# FUN3D v12.4 Training

# Session 12: Suggar++ Basics

Bob Biedron

# Session Scope

- What this will cover
  - Very rudimentary SUGGAR++ operation
- What will not be covered
  - All the useful stuff that Ralph Noack would teach you
  - GVIZ (Ralph's own viewer for overset grid assembly - useful for debugging/assessing hole cutting)
- What should you already be familiar with
  - Basic concept of overset meshes

# Introduction

- Background
  - Use of overset grids in FUN3D requires SUGGAR++
    - Disclaimer: I am not a SUGGAR++ expert - just a user for limited applications; this presentation may contain factual errors or other misinformation
- Compatibility
  - FUN3D requires both DiRTlib and SUGGAR++ codes from Celeritas http://www.celeritassimtech.com
  - Grid formats: VGRID, AFLR3, FieldView (FV)
- Status
  - Overset simulations done with FUN3D and SUGGAR++ on a frequent basis, primarily for rotorcraft applications.

# SUGGAR++ Documentation

- User's Guide: doc/UsersGuide/UsersGuide.pdf
  - Documents list of input elements (the rules, not much of the "why")
  - Documents command-line options for SUGGAR++
- Programmer's Guide: doc/ProgrammersGuide/ProgrammersGuide.pdf
  - Compilation
  - How to integrate libSUGGAR++ into a flow solver
- Ralph Noack and Dave Boger provided training at the April 2010 FUN3D Training Session
  - Much of the material here is a distillation of the April 2010 slides - but they had a full day to cover this

# Nomenclature (1/4)

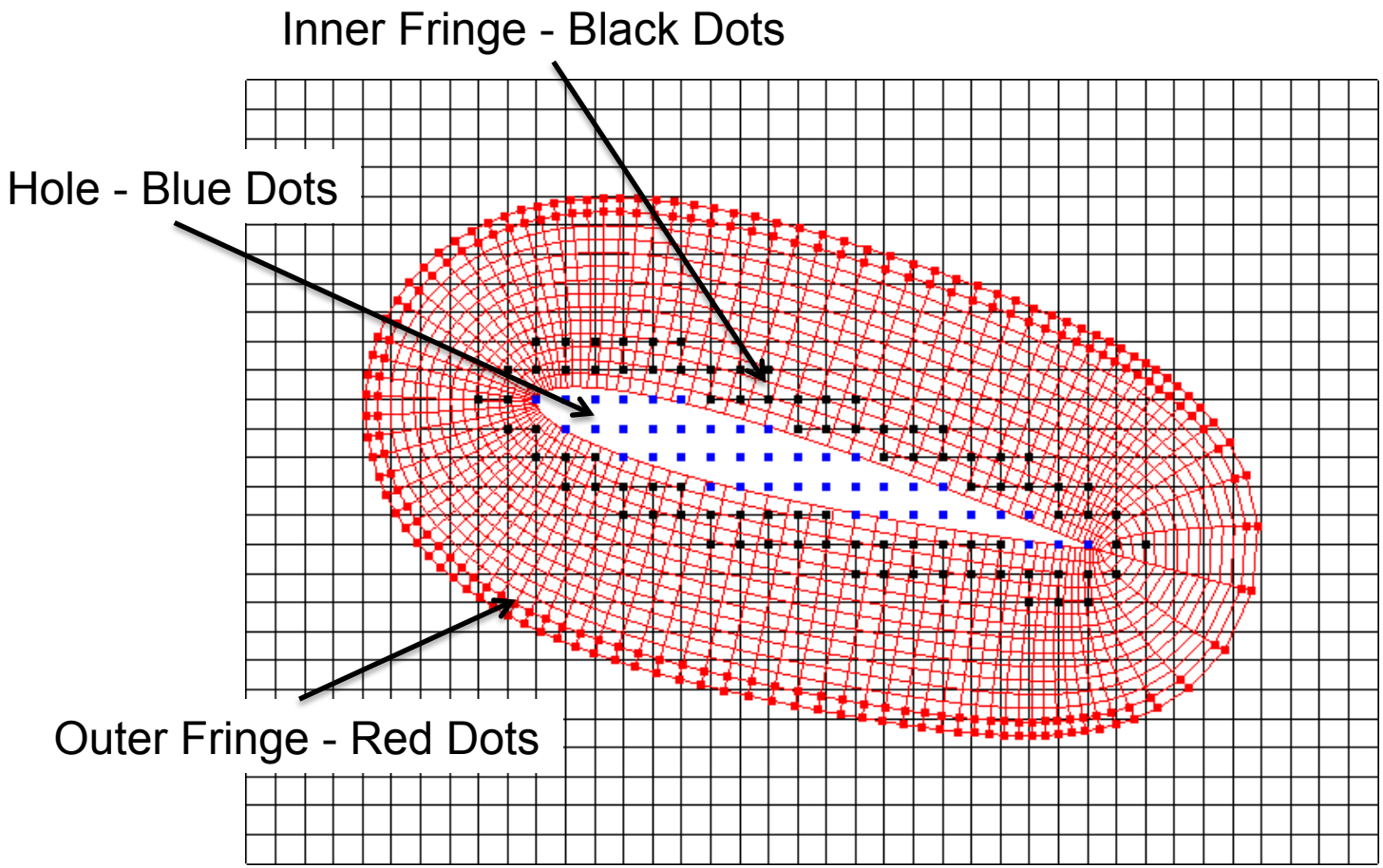
- SUGGAR++: ***S***tructured, ***U***nstructured, ***G***eneralized overset ***G***rid ***A***ssemble***R***
  - PEGASUS-like capability for general grids
  - Stand-alone version plus library version to call within a flow solver
- DiRTlib: ***D***onor ***i***nterpolation/***R***eceptor ***T***ransaction library - used by flow solver to handle data provided by SUGGAR++; no user input (just compile and link to flow solver)
- Component Grid
  - "Independently" generated grid for one piece of the configuration
  - Up to you to create these
- Composite Grid
  - An assembly of component grids
  - Created by SUGGAR++ based on your input

# Nomenclature (2/4)

- Overset grid point classification
  - In or Active: flow solver updates these points by solving the governing equations at these locations
  - Out or Hole: flow solver need not update these points as they have been removed from the domain
    - In practice, especially for moving grids, the flow solver fills in data at these points by averaging neighboring points - done so that as points move from “out” to “in”, they have “reasonable” data
  - Fringe: these points are updated by interpolation from “in” points; fringe points border a hole (inner fringe) or lie along an outer boundary (outer fringe)
  - Donor: the “in” points that supply data to fringe points
  - Orphan: fringe points for which too few or no donor points can be found; undesirable; solver fills in data at these points by averaging solution at neighboring points

# Nomenclature (3/4)

- Flow solver point classification - example



# Nomenclature (4/4)

- DCI file
  - ***D***omain ***C***onnectivity ***I***nformation file
  - Created by SUGGAR++; contains information about point classifications (hole, fringe, etc) for points in composite mesh, plus interpolation stencil data
  - Calls to DiRTlib within FUN3D read the DCI file and utilize the data within to update the solution at fringe points via interpolation from donor points
  - If grid is static, only need one DCI file
  - If grid is dynamic, must either have pre-computed DCI files available for the grid positions at each time step, or utilize libsuggar calls within FUN3D to compute DCI data “on the fly” (separate presentation)

# XML Basics (1/2)

- SUGGAR++ input is based on XML
  - eXtensible Markup Language (HTML-like, but not web centric)
  - XML element is enclosed in a tag "< >" , with corresponding end tag

    **<body>** … **</body>** (start and end can also span multiple lines)
  - Elements can have attributes/data: **<body name="wing">**
  - Elements can have an implicit end tag; elements can be empty - no attributes: **<dynamic/>**
  - XML elements can be embedded in other XML elements to create parent-child relationships ( wing and store are children of aircraft)

```
<body name="aircraft">
  <body name="wing">
  </body>
  <body name="store">
  </body>
</body>
```

# XML Basics (2/2)

- Element attributes are name/value pairs associated with an element
  - Always in the start tag, value must be in quotes (single or double)

```
<body name='blade_1'> … </body>
<translate axis="x" value="1.0e0"/>
```

- Comments start with `<!--` and end with `-->` and cannot be within a tag

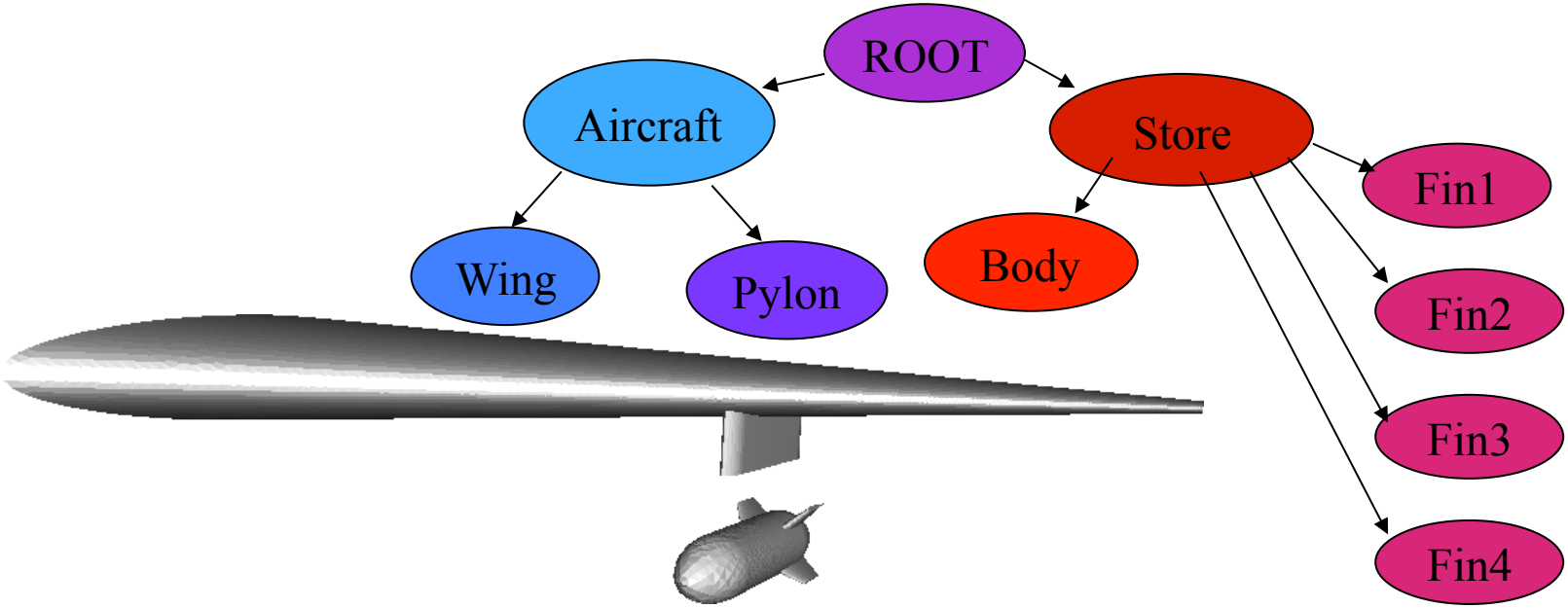
```
<!-- <body name="aircraft"/> -->
```
Correct

```
<body <!-- name="aircraft" --> />
```
Incorrect

- XML syntax must be precise: xmllint is on most(?) systems and can be used to check XML syntax before using SUGGAR++
  - Usage: xmllint myfile.xml
  - If syntax is OK, will simply echo XML file to screen; otherwise it reports the error
- Indentation helps keep XML input readable; xmllint can help here too
  - Usage: xmllint -format my_messy_file.xml > my_neat_file.xml

# Hole-Cutting: Hierarchy

- Parent-Child hierarchy established in XML file minimizes additional input to control hole cutting
- Basic rule: siblings cut each other
  - Geometry in one body (including all children) cut all grids in a sibling body (including all children); Aircraft cuts hole in Store and vice versa



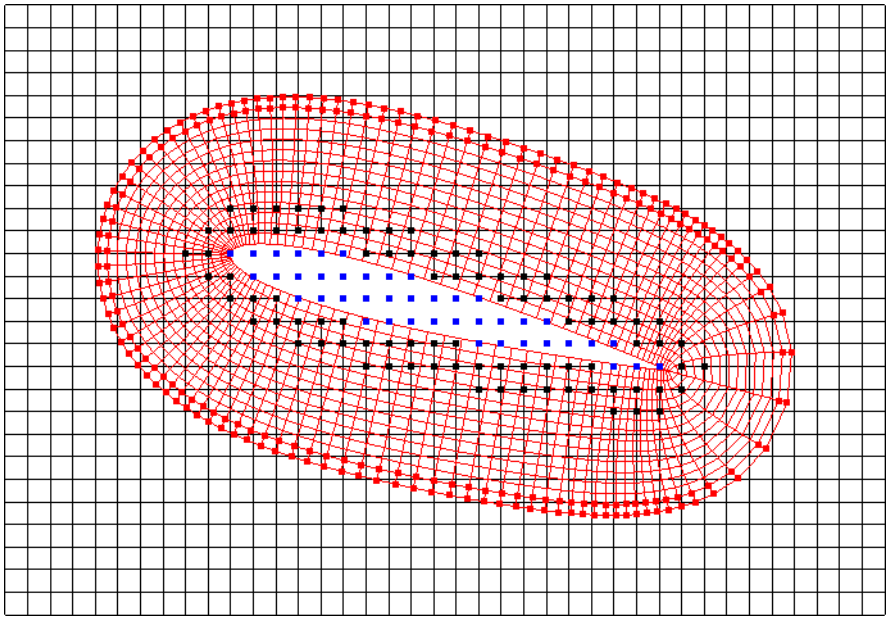
# Hole-Cutting: SUGGAR vs SUGGAR++

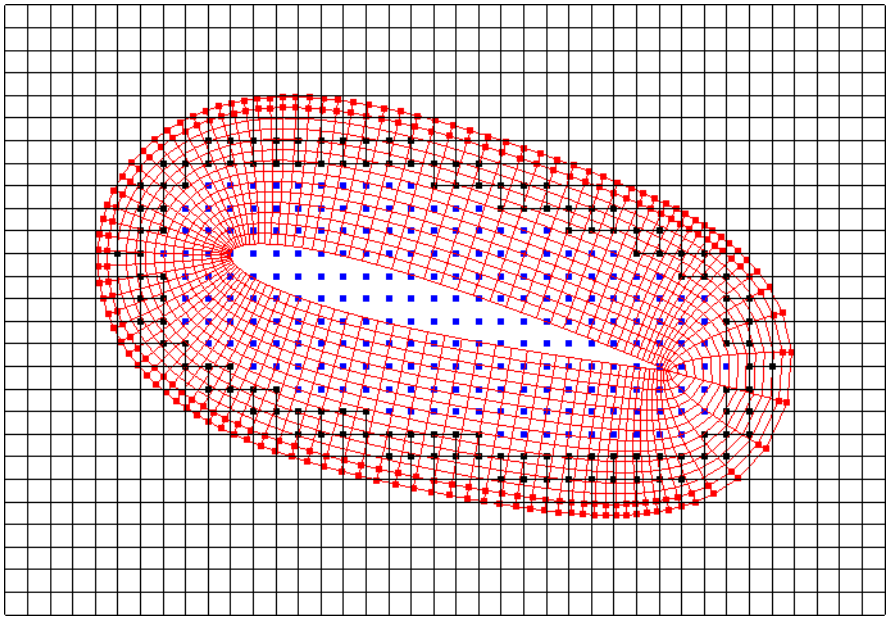
- Older SUGGAR code relied (primarily) on Octree hole cutting - uses Cartesian representation of geometry; hole cutting based on a query approach: Is this point inside (the Cartesian representation of) the body?

- In my experience, the Octree hole cutting approach often needs a lot of tweaking beyond the default behavior
- Newer SUGGAR++ code relies (primarily) on a direct hole cutting approach: Find intersections of geometry and grid; requires watertight geometry
- In my experience very little tweaking has been required with SUGGAR++
- SUGGAR++ supports the older Octree approach too; other hole-cutting options are available but are beyond the scope here
- There are pros and cons to any approach…

# Hole Cutting: Overlap Minimization

- Solution quality usually improved by reducing amount of overlap
- Goal is to have donors and receptors of similar size
- Enabled by element **`<minimize_overlap>`**
- For moving grids: **`<minimize_overlap keep_inner_fringe="yes"/>`**
  - Instead of blanking out points removed in overlap minimization, keeps them as fringes so they are *interpolated* rather than *averaged* - presumably better for when these points later emerge from the hole





# Building Up A SUGGAR++ Input File (1/9)

- **<global>** element serves as the root (parent) element for *every* SUGGAR++ input file: first line in file is **<global>** and last line is **</global>**
- Child elements of **<global>** specify various global parameters, and the body hierarchy
- So on a high level an input file for an aircraft composed of a wing and a store would look something like:

```
<global>
  <!-- global parameters here -->
  <body name="aircraft">
    <body name="wing">
      <!-- wing parameters here -->
    </body>
    <body name="store">
      <!-- store parameters here -->
    </body>
  </body>
</global>
```

# Building Up A SUGGAR++ Input File (2/9)

- Common child elements of **<global>** (see documentation for more info)
  - **<donor_quality value="0.9"/>** (lower stencil quality standard to reduce number of orphans)
  - **<minimize_overlap keep_inner_fringe="yes"/>**
  - **<output>** (governs output of composite mesh and DCI file)
  - Principal children of **<output>**
    - **<composite_grid filename="file" style="style"/>**
    - **<domain_connectivity filename="file" style="style"/>**
  - **<composite_grid/>** style attributes compatible with FUN3D:
    - **"unsorted_vgrid_set", "fvuns", "aflr3", "ugrid"**
    - Note: **"vgrid_set"** is ***not*** valid output option for node-centered grids (FUN3D is node centered)

# Building Up A SUGGAR++ Input File (3/9)

- **<body>** element can be child of **<global>** or another **<body>**
  - Required attribute **is name="body_name"**
- Common child elements of **<body>** (see documentation for more info)
  - **<volume_grid name="wing" filename="Grids/wing" style="vgrid_set"/>** (associates a volume grid with a body)
  - **<dynamic>** (declares a body as moving; also determines how the element **<transform>** is handled)
  - **<transform>** (to manipulate body: scale, rotate, translate, etc.)
    - If **<transform>** is child of **<body>,** transform is "static" - input grid coordinates are actually altered by the transform specified
      - Use to move component grids into place for composite mesh
    - If **<transform>** is child of **<dynamic>,** transform is "dynamic" - input grid coordinates are *not* altered by the transform; the transform is only used internally to compute overset data
      - Use to specify grid motion from static position

# Building Up A SUGGAR++ Input File (4/9)

- Subtle (important) effect of <dynamic> tag:
  - Flags the associated grid as dynamic in the DCI file
  - FUN3D will need this info up front for dynamic grid simulations
- When setting up input file to generate composite mesh / initial DCI file:
  - Add a "self-terminating" **<dynamic/>** child to any body that will subsequently be in motion:

```
<body name="store">
  <dynamic/>
  <transform>
    <translate axis="x" value="  7.6520E-01"/>
  </transform>
</body>
```

  - Because the **<dynamic/>** element self terminates, **<transform>** is not a child of it, and the usual static transform is applied to position component "store" in the composite mesh

# Building Up A SUGGAR++ Input File (5/9)

- Children of **<transform>**:
  - **<translate>**
  - **<rotate>** (used to rotate about x, y, or z)
  - **<rotate_about_v>** (used to rotate about arbitrary vector axis )
  - **<scale>**

```
<body name="store">
  <dynamic/>
  <transform>
    <translate axis="x" value="  7.6520E-01"/>
  </transform>
</body>
```

  - The order of transforms is important; transforms applied in order specified in the input file
- Refer to documentation for complete rules about which elements are allowed as children, which are allowed as parent, allowable attributes, etc.

# Building Up A SUGGAR++ Input File (6/9)

- More complex example of **<transform>** from rotorcraft application

```
<body name="rotor1_blade2">
  <dynamic/>
    <transform>
      <translate axis="x" value="  7.6520E-01"/>
      <translate axis="y" value="  0.0000E+00"/>
      <translate axis="z" value="  7.9600E-01"/>
      <rotate_about_v axis_vector="0.0E+00,  1.0E+00, 0.0E+00" value="0.0E+00"
   originx="7.652E-01" originy="0.0E+00" originz ="7.96E-01"/>
      <rotate_about_v axis_vector="1.0E+00,  0.0E+00, 0.0E+00" value="0.0E+00"
   originx="7.652E-01" originy="0.0E+00" originz ="7.96E-01"/>
      <rotate_about_v axis_vector="0.0E+00,  0.0E+00, 1.0E+00" value="0.0E+00"
   originx="7.652E-01" originy="0.0E+00" originz ="7.96E-01"/>
      <rotate_about_v axis_vector="0.0E+00, -1.0E+00, 0.0E+00" value="0.0E+00"
   originx="7.652E-01" originy="0.0E+00" originz ="7.96E-01"/>
      <rotate_about_v axis_vector="0.0E+00,  0.0E+00, 1.0E+00" value="9.0E+01"
   originx="7.652E-01" originy="0.0E+00" originz ="7.96E-01"/>
    </transform>
  <volume_grid name="rotor_w_cutout_1_correct_pitch" style="vgrid_set"
   filename="rotor_w_cutout_1_correct_pitch" format="unformatted"
   precision="double">
  </volume_grid>
</body>
```

# Building Up A SUGGAR++ Input File (7/9)

- Boundary conditions
  - SUGGAR++ needs to know some boundary condition information, e.g. which are the solid (body) boundaries, which outer boundaries need to be interpolated from other grids
    - SUGGAR++ input has provision for specifying the required SUGGAR ++ BC's via XML elements
    - An alternative is to provide SUGGAR++ with a separate file with the BC data
  - I strongly recommend the first approach - set the BC's via XML, since the SUGGAR++ BC files are not *required*, and if you move things around and forget the BC files, SUGGAR++ will run with defaults, likely not what you want
    - One exception: if VGRID grids are used exclusively, SUGGAR++ will use BC's from VGRID's mapbc file, which FUN3D will also require, so you will always have consistent BC's.

# Building Up A SUGGAR++ Input File (8/9)

- SUGGAR++ needs BC info for each *component* grid
- **<boundary_condition>** is a child of **<boundary_surface>** which is a child of **<volume_grid>**
- Examples (syntax for each grid type a little different)
  - AFLR grid

```
<boundary_surface find="yes" name="Surf=2”>
  <boundary_condition type=”overset”/>
</boundary_surface>
```

surface corresponds to 2nd patch in grid file

  - FV grid

```
<boundary_surface find="yes" name="airfoil_surface">
  <boundary_condition type="solid"/>
</boundary_surface>
```

must be *SAME* name that is set in grid file

  - VGRID grid (shown completeness – generally don't need)

```
<boundary_surface find="yes" name="Surf=3:bc=4”>
  <boundary_condition type="solid”/>
</boundary_surface>
```

need surface/patch no. *AND* bc type

# Building Up A SUGGAR++ Input File (9/9)

- Principal options for **`<boundary_condition type= >`**
  - “overlap”
  - “non-overlap”
  - “solid”
  - “non-solid”
  - “symmetry”
  - “farfield”
  - “freestream”
  - “periodic”
  - “axis”
- 2D Cases
  - Add as child of <global>

```
<symmetry_plane axis="Y" both_directions="yes"/>
<ignore_direction dir="Y"/>
```

# Running SUGGAR++: Static / T=0 (1/3)

- Ralph recommends creating a “Grids” subdirectory and an “Input” subdirectory for each case
  - I never do this however
  - By default SUGGAR will look to read Input/Input.xml, so if you don’t have this you simply have to explicitly give the input file name
- You will want to redirect stdout and stderr (stdout has LOTS of output); below, file name **`Input.xml_0`** is explicitly given
  - c-shell

```
(./suggar++ Input.xml_0 > suggar++.output) > & suggar++.error
```

  - bourne-shell

```
./suggar++ Input.xml_0 1> suggar++.output 2> suggar++.error
```

  - Simpler trick: **`./suggar++ -reopen Input.xml_0`**
    - stdout and stderr automatically go to **`out.stdout++ and out.stderr++`**

# Running SUGGAR++: Static / T=0 (2/3)

- Principal output: DCI and composite grid files specified in the XML file
- A concise summary of SUGGAR++ info is written to **`summary.log`**

```
start time: Wed Jul  7 18:49:17 2010
host: i16n1
last git commit:
command line: ./suggar++ Input.xml_0
number of processors: 1
number of threads: 1
total number of out: 9657
total number of fringes: 166124
total number of min fringes: 145265
total number of orphans: 199
number of orphans due to poor quality donors: 199
wall clock to perform assembly (seconds): 4.98748
memory used (MB): 1018.83
max interpolation deviation: 7.32747e-15
fringe donor quality: 0.904761
min fringe donor quality: 1
```

# Running SUGGAR++: Static / T=0 (3/3)

- SUGGAR++ *can* use multiple threads
  - Via command line **`-n_threads N`** (for N threads)
  - Via input element **`<threads n="N"/>`**
  - Never found this particularly worthwhile (YMMV)
- SUGGAR++ *can* be run in parallel
  - So far scaling achieved has been fairly poor - nowhere near linear, even for small (~8) processor counts
  - Requires a separate partitioning step, which is at odds with current FUN3D parallel-processing paradigm; “optimum” SUGGAR++ partitioning bears *no* resemblance to optimal flow solver partitioning
  - For these reasons, there has been minimal incentive to utilize the parallel capability for SUGGAR++ processing
  - Hopefully SUGGAR++ parallel scaling will improve in the future

# Running SUGGAR++ : Moving Grid (1/3)

- For FUN3D applications involving moving grids, SUGGAR++ must be run *at least* one time, to create the composite mesh and initial (T=0) DCI file
  - FUN3D can call SUGGAR++ routines to compute the DCI data for each time step *after* T=0, "on the fly"
    - Works for the most general case involving deforming bodies/grids or where motion is not known a priori (6DOF/aeroelastic)
    - Creates a serial bottleneck in FUN3D execution, but is the easiest option to use
    - More details in "Overset-Grid Simulations" Session
- For rigid grids with prescribed motion can run SUGGAR++ with a "motion file"
  - Can be done "embarrassingly parallel" – simultaneous runs with different motion files
  - Potentially can use SUGGAR++ in parallel mode as well

# Running SUGGAR++ : Moving Grid (2/3)

- To run SUGGAR++ with a motion file called (e.g.) “motion.xml”:
  - **`(./suggar++ Input.xml_0 -play_motion motion.xml > suggar++.output) > & suggar++.error`**
    - **`Input.xml_0`** is the same xml file used to create the composite grid and static / T=0 DCI file
- Motion file:
  - Each time step is contained in a complete <global> element
  - Typical motion file will have multiple time steps
  - Output specification of DCI file for the time step should be placed before and <body> specifications
  - Up to the user to make sure the specified motion is the same as that which will later be applied by FUN3D during execution
  - Should include one “motion” step with no motion if you want to generate the static / T=0 dci file in the same execution of SUGGAR++

# Running SUGGAR++ : Moving Grid (3/3)

- Simple example of motion file with 2 time steps: T= 0 and T=deltaT

```
<global>
  <output>
    <!--- This is to generate the T=0 dci file    note: no number after [project] name -→
    <domain_connectivity style="ascii_gen_drt_pairs" filename="./wingstore.dci"/>
  </output>
  <body name="wingstore">
    <body name="wing">
    </body>
    <body name="store">
      <dynamic>
      <transform>
        <translate axis="z" value="   0.000000000000E-00"/>
      </transform>
      </dynamic>
    </body>
  </body>
</global>

<global>
  <output>
    <!--- This is to generate the T=delta_t dci file (timestep1) -→
    <domain_connectivity style="ascii_gen_drt_pairs" filename="./wingstore1.dci"/>
  </output>
  <body name="wingstore">
    <body name="wing">
    </body>
    <body name="store">
      <dynamic>
      <transform>
        <translate axis="z" value="  -2.120800000000E-01"/>
      </transform>
      </dynamic>
    </body>
  </body>
</global>
```

# GVIZ

- Ralph has a “home-brew” interactive visualizer for looking at the overset grid assembly, called GVIZ
  - Allows visualization of the meshes, hole points, fringe points, etc.
  - Can be useful for debugging
  - I don’t have enough skill with GVIZ to even begin to explain how to use it

# Troubleshooting

- Lots of orphans could mean:
  - Improper BC's
  - Non watertight geometry (default direct hole cutting requires watertight surfaces) – likely if virtually all points end up as hole points

# List of Key Input/Output Files

- Input
  - **`Input/Input.xml`** (default; any name OK if explicitly specified)
  - Motion file (any name OK, used with **`-play_motion`**)
  - Component grids (name and grid format vary; for FUN3D: VGRID, AFLR3, Fieldview formats)
- Output
  - Composite grid; name and grid format vary
  - **`filename.dci`** (filename set in XML file)
  - **`summary.log`** concise summary by point type (out, fringe, orphan…)
  - **`SUGGAR++_motion.log`** (if **`-play_motion`** ) echo of motion file