatch between the FUN3D driver and SFE. The one exception is static aeroelastic analysis where setting `time_accuracy` = 0 permits running SFE in steady mode while coupled to the unsteady modal structural solver.

`ignore_restart_cfg = .false.`

Toggles reading of `sfe_restart.cfg` during a restart.

B.15.10 Adjoint

```
adjoint               =  .false.
cost_function         =  8
use_far_field_forces  =  .false.
```

`adjoint = .false.`

Toggles adjoint solution mode.

`cost_function = 8`

output function of interest for the adjoint.

`'1,2,3'` Cx,Cy,Cz force coefficients.

`'4,5,6'` CMx,CMy,CMz, moments coefficients.

`'7,8'` CL, CD.

`use_far_field_forces = .false.`

Toggles use of far-field boundaries to calculate forces.

Calculation of forces using far-field boundaries may produce a smoother adjoint field when performing output-based mesh adaptation with strong enforcement of solid wall boundary conditions.

B.16 Yoga Input Files

Yoga has an optional ASCII input file for setting the order, node counts, and grid-imesh values for the component grids within the composite grid file. This file can be used in lieu of setting the values via namelist (as described in the&overset_data namelist section). If Yoga is used to create the composite grid, Yoga will generate this file as a by-product. For example, if Yoga is used to generate a composite grid from two component grids, where the first grid has 700 nodes and the second grid has 600 nodes, Yoga will generate a file with the default name *imesh.dat* with the following contents:

```
2
700 1
600 0
```

From this file, FUN3D will determine which nodes in the composite belong to each component grid and what *imesh* value to assign each component.

Appendix C

Troubleshooting

The goal of the FUN3D developers is to produce as robust a solver as possible. However, there are times when the code fails to produce a converged solution. For example, taking the square root of a negative quantity (due to a negative density or pressure) results in a NaN. We hope that these suggestions are helpful. If none of the suggestions listed here remedy your problem, please contact `Fun3D-Support@lists.nasa.gov`.

C.1 What if I cannot run the solver in parallel?

While the predominant way of executing FUN3D is on parallel architectures, the use of MPI for concurrent processing introduces additional complexity. It is critical that consistency is maintained between the build and execution environments for not only FUN3D but also its dependent third party libraries. The consistency holds true for maintaining the same MPI implementation and Fortran compiler across the builds and execution. When checking for consistency one might use the following in the build environment,

```
$ which mpif90     # to show the MPI installation used
$ mpif90 -show     # to show the underlying Fortran compiler used with MPI
```

In the execution environment of all hosts,

```
$ which mpirun     # to show the MPI installation used
$ which $FC        # to show the Fortran compiler and runtime
```

The MPI installations and Fortran compilers must agree for successful concurrent execution.

C.2 What if the solver has trouble starting or reports NaNs or infinities?

Check that the freestream, reference, and boundary conditions are specified correctly. Visualize the solution, especially near the location of the maximum residual. If the problem is widespread, run the simulation again and visualize the solution a few iterations before the problem happens. Look for extremely large Mach numbers, low pressures, low densities, or reversed flow at boundaries.

If the solver gave you `nan_locations_#.dat` or `inf_locations_#.dat` (infinities) files (where `#` corresponds to the processor number), visualize their locations by loading them (and your boundaries) into your plotting software, e.g., for Tecplot™, load your boundary data, and then "add," "by position," a scatter plot layer of spheres. Their locations may point to regions of poor grid quality (typically too coarse to resolve high-gradient physics) or areas where the linearized Riemann fluxes are not valid, [147] e.g., typically expansions severe enough to produce a vacuum condition during flux reconstruction. You might try one or more of the following: generate a finer grid in the affected area(s), initialize stagnant flow in the area(s) (typically wake regions),[C1] add a flux-reconstruction limiter, and/or use a different flux function.[C2]

Examining the residual history may help to isolate the problem to the mean flow or turbulence model. Lowering the CFL numbers of the mean-flow and turbulence equations can aid linear and nonlinear convergence, see section B.4.15. If the linear system is diverging (the linear system can be examined with the `--monitor_linear` command line option), utilize a more dissipative flux linearization `flux_construction_lhs` or the Krylov projection method `linear_projection=.true.`, see section B.4.19.

Try flow field initialization in the problematic region of the domain (e.g., engine plenum), see section B.4.26. Start with some `first_order_iterations` (section B.4.9) or try continuation from less challenging freestream condition (e.g., lower angle of attack, subsonic Mach number). The time accurate flow solver may be able to survive initial transients better than the steady solver.

C.3 What if the forces and moments aren't steady or residuals don't converge to steady-state?

Try lowering the CFL numbers of the mean flow and turbulence model, see section B.4.15. Examining the residual history may help to isolate the problem to the mean flow or turbulence model. The problem may be unsteady; try restarting the solution with a time-accurate simulation.

C.4 What if the solver dies unexpectedly?

You may need to set your shell limits to unlimited,

```
$ ulimit unlimited # for bash
$ unlimit          # for c shell
```

An example of dying unexpectedly is a `SIGSEGV` Signal 11 fault up to or during the first time step, or during post processing.

If your operating system reports a `SIGKILL` Signal 9 or `SIGSEGV` Signal 11 and you have already tried removing shell restrictions, you have likely hit

[C1] See section B.4.26 for flowfield initialization.
[C2] See section B.4.9 for flux and flux-limiter options.

the memory limit of your machine. Try reducing the number of mesh points, running on more nodes, or installing more memory on your machine.

C.5 What if the compiler complains, "This module file was not generated by any release of this compiler"?

This indicates that you have compiled files from multiple versions of a compiler or compilers from different vendors. Use,

```
$ make -j clean
```

to remove inconsistent files before attempting to re-compile.

C.6 What if the solver dies with an error like "Probable incomplete read of namelist: &some_namelist"?

If this message persists after carefully checking for missing quotation marks, ampersands, array indices, etc., try checking for special, non-ASCII characters, e.g., `cat -v fun3d.nml`.

C.7 What if a segmentation fault occurs after "Calling ParMETIS"?

More recent implementations of MPI (e.g., OpenMPI) internally manage the handles (i.e., communicators) differently from previous versions of MPI. However, ParMETIS 3.x uses an older paradigm, which has been updated in ParMETIS 4.x. Upgrade to ParMETIS 4.x.

C.8 What if the solver dies with an error like "input statement requires too much data" after echoing the number of tetrahedra and nodes for a VGRID mesh?

The endianness (section 4.1) of the grid files (section 4) may be different than FUN3D expects. Single-segmented VGRID grids over 20 million nodes exceed the allowable record length. Use `postgrid` to save grid as a multisegmented format (option `05` in batch mode).

C.9 What if the solver reports that the Euler numbers differ?

The Euler number is a global indicator of consistent node, element, and face connectivity. There is some limited evidence that suggests there may be times when it reports a problem that may not be an issue for the solver, but the failure of the Euler number check indicates a problem with the grid in a majority of cases. Instructions to determine if the Euler number will impact your solution follow this description of the Euler number.

C.9.1 Euler Number Description

A valid grid is composed of four elemental volume types (tetrahedra, pyramids, prisms, and hexahedra) face-connected either to each other or one of two boundary face types (triangles or quadrilaterals). Each boundary edge connects to precisely two boundary faces. Two neighboring boundary faces share exactly one boundary edge. For each boundary face connecting to a boundary node, every other boundary face connecting to the same boundary node can be found by a path through a connected-edge/connected-face traverse starting from that boundary face.

The above restrictions are meant to exclude certain topologies such as two spherical boundaries coming together at a point or two rectangular boundaries connecting along an edge. These restrictions are not checked explicitly but will cause the Euler number check described below to fail.

The Euler Number computed from boundary data (EN_b) is

$$EN_b = N_b - E_b + F_b \tag{C1}$$

where

$$N_b \equiv \text{boundary nodes (counted)} \tag{C2}$$

$$E_b \equiv \text{boundary edges (inferred from} N_{tri} \text{ and } N_{quad}) \tag{C3}$$

$$F_b \equiv \text{boundary faces (inferred from } N_{tri} \text{ and } N_{quad}) \tag{C4}$$

The Euler number is a characteristic number for the topology of the boundary or boundaries. N_{tri} and N_{quad} are the number of triangular and boundary faces, respectively.

The Euler Number computed from volume data (EN_v) is

$$EN_v = 2(N - E + F - C) \tag{C5}$$

where

$$N \equiv \text{volume nodes (counted)} \tag{C6}$$

$$E \equiv \text{volume edges (counted)} \tag{C7}$$

$$F \equiv \text{volume faces (inferred from } C \text{ and } F_b) \tag{C8}$$

$$C \equiv \text{volume cells (counted)} \tag{C9}$$

The formula that is checked is

$$EN_v - EN_b = 0 \tag{C10}$$

Barth [148] derived this formula for tetrahedra and noted the formula does not hold in certain cases, such as two simplices that share only one common edge or two simplices that share only one common node. Barth notes that the above formulas are specific forms of the general Dehn-Sommerville formula, reported in Wikipedia to hold for simplicial polytopes and simple polytopes. The pyramid is not a simple polytope. It can be proved by induction [149] that the formula holds for every valid grid as defined above.

Try this checklist to diagnose the problem:

1. Check the EN_b with your expectations for this case:

   2 for a spherical topology, simple 3D wing with symmetry, ...

   0 for a torus, donut, ...

   -2 for a double torus, ...

   -4 for a triple torus, pretzel, ...

   4 for a sphere within a sphere, ...

   6 for two spheres within a sphere, ...

2. Ensure the number of boundary nodes N_b and faces F_b reported match expected values.

3. Ensure the number of nodes N and cells C reported match expected values.

4. The difference between EN_b and EN_v points to inconsistencies in edge counts, i.e., $\delta(E) = 2(EN_b - EN_v) \neq 0$. The inequality $EN_b > EN_v$ implies you have more edges than expected. When this occurs, the reported face counts will differ from an actual count. An error of this type would arise when there are adjacent faces that are inconsistent, such as a quadrilateral face shared between two elements that is cut into two triangular faces by different edges.

C.9.2 Determining the Impact of an Euler Number Mismatch

A freestream residual problem localization technique is described. However, the best practice is to not proceed without repairing the grid to ensure $EN_b = EN_v$. The `ignore_euler_number` namelist variable does just what the name implies, and allows the solver to proceed. The `--test_freestream` option can be used as a secondary check on the mesh. On a valid mesh, the solver should preserve the freestream for an arbitrary number of iterations. You should run 20–50 iterations, which may require a lowering of the `stopping_tolerance` to 1e-20 so the solver does not automatically stop. All residuals should hover around machine zero, and not slowly increase (there will be iteration-to-iteration variation in the exact number, however).

If freestream is maintained by this test, you could proceed with your intended computation using only `ignore_euler_number`, but this is not recommended. If freestream is not maintained, this test confirms that there is a problem with the mesh and the location of the max residual may give a clue as to where in the mesh to start looking for the problem.

C.10 What if the initial residuals reported by the elasticity solver are unusually large when the input surface mesh is intended to recover the existing mesh?

This frequently occurs when parameterizing surface meshes for design optimization. Quite often the global point numbers being provided within the input surface mesh file are not consistent with the volume mesh that FUN3D is using internally. When FUN3D tries to associate coordinates from the input surface(s), it associates them with the wrong points in the global mesh. This results in what appear to be very large mesh movements (large initial residual in the elasticity system).

Plot the input surface mesh file(s) and verify that it appears as expected. If so, the global point numbers contained in the file are likely inconsistent with the global mesh file that FUN3D is using. The input surface mesh should be regenerated such that the global point numbers are consistent.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704–0188*

1. REPORT DATE *(DD-MM-YYYY)* 01-04-2023	**2. REPORT TYPE** Technical Memorandum	**3. DATES COVERED** *(From - To)*
4. TITLE AND SUBTITLE FUN3D Manual: 14.0.1	**5a. CONTRACT NUMBER**	
	5b. GRANT NUMBER	
	5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Anderson, William K.; Biedron, Robert T.; Carlson, Jan-Reneé; Derlaga, Joseph M.; Diskin, Boris; Druyor Jr., Cameron T.; Gnoffo, Peter A.; Hammond, Dana P.; Jacobson, Kevin E.; Jones, William T.; Kleb, Bil; Lee-Rausch,Elizabeth M.; Liu, Yi; Nastac, Gabriel C.; Nielsen, Eric J.; Park, Michael A.; Rumsey, Christopher L.; Thomas, James L.; Thompson, Kyle B.; Walden, Aaron C.; Wang, Li; Wood, Stephen L.; Wood, William A.; Zhang, Xinyu	**5d. PROJECT NUMBER**	
	5e. TASK NUMBER	
	5f. WORK UNIT NUMBER 109492.02.07.01.01.01	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, Virginia 23681-2199	**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA	
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/TM–20230004211	
12. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified-Unlimited Subject Category 02 Availability: NASA STI Program (757) 864-9658		
13. SUPPLEMENTARY NOTES An electronic version can be found at http://ntrs.nasa.gov.		

14. ABSTRACT

This manual describes the installation and execution of FUN3D version 14.0.1, including optional dependent packages. FUN3D is a suite of computational fluid dynamics simulation and design tools that uses mixed-element unstructured grids in a large number of formats, including structured multiblock and overset grid systems. A discretely-exact adjoint solver may be used for formal design optimization, error estimation, and mesh adaptation. FUN3D also offers a reacting, real-gas capability and provides GPU acceleration of many common simulation options.

15. SUBJECT TERMS

FUN3D; Aerodynamics; Computational fluid dynamics; Fluid Mechanics; Mathematics; Propulsion Systems

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Information Desk (help@sti.nasa.gov)
U	U	U	UU	504	**19b. TELEPHONE NUMBER** *(Include area code)* (757) 864-9658