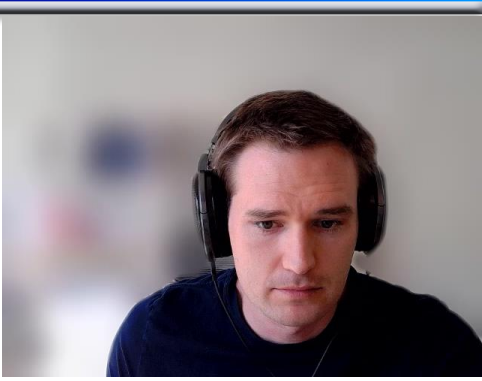
FUN3D v14.0 Training
Overset simulations with Yoga
Cameron Druyor

Overview

- What is Yoga?
  - Library – what and why
  - Executable – purpose and usage
- Demo simple case
  - Generating composite grid
    - Input files
    - Spot check / visualize
    - Output files
  - Standalone domain assembly
    - Input files
    - Spot check / visualize
- Demo rotorcraft case
  - Auto-generate Yoga inputs from FUN3D inputs
  - Repeat steps from previous demo case

FUN3D
Fully Unstructured Navier-Stokes

Yoga



This tutorial assumes some familiarity using FUN3D with Suggar++ as demonstrated in previous tutorials and workshops, which are available on the FUN3D website:

https://fun3d.larc.nasa.gov/tutorial-2.html#overset_moving_grids

https://fun3d.larc.nasa.gov/training-7.html

What is Yoga?

- Yoga library
  - Alternative to Suggar++ library for FUN3D with minimal workflow changes
  - Performs domain assembly in parallel with dynamic load balancing
  - Called by FUN3D at each time step
    - Determine overset boundary locations
    - Identify donors/receptors
    - Calculate interpolation weights
    - Exchange/interpolate solution values
  - Runs on same MPI ranks as FUN3D
    - No extra, high memory node required
    - Scalable to hundreds of millions of grid points on thousands of MPI ranks

- Yoga executable
  - Self documenting
  - Combine component grids -> composite grid
  - Standalone domain assembly
    - Spot check donor/receptor/orphan counts
    - Visualize via ParaView / Tecplot
    - Export FUN3D partition files
  - Extract rotorcraft-specific input data from FUN3D input files to maintain consistency

Building FUN3D with Yoga

FUN3D v14 ships with Yoga, which can be enabled by adding the following pair of options:

```
--enable-yoga --with-nanoflann=/path/to/nanoflann
```

Where *nanoflann* is a header-only nearest-neighbor library that Yoga uses:

https://github.com/jlblancoc/nanoflann/archive/refs/tags/v1.3.0.tar.gz

YOGA is incompatible with the latest version of nanoflann

Please use version 1.3.0 for YOGA compilation

Yoga executable

The Yoga executable uses a sub-command model, much like other tools like `git`

yoga <subcommand> <args>

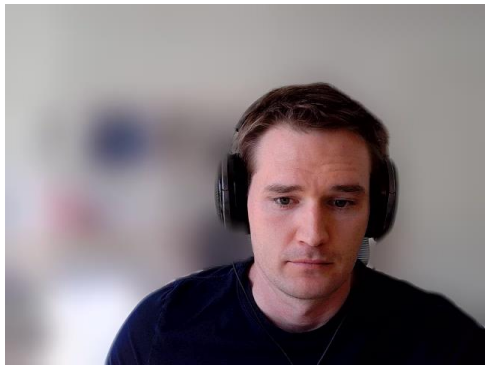
Yoga is self documenting via `-h` or `--help`, like many Unix utilities:

yoga --help
yoga -h

Which will print out all available subcommands and a description of what they do. Each subcommand is also self documenting, so

yoga <subcommand> --help

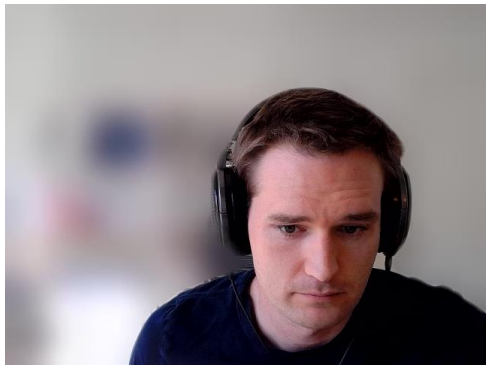
Will print out information about argument names, defaults, and descriptions for the selected subcommand



Store separation tutorial case

This tutorial case walks through the steps for creating a composite grid for FUN3D, running a steady-state case, and finally running with specified motion.

All required input files are provided along with a README in each directory to describe each phase.



store_separation/
- README
- grids/
  - README
  - composite.txt
  - wing.lb8.ugrid
  - wing.mapbc
  - store.lb8.ugrid
  - store.mapbc
  - yoga_bcs.txt
- steady_state/
  - README
  - fun3d.nml
- rigid_motion/
  - README
  - fun3d.nml
  - moving_body.input

Examine component grids

First, let's take a look at the component grids to make sure that everything looks reasonable geometrically. By looking at the mapbc files of each component, we can see that the solid surfaces of the wing are defined by boundary tags 6–14 and the solid surfaces of the store are defined by tags 1–44.

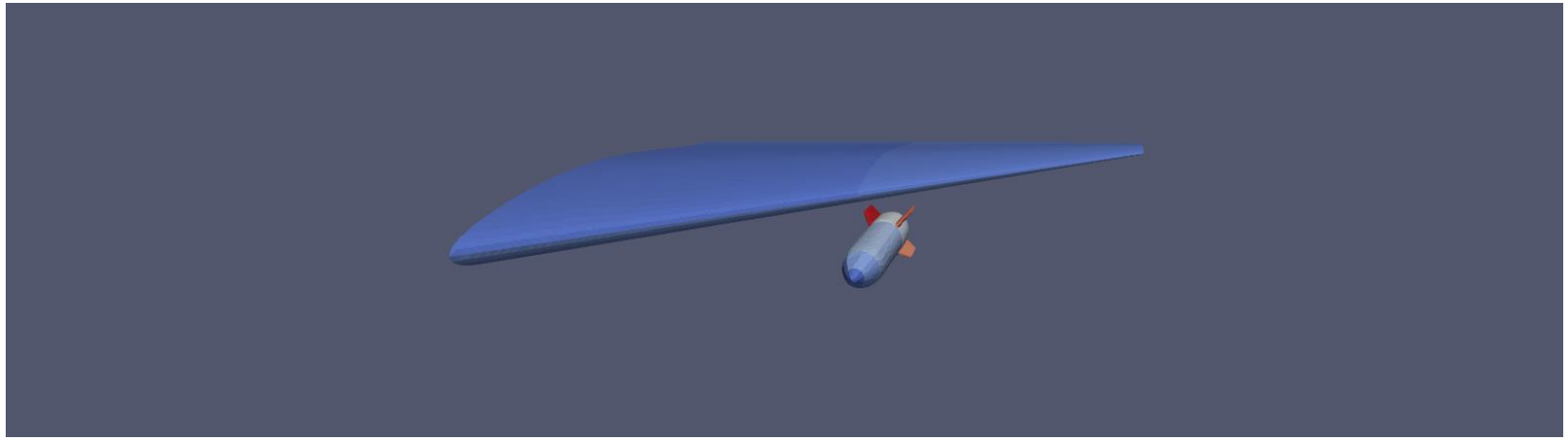
Let's extract the surfaces from each grid and export to vtk (or change the extension to *.plt to get TecPlot instead):

```
cd grids

inf plot --mesh wing.lb8.ugrid --tags 6:14 -o wing-surface.vtk

inf plot --mesh store.lb8.ugrid --tags 1:44 -o store-surface.vtk
```

Now, we can open in ParaView (or TecPlot) and verify that the boundaries are where we expect:



Creating a composite mesh

Yoga's help system lists information about input arguments for each subcommand, e.g.,

```
yoga make-composite --help
```

Required arguments are marked as such, and the one we are interested in for this case is:

```
 -f --file <value> [REQUIRED]
      name of composite builder script
```

which is the yoga input file that specifies the names of the component grids, their boundary condition files, and any translations or rotations to apply to the initial composite grid.

So, the full command for generating the composite grid with Yoga, from within the *grids* subdirectory, is:

```
yoga make-composite --file composite.txt -o wingstore.b8.ugrid
```

Analogous Suggar++ command:

```
suggar++ Input.xml
```

Creating a composite mesh

Three files are created by this command:

1. wingstore.b8.ugrid
2. wingstore.mapbc
3. imesh.dat

Where 1 and 2 contain the composite grid, and 3 contains metadata about each component grid in the composite. In particular, the *imesh.dat* file is plain text:

```
2
62014 1
79012 0
```

The first line is the number of component grids. Then there is a line for each component that lists the number of grid points in the component and the "imesh" value for that component.

imesh.dat is analogous to xxxxx.dci *in the Suggar++ workflow*

We can verify that the order is correct by checking the grid point counts of the component grids, e.g.,

```
inf examine --mesh store.lb8.ugrid
```

which will print:

```
Nodes in grid      : 62,014
Total cells in grid: 358,759
TRI_3: 15,258
TETRA_4: 343,501
```

(repeat for wing.lb8.ugrid)

Yoga – Composite grid input file

```
# Comment lines beginning with `#` are ignored
# Input grids can be little endian, big endian,
# or a combination of the two.

grid store.lb8.ugrid
mapbc store.mapbc

# For moving body simulations, FUN3D's numbering convention
# dictates that moving body "imesh" values start at 1,
# and the stationary, or "background", grid should
# have an "imesh" value of 0.  To facilitate this,
# Yoga assumes that the last component grid is the background
# grid and will assign it an index of 0 in the "imesh.dat" file.
# The other component grids can be in any order, but
# the last grid is expected to be the stationary one (which
# can be overridden by manually changing imesh.dat if necessary)

grid wing.lb8.ugrid
mapbc wing.mapbc
```

For Suggar++, component grids and their transformations are specified similarly in Input.xml. Note that boundary conditions are also set directly in Input.xml, whereas Yoga extracts boundary tag information from the *.mapbc files.

From demo case: /store_separation/grids/composite.txt

Yoga – Composite grid input file

Yoga's help system also describes syntax for input files, which can be accessed via:

```
yoga show-syntax --help
```

In particular,

```
yoga show-syntax --composite
```

Shows how different translations, rotations, and arbitrary movement transformation matrices can be applied to individual component grids if required for a particular simulation.

Domain assembly dry run

The following command will run Yoga in standalone mode on 8 MPI ranks and generate a visualization of the assembled grid system:

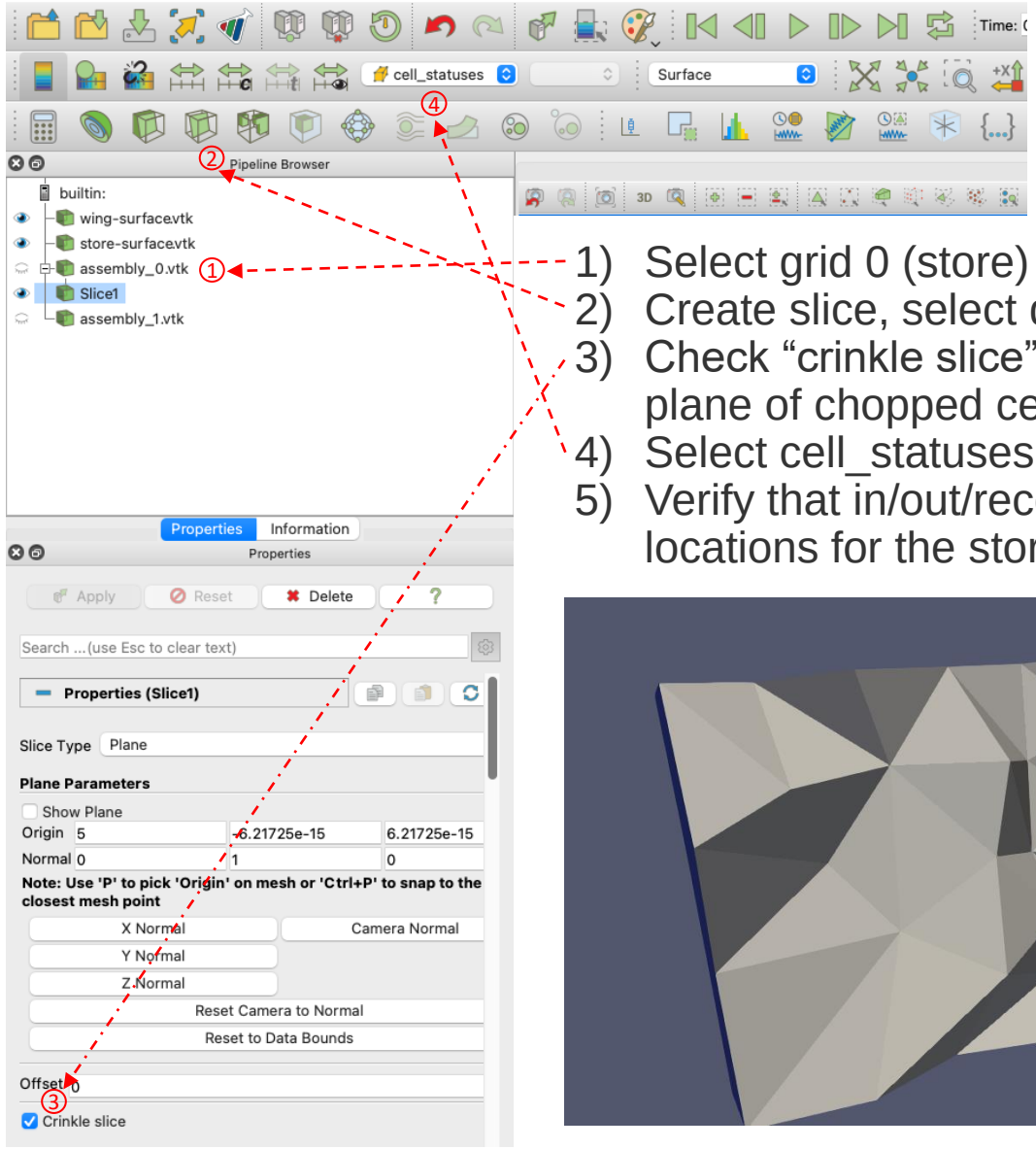
```
mpiexec_mpt -np 8 yoga assemble --file composite.txt --bc yoga_bcs.txt --viz assembly.vtk
```

(here again, changing the extension from .vtk -> .plt will yield a TecPlot binary)

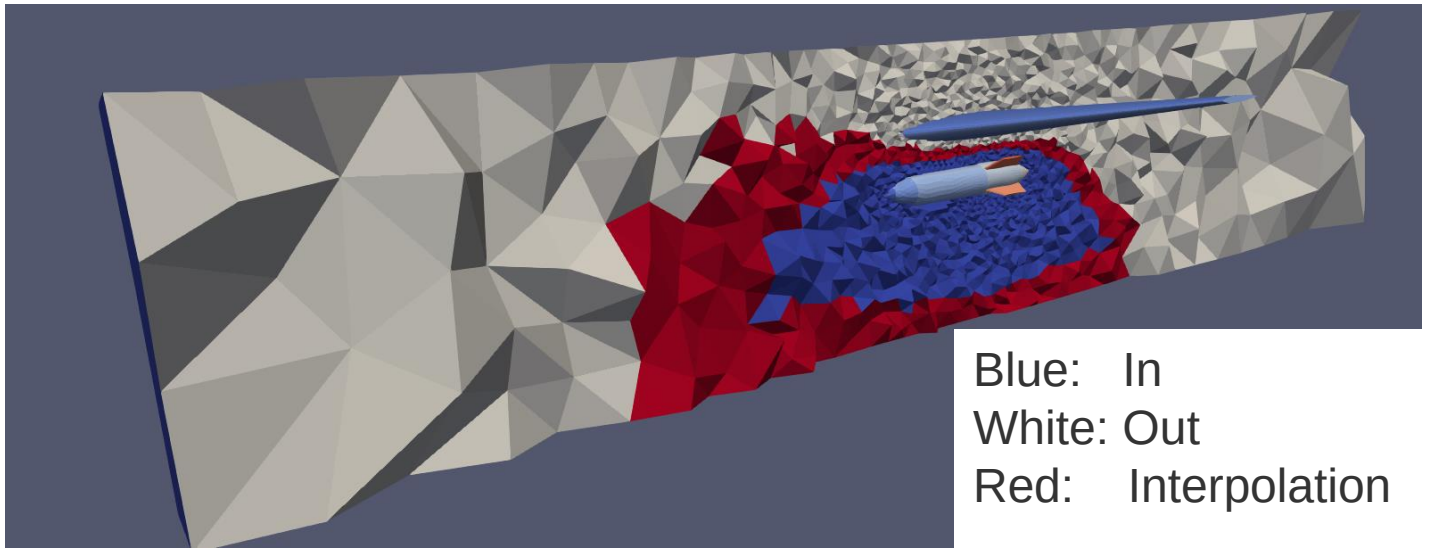
Yoga prints some diagnostics during assembly, and we're interested in the section at the end of the output that prints the global counts of in/out/receptor/orphan nodes. Zero is the target number of orphans, but larger and more complex cases will often have a small fraction of orphan points. If this number of orphans is more than ~1% of the number of receptors, that indicates that the something might be wrong with the case setup (incorrect boundary conditions, poorly matched grid spacing, incorrect grid locations, etc).

```
Yoga: in: 70229 out: 64441 receptor: 6356 orphan: 0
Total assembly time: 0.749044 seconds
max mean imbalance name
0.12 0.11 1.11 buildReceptorCollections
0.12 0.11 1.15 overdecompose
0.01 0.01 1.42 pack
```

ParaView for Overset Grids



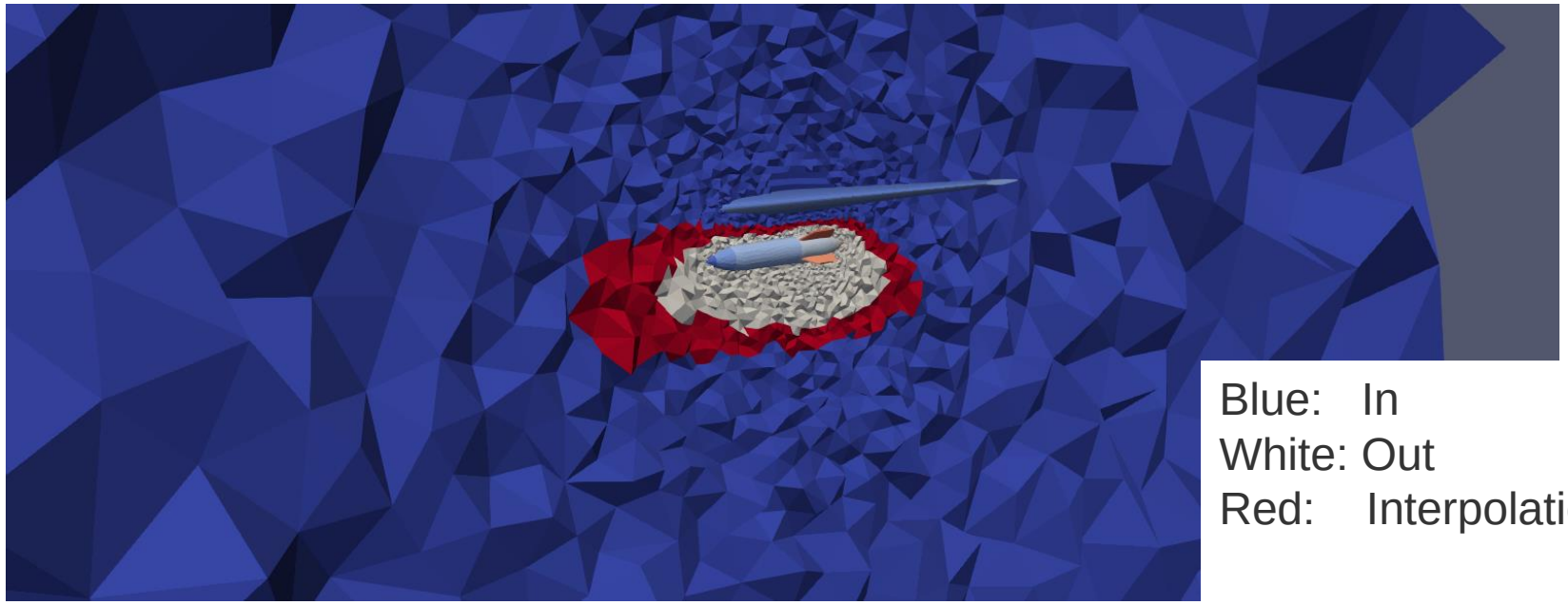
1) Select grid 0 (store)
2) Create slice, select direction/location
3) Check “crinkle slice” to get full cells (not 2D plane of chopped cells)
4) Select cell_statuses
5) Verify that in/out/receptor cells are in expected locations for the store grid:



Blue: In
White: Out
Red: Interpolation

ParaView for Overset Grids

Then take a slice of the wing grid at the same location (y=0.0):



Blue: In
White: Out
Red: Interpolation

- ✓ Zero / few orphan points
- ✓ Interpolation boundaries
  - ✓ Contiguous
  - ✓ Relatively far from solid walls of each body
- ✓ In points near solid walls on each grid
- ✓ Points outside interpolation boundaries marked out

Steady state simulation

The provided fun3d.nml file has the following addition:

```
&overset_data
  assembler = 'yoga'
  input_imesh_file = 'imesh.dat'
  dci_on_the_fly = .true.
  overset_flag = .true.
/
```

- FUN3D uses Suggar++ by default, so we set *assembler* to *yoga* to override
- *imesh.dat* is the text file that contains component grid node-counts and ids
- *dci_on_the_fly* is only ever false when running with Suggar++ .dci files
- *overset_flag* should always be set to true when running with Yoga or Suggar++

Note that the last two options could also be specified at the command line when running FUN3D, but setting via namelist is recommended.

A number of FUN3D options that are available when using Suggar++ have no effect when using Yoga, including:

- *skip_dci_output*
- *reuse_existing_dci*
- *dci_period*
- *reset_dci_period*
- *dci_freq*
- *dci_dir*
- *input_xml_file*

Steady state simulation

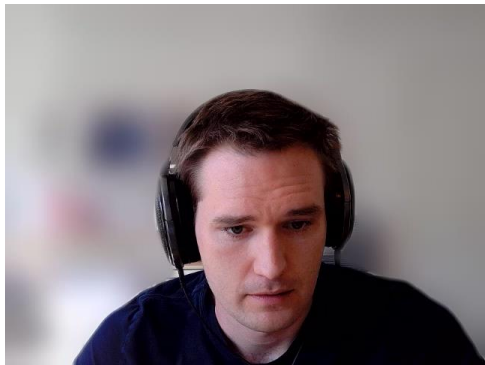
In the steady_state/ directory, create soft links to the composite grid files:

```
ln -s ../grids/wingstore.b8.ugrid .
ln -s ../grids/wingstore.mapbc .
ln -s ../grids/imesh.dat .
```

Then launch FUN3D using the appropriate mpi command for your system, e.g.,

```
mpiexec_mpt -np 8 nodet_mpi
```

Then FUN3D will run the prescribed 500 iterations and write out a restart file that we will use as a starting point for the dynamic version of this case.



Prescribed motion simulation

Moving grid is enabled via the *global* namelist:

```
&global
  moving_grid = .true.
  boundary_animation_freq = 5
/
```

With *moving_grid* enabled, FUN3D will read *moving_body.input* to determine which bodies need to be moved and what type of motion to use:

```
&body_definitions
  n_moving_bodies = 1,       ! number of bodies in motion
  body_name(1) = 'store',    ! name must be in quotes
  n_defining_bndry(1) = 1,   ! number of boundaries that define this body
  defining_bndry(1,1) = 4,   ! index 1: boundary number index 2: body number
  mesh_movement(1) = 'rigid', ! 'rigid', 'deform'
  motion_driver(1) = 'forced' ! motion is specified below
/

&forced_motion
  translate(1) = 1,          ! translation type: 1=constant rate 2=sinusoidal
  translation_rate(1) = -0.2, ! translation velocity (mach)
/
```

Prescribed motion simulation

Again, link the composite grid files to the current directory:

```
ln -s ../grids/wingstore.b8.ugrid .
ln -s ../grids/wingstore.mapbc .
ln -s ../grids/imesh.dat .
```

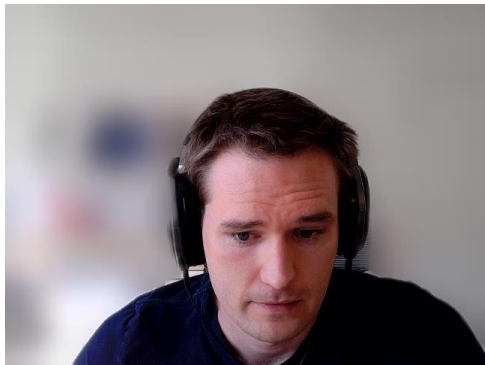
Also, copy over the restart file from the steady run to minimize the effect of the initial transients.

```
cp ../steady_state/wingstore.flow .
```

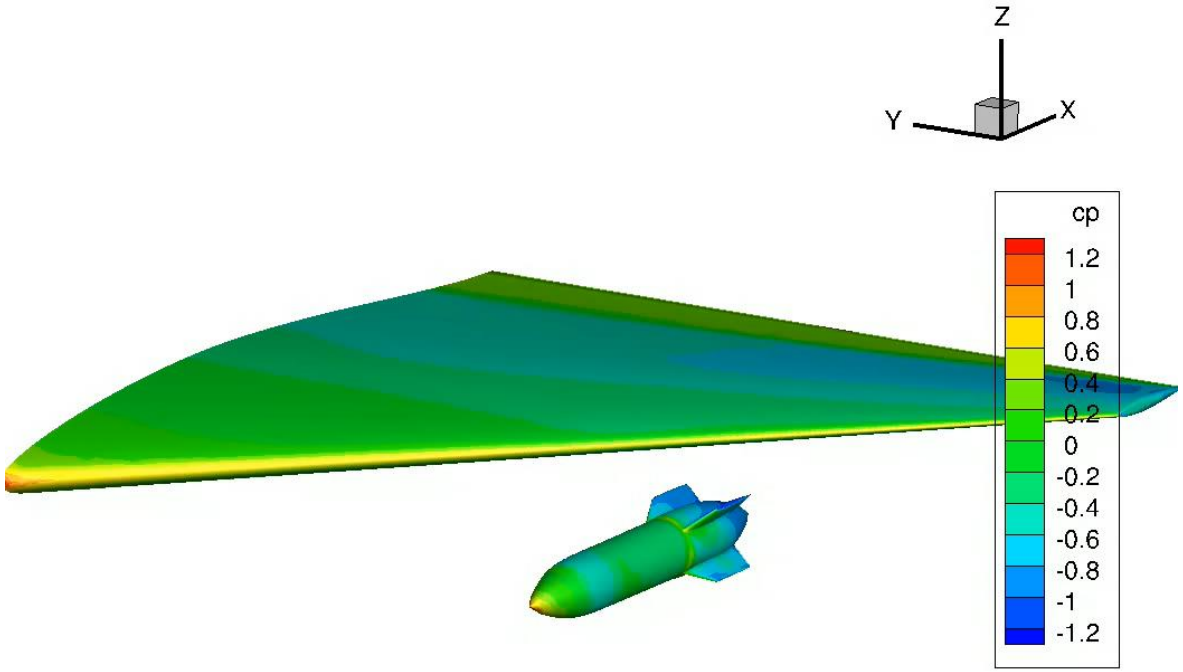
Then launch FUN3D using the appropriate mpi command for your system, e.g.,

```
mpiexec_mpt -np 8 nodet_mpi
```

This time, FUN3D will generate a boundary plot every 5 time steps that we can use to verify that the prescribed motion is correct.



Viewing grid motion in TecPlot

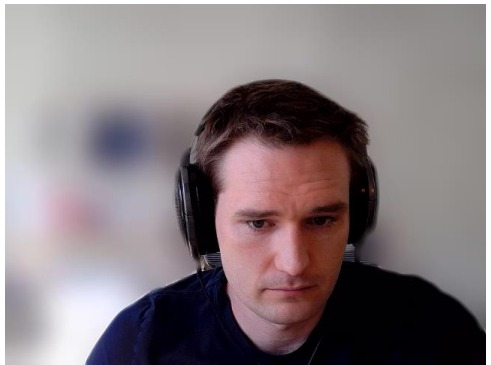


Rotorcraft tutorial case

This tutorial case walks through the same steps as the previous example, with some additions that are unique to rotorcraft simulations.

All required input files are again provided along with a README in each directory to describe each phase.

quadrotor/
- README
- grids/
  - README
  - blade-cw.b8.ugrid
  - blade-cw.mapbc
  - blade-ccw.b8.ugrid
  - blade-ccw.mapbc
  - box.b8.ugrid
  - box.mapbc
  - yoga_bcs.txt
- flow/
  - README
  - fun3d.nml
  - rotor.input
  - moving_body.input

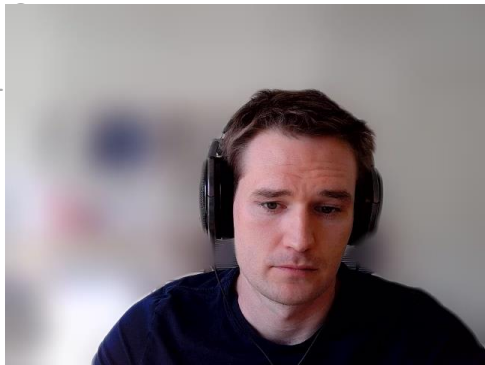


Rotorcraft tutorial case

rotor.input specifies:

- Number of rotors
- Blades per rotor
- Location of each rotor
- Orientation of each rotor

```
  # Rotors  Vinf_ratio  Write Soln   Force Ref Momment Ref
        4     0.3732        1500         1.0        1.0
=== Front Right Rotor ========================================
Rotor Type   Load Type    # Radial    # Normal  Tip Weight
         1           1          50         180         0.0
  X0_rotor    Y0_rotor    Z0_rotor        phi1        phi2        phi3
-12.441437   12.441437     2.30397        0.00        0.00        0.00
  Vt_ratio  ThrustCoff   PowerCoff        psi0  PitchHinge      DirRot
       1.0     0.00457       -1.00        0.00         0.0
  # Blades   TipRadius  RootRadius  BladeChord   FlapHinge     LagHi
         3      9.2150      1.1058     0.57657         0.0    0.0
```



Rotorcraft tutorial case

Yoga provides a subcommand to extract information from *rotor.input* and *moving_body.input* to generate an input file with the proper translations/rotations for each component grid.

So first, from the grids directory, link the input files:

```
cd grids
ln -s ../flow/rotor.input .
ln -s ../flow/moving_body.input .
```

Then run the following command to generate Yoga's input file:

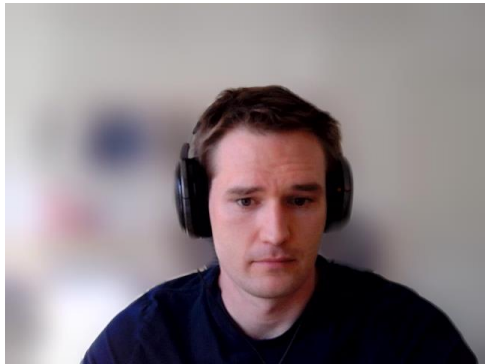
```
yoga composite-rotor
```

Analogous to running `dci_gen`
to generate `Input.xml` for `Suggar++`

Generating Yoga input file

Inside the subcommand *yoga composite-rotor*, Yoga first parses the FUN3D input files to determine how many rotors and blades there are and where they go. Then it prompts the user to fill in the grid names for each rotor and the background/fuselage grid. Note that Yoga will provide a suggested name, which can be selected via "." if the guessed name is correct. (user input highlighted)

```
Reading: rotor.input
Reading: moving_body.input
Number of rotors: 4
Enter blade grid name for <rotor-0> <counter-clockwise> (or '.' to use rotor.b8.ugrid): blade-ccw.b8.ugrid
Enter mapbc name (or '.' to use blade-ccw.mapbc): .
Enter blade grid name for <rotor-1> <clockwise> (or '.' to use blade-ccw.b8.ugrid): blade-cw.b8.ugrid
Enter mapbc name (or '.' to use blade-cw.mapbc): .
Enter blade grid name for <rotor-2> <clockwise> (or '.' to use blade-cw.b8.ugrid): .
Enter mapbc name (or '.' to use blade-cw.mapbc): .
Enter blade grid name for <rotor-3> <counter-clockwise> (or '.' to use blade-cw.b8.ugrid): blade-ccw.b8.ugrid
Enter mapbc name (or '.' to use blade-ccw.mapbc): .
Enter background grid filename: box.b8.ugrid
Enter mapbc name (or '.' to use box.mapbc): .
```



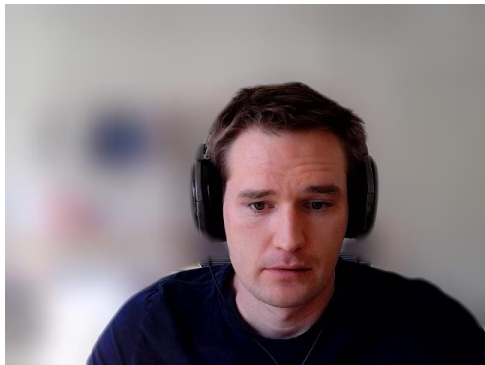
Generating Yoga input file

The generated *composite.txt* file should now contain a section for each rotor blade that looks like the following:

```
grid rotor.b8.ugrid
mapbc rotor.mapbc
domain rotor1_blade2
move
-1.0000000000000000e+00 -1.2246467991473532e-16 0.0000000000000000e+00 5.0700000000000012e+00
1.2246467991473532e-16 -1.0000000000000000e+00 0.0000000000000000e+00 -1.8750000000000000e+01
0.0000000000000000e+00 0.0000000000000000e+00 1.0000000000000000e+00 6.7300000000000004e+00
0.0000000000000000e+00 0.0000000000000000e+00 0.0000000000000000e+00 1.0000000000000000e+00
```

Name extracted from *moving_body.input*

Translations/Rotations calculated from *rotor.input*



Generating the composite grid

Now, create the composite mesh as in the previous tutorial, but using the generated input file:

```
yoga make-composite --file composite.txt -o quadrotor.b8.ugrid
```

The composite mapbc file should now contain boundary conditions for all 12 blades and the background grid:

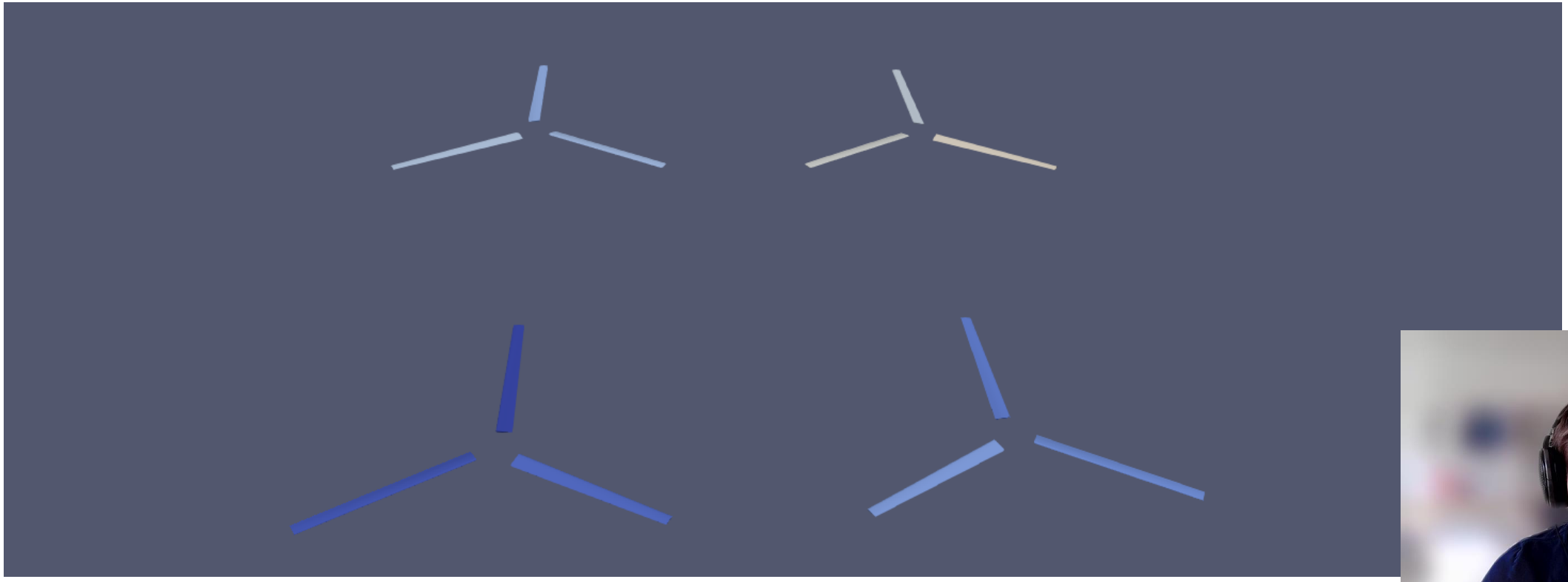
```
25
1 4000 rotor1_blade1
2 -1 rotor1_blade1_outer
3 4000 rotor1_blade2
4 -1 rotor1_blade2_outer
5 4000 rotor1_blade3
...
25 5000 box
```

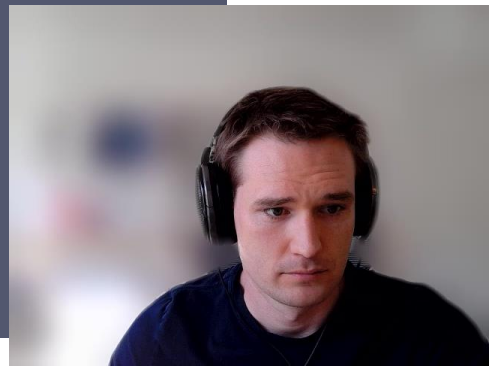
Automatically renamed to match body name in *moving_body.input*

Visual inspection / spot check

Now, let's plot all 12 of the rotor blades by selecting their tags to make sure that they are in the expected locations:

```
mpiexec_mpt –np 8 inf plot --mesh quadrotor.b8.ugrid --tags 1,3,5,7,9,11,13,15,17,19,21,23 -o rotor-surfaces.vtk
```

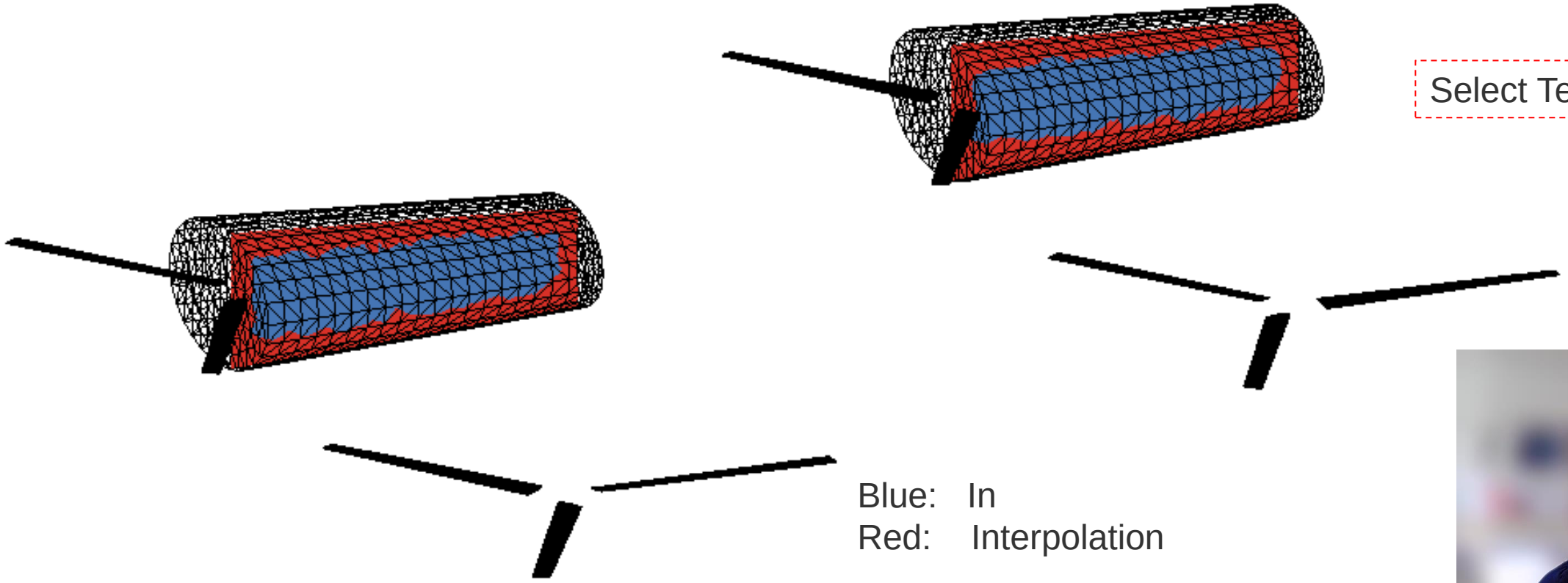




Domain assembly dry-run

Again, we can check the domain assembly by running Yoga in standalone mode to make sure everything looks ok before running with FUN3D. **Note**, you will need at least the number of ranks as the number of component grids due to the I/O process used by standalone Yoga (13 for this case).

```
mpiexec_mpt –np 13 yoga assemble --file composite.txt --bc yoga_bcs.txt --viz assembly.plt
```

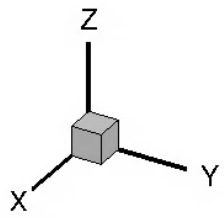


Select Tecplot this time

Blue: In
Red: Interpolation

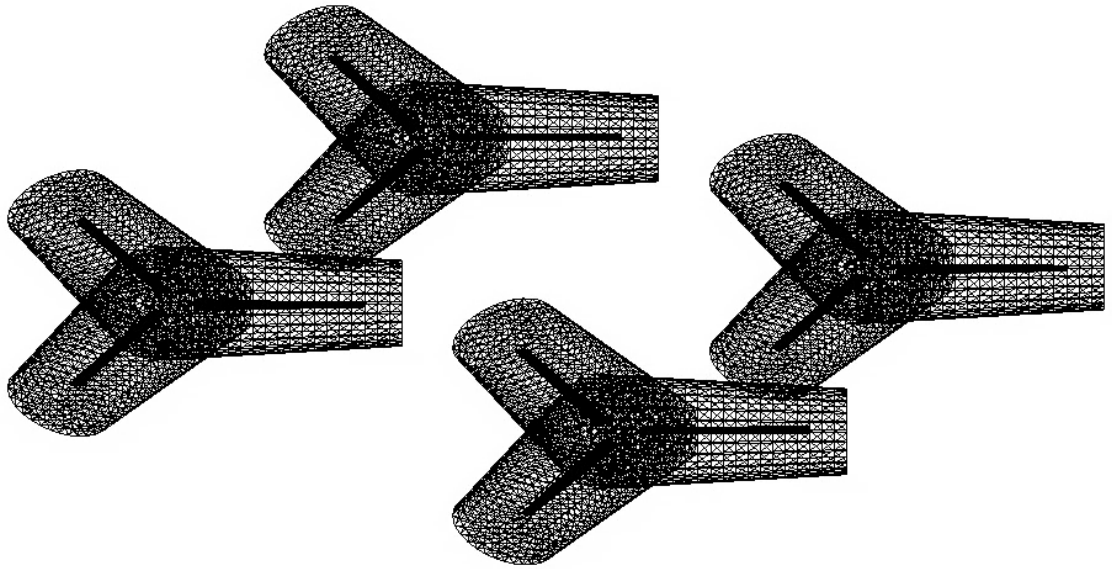
Checking motion inputs

Turn on grid-motion-only and run ~45 steps with boundary animation to make sure the rotors are turning in the expected directions. Note that the *&overset_data* namelist has no additional requirements beyond those covered for a non-rotorcraft case.



From within the flow directory, link the composite mesh files:

```
cd ../flow
ln –s ../grids/quadrotor.b8.ugrid .
ln –s ../grids/quadrotor.mapbc .
ln –s ../grids/imesh.dat .
```



Uncomment the line in the fun3d.nml:

```
grid_motion_only = .true.
```

And change the boundary output frequency:

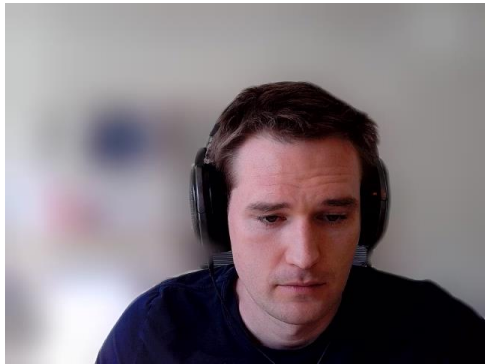
```
boundary_animation_freq = 5
```

Then run FUN3D (Note, any number of cores may be used)

```
mpiexec_mpt –np 40 nodet_mpi –ncyc 45 | grep Yoga | grep orphan
```

This will take some time as the pipes buffer the output. Verify that the resulting output for each step has similar numbers of in/out/receptor/orphan points. They should be different as the grids move, but not drastically so.

```
Yoga: in: 992837 out: 1398 receptor: 17099 orphan: 0
```



Running FUN3D

After confirming that:

- Domain assembly appears correct
  - Few orphan points relative to receptor points
  - Interpolation boundaries in plausible locations
- Motion inputs are correct
  - Blades move as expected with *grid_motion_only*
  - Domain assembly statistics consistent across time steps

Comment the line in the fun3d.nml:

```
!grid_motion_only = .true.
```

The final step is to run FUN3D in the usual way:

```
mpiexec_mpt –np 40 nodet_mpi
```

What We Learned

- How to use *yoga* executable and subcommands to:
  - Create composite grids
  - Spot check assembly
    - Visually in Tecplot / ParaView
    - By checking in/out/receptor/orphan counts
  - Extract rotorcraft-specific input data via Yoga utility
    - Rotor position, orientation, blade-count from *rotor.input*
    - Body names from *moving_body.input*
- Input file format and requirements
  - Yoga – composite grid input
  - Yoga – boundary condition input
  - FUN3D – namelist parameter changes