NASA/TM–2018-220089

Sensitivity Analysis for Multidisciplinary Systems (SAMS)

Robert T. Biedron, Kevin E. Jacobson, William T. Jones, Steven J. Massey
Eric J. Nielsen, and William L. Kleb
Langley Research Center, Hampton,Virginia

Xinyu Zhang
Analytical Mechanics Associates, Inc., Hampton, Virginia

September 2018

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

- E-mail your question to help@sti.nasa.gov

- Phone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM–2018-220089

Sensitivity Analysis for Multidisciplinary Systems (SAMS)

Robert T. Biedron, Kevin E. Jacobson, William T. Jones, Steven J. Massey,
Eric J. Nielsen, and William L. Kleb
Langley Research Center, Hampton,Virginia

Xinyu Zhang
Analytical Mechanics Associates, Inc., Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

September 2018

Available from:

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

Contents

List of Acronyms, Abbreviations, and Symbols

AGARD	Advisory Group for Aerospace Research and Development
BSCW	Benchmark Supercritical Wing
CRM	Common Research Model
FSI	Fluid-Structure Interaction
KS Failure	Kreisselmier-Steinhauser Failure
SAMS	Sensitivity Analysis for Multidisciplinary Systems
SLSQP	Sequential Least Squares Quadratic Programming
PBS	Portable Batch System
TACS	Toolkit for the Analysis of Composite Structures
uCRM	undeflected Common Research Model
VIV	Vortex-Induced Vibrations
α	angle of attack
$a^*_{\inf}$	reference speed of sound (e.g., in m/s)
c	damping coefficient
δ	structural displacement
ζ	damping ratio
E	energy
f	frequency
f^*	characteristic frequency
F_a	aerodynamic force
$\hat{F}_a$	generalized aerodynamic force
h	displacement
i	loop index
Φ	state-transition matrix
Θ	convolution integral of [Φ]
D	damping matrix
I	identity matrix
K	stiffness matrix
L^*_{ref}	reference length of the physical problem (e.g., chord in ft)
L_{ref}	corresponding length in the grid (dimensionless)
M	mass matrix
n	iteration count
$q_{\inf}$	freestream dynamic pressure
Re	Reynolds number
t_{chr}	characteristic time
Δt	non-dimensional time step
x	vector of generalized displacements and velocities
ω_i	natural frequency of ith mode
$\phi_i(\mathrm{x})$	i^{th} mode shape
υ_n	series coefficient for the representation of $\dot{\mathrm{x}}$

Introduction

This report describes the research conducted under an interagency collaboration agreement between the Aerospace Systems Directorate of the Air Force Research Laboratory (AFRL/RQ) and the Computational AeroSciences Branch of NASA Langley (NASA LaRC). Both organizations have a long-term goal of developing a modular computational system for coupling fluids and structures to enable both analysis and optimization of aerospace vehicles. Ultimately, the system should support multiple solvers within the fluid and structure domains to allow the best combination for the task at hand, as well as to allow for institutional preferences of specific software components. Towards this goal, the current research was focused on enhancing the existing modal aeroelastic analysis in the NASA FUN3D software (Biedron et al. 2018), as well as developing new aeroelastic analysis and optimization capabilities based on a non-linear finite-element method. The methods and enhancements described in this document pertain to FUN3D Version 13.4.

Enhancements to existing capabilities include:

- modularization of the modal solver and modal fluid-structure transfer routines
- modal sensitivity analysis via complex variables

Completely new capabilities include:

- steady and unsteady FEM-based aeroelastic analysis
- coupled CFD-FEM adjoint-based sensitivities for design optimization

Modularization of the modal aeroelastic routines breaks the traditional tightly-integrated software design paradigm of FUN3D, and allows for these software components to be used outside of the FUN3D ecosystem.

Capabilities for FEM-based aeroelastic analysis and sensitivity analysis were leveraged from work done by Georgia Tech under a NASA NRA entitled, An Efficient Scalable Framework for Aeroelastic Analysis and Adjoint-based Sensitivities Using FUN3D and TACS (NNX15AU22A). The FUNtoFEM (Kiviaho et al. 2017) framework was developed to couple FUN3D with the structural solver TACS (Kennedy and Martins 2014). In addition to FUNtoFEM, a Hermes-based (Snyder 2017) client-server was developed to transfer loads and displacements to and from the flow solver and the structural solver.

Three benchmark aeroelastic test cases were added to the FUN3D test suite which include: Vortex Induced Vibration (VIV) of a cylinder, the Benchmark Supercritical Wing, (BSCW), and the AGARD 445.6 wing (see Appendix A).

Modal Aeroelastic Analysis Formulation in FUN3D

The coupled linear structural dynamics equations can be written as

$$[\mathrm{M}]\ddot{\delta}+[\mathrm{D}]\dot{\delta}+[\mathrm{K}]\delta=\mathrm{F}_a \tag{1}$$

where $[\mathrm{M}]$ is the mass matrix, $[\mathrm{D}]$ the damping matrix, $[\mathrm{K}]$ the stiffness matrix, $\delta(\mathrm{x},t)$ the displacement, and $\mathrm{F}_a(t)$ the loading vector, here assumed to be from aerodynamic forces only. The mass and stiffness matrices are diagonal matrices. In this implementation it is assumed that the damping matrix is also diagonal. The displacements are written as an expansion in terms of natural vibration modes $\{\phi_i(\mathrm{x})\}$

$$\delta=\sum_{i=1}^{N_{\mathrm{modes}}}\mathrm{q}_i(t)\phi_i(\mathrm{x}),$$

where the coefficients of the series, $\{\mathrm{q}_i\}$, are referred to as the generalized coordinates. The vibration modes have associated natural frequencies $\{\omega_i\}$ and are orthonormalized, so that $\phi_i^T[\mathrm{M}]\phi_i=[\hat{\mathrm{M}}]$, where $[\hat{\mathrm{M}}]$ is the generalized mass matrix. Substitution of the series representation into Eq. (1) and multiplying by ϕ^T yields

$$[\hat{\mathrm{M}}]\ddot{q}+[\hat{\mathrm{D}}]\dot{q}+[\hat{\mathrm{K}}]q=\phi^T\mathrm{F}_a=\hat{\mathrm{F}}_a, \tag{2}$$

where $\hat{\mathrm{F}}_a$ is the vector of generalized aerodynamic forces, $[\hat{\mathrm{D}}]$ the generalized damping matrix, and $[\hat{\mathrm{K}}]$ the generalized stiffness matrix. The system represented by Eq. (2) can be added to the identity system of equations $[\mathrm{I}]\dot{q}+[0]\ddot{q}+[0]q-[\mathrm{I}]\dot{q}=[0]$, and the second derivatives may be converted to first derivatives through the substitution $\mathrm{x}=(\mathrm{q},\dot{\mathrm{q}})^T$ to give the system

$$\begin{bmatrix}\mathrm{I} & 0\\ 0 & \hat{\mathrm{M}}\end{bmatrix}\dot{\mathrm{x}}+\begin{bmatrix}0 & -\mathrm{I}\\ \hat{\mathrm{K}} & \hat{\mathrm{D}}\end{bmatrix}\mathrm{x}=\begin{bmatrix}0\\ \hat{\mathrm{F}}_a\end{bmatrix}=\mathrm{u}(t)$$

This may be cast as a system of first order ordinary differential equations, one for each mode, subject to time-dependent forcing terms

$$\dot{\mathrm{x}}+\mathrm{Ax}=\mathrm{Bu}(t) \tag{3}$$

where

$$\mathrm{A}=\begin{bmatrix}0 & -\mathrm{I}\\ \hat{\mathrm{M}}^{-1}\hat{\mathrm{K}} & \hat{\mathrm{M}}^{-1}\hat{\mathrm{D}}\end{bmatrix}\qquad \mathrm{B}=\begin{bmatrix}\mathrm{I} & 0\\ 0 & \hat{\mathrm{M}}^{-1}\end{bmatrix} \tag{4}$$

The solution of Eq. (3) is performed using the **`libmodalstructure`** library described in the next section, with assistance from **`libmodalfsi`** to evaluate u(t) based on the flow solution from a CFD solver such as FUN3D. The library can solve Eq. (3) using either a 2nd-order predictor-corrector method (recommended) or a family of backward-difference schemes from 1st to 3rd-order. The predictor-corrector scheme is described in Biedron and Thomas (2009), while the backward-difference schemes are the same as used

within the FUN3D flow solver, described in Biedron et al. (2005). The predictor-corrector scheme is recommended for two reasons: 1) a long history of use for a range of aeroelastic applications, and 2) limited testing of the backward-difference schemes indicate that additional fluid-structure subiterations are required to attain the same solution that the predictor-corrector scheme provides without the cost of additional FSI subiterations.

Static (steady-state) solutions can be obtained from Eq. (3) with large damping values and large time steps. Eq. (3) is solved in dimensional form within **`libmodalstructure`**, so the calling application must provide inputs in dimensional form.

Modularization

The original modal solver in FUN3D was not implemented in such a way as to be an independent piece of software, callable by flow solvers other than FUN3D. Furthermore, the solution of the linear (modal) structural dynamics equations (Eq. 1) was entangled with the necessary, but independent, processes of transferring loads from the flow solver to the modal solver, and transferring displacements from the modal solver to the flow solver. As a part of the SAMS effort, two subroutine libraries were developed. The first, **`libmodalstructure`**, solves the linear structural dynamics equations, given inputs such as mode shapes, mode frequencies, and time step. The second, **`libmodalfsi`**, orchestrates the transfer of force and displacement data. In this manner, the solution of a linear structural dynamics problem with FUN3D is more closely aligned with the solution of a nonlinear dynamics problem using, for instance, TACS and FUNtoFEM.

Both **`libmodalstructure`** and **`libmodalfsi`** are simply collections of subroutines with defined interfaces. Each library has an **`include`** subdirectory with a **`.inc`** file defining the interfaces. The interfaces contain what might be termed as "plain old data" (POD) within the FORTRAN programming language: integers, logicals, character strings, and reals. No derived types are used, although arrays of POD with rank 1 and 2 are. Subroutines fall into three categories – "setters", "getters", and "actions". Within **`libmodalstructure`**, subroutine names begin with **`struc_`**, while within **`libmodalfsi`**, subroutine names begin with **`fsi_`**. In both libraries, setter routine names contain **`set_`** and getter routines contain **`get_`**. Action routines follow a less defined naming convention, but an attempt has been made to provide each with a descriptive name. Thus, for example, **`struc_set_nmode`** provides the number of modes to use within the modal solver, while **`fsi_get_fluid_movement`** retrieves the movement (displacement, velocity, and acceleration) of the fluid side of the fluid-structure interface. A call to **`struc_update_solution`** will update the solution of the linear structural dynamics equations, and a call to **`fsi_fld_to_str_force_xfer`** will transfer forces from the fluid side of the interface to the structure side. Note: **`libmodalfsi`** is not intended to be a general fluid-structure transfer package like FUNtoFEM. A simplifying assumption for **`libmodalfsi`** is that both sides of the fluid-structure interface must be defined by the same points in space. The points need not be ordered the same on each side of the interface however.

Both libraries are designed to operate in parallel, with a portion of the entire fluid-structure interface residing on a given processor. Parallelism is obtained via MPI, and specifically the MPI wrappers that are provided by FUN3D's **`lmpi`** library – the use of these MPI wrappers are the last entanglement between FUN3D and **`libmodalstructure`** or **`libmodalfsi`**. Both libraries may be "complexified" (i.e., converted to source code that uses complex arithmetic rather than real-valued arithmetic) to allow the evaluation of arbitrarily-accurate derivatives of the modal response (generalized displacement, velocity, acceleration and force). Complex versions of both libraries are built when the FUN3D source code is complexified. The pseudo code presented in Appendix B illustrates the code flow required to use these new libraries.

Steady and Unsteady Modal Aeroelastic Sensitivity Analysis

Analysis

This section provides an example of running an aeroelastic analysis simulation with the FUN3D modal solver. The AGARD 445.6 test case is used in this example.

Steady Simulation

A steady-state solution is needed for both the static and the dynamic aeroelastic analysis. The inputs are defined in the namelist file (**`fun3d.nml`**) and as command-line options in the Portable Batch System (PBS) script (**`sub.fun3d`**).

Inputs

1. Grid file (**`agard1pw.b8.ugrid`**)
2. Boundary conditions definition file (**`agard1pw.mapbc`**)
3. FUN3D namelist file (**`fun3d.nml`**)
4. Script for running FUN3D on K-cluster (**`sub.fun3d`**). This file will need to be edited depending on the computing platform and to set command-line options (CLO), environment variables, paths, and so forth.

Running the Simulation

1. Create a soft link to **`*.ugrid`** and **`*.mapbc`** files in the **`Steady`** directory.

```
cd Steady
ln -s ../Grids/agard1pw.* .
```

2. Copy the FUN3D namelist file for the steady run into the **`Steady`** directory:

```
cp ../InputsScripts/fun3d.nml_Steady fun3d.nml
```

3. Copy the PBS script file for the steady run into the **`Steady`** directory:

```
cp ../InputsScripts/sub.fun3d_Steady sub.fun3d
```

   Make sure that the paths are set correctly in the PBS script.

4. Submit the job:

```
qsub sub.fun3d
```

Outputs of Interest

1. Time history file (**`agard1pw_hist.dat`**). To verify if the solution has reached steady state or not.
2. Restart file (**`agard1pw.flow`**). Needed to start the dynamic simulation.
3. Mapping files (**`agard1pw_massoud_body1.dat`**, **`agard1pw_ddfdrive_body1.dat`**). FUN3D surface connectivity information used in generating mode shapes. In this example it is assumed that the mode files are already provided in the **`Modes`** directory. The user only needs the restart file after running this step.

It is assumed that the user is already familiar with the basic steady-state, time-dependent, and dynamic-mesh solver operations and controls, especially related to deforming meshes, as well as basic flow visualization of FUN3D output. Please refer to FUN3D user's manual (Biedron et al. 2018) for details on these topics.

A portion of the **fun3d.nml** file used for running the steady case is shown here. Input parameters relevant to the aeroelastic simulations are highlighted in blue. Please note that the **.moving_grid.** is set to **false**.

fun3d.nml (steady run)

```
! ------------------------------------------------------------------------------
! - fun3d namelist file (agard steady run)
! ------------------------------------------------------------------------------

&project
  project_rootname = 'agard1pw'
/
&raw_grid
  grid_format   = 'aflr3'
  patch_lumping = 'family'
/
&governing_equations
  viscous_terms = 'inviscid'
/
&reference_physical_properties
  mach_number = 0.9
/
&force_moment_integ_properties
  area_reference  = 548.0
  x_moment_length = 22.0
  y_moment_length = 30.0
  x_moment_center = 3.0
/
&nonlinear_solver_parameters
  schedule_cfl = 100.0  100.0
/
&code_run_control
  steps              = 1
  restart_write_freq = 200
  restart_read       = 'on'   ! restarting from a previous run
/
&massoud_output
  n_bodies         = 1
  nbndry(1)        = 1
  boundary_list(1) = '3'      ! note: this is the numbering after lumping
/
&global
    moving_grid = .false.
    boundary_animation_freq = -1
/
```

The **&massoud_output** namelist is used to generate output for providing the interface with geometry parameterization software.

```
&massoud_output
  n_bodies         = 1        ! parameterize one body
  nbndry(1)        = 1        ! # of bounds, which comprise body 1
  boundary_list(1) = '3'      ! note: this is the numbering after lumping
/
```

Other options under this namelist are described in the FUN3D user's manual. In addition to the **&massoud_output** namelist, the command-line option **--write_aero_loads** is specified when running the simulation. The aero loads output is written to **[project]_ddfdrive_bodyN.dat** file (Tecplot™ ASCII format by default).

Most aeroelastic problems require an integer tag that maps a surface point to the corresponding volume grid and this tag needs to be preserved throughout any manipulation (e.g., when the surface geometry is updated or mode shapes are mapped onto the surface). The command-line option **--write_massoud_file** needs to be specified in order to write this file. The output is written to **[project]_massoud_bodyN.dat** (Tecplot™ ASCII format by default):

agard1pw_massoud_body1.dat

```
title="surface points and l2g id for massoud"
variables="x","y","z","id"
zone t="mdo body 1", i=20497, j=40792, f=fepoint, solutiontime= 0.2203000E+04,
strandid=0
   0.219960000000000E+002  0.000000000000000E+000  0.000000000000000E+000          1
   0.000000000000000E+000  0.000000000000000E+000  0.000000000000000E+000          2
   0.463800000000000E+002  0.300000000000000E+002  0.000000000000000E+000          7
   0.318840000000000E+002  0.300000000000000E+002  0.000000000000000E+000          8
   0.728296446815993E-001  0.685262121207643E-001  0.000000000000000E+000         89
   0.150486858624087E+000  0.141594735155298E+000  0.000000000000000E+000         90
   0.233265620261247E+000  0.219482176472447E+000  0.000000000000000E+000         91
[…]
```

A portion of the **sub.fun3d** file used for running the steady case is shown here. Input parameters relevant to the aeroelastic simulations are highlighted in blue.

sub.fun3d (steady run)

```
# ------------------------------------------------------------------------------
# - pbs script for running fun3d on K-cluster
# ------------------------------------------------------------------------------

#PBS -S /bin/csh
#PBS -q K3-standard
#PBS -N agard_steady
#PBS -r n
#PBS -j oe

#PBS -l select=2:ncpus=12:mpiprocs=12
#PBS -l walltime=1:00:00

# ------------------------------------------------------------------------------
# - set environment variables
# ------------------------------------------------------------------------------

setenv F_UFMTENDIAN big
setenv FUN3D_TUTORIALS /lustre2/hpnobackup1/nnahmad/fun3d/aeroelastic/AGARD_445_6/
setenv WORKDIR $FUN3D_TUTORIALS/flow_modal_aeroelasticity/Steady

# ------------------------------------------------------------------------------
# - load modules
# ------------------------------------------------------------------------------

source /usr/local/pkgs/modules/init/tcsh
module purge
module use --append /u/shared/fun3d/fun3d_users/modulefiles
module load MASSOUD/2.2.1
module load FUN3D/13.1
module add PORT_1.0

# ------------------------------------------------------------------------------
# - go to work directory and run fun3d
# ------------------------------------------------------------------------------

cd $WORKDIR

mpiexec nodet_mpi --write_aero_loads_to_file --write_massoud_file
```

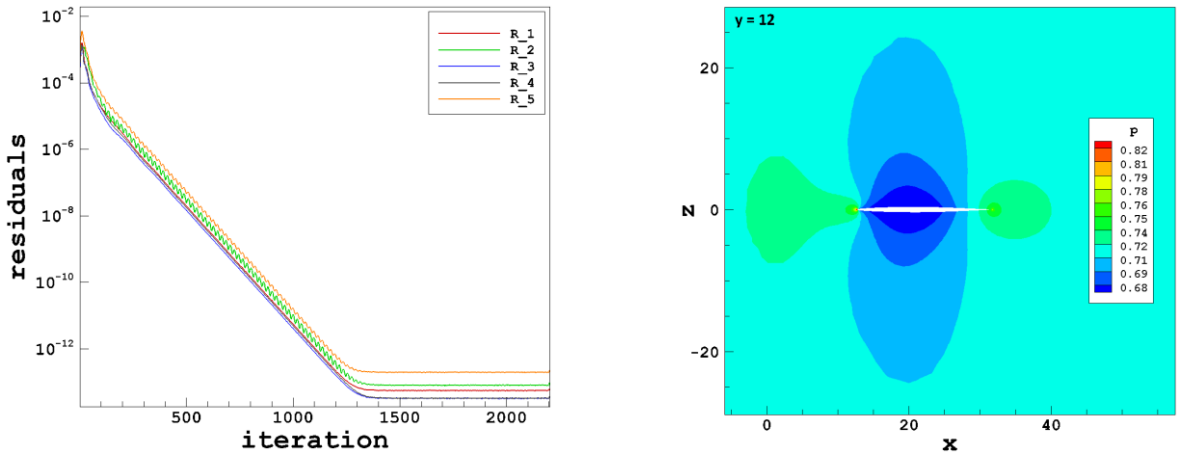


Figure 1: AGARD 445.6 wing; Mach number = 0.9; α = 0. Inviscid solution. Residuals (left); and the pressure field in the *xz*-plane at *y* = 12 (right).

Static Aeroelastic Simulation

For the general case, the next step would be to run a static aeroelastic simulation in a manner very similar to the dynamic simulation described in the next section, but with the critical damping ratio for each mode set to a value close to one (e.g. 0.99), and a large time step. However, the AGARD 445.6 wing is symmetrical, so that the static aeroelastic deflection will be zero. In practice, small asymmetries in the mesh and solution algorithm will result in a very small static deflection, but since these are small, the intermediate static aeroelastic step is ignored here.

Dynamic Simulation

Inputs for and the outputs from running the dynamic aeroelastic simulation are described next. The inputs are defined in the namelist file (`fun3d.nml`), the `moving_body.input` file, and as command-line-options in the PBS script (`sub.fun3d`).

Inputs

1. Grid file (`agard1pw.b8.ugrid`).
2. Boundary conditions definition file (`agard1pw.mapbc`).
3. Restart file (`agard1pw.flow`) from the `Steady` run.
4. Mode shape files (`agard1pw_body1_*.dat`) from the `Modes` directory. See Figure 2.
5. FUN3D namelist file (`fun3d.nml`).
6. `moving_body.input` file.
7. Script for running FUN3D on K-cluster (`sub.fun3d`). This file will need to be edited depending on the computing platform and to set environment variables, paths, and so forth.

Running the Simulation

1. Create a soft link to `*.ugrid` and `*.mapbc` files in the `Dynamic` directory.

```
cd Dynamic
ln -s ../Grids/agard1pw.* .
```

2. Copy input files into the `Dynamic` directory:

```
cp ../InputsScripts/fun3d.nml_Dynamic fun3d.nml
cp ../InputsScripts/sub.fun3d_Dynamic sub.fun3d
cp ../InputsScripts/moving_body.input .
```

```
cp ../Steady/agard1pw.flow .
cp ../Modes/agard1pw_* .
```

3. Submit the job:

```
qsub sub.fun3d
```

Outputs of Interest

1. The time histories of generalized displacement, force, etc. (**`aehist_body1_mode[1-4].dat`**). The time histories of generalized displacements from these files are plotted in Figure 3.
2. Surface files (**`agard1pw_tec_boundary_timestep*.szplt`**). The output frequency is specified in the namelist file.

A portion of the **`fun3d.nml`** file used for running the dynamic case is shown here. Input parameters relevant to the aeroelastic simulations are highlighted in blue. The **`.moving_grid.`** option is set to **`true`**.

fun3d.nml (dynamic simulation)

```
! ------------------------------------------------------------------------------
! - fun3d namelist file (agard dynamic run)
! ------------------------------------------------------------------------------

&project
  project_rootname = 'agard1pw'
/
&raw_grid
  grid_format   = 'aflr3'
  patch_lumping = 'family'
/
&global
  moving_grid = .true.
/
&governing_equations
  viscous_terms = 'inviscid'
/
&reference_physical_properties
  mach_number = 0.9
/
&force_moment_integ_properties
  area_reference  = 548.0
  x_moment_length = 22.0
  y_moment_length = 30.0
  x_moment_center = 3.0
/
&nonlinear_solver_parameters
  time_accuracy        = '2ndorder'
  time_step_nondim     = 3.6
  subiterations        = 25
  schedule_cfl         = 50.0  50.0
  temporal_err_control = .true.
  temporal_err_floor   = 0.01
/
&code_run_control
  steps              = 10000
  restart_write_freq = 1000
  restart_read       = 'on_nohistorykept'
/
&special_parameters
  large_angle_fix = 'on'
/
```

In the AGARD case, it is known from experiment that the flutter frequency at Mach 0.9 is ω* ~120 rad/s. Therefore, we need to resolve at least up to this frequency. This determines the nondimensional time step (**`time_step_nondim`**) in the **`fun3d.nml`** for the dynamic simulation.

$$t_{chr} = \left(\frac{1}{f^*}\right) a^*_{\inf} \left(\frac{L_{ref}}{L^*_{ref}}\right) = \left(\frac{2\pi}{\omega^*}\right) a^*_{\inf} \left(\frac{L_{ref}}{L^*_{ref}}\right), \tag{5}$$

where, $\frac{L_{ref}}{L^*_{ref}} = 1$ and $a^*_{\inf} = \frac{U^*_{\inf}}{Ma} = \frac{11680.8}{0.9} = 12978.67$ in/s. $U^*_{\inf}$ is specified in the **`moving_body.input`** file.

$$t_{chr} = \left(\frac{2\pi}{\omega^*}\right) a^*_{\inf} \left(\frac{L_{ref}}{L^*_{ref}}\right) = \left(\frac{2\pi}{120}\right) \cdot 12978.67 = 679.56\,. \tag{6}$$

If we need 200 steps to resolve this frequency, then

$$\Delta t = \frac{t_{chr}}{N} = \frac{679.56}{200} = 3.39\,. \tag{7}$$

The nondimensional time step of 3.6 is used for the dynamic simulation. In practice, a time step refinement study will be required to verify if the specified time step is adequate.

`moving_body.input (dynamic simulation)`

```
! ------------------------------------------------------------------------------
! - moving_body.input file (agard dynamic run)
! ------------------------------------------------------------------------------

&body_definitions
      n_moving_bodies = 1          ! define bodies as collection of surfaces
         body_name(1) = 'airfoil'  ! identifier
  n_defining_bndry(1) = -1         ! use all solid surfaces
     motion_driver(1) = 'aeroelastic'
     mesh_movement(1) = 'deform'
/
&aeroelastic_modal_data
    plot_modes = .true.            ! write back mode shapes for verification
      nmode(1) = 4                 ! 4 modes for this body
       uinf = 11680.8              ! free stream velocity (in/s)
       qinf = 0.52083              ! free stream dynamic pressure (psi)
     freq(1,1) =  60.3135016       ! mode 1 frequency (rad/s)
     freq(2,1) = 239.7975647       ! mode 2 frequency (rad/s)
     freq(3,1) = 303.7804433       ! mode 3 frequency (rad/s)
     freq(4,1) = 575.1924565       ! mode 4 frequency (rad/s)
  gmass(1:4,1) = 4*1.0             ! generalized mass (nondimensional)
  gvel0(1:4,1) = 4*0.1             ! nonzero initial velocity to kick off dynamic
/                                  ! response. should be set to zero for restart.
```

A portion of the **sub.fun3d** file used for running the dynamic case is shown here. Parameters relevant to the aeroelastic simulations are highlighted in blue. The command-line option **--aeroelastic_internal** is needed to run FUN3D for aeroelastic analysis.

sub.fun3d (dynamic simulation)

```
# ------------------------------------------------------------------------------
# - pbs script for running fun3d on K-cluster
# ------------------------------------------------------------------------------

#PBS -S /bin/csh
#PBS -q K3-standard
#PBS -N agard_dynamic
#PBS -r n
#PBS -j oe

#PBS -l select=2:ncpus=12:mpiprocs=12
#PBS -l walltime=12:00:00

# ------------------------------------------------------------------------------
# - set environment variables
# ------------------------------------------------------------------------------

setenv F_UFMTENDIAN big
setenv FUN3D_TUTORIALS /lustre2/hpnobackup1/nnahmad/fun3d/aeroelastic/AGARD_445_6/
setenv WORKDIR $FUN3D_TUTORIALS/flow_modal_aeroelasticity/Dynamic

# ------------------------------------------------------------------------------
# - load modules
# ------------------------------------------------------------------------------

source /usr/local/pkgs/modules/init/tcsh
module purge
module use --append /u/shared/fun3d/fun3d_users/modulefiles
module load FUN3D/13.1

# ------------------------------------------------------------------------------
# - go to work directory and run fun3d
# ------------------------------------------------------------------------------

cd $WORKDIR

mpiexec nodet_mpi --aeroelastic_internal
```

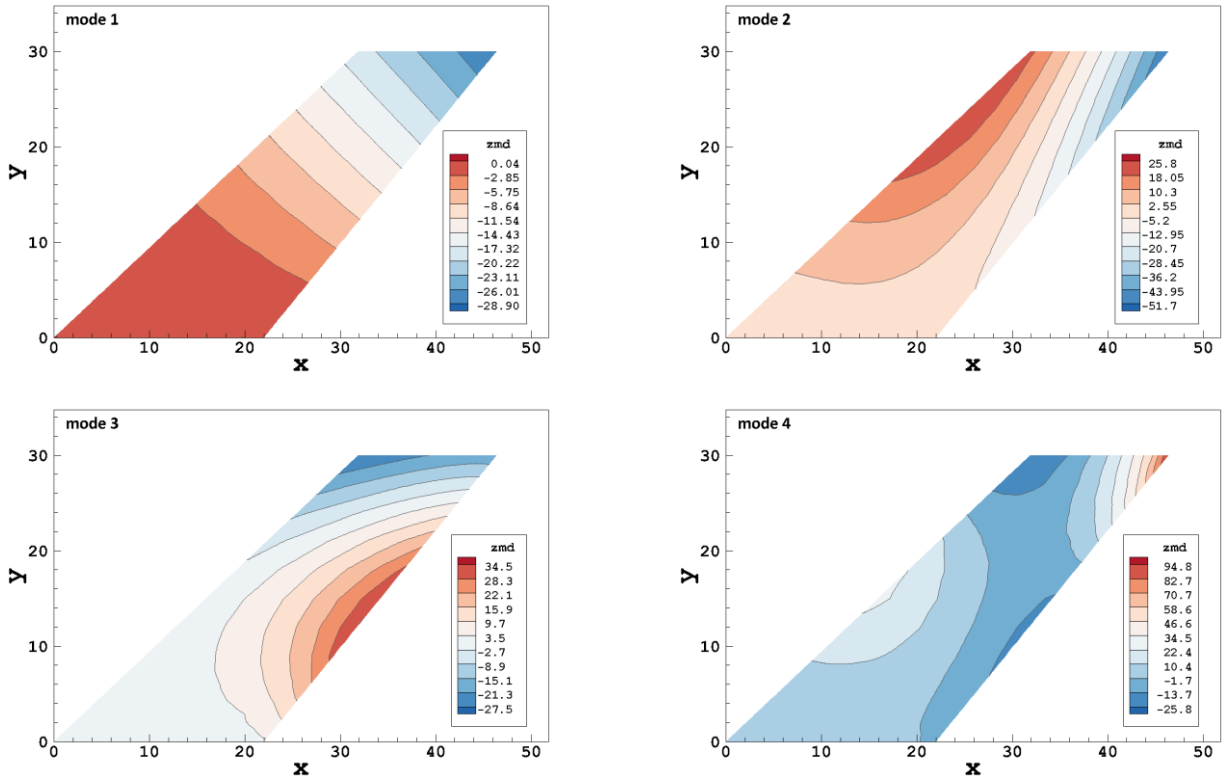


Figure 2: AGARD 445.6 wing. First bending mode (top left); first torsion mode (top right); second bending mode (bottom left); second torsion mode (bottom right).

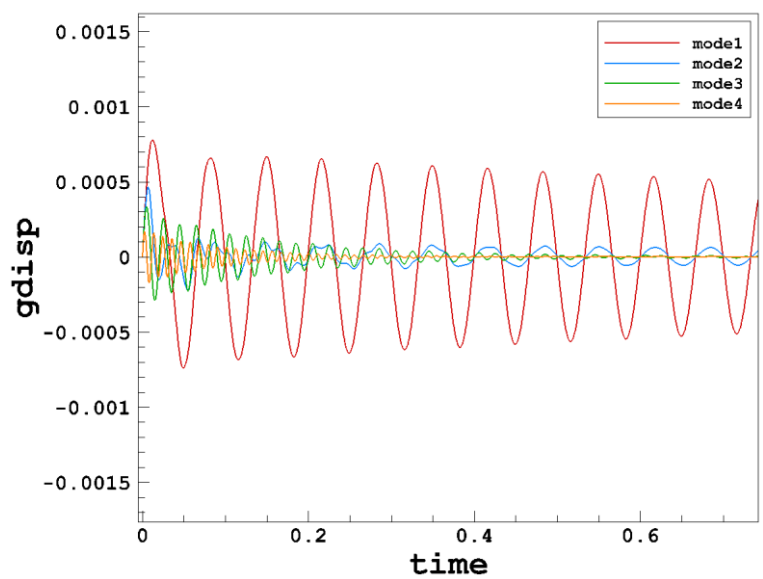


Figure 3: AGARD 445.6 wing; Mach number = 0.9; α = 0. Inviscid solution. The aeroelastic response is shown in the time history of generalized displacements.

Sensitivities via Complex Variables

Use of complex step method for estimating derivatives was first proposed by Lyness and Moler (1967) and first used for aerodynamic applications by Newman et al. (1998). The method has been described and evaluated in detail in the past (Anderson et al. 1999; Squire and Trap 1998). Consider a function f of a real variable x, and perturb x by a small positive and small negative value h:

$$f(x+h) = f(x) + hf'(x) + h^2 f''(x) + O\left(h^3\right), \tag{8}$$

and,

$$f(x-h) = f(x) - hf'(x) + h^2 f''(x) - O\left(h^3\right). \tag{9}$$

Subtracting Eq. (9) from Eq. (8),

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O\left(h^2\right). \tag{10}$$

Eq. (10) is the classic central difference accurate to $O(h^2)$. Note however, that if the positive and negative perturbations and are very close to each other – as is desired for high accuracy – then the numerical results using finite-precision arithmetic may suffer from subtractive errors not accounted for in the $O(h^2)$ truncation error. However, if for a function of a complex variable $x + ih$, where h is again a small step, one can write

$$f(x+ih) = f(x) + ihf'(x) - h^2 f''(x) - O\left(ih^3\right). \tag{11}$$

Taking the imaginary part gives

$$f'(x) = \mathrm{Im}\left[f(x+ih)\right] - O\left(ih^2\right). \tag{12}$$

The derivative in Eq. (12) is also accurate to $O(h^2)$, but is not subject to subtractive error. Thus, h may be taken to be very small, yielding essentially exact derivatives. In practice, complex step sizes of 10^{-20} or even 10^{-50} are used. The simple idea above may be applied to software, which in effect provides an output f given an input x. Thus, by creating a complex-arithmetic version of the code, providing some facility to provide the desired input x with a small complex perturbation ih, taking the imaginary part of the desired output f, and finally dividing by the step size, gives the desired derivative.

The FUN3D build environment has the infrastructure to create a complex version of itself and its associated libraries. The complex versions of **`libmodalstructure`** and **`libmodalfsi`** are automatically generated when a complex version for FUN3D is compiled. Only minor modifications to the libraries were required to accommodate the complex arithmetic beyond what is already provided by the FUN3D complexification process, principally regarding output formatting. A flag, **`complex_mode`**, can be passed to **`libmodalstructure`** during initial set up via the "setter" function **`struc_set_complex_mode`**. When this flag is set to **`.true.`** (default is **`.false.`**) and the complex version is used, in addition to the usual output of modal response (i.e. the **`aehist`** files), the imaginary part of the modal response is also output, with a suffix **`_imaginary`** added to the file name to distinguish from the corresponding real part. The values in the **`_imaginary`** file(s) are *not* divided by the step size. To obtain the actual derivatives, the user must perform that simple operation.

Please note that the **`libmodalstructure`** does not have any procedure to specify the complex step size. If the user wishes to employ the complex version to generate sensitivities of the modal response with respect to say, the freestream dynamic pressure (q_{inf}), the value of q_{inf} that is passed to the library via the call to

struc_set_qinf must already contain the complex perturbation. In other words, it is up to the calling application to add a complex perturbation to the input variable of interest. To this end, FUN3D has been modified to allow complex perturbations to the following: freestream velocity, freestream dynamic pressure, the critical structural damping coefficient of any mode, the frequency of any mode, the generalized mass of any mode, the initial generalized velocity of any mode, and the initial generalized displacement of any mode. These perturbation options are defined in more detail in the FUN3D Manual, in the **&aeroelastic_modal_data** namelist documentation.

The **perturb.input** file (shown below) is required when running FUN3D in the complex mode. For aeroelastic sensitivities the user needs this file in the current working directory with the input **PERTURB** set to zero and **EPSILON** set to the desired complex step size. Aeroelastic sensitivity inputs are specified in the **moving_body.input** file.

perturb.input (complex run)

```
! ------------------------------------------------------------------------------
! - perturb.input file
! ------------------------------------------------------------------------------

   PERTURB   EPSILON GRIDPOINT
         0    1.e-20       666

0 = No perturbation
1 = Mach number
2 = Alpha
3 = Shape

! ------------------------------------------------------------------------------
! - end of file
! ------------------------------------------------------------------------------
```

moving_body.input (complex run)

```
! ------------------------------------------------------------------------------
!  - moving_body.input file
! ------------------------------------------------------------------------------

&body_definitions
      n_moving_bodies = 1          ! define bodies as collection of surfaces
         body_name(1) = 'airfoil'  ! identifier
  n_defining_bndry(1) = -1         ! use all solid surfaces
     motion_driver(1) = 'aeroelastic'
     mesh_movement(1) = 'deform'
/
&aeroelastic_modal_data
  modal_ae_complex_to_perturb = 3  ! 1=qinf,  2=uinf, 3=damp, 4=freq, 5=gmass,
                                   ! 6=gvel0, 7= gdisp0
  modal_ae_mode_to_perturb = 2     ! #3-7 need a mode specified
  modal_ae_body_to_perturb = 1     ! #3-7 need a mode specified
  uinf  = 973.4,
  grefl = 1.00,
  qinf  = 75.0,
  nmode(1)  = 4,
  freq(1,1) = 60.3135016    ! index 1: mode number index 2: body number
  freq(2,1) = 239.7975647,
  freq(3,1) = 303.7804433,
  freq(4,1) = 575.1924565,
  gmass(1,1)  = 1.0,
  gmass(2,1)  = 1.0,
  gmass(3,1)  = 1.0,
  gmass(4,1)  = 1.0,
  damp(1,1)  = 0.999,
  damp(2,1)  = 0.999,
  damp(3,1)  = 0.999,
  damp(4,1)  = 0.999,
  gvel0(1,1) = 0.0 ! 0.1    ! nonzero only first time
  gvel0(2,1) = 0.0 ! 0.1
  gvel0(3,1) = 0.0 ! 0.1
  gvel0(4,1) = 0.0 ! 0.1
/
```

When running FUN3D in the complex mode, in addition to the **aehist_bodyN_modeN.dat** file(s), the aeroelastic sensitivity data is written to **aehist_bodyN_modeN_imaginary.dat** file(s). As mentioned earlier, the imaginary part given in the **aehist_bodyN_modeN_imaginary.dat** file(s) needs to be divided by the complex step size (10^{-20} in the current example) in order to obtain the derivatives.

aehist_body1_mode4_imaginary.dat

```
variables = "time", "Im(gdisp)", "Im(gvel)", "Im(gforce)" "Im(gaccel)"
zone t = "modal history for body   1, mode   4"
   0.000000000E+00    0.000000000E+00    0.000000000E+00    0.000000000E+00    0.000000000E+00
   2.588863776E-02    9.254199952E-28    2.546329696E-30    5.102685639E-23    3.061720055E-22
   5.177727552E-02    1.898992484E-27    2.678811711E-30    5.368168541E-23    1.592815974E-23
[…]
```

Compiling FUN3D in Complex Mode

Steps for compiling FUN3D in complex mode are described in this section. The selection of compilers is based on Langley's K-cluster. The users will need to modify this selection based on their computing environment and available resources.

Compiling FUN3D in Complex Mode

```
cd fun3d

. /usr/local/pkgs/modules/init/bash
module purge
module load intel_2017.2.174
module load mpt-2.14

module use --append /u/shared/fun3d/fun3d_users/modulefiles
module load hdf5/1.8.17-mpt-2.14-intel_2017.2.174
module load Suggar++/2.5.1-mpt-2.14-intel.2017.2.174
module load cgnslib/3.3.0-mpt-2.14-intel_2017.2.174

./bootstrap

mkdir Complex_mpi_build
cd Complex_mpi_build/

../configure \
 --prefix=$PWD \
 --enable-ftune \
 --enable-complex \
 --enable-full-precision \
 --with-mpi=/opt/sgi/mpt/mpt-2.14 \
 --with-mpiexec=mpiexec_mpt \
 --with-parmetis=/u/shared/fun3d/fun3d_users/modules/ParMETIS/4.0.3-mpt-2.14-intel_2017.2.174 \
 --with-SPARSKIT=/u/shared/fun3d/fun3d_users/modules/SPARSKIT/2-mpt-2.14-intel_2017.2.174/lib \
      FCFLAGS="-fp-model precise"

make -j 8
make install
make -j 8 complex
make install
```

Verification of the Complex-Step for Modal Sensitivities

In this section, the "complexifed" versions of **`libmodalstructure`** and **`libmodalfsi`** are used to compute the sensitivities (derivatives) of several quantities of interest, and the results are verified against the more traditional finite-difference approach. Central differences are used for all finite-difference results. Both methods have errors that scale as the square of the step size, and as noted earlier, the finite-difference method is also susceptible to subtractive errors.

Three configurations are considered: Benchmark Supercritical Wing (BSCW) from the Aeroelastic Prediction Workshop (Chwalowski et al. 2017), the Vortex Induced Vibration (VIV) of a cylinder (Anagnostopoulos and Bearman 1992), and the AGARD 445.6 Wing (Yates 1987; Lee-Rausch and Batina 1993). Flow regimes range from incompressible to transonic. Both static/steady and time-dependent sensitivity derivatives are evaluated for the BSCW configuration, while time-dependent sensitivities are evaluated for the other two configurations.

Benchmark Supercritical Wing (BSCW)

The first verification example for modal sensitivity derivatives via complex arithmetic corresponds to a static aeroelastic condition for the BSCW on a grid of approximately three million nodes. Note that in FUN3D, a static aeroelastic case is run as an unsteady case, with a large time step and a critical damping ratio of $O(1)$ used to quickly reach static equilibrium. In particular, the critical damping ratio was taken as 0.999, and the nondimensional time step of 50 used in FUN3D corresponded to approximately 0.0082 seconds at the freestream speed of 4508.4 in/s. The Mach number was 0.74, with a Reynolds number of 278,400 per inch. The flow was assumed fully turbulent and the Spalart-Allmaras turbulence model was employed. The BSCW tests were conducted in a working fluid other than air, however, $\gamma = 1.4$ was assumed for the computations shown here. All simulations were run for 600 time steps, which were sufficient to reach static equilibrium. The freestream dynamic pressure was 1.1722 lb/in^2. The complex step size was 1×10^{-50} while the finite-difference step size was 1×10^{-6}. Table 1 shows selected sensitivities (f) with respect to several independent variables (x) of interest for modal aeroelastic analysis.

Table 1: Comparison of Complex vs. Finite Difference Sensitivities – BSCW Static Deflection

x	***f***	**Complex**	**Finite Difference**
freestream α	tip pitch angle	0.25492	0.25551
	gdisp mode 1	0.11623	0.11633
	gforce mode 2	-27.4202	-27.5016
freq mode 1	tip pitch angle	-0.8740×10^{-5}	-1.0×10^{-5} *
	gdisp mode 1	-0.11351	-0.11351
	gforce mode 2	-162.6105	-162.6057
q_{inf}	tip pitch angle	1.51058	1.51058
	gdisp mode 1	1.15195	1.15195
	gforce mode 2	-162.5892	-162.5990
initial displacement mode 1	tip pitch angle	-0.1278×10^{-6}	0*
	gdisp mode 1	1.4217×10^{-8}	0*
	gforce mode 2	3.3082×10^{-5}	0*

Complex step size = 1×10^{-50} and the central difference step size = 1×10^{-6}

* Need more digits output or bigger finite difference step

BSCW – Time-Dependent Sensitivity Derivatives

The second verification case is also for the BSCW, but this time using a smaller time step so that some level of time-accuracy is retained. Furthermore, the configuration is perturbed from a static solution in the manner typical of flutter-onset analysis. Note that this case is for zero angle of attack. The dynamic sensitivity study used the same flow conditions (excepting angle of attack), and turbulence model as the static sensitivity study. For the dynamic sensitivities, the critical damping ratio was set to zero, and both modes were given an initial generalized velocity of 0.1. For this case sensitivities of the generalized response (displacement, velocity, force) with respect to the frequency of mode 1 (plunge) are computed. The complex step size is again 1×10^{-50}, but the finite-difference step size was increased to 1×10^{-5} after some preliminary results suggested the smaller finite-difference step size was in the range of error subject to subtractive cancellation and/or insufficient subiterative convergence. Figure 4 shows 3750 time steps (each corresponding to 0.19697×10^{-3} seconds) after an initial static solution was obtained. It can be seen that the results from the complex step and finite-differences are indistinguishable.

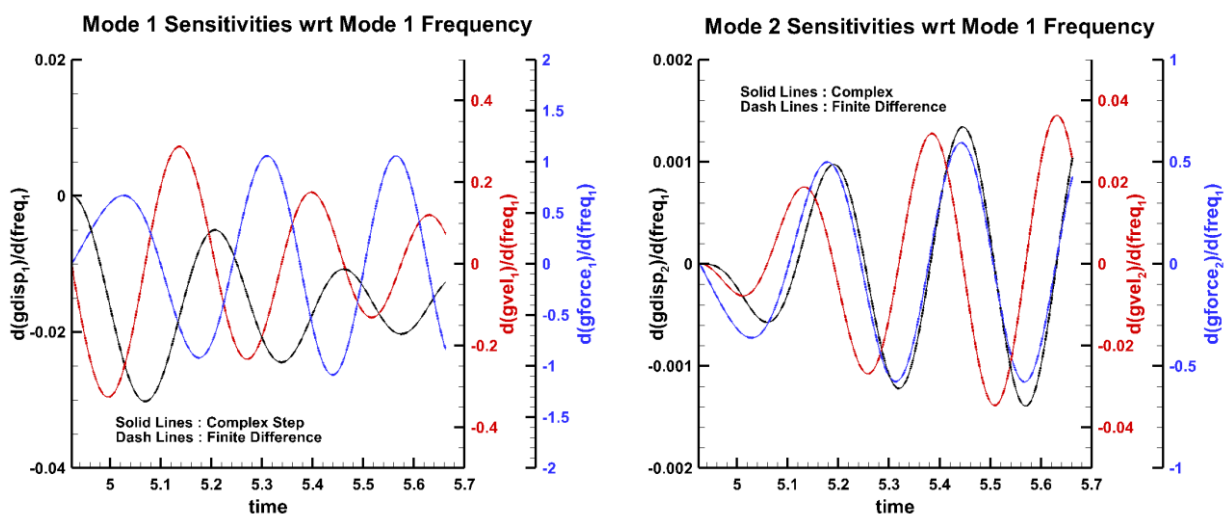


Figure 4. Sensitivities of the generalized response for mode 1 (plunge, left) and mode 2 (pitch, right) with respect to the frequency of mode 1, computed by the complex-step method (solid lines) and the finite-difference method (dashed lines).

VIV – Time-Dependent Sensitivity Derivatives

For the VIV configuration, only one mode (plunge) is active. The sensitivities of the generalized response (displacement, velocity, force) with respect to the critical damping ratio are examined. The nominal critical damping ratio corresponding to the experimental setup was determined to be 0.00135942. The flow conditions corresponded to a Reynolds number of 120 (based on diameter), with a freestream speed of 0.0670842 m/s and a dynamic pressure of 2.23626 N/m^2 (kg/ms^2). The modal frequency is 44.0828 rad/s, and the generalized mass was taken as 0.000476666 kg for this 2D simulation. The grid contained approximately 10,000 nodes on the *x-z* plane. The Mach number for this case is very small, $O(1\times10^{-5})$, so the incompressible option was used in FUN3D, with an artificial compressibility parameter taken as the default value, 15.0. Laminar flow is assumed for this low Reynolds number.

As this is a 2D simulation on a relatively coarse grid many time steps were performed - 10,000 steps beyond an initial unsteady solution during which the cylinder was held fixed. The initial unsteady solution at this Reynolds number was sufficient to excite a dynamic response without an additional perturbation to the generalized velocity as was used in the BSCW example. A FUN3D nondimensional time step of 0.05 was chosen, corresponding to 0.0011925 seconds at the given freestream speed. A complex step size of

1×10^{-50} was used, and the finite-difference step size was 1×10^{-5}. Figure 5 shows close-up views of the initial and final sensitivities of the generalized response with respect to the critical damping ratio, omitting the many intervening time steps for clarity. As for the BSCW case, the complex-step and finite-difference sensitivities are indistinguishable to plotting accuracy.

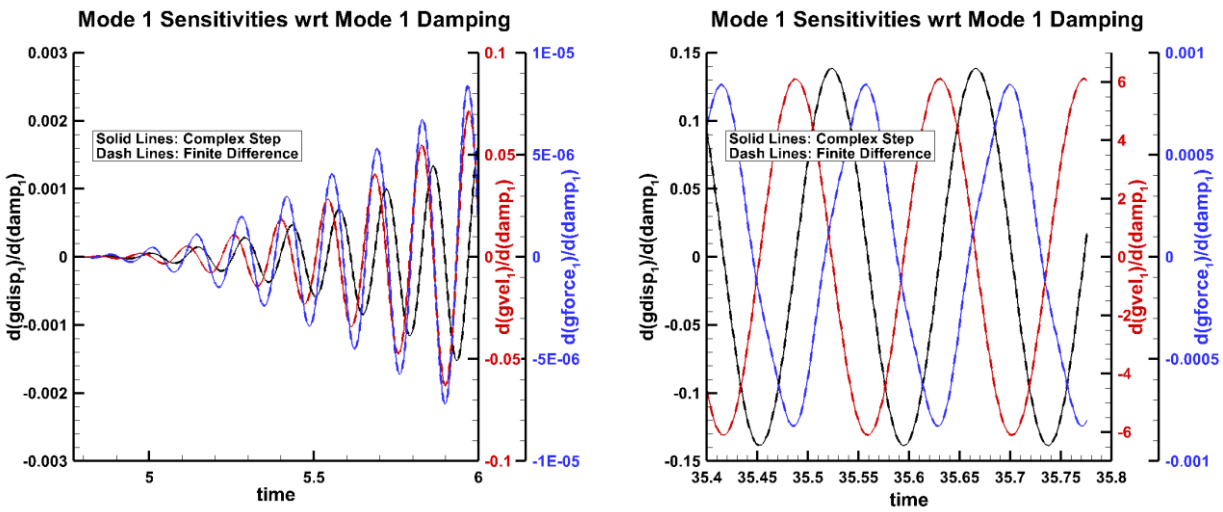


Figure 5. Sensitivities of the generalized response of the cylinder with respect to the critical damping ratio, computed by the complex-step method (solid lines) and the finite-difference method (dashed lines); left, initial sensitivities; right, final sensitivities.

AGARD 445.6 Wing – Time-Dependent Sensitivity Derivatives

The final configuration considered for sensitivity verification is the AGARD 445.6 wing. The sensitivities of the generalized response (displacement, velocity, force) with respect to the freestream dynamic pressure and with respect to the frequency of mode 1 (first bending) are examined. Inviscid flow at Mach 0.9 is assumed, and the solution is initiated from a rigid steady state at zero angle of attack. The unsteady simulations were run for 5000 time steps past this steady initial state, using a FUN3D non-dimensional time step of 3.6, corresponding to 0.00027738 seconds at a freestream speed of 11680.8 in/s. Each of the four modes was given an initial generalized velocity of 0.1 to initiate the dynamic response. The nominal freestream dynamic pressure was 0.52083 lb/in^2.

Figure 6 shows close-up views of the initial and final sensitivities of the modal response with respect to q_{inf}, omitting the many intermediate time steps for clarity. A complex step size of 1×10^{-50} was used, and the finite-difference step size was 1×10^{-5}. Similarly, Figure 7 shows the initial and final history of the modal response sensitivity with respect to mode 1 frequency. A complex step size of 1×10^{-20}, and the finite-difference step size was 1×10^{-6} was used in these computations.

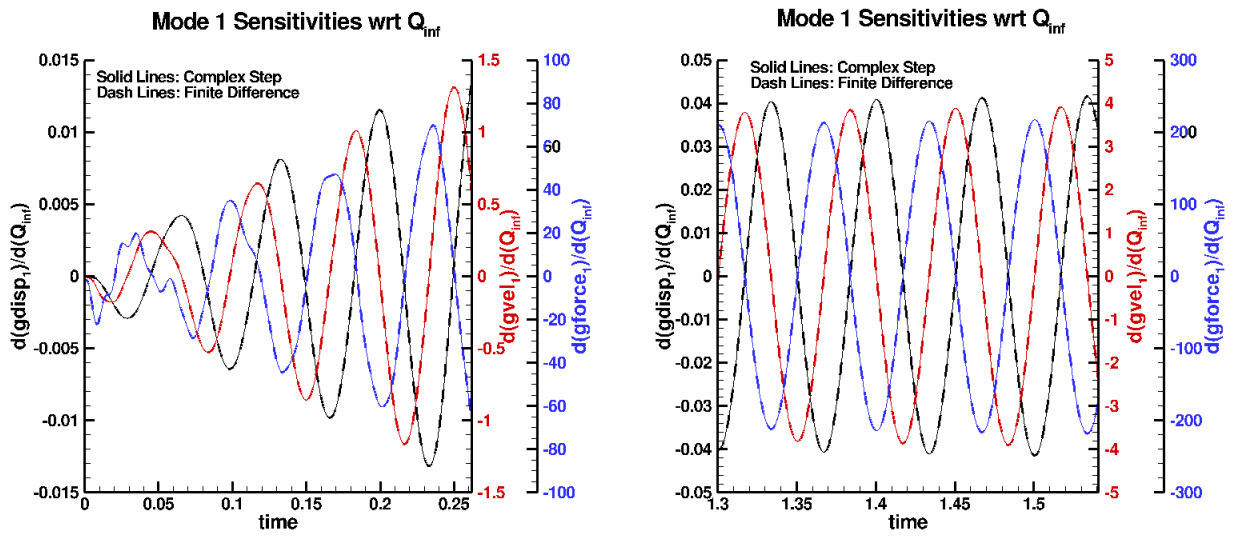


Figure 6. Sensitivities of the generalized response of mode 1 for the 445.6 wing with respect to the freestream dynamic pressure, computed by the complex-step method (solid lines) and the finite-difference method (dashed lines); left, initial sensitivities; right, final sensitivities.

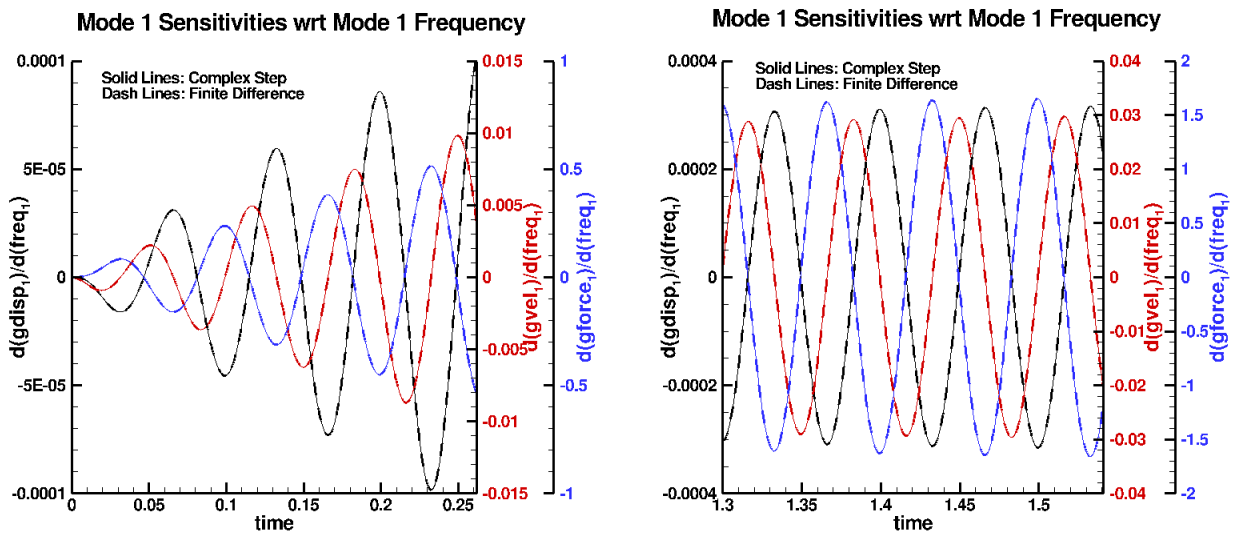


Figure 7. Sensitivities of the generalized response of mode 1 for the 445.6 wing with respect to the frequency of mode 1, computed by the complex-step method (solid lines) and the finite-difference method (dashed lines); left, initial sensitivities; right, final sensitivities.

FEM-based Aeroelastic Analysis

The FEM-based aeroelastic analysis is performed with the FUNtoFEM framework (Kiviaho et al. 2017). FUNtoFEM is a modular Python-based framework developed for adjoint-based aeroelastic optimization. The FUNtoFEM framework provides coupling algorithms for both steady and unsteady aeroelastic problems in addition to implementations of load and displacement transfer schemes to exchange data in-core between the structural and fluid solvers. Kiviaho et al. (2017), and Jacobson et al. (2018) have performed analysis and calculated sensitivities with solver interfaces to FUN3D, TACS, and the FUNtoFEM transfer scheme. These solver interfaces contain direct calls to the Python wrapper of the codes. For SAMS, the direct Python calls to FUN3D, TACS, and the transfer scheme implementation have been replaced with a Hermes-based client-server model (Snyder 2017).

Hermes-based Client-Server Model

The direct Python mode of FUNtoFEM is depicted in Figure 8. The FUNtoFEM driver orchestrates the coupling. Solvers are added through their Python interfaces. There are three MPI communicators in the problem: one for the aerodynamic solver, one for structural solver, and a global communicator which is the union of the other two. Note that the structural communicator may be a subset of the aerodynamic communicator making the global and aerodynamic communicators identical. The FUNtoFEM driver operates with distributed aerodynamic vectors as determined by the aerodynamic solver's domain decomposition. This is illustrated by the multiple blue lines in Figure 8 where each line represents an instruction or transfer of data on an MPI rank of the aerodynamic communicator. In the same manner, the FUNtoFEM driver uses the distributed structural vectors as illustrated by the multiple orange lines.

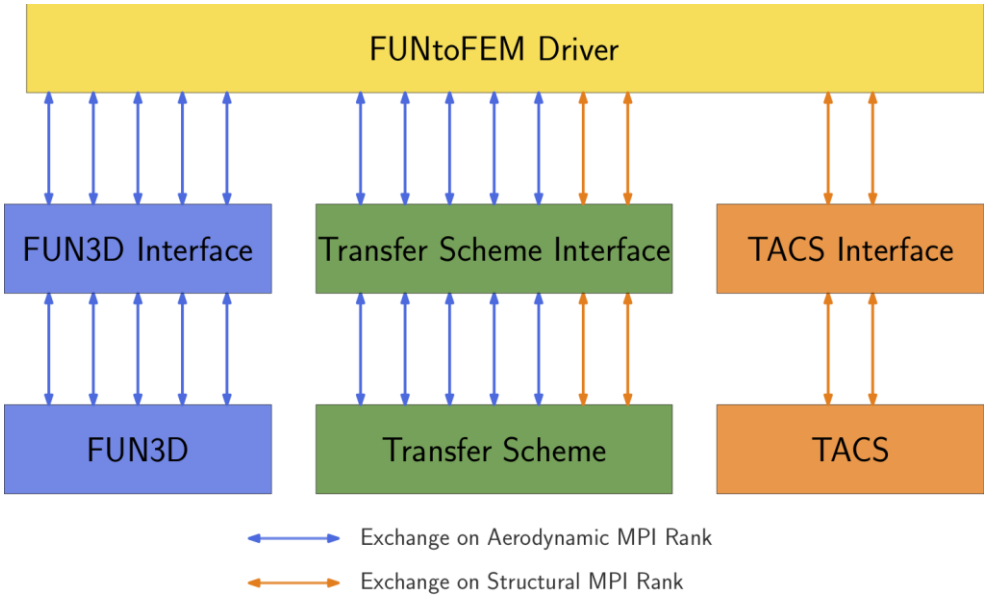


Figure 8: Direct Python mode in FUNtoFEM.

Figure 9 shows how the direct Python mode has been modified for the client and server model in FUNtoFEM. The clients replace the interface classes and implement the same methods, but they send requests to the corresponding server instead of direct calls to the codes' Python wrappers.

Serialization of data for network communications would be a serious bottleneck because multidisciplinary problems involve large amounts of data transfer between the solvers. Therefore, the client server model has been implemented to avoid serializing data and maintain the distributed representation of

the aerodynamic and structural vectors. Each server is started as an MPI process where every rank listens on a separate port, i.e., rank `i` of the server will listen on port `(base port number) + i`. The FUNtoFEM driver is then started as an MPI process with a global communicator that is the same size as the transfer scheme server, an aerodynamic communicator that is the same size as the FUN3D server, and a structural communicator that is the same size as the TACS server. The FUNtoFEM driver exchanges data with and gives instructions to the clients. Each MPI rank of the clients then sends a request to the corresponding port of the server as represented by the dashed lines in Figure 9.

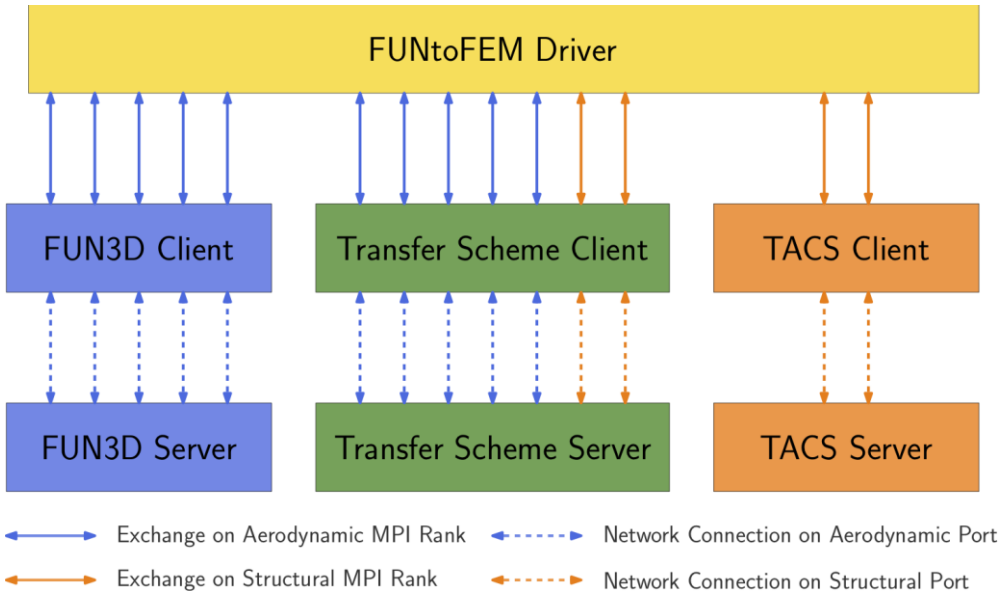


Figure 9: Hermes client-server mode in FUNtoFEM.

Steady Aeroelastic Analysis

For steady aeroelastic analysis, FUNtoFEM uses a nonlinear block Gauss-Seidel algorithm which is represented by the red path in Figure 10. Each block in Figure 10 represents an evaluation of a residual in the algorithm, and the color of the block corresponds to the server in Figure 9 that the evaluation occurs on. The first block, D, is the displacement transfer which takes displacements from the structural mesh and calculates the displacement of the aerodynamic surface. The grid deformation, G, moves the aerodynamic volume mesh to account for the deformed surface mesh. The aerodynamic solver, A, then calculates the aerodynamic solution on the new mesh. The forces on the aerodynamic surface are calculated from the new aerodynamic state (block F). Next, the load transfer, L, determines the forces on the structural model from the forces on the aerodynamic surface. The structural solver, then updates the structural displacements based on those loads. The process is repeated until the problem converges to a steady solution. Aitken's acceleration (Irons et al. 1969) is applied for stability of the coupled solver.

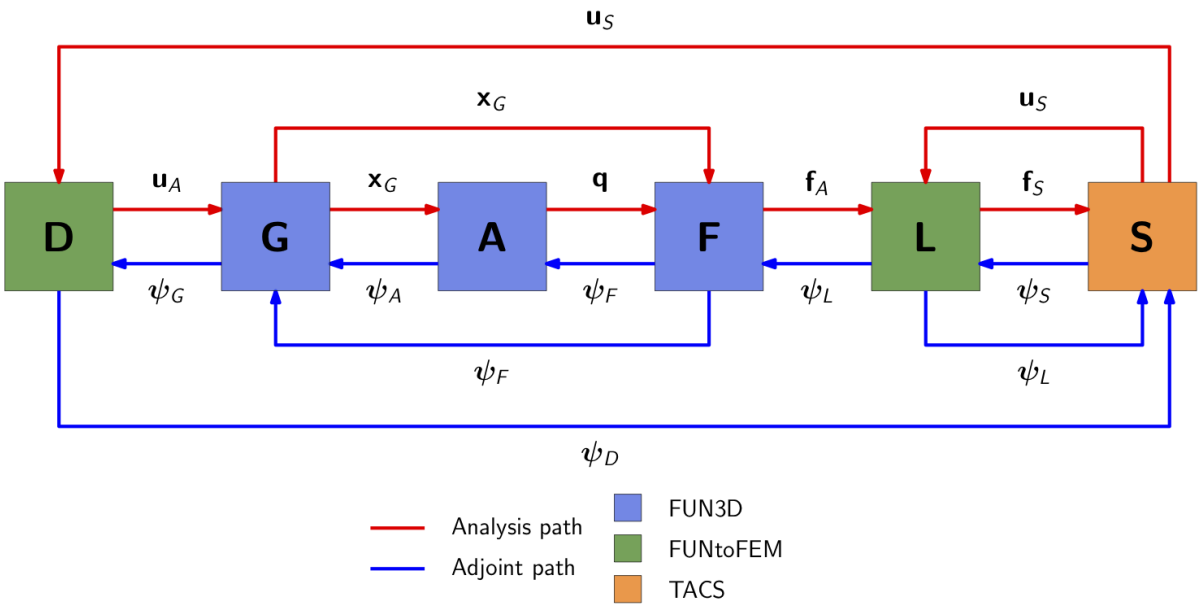


Figure 10: Flow of information in FUNtoFEM for steady aeroelastic coupling.

Steady Aeroelastic Sensitivities

For adjoint-based sensitivities, there is an adjoint residual that corresponds to each of the primal residuals. For the coupled adjoint problem, the flow of information is reversed compared to that of the primal analysis as represented by the blue lines in Figures 10. A linear block Gauss-Seidel algorithm is applied in the FUNtoFEM steady adjoint analysis. Like the primal analysis, the steady solution is found by iteratively evaluating the set of residuals until the problem converges. The full set of adjoint equations are provided in Kiviaho et al. (2017) for the steady problem. The reference also presents the expressions which relate the adjoint solution to the sensitivities of functions of interest to design variables.

Like the primal analysis, the FUNtoFEM adjoint formulation is modular. Aerodynamic solvers do not need to have any knowledge of structural functions or design variables and vice versa.

Steady Aeroelastic Simulation Setup

There are five parts to the steady problem setup:

1. Defining the FUNtoFEM model
2. The FUN3D server
3. The TACS server
4. The FUNtoFEM transfer scheme server
5. The FUNtoFEM driving script/client

These steps are illustrated below with the undeflected CRM (uCRM) example.

FUNtoFEM model

The FUNtoFEM model defines the problem that is going to be solved. The model is made up of bodies (the wing in the uCRM case) and scenarios that describe parameters such as the flow conditions and output functions of interest. For the uCRM case, a wing body is defined and given a set of design variables that define the thickness of panels of the wing box structure. A cruise scenario is defined, and the angle of attack is defined as a design variable. The scenario is also given function definitions for the problem which are a KS failure function that represents the maximum stress in the structure, lift, and drag.

This **`build_model`** module that defines the FUNtoFEM model in Python is invoked by other parts of the problem setup.

build_model.py (FUNtoFEM uCRM simulation)

```
from pyfuntofem.model import *
from funtofem import TransferScheme

def build_model():
    crm = FUNtoFEMmodel('crm')

    wing = Body('wing',group=0,boundary=3)

    if TransferScheme.dtype==complex:
        thickness = np.loadtxt('sizing_complex.dat',dtype=TransferScheme.dtype)
    else:
        thickness = np.loadtxt('sizing.dat',dtype=TransferScheme.dtype)

    for i in xrange(thickness.size):
        wing.add_variable('structural',Variable('thickness '+
str(i),value=thickness[i],lower = 0.001, upper = 0.1))

    crm.add_body(wing)

    cruise = Scenario('cruise',group=0,steps=300)

    cruise.set_variable('aerodynamic',name='AOA',value=3.0,lower=-15.0,upper=15.0)

    ks = Function('ksfailure',analysis_type='structural')
    cruise.add_function(ks)

    drag = Function('cd',analysis_type='aerodynamic')
    cruise.add_function(drag)

    lift = Function('cl',analysis_type='aerodynamic')
    cruise.add_function(lift)

    crm.add_scenario(cruise)
    return crm
```

FUN3D Server

The FUN3D part of the problem set up is similar to a standard problem setup with a couple of exceptions. In **fun3d.nml**, the **aero_loads_dynamic_pressure** in the **massoud_output** namelist specifies the dynamic pressure for dimensionalization of the surface forces. Additionally, **moving_grid** in the global namelist needs to be set to true.

fun3d.nml (FUNtoFEM uCRM simulation)

```
&project
  project_rootname = 'ucrm'
/
&raw_grid
  grid_format   = 'aflr3'
  patch_lumping = 'family'
/
&global
  moving_grid = .true.
  boundary_animation_freq = 100    ! write *tec_boundary* files every 100 iter
/
&governing_equations
  viscous_terms = 'inviscid'
/
&reference_physical_properties
  mach_number = 0.84
  temperature = 216.66
/
&nonlinear_solver_parameters
  time_accuracy        = 'steady'
  time_step_nondim     = 0.0
/
&code_run_control
  steps              = 300
  restart_write_freq = 300
  restart_read       = 'off'
/
&elasticity_gmres
  tol=1e-15
  tol_abs=1e-15
/
&massoud_output
   aero_loads_dynamic_pressure = 9510.486
/
```

In the **body_definitions** namelist of **moving_body.input**, the motion driver for the moving body is specified as '**funtofem**'. This tells FUN3D that it should calculate nodal forces on the surface and use displacements of the surface received via the FUN3D Python extension module.

moving_body.input (FUNtoFEM uCRM simulation)

```
&body_definitions
   n_moving_bodies = 1
   body_frame_forces = .false.
   body_name(1) = 'wing'
   motion_driver(1) = 'funtofem'
   n_defining_bndry(1) = 1
   defining_bndry(1,1) = 3
   mesh_movement(1)    = 'deform'
/
```

Once the input files and meshes have been set up, the FUN3D server can be started:

```
mpirun -n X python flow_server.py &
```

where **X** is the number of processors that the flow solver will use. The FUN3D server will then listen for communication from the client. **flow_server.py** is the FUN3D Hermes server included in the FUN3D repository.

Transfer Scheme Server

For the transfer scheme server, the user writes a Python script. The script has three parts. First, the MPI communicator is split, so that the transfer scheme server knows which ranks/ports to use for aerodynamic data versus structural data. Next, the options are selected for the transfer scheme itself which includes things like which transfer scheme to use and whether there is a symmetry plane. The final section of the Python script is to start the transfer scheme server. This script is run in the same manner as the FUN3D flow server:

```
mpirun -n X python transfer_server.py &
```

where again, **X** is the number of aerodynamic processors (assuming that the number of aerodynamic processors is greater than the number of structural processors in the simulation).

transfer_server.py (FUNtoFEM uCRM simulation)

```
import zmq

from mpi4py        import MPI
from funtofem_server import Server

if __name__ == "__main__":

    # split the communicator
    n_struct_procs = 8
    comm = MPI.COMM_WORLD

    world_rank = comm.Get_rank()
    if world_rank < n_struct_procs:
        color = 55
        key = world_rank
    else:
        color = MPI.UNDEFINED
        key = world_rank
    struct_comm = comm.Split(color,key)

    # set the transfer scheme options
    transfer_options = {}
    transfer_options['scheme'] = 'MELD'
    transfer_options['isym']   =  1
    transfer_options['beta']   =  0.5
    transfer_options['npts']   =  200

    # start the server
    context = zmq.Context()
    endpoint = 'tcp://*:' + str(43200+comm.Get_rank())
    server = Server(comm,struct_comm, context=context, endpoint=endpoint,
                    type_=zmq.REP, transfer_options=transfer_options)
    server.serve()
    server.close()
    context.destroy()
```

TACS Server

Like the transfer server, the TACS server requires writing some Python to set up. In **`tacs_server.py`**, a problem specific server class inherits the server functionality from a TACS server base case then the reading of the mesh and set up of TACS itself is added to the constructor (parts of this constructor have been left out of this example for brevity). The main function in **`tacs_server.py`** loads the function information in the FUNtoFEM model defined by the **`build_model`** module then launches the **`CrmServer`**. The server is started by running:

```
mpirun -n Y python tacs_server.py &
```

where **`Y`** is the number of processors being used to solve the structural problem.

`tacs_server.py (FUNtoFEM uCRM simulation)`

```
from tacs import TACS, elements, functions, constitutive
from tacs_steady_server import Server
from build_model import build_model

from mpi4py import MPI
import numpy as np
import zmq

class CrmServer(Server):
    def __init__(self, comm, context, endpoint, type_, ndof, model):
        super(CrmServer,self).__init__(comm, context, endpoint, type_, ndof, model)

        struct_mesh = TACS.MeshLoader(comm)
        struct_mesh.scanBDFFile("CRM_box_2nd.bdf")

        # Set constitutive properties
        rho = 2500.0  # density, kg/m^3
        E = 70.0e9 # elastic modulus, Pa
        nu = 0.3 # poisson's ratio
        kcorr = 5.0 / 6.0 # shear correction factor
        ys = 350e6  # yield stress, Pa
        min_thickness = 0.001
        max_thickness = 0.100

        thickness = 0.015
        spar_thick = 0.015

        # Loop over components in mesh, creating stiffness and element
        # object for each
        self.num_components = struct_mesh.getNumComponents()
        for i in xrange(self.num_components):
            descript = struct_mesh.getElementDescript(i)
            comp = struct_mesh.getComponentDescript(i)
            stiff = constitutive.isoFSDT(rho, E, nu, kcorr, ys, thickness, i,
                                         min_thickness, max_thickness)
            element = None
            if descript in ["CQUAD", "CQUADR", "CQUAD4"]:
                element = elements.MITCShell(2,stiff,component_num=i)
            struct_mesh.setElement(i, element)
       .
       .
       .
        # Initialize member variables pertaining to TACS
        self.tacs = tacs
        self.res = res
        self.ans = ans
```

```
        self.mat = mat
        self.pc = pc
        self.struct_X = struct_X
        self.struct_nnodes = struct_nnodes
        self.gmres = gmres
        self.svsens = tacs.createVec()
        self.struct_rhs_vec = []

if __name__ == "__main__":
    comm = MPI.COMM_WORLD

    context = zmq.Context()
    endpoint = 'tcp://*:'+ str(44200+comm.Get_rank())

    model = build_model()

    server = CrmServer(comm, context=context, endpoint=endpoint, type_=zmq.REP,
ndof=6, model=model)
    server.serve()
    server.close()
    context.destroy()
```

FUNtoFEM driver/client

In the FUNtoFEM main run script, the user first splits the communicator so that the structural and transfer scheme clients know which processors/ports to use for communication with the servers. The driver script then gets the FUNtoFEM model definition from **`build_model`**. Next, it creates a Python dictionary of the solver clients which the FUNtoFEM driver will access the solvers. These clients have basic functions defined such as "initialize", "iterate", "get_function", etc. Within these methods, the client makes requests to the server. The options dictionary in the main script typically tells the FUNtoFEM driver what options to use for the transfer scheme, but for the Hermes example, it tells the driver to use the transfer scheme client instead of directly using the transfer scheme (the transfer scheme options are specified in transfer scheme server set up). The final steps for the main script are to instantiate the driver object and call the **`solve_forward`** and **`solve_adjoint`** methods of that object. After starting all the servers, the main script is run with:

```
mpirun -n X python hermes_driver.py
```

where **`X`** is the number of processors used to solve the flow problem.

`hermes_driver.py (FUNtoFEM uCRM simulation)`

```
from mpi4py import MPI
from pyfuntofem.model import *
from pyfuntofem.driver import *
from pyfuntofem.fun3d_client import Fun3dClient
from pyfuntofem.hermes_structure import *
from build_model import build_model

# split the communicator
n_tacs_procs = 8
comm = MPI.COMM_WORLD

world_rank = comm.Get_rank()
if world_rank < n_tacs_procs:
    color = 55
    key = world_rank
else:
    color = MPI.UNDEFINED
    key = world_rank
tacs_comm = comm.Split(color,key)


# build the model
crm = build_model()

solvers= {}

# instantiate the fem_solver
solvers['structural'] = TacsHermes(comm,tacs_comm,crm)

# instantiate the flow_solver
solvers['flow'] = Fun3dClient(comm,crm)

options = {'scheme': 'hermes'}

# instantiate the driver
driver =
FUNtoFEMnlbgs(solvers,comm,tacs_comm,0,comm,0,model=crm,transfer_options=options)
```

```
# run the forward analysis
fail = driver.solve_forward()

vrs = crm.get_variables()
funcs = crm.get_functions()
if comm.Get_rank() ==0:
    for func in funcs:
        print 'FUNCTION: ' + func.name + " = ", func.value


# run the adjoint
fail = driver.solve_adjoint()

derivatives = crm.get_function_gradients()
if comm.Get_rank() ==0:
    for i, func in enumerate(funcs):
        print 'FUNCTION: ' + funcs[i].name + " = ", funcs[i].value
        for j, var in enumerate(vrs):
                print ' var ' + var.name, derivatives[i][j]
```

Steady Aeroelastic Verification

The Hermes-based client-server model is verified by comparing results to the direct Python mode of FUNtoFEM. The test case for verification is the uCRM wing set up in the previous section. The Euler aerodynamic model has 24,187 nodes that represent the CRM wing. The structural model has 10,584 linearized shell elements that represent the wing box. Table 2 compares output functions from the primal analysis. The lift and drag values are aerodynamic quantities from the coupled simulation. The KS-failure function is an aggregated approximation of the maximum stress in the structure. The table indicates that there is no difference between the real results of the direct Python and Hermes simulations indicating that the client-server model has been implemented properly.

Table 2: Steady uCRM aeroelastic primal results.

	Lift	Drag	KS Failure
Direct Python - complex	123.630842869	9.10252216846	0.574068911078
Direct Python - real	123.630842860	9.10252216776	0.574068911109
Hermes - real	123.630842860	9.10252216776	0.574068911109

Tables 2–5 compare the sensitivities calculated by the complex step method with the adjoint-based sensitivities from the direct Python and client-server modes of FUNtoFEM. These sensitivities are shown for a structural design variable and an aerodynamic design variable. Like the primal analysis outputs, the direct Python and Hermes-based client-server sensitivities agree exactly. The complex step and adjoint-based derivatives match between 8–10 digits which is about the same level of agreement as the primal results.

Table 3: Comparison of steady uCRM aeroelastic sensitivities for lift.

	Panel thickness 0	Angle of attack
Direct Python - complex step	48.770483361	23.041073184
Direct Python - adjoint	48.770483348	23.041073169
Hermes - adjoint	48.770483348	23.041073169

Table 4: Steady uCRM aeroelastic sensitivities for drag.

	Panel thickness 0	Angle of attack
Direct Python - complex step	6.415322146 79	3.029320468 98
Direct Python - adjoint	6.415322146 52	3.029320468 64
Hermes - adjoint	6.415322146 52	3.029320468 64

Table 5: Steady uCRM aeroelastic sensitivities for KS failure.

	Panel thickness 0	Angle of attack
Direct Python - complex step	-4.6136009 9686	0.0755157643653
Direct Python - adjoint	-4.6136009 8971	0.0755157643636
Hermes - adjoint	-4.6136009 8971	0.0755157643637

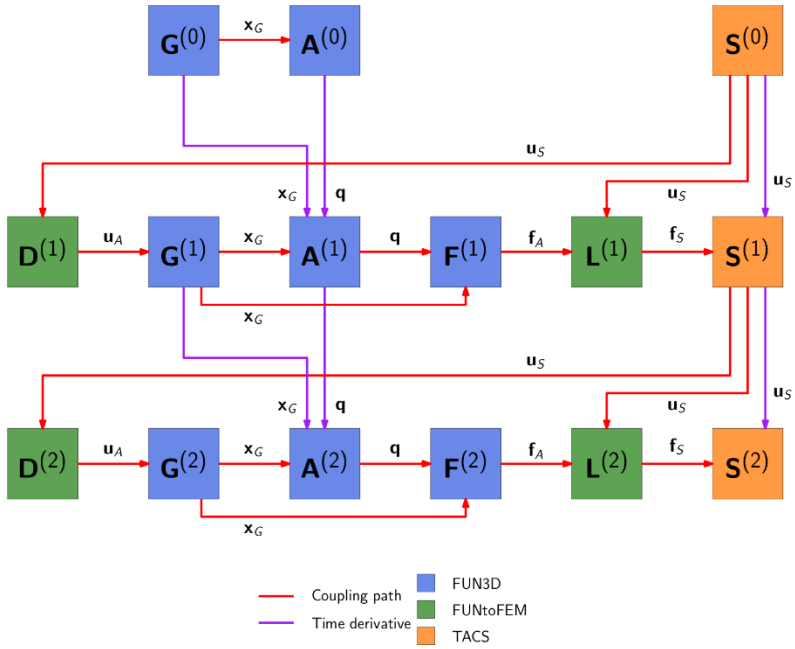
FEM-based Unsteady Aeroelastic Analysis and Sensitivities

Unsteady Aeroelastic Primal Analysis

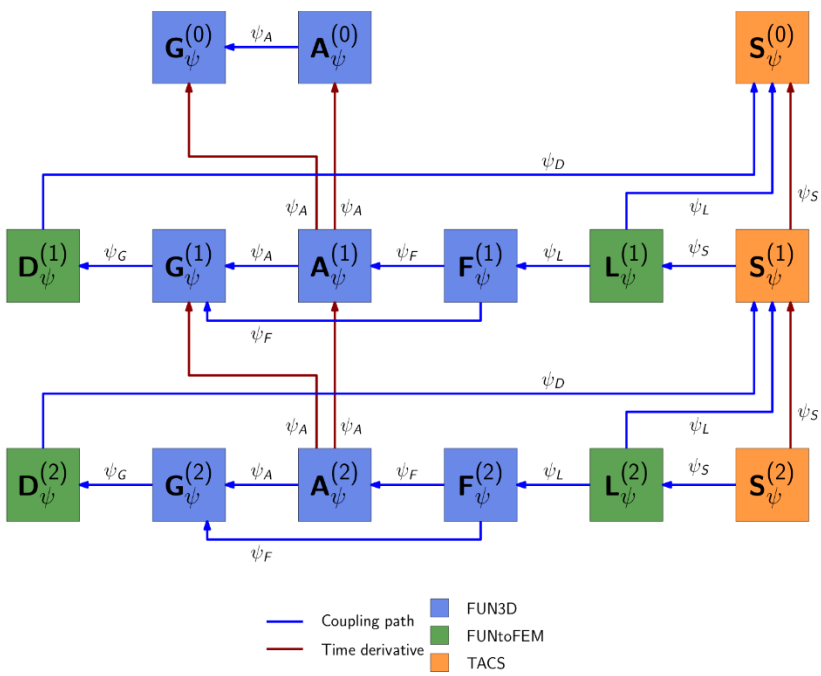
FUNtoFEM extends the nonlinear block Gauss-Seidel algorithm from the steady analysis to unsteady analysis by staggering the structural displacements for each time step. That is, the displacement transfer at time step `n` is dependent on the structural displacements at step `n-1`. The flow of information in the unsteady primal analysis is illustrated in Figure 11a. Each row in figure represents a time step, and the unsteady analysis can be solved by marching from left to right and top to bottom in the flow chart. In the unsteady aeroelastic analysis, there are time derivatives of quantities such as the flow state and the displacement of the structure. When discretized, these time derivatives create dependencies of the aerodynamic and structural residuals on states from previous time steps represented by the purple lines in Figure 11a. However, these time derivatives are confined within the individual disciplinary solvers. Therefore, the aerodynamic and structural solvers can use any time marching method inside this coupling algorithm as long as the time step size matches.

Unsteady Aeroelastic Sensitivities

The flow of information in the unsteady adjoint analysis is illustrated in Figure 11b. As in the steady case, the dependencies have been reversed. Each row in the figure represents a time step, and the unsteady adjoint can be solved by reverse time marching, i.e., from right to left and bottom to top in the flow chart. The full set of adjoint equations and sensitivity expressions are provided in Jacobson et al. (2018).



a) Forward Analysis



b) Adjoint Analysis

Figure 11. Flow of information in FUNtoFEM for unsteady aeroelastic coupling.

Unsteady Aeroelastic Simulation Setup

The unsteady problem setup only has a few differences from the steady aeroelastic version which are highlighted here in `blue` The transfer scheme server does not change. In the `model_builder` module, the scenario is specified as unsteady:

build_model.py (unsteady modifications)

```
steps = 300
cruise = Scenario('cruise',group=0,steps=steps,steady=False)
```

In the FUN3D namelist, the solver is instructed to use time dependent analysis:

fun3d.nml (unsteady modifications)

```
&nonlinear_solver_parameters
  time_accuracy        = '2ndorderOPT'
  time_step_nondim     = 0.1
  subiterations        = 10
/
```

Like the steady problem, the TACS script creates a server class that inherits all the required functionality of the server from a base class, then adds the TACS initialization to the constructor. The structural server uses a TACS utility called TACSBuilder to help set up the unsteady problem. One additional change is that the server is given a Python dictionary of time integrator options when it is instantiated. This dictionary tells TACS the number of time steps, the time step size, and other integration related options.

tacs_server.py (unsteady uCRM version)

```
import zmq
from mpi4py import MPI
from tacs_unsteady_server import Server
from tacs_builder import *
from tacs import TACS
from build_model import *

class CRMServer(Server):
    def __init__(self, comm, context, endpoint, type_, ndof, model,
integrator_options):
        rho=2500.0
        E=70.0e9
        nu=0.3
        kcorr=5.0/6.0
        ys=350.0e6
        thickness=0.015
        tmin=1.0e-4
        tmax=1.0
        tdv=0

        # Create an instance of TACS
        self.builder = TACSBuilder(comm)

        shellStiff = ShellStiffness(rho,E,nu,kcorr,ys,thickness,tmin,tmax)
        wing =
self.builder.addMITCShellBody('wing','CRM_box_2nd.bdf',0,shellStiff,isFixed=False)

        super(CRMServer,self).__init__(comm, context, endpoint, type_, ndof, model,
integrator_options)

if __name__ == "__main__":
    comm = MPI.COMM_WORLD
```

```
    context = zmq.Context()
    endpoint = 'tcp://*:'+ str(44200+comm.Get_rank())

    model = build_model()
    steps = model.scenarios[0].steps
    options = {'integrator': 'BDF',     'start_time': 0.0,      'step_size': 0.001,
               'steps': steps,          'integration_order': 2, 'solver_rel_tol':
1.0e-10,
               'solver_abs_tol':1.0e-9, 'max_newton_iters': 50, 'femat':1,
               'print_level': 1,        'output_freq': 10,      'ordering':
TACS.PY_RCM_ORDER }

    server = CRMServer(comm, context=context, endpoint=endpoint, type_=zmq.REP,
ndof=6, model=model, integrator_options=options)
    server.serve()
    server.close()
    context.destroy()
```

Apart from these changes described, the unsteady forward and adjoint problems is set up and run in the same way as the steady problem.

Unsteady Aeroelastic Verification

The unsteady verification is performed with the uCRM and the vortex induced vibration (VIV) cases. For the VIV case, there is one difference between the analysis described in the Appendix and the simulations used for verification. For sensitivity verification, only pressure forces are considered in the load transfer. Traditionally, viscous forces have a negligible effect on the structure in aerospace aeroelastic problems, and the sensitivity terms for the viscous force transfer have not yet been implemented; however, these are very low Reynolds number cases where the viscous forces are significant if the correct aeroelastic response is desired.

For the uCRM verification, the wing starts at the jig shape and free stream flow conditions. The simulation is run for 10 time steps and the lift and KS failure functions are calculated. In Table 6, the calculated lift and KS failure values match the direct Python results. Tables 7 and 8 show the comparison of the derivatives of the functions of interest with respect to the thickness of one of the structural panels and the angle of attack. The tables show at least 10 digits of agreement between the complex step method and the two adjoint methods.

Table 6: Unsteady uCRM aeroelastic primal results.

	Lift	KS Failure
Direct Python - complex	0.000192515603246	0.0259753355763
Direct Python - real	0.000192515603246	0.0259753355763
Hermes - real	0.000192515603246	0.0259753355763

Table 7: Unsteady uCRM lift sensitivities.

	Panel thickness 0 (10^{-5})	Angle of Attack
Direct Python - complex	-2.00547262678	2.89636955472
Direct Python - adjoint	-2.00547262677	2.89636955472
Hermes - adjoint	-2.00547262677	2.89636955472

Table 8: Unsteady uCRM KS failure sensitivities.

	Panel thickness 0	Angle of Attack (10^{-6})
Direct Python - complex	-0.0695002469313	5.1745426714
Direct Python - adjoint	-0.0695002469313	5.1745426714
Hermes - adjoint	-0.0695002469313	5.1745426714

The VIV case is treated as an energy harvesting problem where the selected function of interest is the energy dissipated by the damper over a set time period. The design variables are the damping, the spring stiffness, and the angle of attack. Table 9 shows good agreement for the calculated harvested energy between the client-server mode and the direct Python mode. Table 10 compares the derivatives of the energy harvested with respect to the spring stiffness and damping values. Like the unsteady uCRM, the agreement between the different versions is at least 10 digits.

Table 9: VIV aeroelastic primal results.

	Energy Harvested (10^{-9})
Direct Python - complex	8.40950521459
Direct Python - real	8.40950521459
Hermes - real	8.40950521459

Table 10: VIV energy sensitivities.

	Angle of attack (10^{-8})	Stiffness (10^{-9})	Damping (10^{-8})
Direct Python - complex	6.7947745116**1**	4.2440343618**9**	8.40211347566
Direct Python - adjoint	6.7947745114**2**	4.2440343618**6**	8.40211347566
Hermes - adjoint	6.7947745114**2**	4.2440343618**6**	8.40211347566

VIV Optimization

An energy harvesting optimization of the VIV case was performed with FUNtoFEM. The objective function was the energy extracted by the damper attached to the cylinder.

$$E = \int_{t_1}^{t_2} c\dot{h}(t)^2 \, dt \,, \tag{13}$$

where c is the damping coefficient and h is the displacement of the cylinder. The design variables were the spring stiffness and the damping coefficient. To avoid the effects of the initial transients, the energy harvesting window was the final 3,000 time steps of the 10,000 step simulation. As in the sensitivity verification, only the pressure forces were considered in the load transfer. Therefore, the optimization is intended to demonstrate the design capability rather than draw meaningful scientific conclusions from optimization results.

The optimization used the sequential least squares quadratic programming (SLSQP) from PyOpt. After 11 design cylces, the sensitivites of the energy harvest were -5.76×10^{-4} and 8.94×10^{-5} for the spring stiffness and damping coefficient respectively which are close to zero indicating that the optimization had converged

to a locally optimal result. Figure 12 shows that the optimization convergence and that the energy extracted increased by more than a factor of 8. The history of the design variables is given in Figure 13. The initial stiffness value was selected to have a natural frequency near the rigid cylinder shedding frequency to produce large amplitudes of the motion. Over the optimization, the stiffness more than doubled, and the damping ratio also increased significantly. The higher spring stiffness increases the frequency of the cylinder motion; this produces more oscillations (29 stationary points versus 25) over the window of measured energy harvest as illustrated by the displacement history in Figure 14 and the vortex shedding in Figure 15. The higher damping allows more energy to be harvested per cycle despite the lower amplitude of the motion.

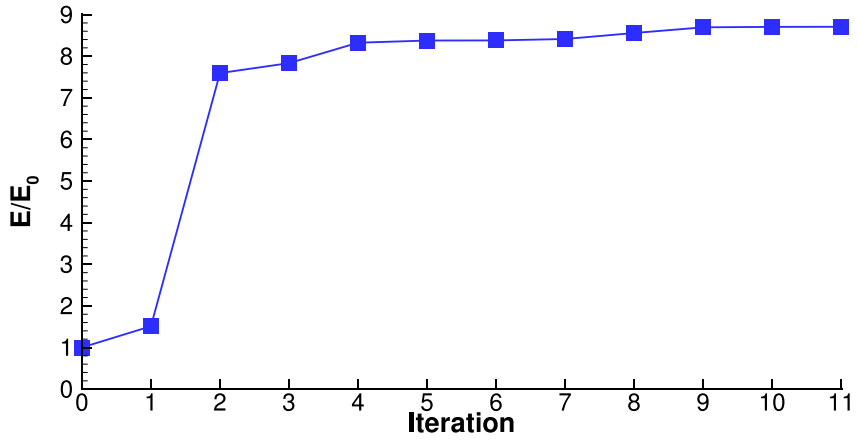


Figure 12. Optimization history of VIV energy harvested normalized by the initial design's value.

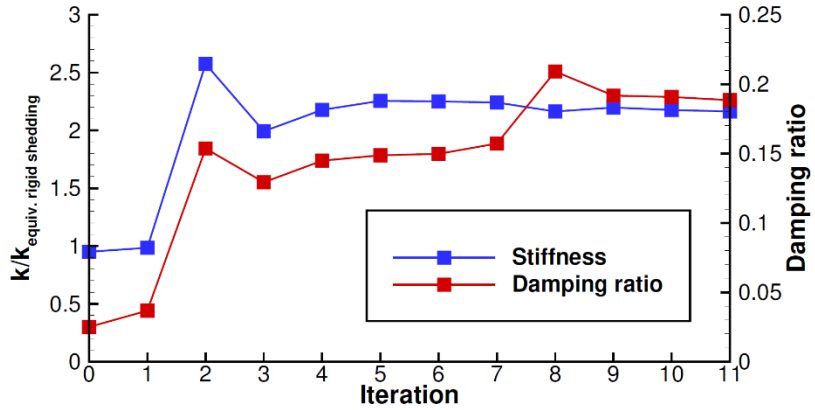


Figure 13. Design variable history for the VIV optimization. The spring stiffness is normalized by the stiffness that corresponds to the natural frequency matching the shedding frequency of the rigid cylinder.

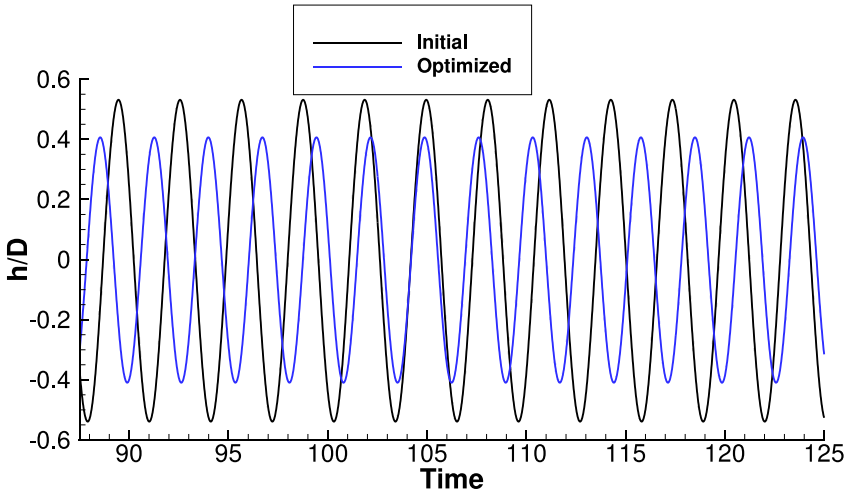


Figure 14. Comparison of the displacement of the cylinder over the time window of energy harvesting for the initial and final design.

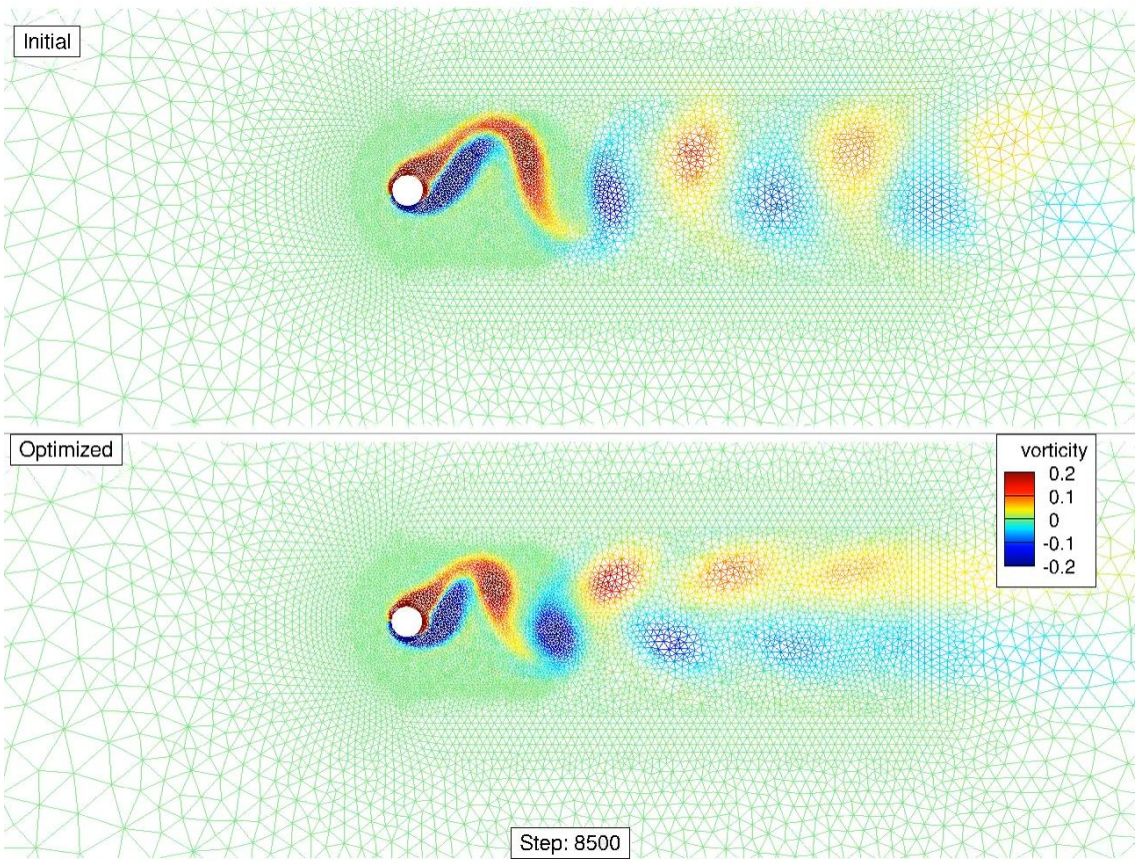


Figure 15. The VIV mesh colored by the vorticity for initial and optimal design at a time step within the energy harvesting window.

Appendix A: Benchmark Test Cases

Three benchmark aeroelastic test cases have been added to the FUN3D test suite. This was done in order to ensure that any future FUN3D code development will not break the current implementation of aeroelastic capabilities. The test suite includes three cases: Vortex Induced Vibrations (VIV), the Benchmark Supercritical Wing (BSCW), and the AGARD 445.6 wing. These tests are run on a weekly basis on the NASA Langley's K-cluster. The cases' grids and inputs have been placed in the FUN3D `git` repository.

The tests are run via Jenkins and metrics (plots/tables) are generated upon their successful completion. Resources allocated for the tests are given in Table A.1.

Table A.1: Resources used for the Aeroelastic Benchmark Tests

Case	Wall Time	Number of Cores
VIV (19,840 nodes)		48
Re=112	~2hr 30min	
Re=120	~3hr	
Re=130	~2hr 30min	
BSCW (2,968,550 nodes)	~3hr 40min	120
AGARD (439,415 nodes)	~1hr	72

Vortex-Induced Vibrations (VIV)

The Vortex-Induced Vibrations (Anagnostopoulos and Bearman 1992) case is run at three different Reynolds numbers (Re = 112, 120, and 130). Simulations at Re=112 and 130 are run for a total of 10,000 steps initially with no perturbation. A perturbation in velocity (`gvel=0.02`) is added after 10,000 iterations. The simulation with Re = 120 is run with no perturbation. The FUN3D namelist file and the `moving_body.input` file for Re = 120 is given below.

fun3d.nml (Re=120)

```
&global
    slice_freq = 0
    boundary_animation_freq = 10000
    moving_grid = .true.
/

&project
    project_rootname = "project"
    case_title = "case project"
/


&raw_grid
     grid_format = "aflr3"
     data_format = "stream"
     patch_lumping = "none"
     twod_mode = .true.
/
```

```
&massoud_output
!use with --write_aero_loads_to_file
   massoud_output_freq = -1
   massoud_file_format = 'ascii'
   n_bodies = 1
   nbndry(1) = 1
   boundary_list(1) = '2'
/

&boundary_output_variables
  number_of_boundaries = 1
  boundary_list = '3'
  y = .false.
  u = .true.
  v = .false.
  w = .true.
  vort_y=.true.
/

&governing_equations
    eqn_type = "incompressible"
    viscous_terms = "laminar"
/

&reference_physical_properties
    reynolds_number = 120
    angle_of_attack = 0.0
    angle_of_yaw =    0.0
/

&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
/

&inviscid_flux_method
    flux_construction = "roe"
    first_order_iterations = 0
    flux_limiter = "none"
/

&nonlinear_solver_parameters
    time_accuracy = "2ndorderOPT"
    time_step_nondim = 0.05
    subiterations = 25
    schedule_iteration = 1 100
    schedule_cfl = 10 10
    schedule_cflturb = 5 5
/


&linear_solver_parameters
    meanflow_sweeps = 15
    turbulence_sweeps = 10
    linear_projection = .false.
/
```

```
&special_parameters
    large_angle_fix = "off"
/

&code_run_control
    steps = 50000
    restart_write_freq = 5000
    restart_read = "off"
/

&elasticity_gmres
    nsearch = 50
    nrestarts = 100
    tol = 1.e-5
     restart_deformation = .true.
    elasticity = 1,
    elasticity_exponent = 1.0,
    restart_deformation = .true.
/
```

moving_body.input (Re=120)

```
! ------------------------------------------------------------------------------
! - ViV - Re=120
! ------------------------------------------------------------------------------

&body_definitions
    n_moving_bodies     = 1
    body_name(1)        = "cylinder"
    n_defining_bndry(1) = 1
    defining_bndry(1,1) = 2
    motion_driver(1)    = "aeroelastic"
    mesh_movement(1)    = "deform"
/

&aeroelastic_modal_data
    nmode(1)               = 1
    grefl                  = 0.0016
    uinf                   = 0.0670842
    qinf                   = 2.23626
    gmass(1,1)             = .000476666
    freq(1,1)              = 44.0828
    damp(1,1)              = 0.00135942
    genforce_include_shear = .true.
    gvel0(1,1) = 0.0    ! don't need to perturb with unsteady starting flowfield
/
```

Figures A.1–A.2 show the generalized displacement for Re=112, Re=120, and Re=130. Upon the successful completion of the test, these plots are posted on Jenkins. The means and variances of generalized displacement peaks of the last 10,000 steps are checked against reference values. The tolerance is set to 0.1% for the mean and 1% for the variance. The reference values are provided in Table A.1.

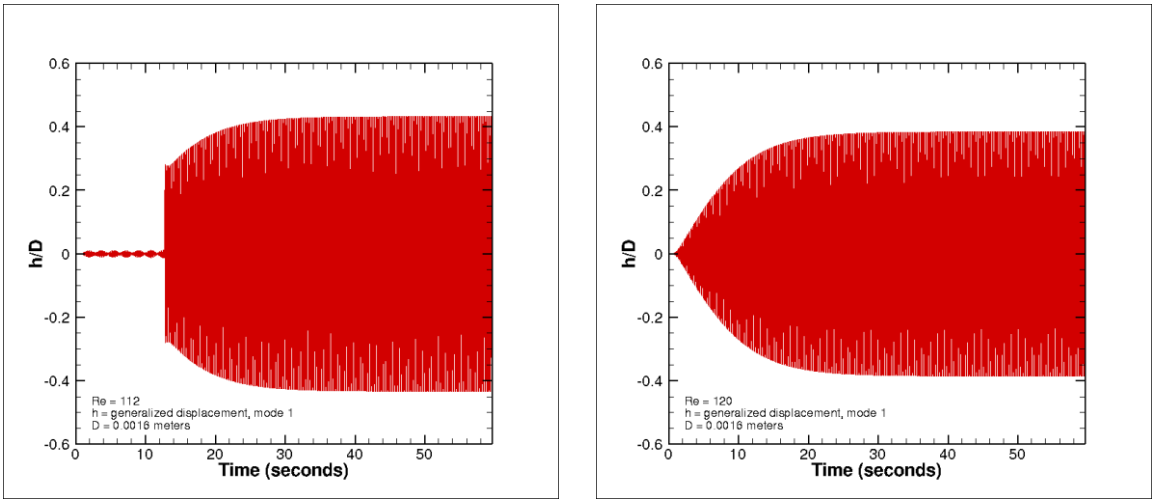


Figure A.1. Generalized displacement for the VIV case. Re = 112 (left), and Re = 120 (right).

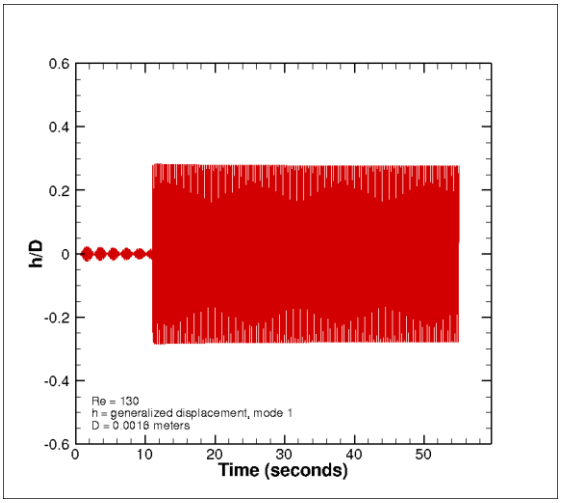


Figure A.2. Generalized displacement for the VIV case. Re = 130.

Table A.2: VIV Reference Values

Re	Reference Mean	Reference Variance
112	6.9261309E-04, -6.9268444E-04	7.9077064E-15, 8.0097116E-15
120	6.1567323E-04, -6.1575137E-04	5.2378125E-15, 5.1437498E-15
130	4.4499669E-04, -4.4501731E-04	6.2714526E-15, 6.1559374E-15

Benchmark Supercritical Wing (BSCW)

The FUN3D namelist file and the **`moving_body.input`** file for the Benchmark Supercritical Wing (Chwalowski et al. 2017) simulation are given below. This simulation is restarted from a restart file (5000 iterations) and run with no perturbation.

fun3d.nml

```
&project
    project_rootname = "bscw_coarse_mixed_nc"
/

&governing_equations
    eqn_type      = "compressible"
    viscous_terms = "turbulent"
/

&reference_physical_properties
    mach_number     = 0.74
    angle_of_attack = 0.00
    reynolds_number = 278399.75
/

&code_run_control
    steps        = 1000
    restart_read = "on"
    restart_write_freq=100
/

&nonlinear_solver_parameters
    time_accuracy    = "2ndorder"
    time_step_nondim = 1.2
    subiterations    = 15
/

&raw_grid
   grid_format = "aflr3"
   data_format = "stream"
/

&global
     moving_grid             = .true.
     boundary_animation_freq = -1
     volume_animation_freq   = 0
/

&boundary_output_variables
     number_of_boundaries = 1
     boundary_list        = "6"
/
```

moving_body.input

```
! ------------------------------------------------------------------------------------
! - BSCW
! ------------------------------------------------------------------------------------

&body_definitions
    n_moving_bodies     = 1
    body_name(1)        = "wing"
    n_defining_bndry(1) = 8
    defining_bndry(1,1) = 1
    defining_bndry(2,1) = 2
    defining_bndry(3,1) = 3
    defining_bndry(4,1) = 10
    defining_bndry(5,1) = 11
    defining_bndry(6,1) = 12
    defining_bndry(7,1) = 13
    defining_bndry(8,1) = 14
    motion_driver(1)    = "aeroelastic"
    mesh_movement(1)    = "deform"
/


&aeroelastic_modal_data
    nmode(1)                = 2
    grefl                   = 1
    uinf                    = 4508.4
    qinf                    = 1.1722
    gmass(1,1)              = 1.0
    gmass(2,1)              = 1.0
    freq(1,1)               = 20.923
    freq(2,1)               = 32.673

/
```

The BSCW test is restarted from 5000 steps and runs for an additional 1000 steps. Plots shown in Figure A.3 are posted on Jenkins on the successful completion of the simulation. Peaks for both the plunging and the pitching modes are compared with the reference values (Table A.3). The tolerance is set to 0.1%.

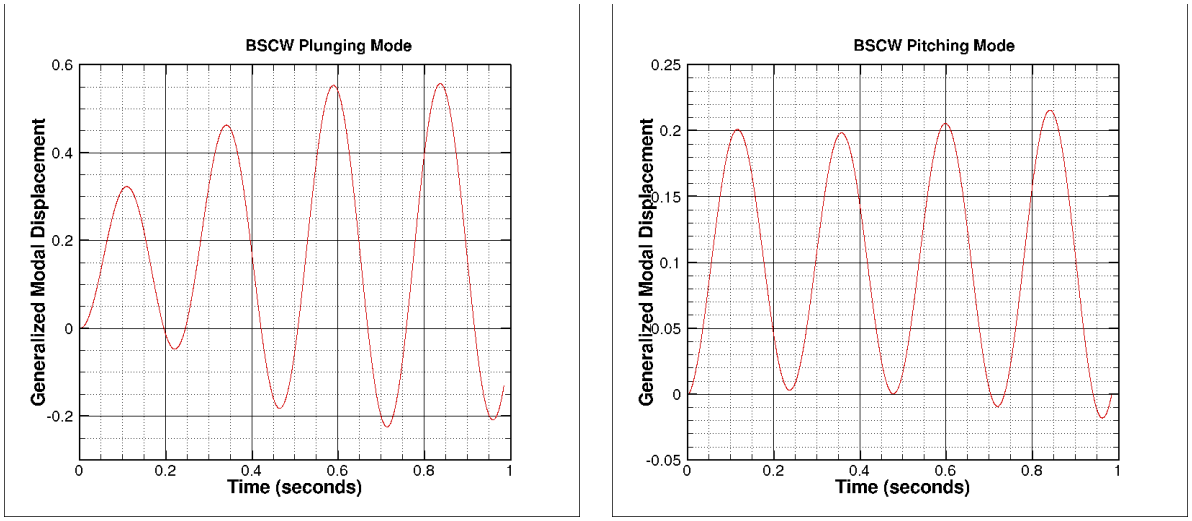


Figure A.3. Generalized displacements for the BSCW case. The plunging mode (left), and the pitching mode (right).

Table A.3: BSCW Reference Values

Mode	Reference Value
Plunging Mode	-0.2083332, 0.5575934
Pitching Mode	-0.0180195, 0.2156386

AGARD 445.6 Wing

The FUN3D namelist file for the AGARD wing (Yates 1987; Lee-Rausch and Batina 1993) simulation is provided below. A steady solution is obtained by running the simulation for 2000 steps. This solution is used as the restart for the dynamic run. A perturbation in velocity (**`gvel=0.4`**) is added for the dynamic run.

fun3d.nml (dynamic simulation)

```
&project
  project_rootname = 'agard1pw'
/

&raw_grid
  grid_format   = 'aflr3'
  patch_lumping = 'family'
/

&global
  moving_grid = .true.
  boundary_animation_freq = -1
/

&boundary_output_variables
  primitive_variables = .true.,
  cp = .true.,
/

&governing_equations
  viscous_terms = 'inviscid'
/

&reference_physical_properties
  mach_number = 0.9
/

&force_moment_integ_properties
  area_reference  = 548.0
  x_moment_length = 22.0
  y_moment_length = 30.0
  x_moment_center = 3.0
/

&nonlinear_solver_parameters
  time_accuracy        = '2ndorder'
  time_step_nondim     = 3.6
  subiterations        = 25
  schedule_cfl         = 50.0  50.0
  temporal_err_control = .true.
  temporal_err_floor   = 0.01
/

&code_run_control
  steps              = 2000
  restart_write_freq = 1000
  restart_read       = 'on_nohistorykept'
/

&special_parameters
  large_angle_fix = 'on'
/
```

The following **moving_body.input** file was used in the simulation:

moving_body.input

```
! ---------------------------------------------------------------------------------
! - AGARD
! ---------------------------------------------------------------------------------

&body_definitions
      n_moving_bodies = 1
         body_name(1) = 'airfoil'
  n_defining_bndry(1) = -1
     motion_driver(1) = 'aeroelastic'
     mesh_movement(1) = 'deform'
/


&aeroelastic_modal_data
    plot_modes = .true.
      nmode(1) = 4
       uinf = 11680.8
       qinf = 0.52083
     freq(1,1) =  60.3135016
     freq(2,1) = 239.7975647
     freq(3,1) = 303.7804433
     freq(4,1) = 575.1924565
  gmass(1:4,1) = 4*1.0
  gvel0(1:4,1) = 4*0.1
/
```

Figure A.4 shows the plot of generalized displacements which is posted on Jenkins on the successful completion of the simulation.

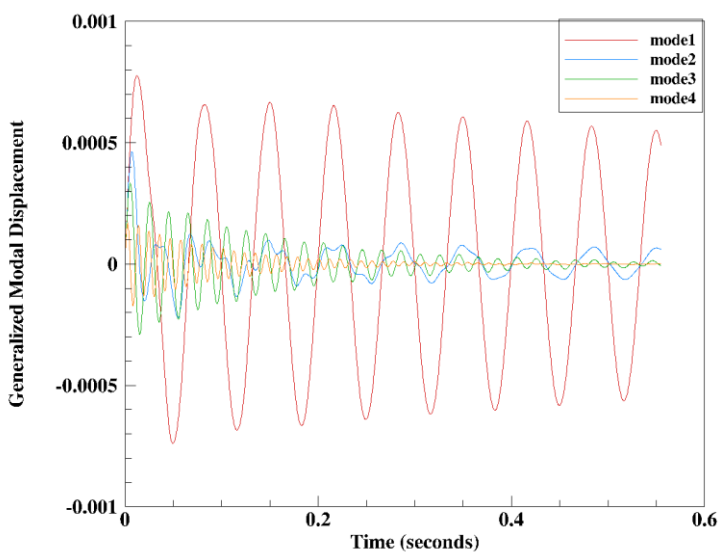


Figure A.4. Generalized displacements for all four modes is shown in the plot.

The damping ratio is found from logarithmic decrement by:

$$\zeta = \frac{1}{\sqrt{1+\left(\frac{2\pi}{\ln\left(x_0 / x_1\right)}\right)^2}}, \tag{A.1}$$

where x_0 and x_1 are two successive peaks of the generalized displacement. If $\zeta \ll 1$,

$$\zeta = \frac{\ln\left(x_0 / x_1\right)}{2\pi}. \tag{A.2}$$

The slope is obtained by performing a linear least square fit of the natural logarithm of generalized displacement peaks

$$\zeta = -\frac{slope}{2\pi f}, \tag{A.3}$$

where, f is the median frequency. The damping ratios and frequencies of the four modes are compared with reference values given in Table A.4. The tolerance is set to 0.1%.

Table A.4: AGARD Reference Values

Mode	Reference Damping Ratio	Reference Frequency (Hz)
1	0.005698	14.980801
2	0.001559	39.722305
3	0.020199	50.011787
4	0.019484	91.099510

Appendix B: Pseudo Code for Modal Fluid-Structure Interaction

A "call struc_XXX" is defined as an interface in libmodalstructure
A "call fsi_YYY" is defined as an interface in libmodalfsi

For readability, no arguments are shown for any subroutine call – all require at least one argument, an output integer indicating success (0) or falure (>0)

Flow solver actions indicated by "FLOW:" are not shown, but the nature of the action is described.

```
!-------------------------------------------------------------------------------
! data registration for modal solver
!-------------------------------------------------------------------------------

call struc_set_time_integration_scheme()  ! default: predictor-corrector

call struc_set_mach()             ! default: 0.0      (for output info only)
call struc_set_uinf()             ! default: 0.0      (for output info only)
call struc_set_qinf()             ! default: 0.0      (for output info only)

call struc_set_dt()               ! default: 0.0        (time step, sec.)
call struc_set_time()             ! default: 0.0        (start time, sec.)
call struc_set_force_basis()      ! default: face       (for or node)
call struc_set_complex_mode()     ! default: .false.    (ouput Im() part)
call struc_set_restart()          ! default: .true.     (is it a restart?)
call struc_set_project_rootname() ! default: my_project (restart file name)

call struc_set_nbodies()          ! default: 1

  body_loop : do body = 1,nbodies

      call struc_set_nmode()

      mode_loop : do j = 1, nmode(body)

! set initial conditions for generalized displacement, velocity and force generally,
! give gvel0 a non-zero value to initiate dynamic response

        call struc_set_gdisp0()    ! default: 0.0
        call struc_set_gvel0()     ! default: 0.0
        call struc_set_gforce0()   ! default: 0.0

! set modal properties

        call struc_set_gmass()     ! default: 0.0
        call struc_set_freq()      ! default: 0.0
        call struc_set_damp()      ! default: 0.0

! not shown: setter calls for never/infrequently-used perturbation options for
! initiating a dynamic response - all these options turned off by default; almost
! always use gvel0 to initiate

      end do mode_loop

  end do body_loop
```

```
! define the modal structural interface from mode-shape files and set mode shapes

! struc_read_mode_shapes reads Jamshid Samareh style mode shape files (Samareh 2001)
! and return the global (unpartitioned) structure interface and modal
! surface(s), but does NOT set up the required data in the modal solver.
! Must call struc_set_interface and struc_set_mode_shape to do that.

! note: modal solver does not know how to partition, but can be fed either
! partitioned or unpartitioned interface / mode shapes

  body_loop2 : do body = 1,nbodies

      mode_loop2 : do mode=1,nmodes

        call struc_read_mode_shapes()

! FLOW: optionally partition mode shapes and interface

        if (mode == 1 ) call struc_set_interface()

        call struc_set_mode_shape()

      end do mode_loop2

  end do body_loop2

!-------------------------------------------------------------------------------
! data registration for FSI module
!-------------------------------------------------------------------------------

    call fsi_set_nbodies()                  ! default: 1
    call fsi_set_force_basis()              ! default: face
    call fsi_set_fsi_mapping_tolerance()    ! default: 1.e-8

! Register both sides of the fluid/structure interface with the FSI module

    set_fsi_interface: do body = 1,nbodies

! FLOW: set convenience arrays to store struc_interface and fluid_interface
! FLOW: fill in the fluid_interface data

! Structure side: first retrieve the interface description from the structure
! module, then pass it to the FSI module

      call struc_get_interface()

      call fsi_set_struc_interface()

      call fsi_set_fluid_interface()

    end do set_fsi_interface


!-------------------------------------------------------------------------------
! Initialize the modal structural solver and FSI module
!-------------------------------------------------------------------------------

    call struc_initialize()

    call fsi_initialize()
```

```
!-----------------------------------------------------------------------------------
! Time step loop - assumes predictor-corrector scheme for structural dynamics
! equations.
!-----------------------------------------------------------------------------------

    time_stepping : do step = 1,nsteps

! FLOW: prepare to take a time step (but do not yet take it)

      call struc_start_timestep()

      call fsi_start_timestep()

      predctor_corrector : do subit = -1,0   ! -1 = predictor; 0 = corrector

        body_loop : do body = 1,nbodies

! FLOW: compute current forces on fluid interface

! Pass fluid forces to FSI module

          call fsi_set_fluid_force()

! Have the FSI module transfer the fluid-side loading to the structure

          call fsi_fld_to_str_force_xfer()

! Retrieve structure-side loading from FSI module

          call fsi_get_struc_force()

! Pass the forces on the structural interface to structure module

          call struc_set_force()

! Update structural solution - subit value routes to either predictor or corrector

          call struc_update_solution()

        end do body_loop

        predictor_only : if (subit == -1) then

! update CFD surface meshes

          body_loop : do body = 1,nbodies

! Retrieve the current interface motion (displacement) from the structural solver,
! and pass to the FSI module

            call struc_get_movement()

            call fsi_set_struc_movement()


! Have the FSI module interpolate movements (disp, vel, accel) from structure side
! fluid side and retrieve the interpolated values

            call fsi_str_to_fld_movement_xfer()

            call fsi_get_fluid_movement()
```

```
! FLOW: Add the displacements to the t=0 surface mesh (xs0, ys0, zs0)
! xs = xs0 + xdisp/length_factor
! ys = ys0 + ydisp/length_factor
! zs = zs0 + zdisp/length_factor

          end do body_loop

! FLOW:
! 1) deform volume mesh, given current xs, ys, zs
! 2) flow solver now takes a time step

        end if predictor_only

      end do predictor_corrector

      call struc_end_timestep()

      call fsi_end_timestep()

    end do time_stepping

! Finalize the modal solver and fsi module; all memory deallocated and all variables
! reset to default values

    call struc_finalize()

    call fsi_finalize()
```

References

Anagnostopoulos, P., Bearman, P. W., "Response Characteristics of a Vortex-Excited Cylinder at Low Reynolds Numbers," *Journal of Fluids and Structures*, Vol. 6, Issue 1, 1992, pp 39–50.

Anderson W. K., Newman, J. C., Whitfield, D. L., Nielsen, E. J., "Sensitivity Analysis for the Navier-Stokes equations on unstructured meshes using complex variables," AIAA Paper 1999–3294.

Bartels, R. E., Rumsey, C. L., Biedron, R. T., "CFL3D Version 6.4 - General Usage and Aeroelastic Analysis," National Aeronautics and Space Administration, 2006, NASA/TM–2006–214301.

Bhatia, M., Beran, P., "MAST: An Open-Source Computational Framework for Design of Multiphysics Systems," AIAA Paper 2018–1650.

Biedron, R. T., Carlson, J., 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., Wood, W. A., "FUN3D Manual: 13.3," National Aeronautics and Space Administration, 2018, NASA/TM–2018–219808.

Biedron, R. T., Thomas, J. L., "Recent Enhancements to the FUN3D Flow Solver for Moving-Mesh Applications," AIAA Paper 2009–1360.

Biedron, R. T., Vatsa, V. N., Atkins, H. L., "Simulation of Unsteady Flows Using an Unstructured Navier-Stokes Solver on Moving and Stationary Grids," AIAA Paper 2005–5093.

Chwalowski, P., Heeg, J., Biedron, R. T., "Numerical Investigations of Benchmark Supercritical Wing In Transonic Flow," AIAA Paper 2017–0190.

Edwards, J. W., Bennett, R. M., Whitlow, W. J., Seidel, D. A., "Time-Marching Transonic Flutter Solutions Including Angle-of-Attack Effects," *AIAA Journal*, Vol. 20, No. 11, 1983, pp. 899–906.

Irons, B. M., Tuck, R. C., "A Version of the Aitken Accelerator for Computer Iteration," *International Journal for Numerical Methods in Engineering*, Vol. 1, No. 3, 1969, pp. 275–277.

Jacobson, K., Kiviaho, J. F., Smith, M. J., Kennedy, G., "An Aeroelastic Coupling Framework for Time-accurate Analysis and Optimization," AIAA Paper 2018–100.

Kennedy, G., Martins, J. R. R. A., "A parallel finite-element framework for large-scale gradient-based design optimization of high-performance structures," *Finite Elements in Analysis and Design*, Vol. 87, 2014, pp. 56–73.

Kiviaho, J. F., Jacobson, K., Smith, M. J., Kennedy, G., "A Robust and Flexible Coupling Framework for Aeroelastic Analysis and Optimization," AIAA Paper 2017–4144.

Lee-Rausch, E. M., Batina, J. T., "Calculation of AGARD Wing 445.6 Flutter Using Navier-Stokes Aerodynamics," AIAA Paper 1993–3476.

Lyness J. N., Moler, C. B., "Numerical differentiation of analytic functions," *SIAM Journal of Numerical Analysis*, Vol. 4, No. 2, 1967, pp 202–210.

Newman, J. C., Anderson, W. K., Whitfield, D. L., "Multidisciplinary Sensitivity Derivatives Using Complex Variables," MSSU-COE-ERC-98-09, Engineering Research Center Report, Mississippi State University, 1998.

Samareh, J. A., "Novel Multidisciplinary Shape Parameterization Approach," *Journal of Aircraft*, Vol. 38, No. 6, 2001, pp 1015–1024.

Snyder R. D., "A Cross-Language Remote Procedure Call Framework," AIAA Paper 2017–3822.

Squire, W., Trapp, G., "Using Complex Variables to Estimate Derivatives of Real Functions," *SIAM Review*, Vol. 40, No. 1, 1998, pp 110–112.

Yates, E. C., "AGARD Standard Aeroelastic Configurations for Dynamic Response. Candidate Configuration I. – Wing 445.6," National Aeronautics and Space Administration, 1987, NASA/TM–1987–100492.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
1-09-2018

2. REPORT TYPE
Technical Memorandum

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Sensitivity Analysis for Multidisciplinary Systems (SAMS)

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
031102.02.07.05.93O4.17

6. AUTHOR(S)
Biedron, Robert T.; Jacobson, Kevin E.; Jones, William T.; Massey, Steven J.; Nielsen, Eric J.; Kleb, William L.; Zhang, Xinyu

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-20932

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-2018-220089

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified
Subject Category 02
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes the research conducted under an interagency collaboration agreement between the Aerospace Systems Directorate of the Air Force Research Laboratory (AFRL/RQ) and the Computational AeroSciences Branch of NASA Langley (NASA LaRC). Both organizations have a long-term goal of developing a modular computational system for coupling fluids and structures to enable both analysis and optimization of aerospace vehicles. Ultimately, the system should support multiple solvers within the fluid and structure domains to allow the best combination for the task at hand, as well as to allow for institutional preferences of specific software components. Towards this goal, the current research was focused on enhancing the existing modal aeroelastic analysis in the NASA FUN3D software (Biedron et al. 2018), as well as developing new aeroelastic analysis and optimization capabilities based on a non-linear finite-element method. The methods and enhancements described in this document pertain to FUN3D Version 13.4.

15. SUBJECT TERMS
Aerodynamics; Computational fluid dynamics; FUN3D; Mathematics; Mechanics; Propulsion system

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 Help Desk (email: help@sti.nasa.gov)
U	U	U	UU	63	19b. TELEPHONE NUMBER *(Include area code)* (757) 864-9658

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18