A Tutorial for Using the CGT Script Library to Generate and Assemble Overset Meshes

Shishir A. Pandya, William M. Chan, Stuart E. Rogers
Computational Aerosciences Branch, NAS Division, NASA Ames Research Center
M/S 258-2, Moffett Field, CA 94035

April, 2018

The purpose of this document is to introduce a new user to the procedures for overset CFD analysis by building scripts based on the CGT Script Library. Parameterized inputs are built into the steps of the process which include creation and manipulation of geometry, and surface and volume meshing. In preparation for performing computations in the flow solver, further steps are constructed for specification of inputs for domain connectivity, flow solver boundary conditions, and components for computation of aerodynamic forces/moments. The JCLV rocket (shown in Fig. 1) will be used as an example geometry for this demonstration.

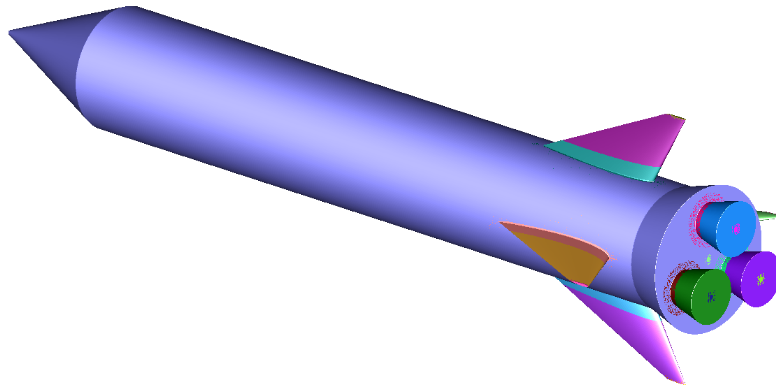


Figure 1. James' Crew/Cargo Launch Vehicle.

1. Introduction

PREPARATION of an overset mesh appropriate for the computation of a flow field is performed in several steps:

- Geometry import or creation
- Surface mesh generation
- Near-body volume mesh generation (meshes associated with surface geometry)
- Off-body volume mesh generation (meshes not associated with surface geometry)
- Domain connectivity input preparation
- Flow solver input preparation (boundary conditions and numerical scheme)
- Component specification for force/moment computation

The Chimera Grid Tools (CGT) package is a collection of tools that provides capabilities for each of these steps. CGT also provides post-processing capability for overset meshes which is not addressed in this tutorial. Three options are available: use of its modules on a command-line in a unix shell, via a graphical user-interface (GUI), and with scripts. At the inception of CGT, the tools were available to the user on a command-line with each tool asking questions interactively, or taking input files or both. This mode of operation requires the user to know which tool is appropriate for a given task and significant manual interaction is needed for each step in the process. Furthermore, the user is responsible for logging all steps in case any part of the process needs to be repeated to correct an error, modify a mesh, or make a configuration change to the geometry. If a mesh parameter in an earlier step is changed, the user must tediously follow the rest of the process manually on the command-line to update the rest of the mesh. At the end of each command-line step, the user has to load the resulting grids in a visualization software such as PLOT3D to check for errors. To improve upon this, an interactive approach is available through a GUI called OVERGRID which provides the means to call various grid generation utilities and view the results within the same interface.

While creating meshes visually and interactively is very attractive, there is limited capability available in OVERGRID to record and parameterize the procedure, making it difficult to modify or repeat the steps. To make the process more efficient and less error-prone, the current CGT best practice is to write a mesh generation script in the Tcl language, making use of the vast number of Tcl grid utility procedures in the CGT Script Library, and to use OVERGRID to view the results of each step of the scripting process. Scripting has several advantages. Geometry and mesh inputs such as maximum stretching ratio and grid spacings are parameterizable. The entire process is recorded for the user, and is thus repeatable. Mesh modifications are simply a minor edit of the script provided there is no topological change. However, scripting also has disadvantages. Each new geometry requires significant development time to build the initial script, and topological geometric changes also demand non-trivial extra development time.

Typically, geometry is given as analytic definitions such as native CAD, IGES or STEP files; or discrete definitions such as unstructured triangulations, or structured surface patches. Any analytic definition can be converted to a discrete definition with little effort using typical CAD or grid generation software. So a user can start with a discrete reference surface (either a triangulation in Cart3D .tri format, or a set of structured patches in PLOT3D format) and use fast algebraic or PDE-based meshing software along with mesh manipulation tools to generate surface and volume meshes. An example of converting CAD files to reference curves and surfaces is provided in the OVERGRID tutorial. In cases where the geometry is not provided, but drawings are made available, simple geometry can be created within CGT. Once overset volume meshes are available, the scripts are also able to address other aspects of pre-processing such domain connectivity input preparation, flow solver boundary conditions specification, and prescription of grid subsets for components definition in forces and moments computation. The goal is to create all files needed to start a CFD flow solver run with the script. Multiple functions are available within the CGT Script Library for setting up inputs for the OVERFLOW flow solver.

1.1. Configuration Management Paradigms

If one were to generate a single mesh using scripts, a script can be written in the Tcl language per CGT best practices. Even for this simple task, the script developer must decide if the script will reside in the same place as the mesh. Another option is that the script resides in a main directory and meshes are generated in a sub-directory. Where should the reference curves and surfaces be stored? Should the grid numbering be inherent in the script following the flow of the script or should there be a reference file that guides the numbering? Where should boundary conditions and other inputs be stored? Now imagine that this must be done for thousands of surface and volume meshes organized in hundreds of components. The management of scripts, grids, and related information becomes an important aspect of mesh generation.

During the development of the script library, two configuration management paradigms have evolved for handling complex configurations. Philosophically, they can be thought of as grid-centric (referred to as the BuildScripts approach), or component-centric, depending on whether the basic conceptual unit for work flow and file storage is a single grid or a single geometric component. We note here that these are just two approaches that have evolved during our experience in handling complex configurations in the last 10 - 20 years. An experienced user can design other configuration management approaches that are more optimized for different applications.

1.1.1. BuildScripts: Grid-Centric

The Build scripts approach has been enabling complex geometry real-world overset CFD analysis for the past two decades. A detailed description of the approach and its capabilities are summarized here. For a more detailed discussion see the manual in the CGT doc directory in the file *doc/scripts.html*, and in AIAA Paper 2000-4216.

Recognizing that most of the information required by these codes is contained in the boundary-condition (BC) input to the flow solver, it was decided to require that the user supply this BC information for each grid at the beginning of the process when the surface grids are built. User-written scripts give the user full control over the creation of the surface grids and the BC files. The Build scripts require that the user provide one or more Makefile(s) in one or more subdirectories that provide instructions for creating the surface and boundary-condition files for each individual grid. The user also supplies two additional input files: *config.tcl*, containing a list of the root-names of each grid to be included in the configuration; and *input.tcl* which defines default values for various input variables, as well as grid-specific values when the user wishes to override the default values. Given the surface grids, their boundary conditions, and these two input files, the script system contains tools which can build all of the volume grids, performs elliptic smoothing where needed on these grids, and builds input files for the MIXSUR, PLOT3D, PEGASUS5, and OVERFLOW programs.

The script system also defines a series of file suffixes that describe the specific contents of the files used in the gridding process. This provides a built-in dependence path for each file in the process. The Build scripts utilize this and are capable of updating the entire grid system automatically when a single input is changed, and does so by performing updates of only the files which are dependent on the modified input, minimizing execution time. In addition, the scripts have the ability to parse "family", "group", and "xmlcomp" information from the boundary condition files if the user desires. A family is a grouping of individual surfaces which comprise an entire component and is used in the force and moment integration (MIXSUR) process. The group designation enables hole-cutting operations, and the xmlcomp designation can be used to define relative body motion in OVERFLOW.

The basic rules to follow using the Build scripts include the following:

- Each surface grid file is named *gridName.srf*
- Each boundary-condition file is named *gridName.ovfi*
- Each volume grid file is named *gridName.vol*
- Each grid is contained in a separate file.
- A Makefile in each subdirectory *subdir* specifies the rules and dependencies for building each surface grid.
- The name of each grid in the configuration is listed in *config.tcl*.
- Input variables and defaults are given in inputs.tcl and *subdir/localinputs.tcl*.
- Build scripts are provided to automatically create (and recreate) the volume grids, and inputs for the entire CFD analysis.

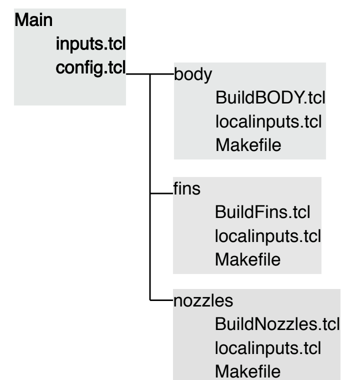


Figure 2. Directory structure for the JCLV using the BuildScripts approach.

This approach also has a very powerful configuration control capability. By defining geometrical options within the *config.tcl* file, an entire series of different versions/configurations of the vehicle/article can be constructed from within one framework.

Once the framework has been set up, the Build scripts enable an end-to-end flow through the entire process of grid and input-file generation. The *BuildSurf* command creates all surface grids and boundary-condition files. The *BuildPlot* command can be used to verify the surface grids, boundary conditions, and family/component definitions. By performing this verification early in the process, errors can be identified and corrected before they propagate further into the process. This can drastically reduce the total cycle time. The *BuildVol* generates the volume grids; it can do so with no additional user input; by reading the boundary conditions it knows enough how to execute the HYPGEN program suitable for most situations. The user can override the default behavior or even provide full control over the volume grid with entries in

the *inputs.tcl* file and in *subdir/localinputs.tcl* files. The *BuildPeg5i*, *BuildMixsuri*, and *BuildOveri* scripts produce input files for Pegasus5, Mixsur, and Overflow, respectively. Again, additional details for this approach are contained in *doc/scripts.html*.

The directory tree for the JCLV example is shown in Fig. 2. The main directory for this approach is *$CGT/trunk/tutorials/jclv/BuildScripts*. All scripts are placed in subdirectories of the user's choosing. The grid directory association is specified by the user in the *config.tcl* file. The user must setup the user scripts in each subdirectory including a *Makefile* which keeps track of the dependency tree and therefore controls if a mesh needs to be regenerated. Example meshing scripts and Makefiles are provided in each subdirectory of this example. The *Build* commands must be executed in the main directory to generate or update meshes. All surface and volume meshes along with their corresponding *.ovfi* files are generated in their respective subdirectories.

1.1.2. Component-Centric

The component-centric approach was targeted for moving-body problems as well as for components that are repeated many times (e.g., blades in a fan or compressor) at its inception. Thus, the user needs to think in terms of components of the geometry (e.g., for an airplane, typical components are fuselage, wing, vertical stabilizer, etc.) while each component usually consists of multiple surface/volume meshes. The rules for this approach are:

- Component names must be listed ahead of time by the user in the *inputs.tcl* file.
- For each component, all grids associated with the component are contained in a single file where the name of the component is the root name.
- Each surface grid file has root name *componentName* while the user can use any extension (common ones include *.sur* and *.srf*.
- Each volume grid file is named *componentName.vol*
- All X-ray files have the extension *.xry*

Here, the master script is in the hands of the user giving the user complete control; with the caveat that the user must write the master script. In practice, users take an example master script and customize it for their needs making the process simpler. However, the user must develop specific rules for a coherent scripting process. We provide some best practices:

- Input variables and defaults should be placed in the *inputs.tcl* file.
- Recommended file name extension: *.sur* for surface grids, *.cut* for X-ray cutters.
- Each component's grids are contained in a separate file.
- The user should provide dependency handling and file clean up instructions in the master script for efficient changes.

This method results in a longer master script and supports PEGASUS5, OVERFLOW/DCF, and C3P for domain connectivity.

The directory structure for this approach is controlled by the user in the master script. In this example, we use a single main directory that holds all the user scripts (see Fig. 3). As long as the location of this main directory is set in each script (see the Par(srcdir) variable in each script), the grids can be generated in any working directory of the user's choosing. The user must execute all scripts with the full path specified. To execute individual scripts, the user can issue "tclsh fullpath/scriptname". Note that putting all user scripts in the main directory is not a requirement. Since the user is writing the master script (*mkjclv*), the directory structure is totally flexible and can be arranged as necessary or desired. In this approach, the surface and volume meshes are typically generated in the working directory of the user's choice. A specific directory can be enumerated in the master script if desired, but in the JCLV example the current directory is used as the working directory. There is no Makefile based dependency system provided in this approach, so the user must specify and manage the dependencies in the master script. An example of this is provided through the *chkDep* procedure in the *mkjclv* master script.

To understand the nuances of each methodology, it is recommended that the user steps through the BuildScripts and componentCentric directories separately. In the rest of this document, we address both methodologies as necessary.

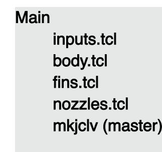


Figure 3. Directory structure for the JCLV using the component-centric approach.

1.1.3. Commonalities and Synergies

It is important to note that while the framework is different for the two approaches, the surface meshing portion of the scripting process (the most time consuming part) is nearly identical in both methods. Often, the BuildScript method uses component names to organize the mesh files in directories, and similarly, the component-centric method can be setup in exactly the same directory framework with use of Makefiles if the user so chooses.

2. Global Variables

Four global arrays (Par, Ovr, Fomo, and BC) are used to demonstrate the capabilities of the scripting process. These arrays are necessary for the proper use and execution of some of the script library functions. Par is used to define all parameters needed by the user as well as some required by scripts in the script library and must be available in almost every routine. Ovr is only used to set flow solver or connectivity inputs, while Fomo is used to set force/moment computation inputs. BC is special and used by several script libray scripts to keep track of boundary condition inputs. If it is desired to use that capability, the user will need the BC variable as a global in the master script.

An additional global variable is the env array. This is a Tcl array that gives the script access to the shell environment variables. This variable must be declared global in the master script.

3. Locations of CGT Executables and Scriptlib

The user must specify the locations of the CGT binaries and script library. After compiling and installing Chimera Grid Tools, set the environment variables *SCRIPTLIB* and *CGTBINDIR* to the location of the *scriptlib* and *bin_dp* directories, respectively. For example, if the Chimera Grid Tools package has been installed in *$HOME/chimera2.2* under C-shell (csh), use these commands:

```
setenv SCRIPTLIB $HOME/chimera2.2/scriptlib
setenv CGTBINDIR $HOME/chimera2.2/bin_dp
```

The *CGTBINDIR* variable sets the location of the CGT executables. This allows the scripts to call command-line codes such as grided (grid editor), or hypgen (hyperbolic volume meshing tool). All user-written grid generation Tcl scripts will need to include the following near the beginning of the script in order to use the Tcl procedures in the Chimera Grid Tools Script Library.

```
lappend auto_path $env(SCRIPTLIB)
InitExecs
```

3.1. Component-Centric: Other Directories

If one is using the component-centric approach, one can use a file called *CheckAndSetDir.tcl* to interpret all directory paths. In this file, the first of these variables, SCRIPTLIB, is used to point to the location of the CGT Script Library. The scripts in the CGT Script Library and their functions are detailed in *doc/scriptlib.html*.

In addition to these, a user may want to define some additional directories. If the geometry was received as a set of surface patches or triangulations, these surfaces and corresponding curves can be placed in a definitions directory called *geomdir*. The locations of connectivity code (PEGASUS5, OVERFLOW, C3P) executable and mpich can also be set if the user wishes to script the mesh connectivity or the flow solver processes. See section 3.3 for details.

Since all user scripts will need to know the information contained in this *CheckAndSetDir.tcl* file, each script needs to source this file in order to assure that the directory information is available to them. Finally, the working directory in the component-centric approach must be specified. The working directory

(designated Par(curdir) in this example) is treated differently. For the component-centric approach, the goal is to be able to generate a mesh in any directory regardless of where the scripts are. Therefore, we define a variable called Par(srcdir) in every component script, and in the master script that points to where the scripts are. In addition to this, a Par(curdir) variable is also defined and set to "pwd" or the current working directory. All working files are placed in Par(curdir). As a result, the Par(srcdir) directory which contains the scripts is never modified when generating meshes and clean up consists simply of removing all contents of the working directory.

3.2. Setting up Multi-Threading

For cases where one mesh does not depend on other meshes, it is more efficient to generate those meshes in parallel using multiple CPUs. While this is possible for surface meshing, the user must decide which meshes depend on which others and make intelligent decisions about what can be done in parallel. One place where parallel execution makes a lot of sense is volume mesh generation since the process of generating volume meshes from surface meshes is completely independent for each mesh. This capability is provided by a function called GenHypVolThread. Multithreading, however, is not activated in Tcl by default. Thus, if the user wishes to use this capability, one additional environment variable must be set.

- setenv THREADLIB $TCL_LIBRARY/libthread2.7.0.so

Here, the 2.7.0 version of the Tcl thread library is pointed to by the THREADLIB variable. The user scripts are made aware of this, by issuing the following command

- package require Thread

within the CheckAndSetDir.tcl file. If "package require" does not find the library, the following command can be used to force the library to be loaded.

- load $env(THREADLIB)

An additional variable called useThreads is set to zero or one to indicate whether the threads capability is available and can be used by user scripts to decide if parallel meshing is possible. We will demonstrate this in component meshing scripts for the component-centric approach. The user can simply not define (or unsetenv) the THREADLIB environment variable to create volume meshes serially.

3.3. Connectivity Codes, MPI and OVERFLOW

Both PEGASUS5 and OVERFLOW2 use MPI parallelization. To enable parallelization for these codes, the user may need to tell the scripts where the MPI executables are. This is done by setting the following environment variable.

- setenv MPI_ROOT $HOME/MPICH

Note that this needs to be adjusted to where the user has installed MPICH. A further option is to let the scripts use PEGASUS5, OVERFLOW, or C3P for hole cutting and connectivity. To do this, define one or all of the locations of these executables as follows:

- setenv OF2DIR $HOME/overflow2.2l/bin_dp
- setenv PEGBIN $HOME/pegasus
- setenv C3PBIN $HOME/c3p

4. The JCLV rocket

James' Crew Launch Vehicle (JCLV) was created as a summer intern exercise by a college freshman and is shown in Fig. 1. It is a model rocket with a 2″ diameter body that is 12″ long with a 2.4″ nose. At the rear of the rocket is a 0.7″ long skirt to cover the rocket motors with a maximum diameter of 2.4″. Three 0.6″ long nozzles sit at the bottom of the skirt with a root diameter of 0.6″ and a tip diameter of 0.9″. Finally, four fins just ahead of the skirt are added for stability. These fins have the NACA0011 airfoil shape at the root and the NACA0010 airfoil shape at the tip. The root chord is 2.4″, while the tip chord is 0.4″ with a leading edge sweep angle of 30°. The fin span is 1.2″.

4.1. Geometry Creation and Surface Meshing

With this definition of the rocket and some additional information about the exact placement of the fins and nozzles, the geometry is created using macros (Tcl procedures) stored in the CGT scriptlib directory. For this step, there is little difference between the BuildScript approach and the component-centric approach with the exception of how and where the grids are saved.

To create geometry, there are many functions available in the CGT Script Library. Though all of them are not used for the JCLV, below is a list of the geometry creation functions available:

- CreatePointxyz: create a point
- CreateLinejkl: create a line from 2 points in an input file
- CreateLinexyz: create a line from 2 points given their coordinates
- CreateCurve: create a curve from an analytic expression using parametric coordinates
- CreatePWLCurve: create a piece-wise linear curve from line segments
- CreateNACAFoil4: create a NACA 4 digit series airfoil
- CreateNACAFoil5: create a NACA 5 digit series airfoil
- CreateParsecFoil: create a PARSEC airfoil with 11 control parameters
- CreateAirfoilComponent: create a section of the wing with NACA or PARSEC airfoils
- CreateCylGrids: create a surface grid for a cylinder
- CreateFrustumGrids: create a surface grid for a frustum
- CreateSphereGrids: create a surface grid for a sphere

All functions in scriptlib are listed and their functions and usage are explained in the *scriptlib.html* document under the doc directory. These scriptlib functions are called from a surface meshing script written by the user.

For the BuildScript approach this user-written script can be placed in any subdirectory. All that is required is that a "make" command in that subdirectory causes the user script(s) to create a surface grid file with suffix *.srf*, and a boundary condition file with suffix *.ovfi*. Each individual grid needs their own single surface-grid file and boundary condition file. As discussed earlier, the individual grid names and their related subdirectories are specified by the user in the *config.tcl* file.

For the component-centric approach, the grid scripts can be named by the user. In the current tutorial, the component meshing scripts are simply named Component.tcl, where Component is either body, nozzles, or fins. The master script (named *mkjclv* in this example) can also be named by the user. As discussed earlier, the directory structure is up to the user, but for this example, all scripts and related files have been placed in a single directory. All meshes and intermediate files are created in the current directory. It is suggested that the user use an empty directory for this purpose. All surface grids for a single component are placed in a single file. For example, the main body is made of 3 grids (See section 4.1.1); all of which are placed in a file in the current directory called body.sur.

4.1.1. Axi-Symmetric Main Body

There are two options for generating an axisymmetric body with the x-axis as the axis of revolution. Option 1 is to generate curve segments in the x-z plane ($z = f(x)$) that define the body from nose to the skirt/base. The segments can define the nose cone, the main cylinder, and the skirt/base separately as shown in Fig. 4. Note that the curved section of the nose is a separate segment for a total of 4 segments. Once the segments defining the geometry have been created, the curve is redistributed to obtain required spacing, and revolved about the axisymmetric axis (x-axis in this case) to obtain the surface grid. A volume grid can then be marched using the hyperbolic volume mesh generation code HYPGEN. An alternative is to generate the "volume mesh" in 2D from the curve and then revolve the entire 2D slice to obtain the volume mesh.

Figure 4. JCLV main body curves.

Option 2 is to use scriptlib routine calls to make the surface meshes in one step. The CreateFrustumGrids call is used to make nose and skirt/base surfaces meshes. In addition to being able to specify diameters, length, and grid spacings, this scriptlib routine allows the user to specify if the ends of the frustum should be open, flat, or hemispherical. Thus, the nose is specified to have a hemispherical front and an open rear, while the skirt is made to have an open front and a flat rear just as we would have done by hand when making the curves in option 1. Note that if the mesh is made so that the j index goes from the nose to the cone break, and the k index goes around the body, then at $j = 1$, all points in the k direction collapse to a single point at the nose creating $k_{max} - 1$ triangles. This degeneracy is referred to as a singular axis condition. To avoid a singular axis, the area in the immediate vicinity of the singular point is replaced by a rectangular cap grid. A similar cap takes care of the singularity at the base ($j = j_{max}$). An added complication is that the caps must be checked visually for surface normals facing the same way as the body's surface normals. In the case of the skirt, the cap normal is facing inwards and this can be verified by stopping the script right after the CreateFrustumGrids call for making the skirt. Therefore, we must call the ReverseInd function in scriptlib to reverse the direction of either j or k so that the surface faces out. Another possible solution is to call the SwapInd function to swap j with k so that $\vec{r}_j \times \vec{r}_k$ points in the opposite direction, where $\vec{r}_j$ and $\vec{r}_k$ are the grid index directions in j and k, respectively. The CreateCylGrids call is used to make the main body surface grids. Since each of these calls takes a centroid location as input, those locations are precomputed from position information to assure that the three pieces line up correctly (See Fig. 5). Finally, a call to GenHypVolGrids is made to generate the volume with HYPGEN.

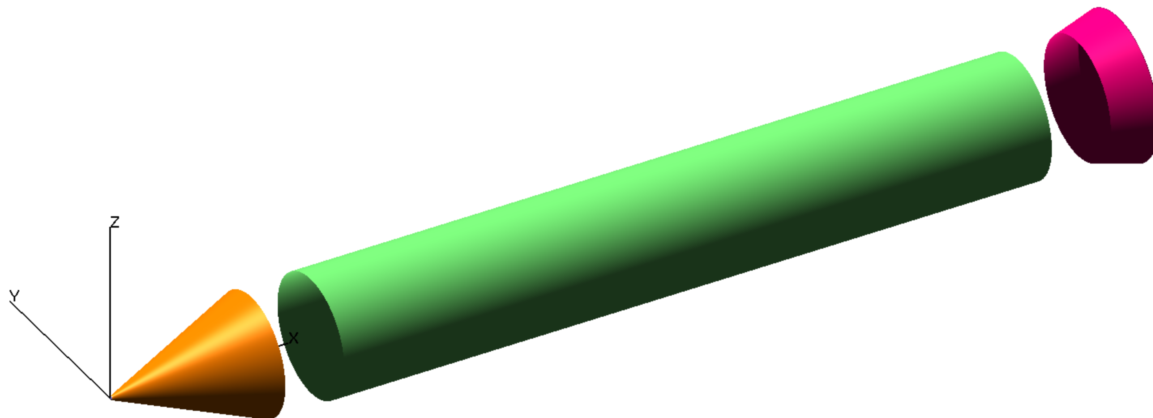


Figure 5. JCLV main body surfaces.

Option 2 is coded into a file called body/BuildBODY.tcl for the BuildScript approach, or body.tcl for the component-centric approach. Note that when using the BuildScript approach, a component's surface meshing script is located in that component's subdirectory; for the component-centric approach, it is located in the main directory along with inputs.tcl, and the master script. At this point, the reader should be able to open the BuildBODY.tcl or body.tcl file in an editor like vi or emacs and try to see if they can follow the code with the logic laid out in the previous paragraph. Because the nose cone, cylindrical main body, and skirt/base are created with the same number of points going around the geometry, those 3 pieces can be

concatenated to make one mesh. The meshes are in 3 different files, so we use the ConcatGrids2 macro to concatenate the nose cone to the cylinder, and then use the same call again to concatenate the newly created surface to the skirt. We then extract the hemispherical nose cap and the skirt/base cap grid, both generated to avoid a singular axis condition at an end of the geometry. For the BuildScript approach, these three grids are placed in 3 separate files in the body sub-directory. They are called: nose.srf, mainBody.srf, and skirt.srf. A Makefile is placed in the body directory to setup the dependencies. For the component-centric approach, all 3 grids associated with the body component are placed in a single file called body.sur, and the dependencies are handled in the master script (See section 8).

4.1.2. Nozzles

The nozzles are also simple frustums at the base of the vehicle. Thus, the first call in BuildNozzles.tcl is CreateFrustumGrids with the appropriate nozzle length and diameter parameters that are pre-specified in *inputs.tcl*. Temporarily, this first nozzle is placed along the centerline of the rocket at $L_{base} = L_{nose} + L_{mainBody} + L_{skirt}$, where L is a length. To specify the centroid of the nozzle, we simply add half of the nozzle length to L_{base}. This length computation is placed in *inputs.tcl* along with other geometry information. With appropriate grid spacings specified, CreateFrustumGrids returns the surface grid for the nozzle shown in Fig. 6(a). Remember that the nozzle geometry needs to be specified so that it is open at the front (where it sits on the skirt) and flat at the rear (closed off nozzle exit).

Since the nozzle sits at the rocket base, a collar grid is needed to connect the nozzle to the base. To do this, we extract the forward most ($j = 1$) curve with the assumption that it is the intersection curve between the base and the nozzle. The strategy is to use this curve to march onto the base surface using the GenHypSurGrids procedure. This routine needs the curve definition, a reference surface (*skirt.srf* and *mainBody.srf*), boundary conditions at each end of the curve and grid spacings in the marching direction. With this information, it predicts the points as the curve is marched along the surface of the base creating a mesh (shown in magenta in Fig. 6(b)) along the reference surface (grey surface in Fig. 6(b)). A small portion of the nozzle near the base can be extracted (seen in orange in Fig. 6(b)) and concatenated to this marched section to create a collar mesh. Swapping or reversing indices may be required to match index directions and surface normals prior to concatenating the two grids. As the intent is to have 3 nozzles positioned 120° apart, *sin* and *cos* functions can be used as arguments to the GedTranslate script to copy and move the nozzles where they belong.

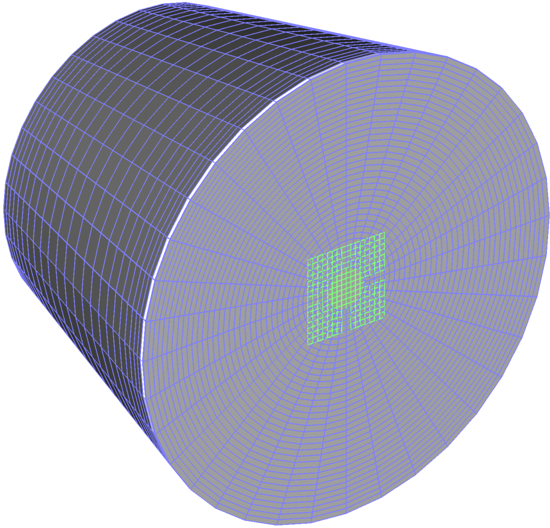
In the BuildScripts example, the script is located in *nozzles/BuildNozzles.tcl*. Also located in that directory is a *Makefile*. This shows an example of how the *make* utility is used to create the required files only if they do not exist or are out-of-date as defined by the dependices in the *Makefile*. For each nozzle, three different surface grids are created: a grid around the nozzle, a cap grid covering the end of the nozzle, and a collar grid where the nozzle is mounted on the rocket base. Note that the *Makefile* also defines actions to be taken when *make clean* is executed (remove intermediate files), and when *make clobber* is executed (remove all target files).

For the component-centric approach, the name of the script is controlled by the user. In the tutorial directories, this file is named *nozzles.tcl*. The resulting 9 surface grids are all placed in a single file called *nozzles.sur*. Note that because we follow the convention that all surface files have the *.sur* extension, we call the CreateNozzles procedure within the *nozzles.tcl* script with the root name of the component as the only argument. A variable called sfile is then defined to be "*$comp.sur*" so that $sfile can be used to denote the resulting surface filename.

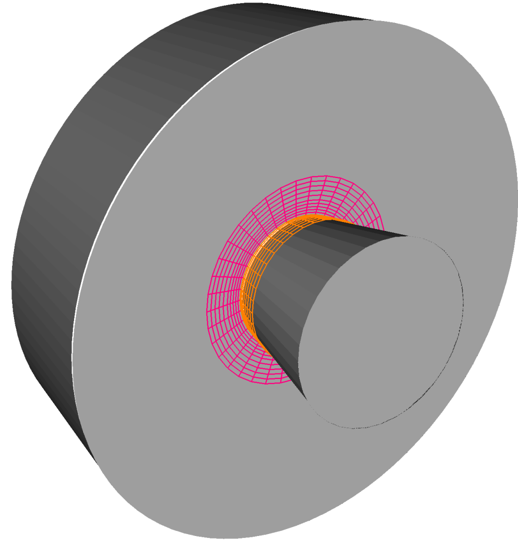
Note also that each of these scripts generate many intermediate files. If these files are not cleaned every time, the working directory will be a large mess. To prevent this, it is recommended that a "clean-up" section at the end of each surface mesh generation step removes all temporary files.

4.1.3. Fins

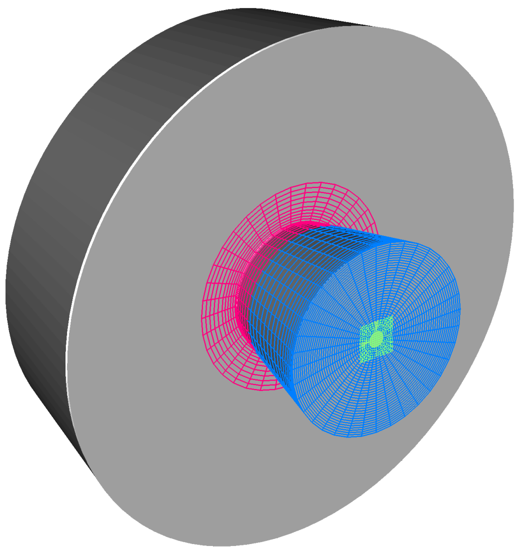
Similar to the other components, geometry information such as root and tip airfoils and their chords along with fin span, sweep, dihedral, and position are specified in the *inputs.tcl* file in the main directory. The *fins/BuildFins.tcl* or *fins.tcl* meshing script then begins with a call to the CreateAirfoilComponent script which is capable of creating NACA or PARSEC airfoils with a given set of parameters. For rocket fins, we choose symmetric airfoils and negative sweep. The negative sweep creates a wing section that is forward



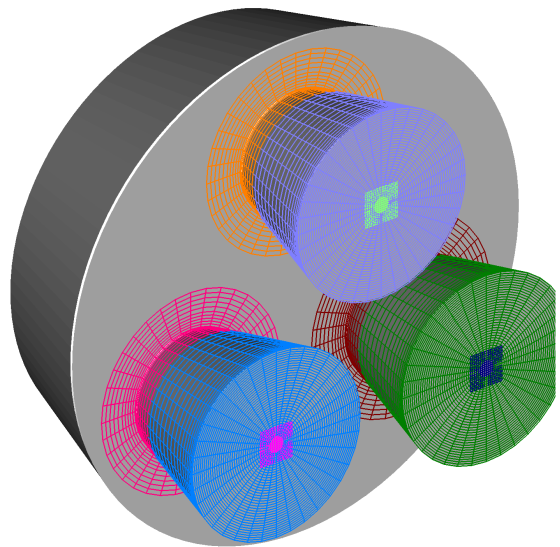
(a) Single nozzle; result of CreateFrustumGrids



(b) Single Nozzle; collar grid



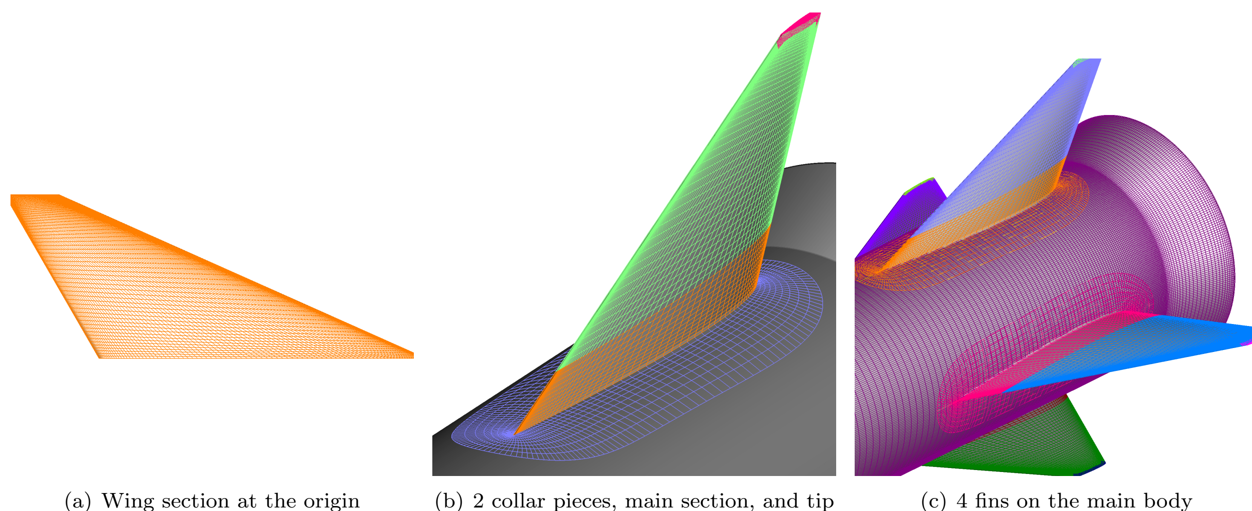
(c) Single nozzle with split collar, main nozzle, and rear cap



(d) Three nozzles defined by three grids each

Figure 6. JCLV nozzles attached to the skirt.

swept. This section can then be rotated 180° so that the sharp trailing edge is pointing into the wind with a high rearward sweep.



(a) Wing section at the origin (b) 2 collar pieces, main section, and tip (c) 4 fins on the main body

Figure 7. JCLV fins.

The symmetry of the airfoils means that the number of points specified for the bottom of the airfoil to CreateAirfoilComponent defines the leading edge. With this information, ExtractGrids can be used to separate the top of the wing section from the bottom. As j is the chord-wise index with $j = 1$ and $j = j_{max}$ at the trailing edge, we use SrapRedist to redistribute from $j = 1$ to j_{LE} and from there to j_{max} separately to achieve user-specified leading edge, and trailing edge grid spacings. The required span-wise spacing is provided by the CreateAirfoilComponent call, so the two redistributed surfaces can be concatenated to obtain a final surface grid. At this point, the wing section is open at both ends. To address this, a collar grid is necessary at the root and a cap grid needs to be created to close the tip. The CreateTruncatedWingCap macro has a complex set of inputs, but is much easier than creating a tip cap by hand.

Before creating a collar grid, we GedRotate the fin 180° around the z axis so that it is facing backwards, and GedTranslate it so that its position is on the outside of the rocket body near the end of the cylinder shaped main body, just ahead of the skirt. To create a collar grid at the root, we assume that the root curve of the wing section is the footprint on the body. While this assumption was a good one for the nozzle footprint because the base where the nozzles attach is a flat surface, the fin sits on a cylindrically curved section of the main body. Thus, we probably should have used the lsect code on the command-line or made a call to lsect using the system call in Tcl to obtain the intersection curve. For simplicity, we use the root curve. The root curve is obtained using the ExtractSubs call, followed by a call to GenHypSurGrids to march that curve onto the main body of the rocket. Here, the user must assure that the collar does not march over the surface discontinuity at the body/skirt junction. If this condition is violated, the variable marching distance option in the GenHypSurGrids call may need to be invoked. A call to GedSplitjkl is used to split out a short section of the wing near the root so that the newly marched piece can be concatenated to that piece to create a collar grid. Note that prior to concatenation, the user must check if the indices of the two pieces match and if the normals of those 2 pieces both face outward. ReverseInd and/or SwapInd will need to be called if the index directions do not match.

Once a complete fin is obtained as 3 grids (collar, wing section, tip), the 3 grids are temporarily placed in a single file so that we can use the DupRotate call to duplicate and rotate by 90° around the x-axis to obtain 4 fins.

4.1.4. User Exercise

To better understand geometry creation and surface meshing, make up a component you would like to add to the simple rocket. This can be a feedline, a systems tunnel, a camera cover, or a pair of booster rockets. Write a script from scratch and add that part to the existing JCLV. Create geometry using scriptlib calls and make a surface mesh that closely matches the spacings on the existing surfaces.

4.2. Making Cutters for X-rays

As the component surface meshes are made, a user intending to employ the DCF code in OVERFLOW that uses the X-rays approach to determine hole cuts must take the opportunity to make cutters for that component. It is recommended that all cutters be closed bodies in order to assure proper input for the X-rays routines. If a component is already closed, the surface mesh can be used as a cutter and nothing needs to be done. This is the case for the body. With the nose and rear meshes closing the body, the *body.sur* or *body.srf* file can be used as a cutter.

This, however, is not the case for the nozzles and fins. In the case of the nozzles, the first nozzle is generated using the CreateFrustumGrids script. A frustum that is open near the front (where it touches the base), but closed at the back is created. If the open end of the frustum is closed immediately, a closed surface cutter can be obtained. However, this surface is flush with the base surface. In order to do a proper hole cut, the cutter surface needs to pierce the surface that is it cutting. This is achieved by extending the frustum into the main body using GedExtrap to extrapolate the frustum. Since a minor overlap is desired, we only need to extend the surface a few points in the $-x$ direction. In the *nozzles.tcl* file, this is done using a 3 point extrapolation. Another option is to extrapolate the surface tangent so that the slope of the frustum is maintained by using the tangent option instead of the $-x$ direction. Either way, proper intersection between the extrapolated nozzle and base is achieved. Once clean intersection is present, the surface can be closed off using the ResetSurfTopo script by collapsing the $j = 1$ ring of points to a singular axis point.

A very similar procedure is followed for the fins. However, since it requires multiple scripting calls before a fin and its cap are generated, the cutter is created later in the process by extrapolating and closing the main fin mesh before it is retracted for the collar grid. An additional point to add is that since there are multiple nozzles and multiple fins in various locations, it is not sufficient to generate a single cutter. The cutter generated from the first nozzle or fin must be duplicated and rotated or translated to its proper location so that the X-rays are computed at the correct locations so that proper hole cutting can be done.

Once the cutters are made, the BuildScript approach generally uses the Tcl CreateXrayMap scriptlib command to create one *.xry* file for each X-ray in the associated sub-directory. The location of all X-ray files should be specified in the *config.tcl* file. A practitioner of the component-centric approach will create and assemble all X-rays in the master script (see section 8.4)

4.2.1. User Exercise

For the geometry and surface you created for section 4.1.4, create an X-ray cutter for each additional component.

4.3. Volume Mesh Generation

4.3.1. BuildScripts

This approach uses a script called *BuildVol* to generate some or all of the volume meshes. The user can specify the default volume meshing parameters for all grids in the *inputs.tcl* file. Specific volume-grid inputs for the individual grids can be specified in either the *inputs.tcl* file or the *<subdirectory>/localinputs.tcl* file. When the BuildVol command is run, it steps into each subdirectory specified in *config.tcl*, finds the surface grid and boundary condition files, and generates each volume grid. The volume grids are created in the same subdirectory as the surface grid file but with a *.vol* suffix. A set of box grid inputs can be specified for use with BuildVol. Typically, such inputs are placed in the *inputs.tcl* file. Typically, a bounding box, mesh spacing, and stretching parameters need to be specified.

The user can instruct *BuildVol* to not generate a specific volume grid. Instead of using the *BuildVol* script, the volume grid can instead be created by any user-defined process, similar to the surface grids. To enable this option for a specific grid, one must add an entry in the *inputs.tcl* file of the form:

```
set MESH(nomakevol) 1
```

where MESH is replaced with the name of the grid in question. In this case, the user can program the *Makefile* so that the volume grid is generated during the surface-grid generation process. Or the user can specify another user-written script or other process(es) to be executed by BuildVol. This is enabled by an entry in the *inputs.tcl* file of the form:

```
set MESH(command) ``exec command1 ; exec command 2 ; ....''
```

Finally, a post-volume-grid generation command can be defined using:

```
set MESH(post) ''exec command1 ; exec command 2 ; ....''
```

See the *docs/scripts.html* documentation for the extensive number of options available when generating the volume grids.

4.3.2. Component-Centric

The user needs to build the script for the generation of the volume meshes. The volume mesh generation script GenHypVolGrids is usually invoked after the surface mesh is generated in the same component script. For example, *body.tcl* calls GenHypVolGrids near the end of the script file to generate volume meshes. These also have the *.vol* extension. The example component meshing scripts also demonstrate the use of GenHypVolThread to generate volume meshes in parallel.

See section 8.1 for details on how to make box grids for this paradigm.

4.3.3. User Exercise

For each surface mesh you created for section 4.1.4, create a volume mesh.

5. *inputs.tcl* and *config.tcl*

The *config.tcl* file is only used by the BuildScripts approach. This file defines the component directory structure and names of all grids in each directory. In this file, a variable called *rootnames* is set to the list of all grid names. The names must be listed in the order that the user wants the grids to appear in the final *grid.in* file to be used as input to the flow solver. If off-body grids are generated by the user, they must also be listed in rootnames. Additionally, if X-rays are to be used to cut holes, the Tcl variable *xraynames* must be similarly defined; this variable is not needed if Pegasus5 is to be used for domain connectivity.

Instead of grid names, the component-centric approach requires component names. The order of the components is followed for the ordering of the grids in the grid.in file. Because a typical component list is much shorter than a grid list (each component is typically made of multiple grids), a *config.tcl* file is deemed unnecessary. A list called Par(complist) is instead defined in the *inputs.tcl* file to list the components. While this list accounts for all grids attached to a body part, off-body grids are most often not contained in the component grid files. So these are added in a list called Par(oblist). The complist and the oblist are concatenated to form the gridlist.

If the X-rays approach is used for hole cutting, an additional list is defined to list all X-ray names (Par(xraylist)). A cutter (a surface grid can be used as a cutter) is associated with each X-ray and each X-ray item in the xraylist has a *.xry* file associated with it. It is often easier to have one X-ray per file, but this is not a requirement and a *.xry* file may contain more than one X-ray map.

The *inputs.tcl* file also holds all geometry related information. This includes geometry dimensions and parameters such as sweep angle, and airfoil twist. If all the geometry is imported from reference surfaces derived from CAD files, then it may not be necessary to include geometry information in *inputs.tcl*. In the JCLV example, all geometry is defined in *inputs.tcl* in variables that are contained in the Par array. These variables are named with component name, and geometry descriptor such as Par(skirt,diam) for the diameter of the skirt, or Par(nose,len) for nose length. More complex names such as Par(fin,root,airfoil,series) are also used to define details of the fin. In addition, position information is also stored in the Par array with variables such as Par(fin,x) which defines the x location of the fin with respect to the origin.

Another section within the *inputs.tcl* file sets all of the meshing parameters including mesh spacing and mesh marching distances. The Par array is also used for this with variables such as Par(ds,nozzle,col) which defines the mesh spacing for the nozzle collar grid. Once defined, all of the parameters are available to use within surface meshing scripts, or in the provided calls such as BuildVol.

In addition to these, the user may also want to define connectivity, flow solver, and force/moment computation inputs. To keep the *inputs.tcl* file from getting too complex, the componentCentric example splits it into separate files that hold that information and are “sourced” (i.e., included) by the main file. These files are named *inputs.overflow.tcl*, *inputs.xrays.tcl*, etc, and contain information in Par, Ovr, and Fomo arrays that is specific to a connectivity code, or for the OVERFLOW solver.

5.1. OVERFLOW Inputs

All OVERFLOW inputs are stored in the Ovr array. For the OVERFLOW variables that are heavily used, there is an Ovr variable. For example, Ovr(nsteps) is the variable that defines how many time steps the solution should run for. The user can simply list all variables in this way for BuildOveri or WriteOver2InpFile to process.

An extra capability exists in WriteOver2InpFile (generally used with the component-centric approach) to write multiple input files directly. To do this, simply extend the name of the variable by an input file number (e.g. Ovr(nsteps,1)). As in OVERFLOW, the last defined input is carried forward. For example, if Ovr(nsteps,1) is set to 1000, and if Ovr(nsteps,2) is not set, Ovr(nsteps,2) is assumed to be 1000. This applies to all supported variables.

5.2. X-rays Inputs

X-rays inputs used by OVERFLOW to compute connectivity using the DCF methodology are placed in the *inputs.xrays.tcl* file. The DCF routines require 3 pieces of information.

- The ID of an X-ray in the *xrays.in* file. We will think of this as a cutterID.
- A list of grids cut by this X-ray. We will think of these as cutee grids which can be grouped into a cutee group.
- An X_δ associated with the cut. This is the offset distance from the X-ray cutter within which grid points from grids in the cutee list will be blanked.

5.2.1. BuildScripts

In OVERFLOW input files, X-ray cutting instructions are contained in the XRINFO namelists. In keeping with this, the XRINFO variable is used to set these 3 items. First, all X-rays are named. This is done in the config.tcl file in the xraynames list in the order that the X-rays will be assembled. There must be a *.xry* file associated with each of these names in the component sub-directories. For each X-ray instruction, the first item specified in the *localinputs.tcl* file in a component sub-directory is the name of the X-ray. To keep track of each cutting instruction, this is done with the XRINFO(instructionName) variable where instructionName is unique for each instruction. In the tutorial files, these are done in the "cutter_cuts_cutee" format which is recommended. The idxray subtag carries the name of the cutter X-ray, the group subtag specifies the group of grids that will be cut by this X-ray with the X_δ specified under the xdelta subtag. This set of triplets is repeated for all cutter/cutee sets for each X_δ.

An efficient way of handling this task is to create groups of grids that will be cut by one or more X-rays. To create a group, the Ovr(gridname,group) variable must be set to the name of one or more groups that the grid belongs to. This is typically done along with boundary condition specification. While this variable can be user specified, the SetGridBC scriptlib call can also be used to do this. Note that the group specifications are typically saved to a .ovfi file using the WriteOvfi call after cutee groups, FOMOCO families, and boundary conditions have been specified for a grid. There is one .ovfi file per grid.

5.2.2. Component-Centric

Here, the master script is used to specify the first 2 items with the X_δ specified in the *inputs.xrays.tcl* file. A procedure in the master script called CreateOvrHoleCuts names all X-rays by using the AddCutterId command. This command takes a list of X-ray names which are usually specified in the *inputs.tcl* file. The cutee list for each X-ray is then specified using the SetCutterCutee scriptlib command. Since all grids are named when generating the surface meshes and all groups are already specified by the time we come to the master script, the SetCutterCutee command can be used to specify a cutter and all the cutees it will be cutting regardless of the X_δ. The X_δ for various cutter-cutee combinations are then stored in Par(cutter,cutee,xdelta) variables in the *inputs.xrays.tcl* file which now only contains X_δ information. The user can simply source this file so that the processing scripts have access to these variables in the master script or in *inputs.tcl*. The SetCutterCutee script is able to use these variables directly to setup cutting instructions.

A less used alternative is available to make this look more like the BuildScripts approach where SetCutterCutee is called with a cutter, a cutee and an X_δ. This option also makes it easier to deal with X-ray files containing multiple X-rays and eliminates the need for a *inputs.xrays.tcl* file. Note that a careful combination of these 2 methods can be employed for more complex situations.

As with the BuildScripts paradigm, groups can be specified for the component-centric approach. In this paradigm, the user must use the AddGroup command which takes group name, component name, and a list of the grids in that component that are part of the group. Group information is written to a *.bc.in* file along with boundary conditions using the WriteBCInfo call.

5.3. Fomoco Inputs

The FOMOCO codes (mixsur and overint) need several pieces of information. While Overint needs things such as fsmach, alpha, beta, Reynolds number, etc., these can be obtained from the *q.save* flow-solver output solution file. For this reason, we generally set these to 0.0 or negative values to indicate copying from information stored in *q.save*. In addition to these, the user must define grid number and index ranges for surface subsets in each grid that will be used to compute forces and moments. Furthermore, multiple surface subsets can be grouped into components or mega-components on which aerodynamic loads are to be computed and reported. These are all defined in the surface meshing scripts for each component.

Note that FOMOCO components and geometric components are frequently alike in that we would usually like to compute the aerodynamic forces/moments acting on a geometric component. This means that for a collar grid that straddles two geometric components, a portion of the grid belongs to the child component, while another portion of the same grid belongs to the parent component. If we think of nozzles as a child of the main body, then a portion of the nozzle collar belongs to the nozzle component and another portion of the same grid belongs to the body component. This is evidenced in the force/moment components section of the *nozzle.tcl* script under the componentCentric directory. Similar logic is applied to the fins.

5.3.1. BuildScripts

To set defaults and to set moment reference center information, the BuildScript approach uses the MIXSUR variable. For example, MIXSUR(refl) is used to set reference length. The force and moment integration surfaces are specified in the boundary condition files. A Ovr(gridname,family) variable is then specified to assign a FOMOCO family name to the appropriate subset. These are written out to the .ovfi file using WriteOvfi. Once all .ovfi files are available, a call to BuildMixsuri interprets these variables and writes a mixsur input file. An example of setting optional mesh priorities is seen in the *inputs.tcl* file in a list variable called *priority*.

5.3.2. Component-Centric

Similarly, the component-centric approach uses the FOMO variable (e.g. Fomo(fsmach)) to specify defaults, reference values, and moment centers. Note that multiple moment centers are specified using the FOMO(refa,1) notation. Surface subsets in this paradigm are specified using AddFomocoSubset call which takes a FOMOCO component name and a list specifying the subset from a given grid in a given component. This information is written to a *component.fomo.in* file using the WriteFomoInfo call. Once all component *.fomo.in* files are available, a call to ProcessFomoInfo in the master script assembles the information needed for writing a mixsur input file. A call to WriteFomocoInpFile writes an input file to disk.

An example of setting optional mesh priorities is shown in the *inputs.fomoco.in* file in variables called Fomo(componentName,pri,pn) where "pri" indicates to ProcessFomoInfo that it is a priority setting and "pn" is the priority number.

5.4. User Exercise

For each surface mesh you created for section 4.1.4 and its corresponding volume mesh, add entries in the *config.tcl*, *inputs.tcl*, *inputs.overflow.tcl*, *inputs.xrays.tcl*, and *inputs.fomoco.tcl* files. Add entries in the individual component meshing script for force/moment computation inputs.

6. Boundary Condition Specification

It is convenient to consider the boundary conditions (BCs) for a mesh at the end of the surface mesh construction step. For example, the surface of the JCLV body is a viscous wall, and the nozzle exits can be specified as viscous walls, or as thrust producing power boundaries. To specify BCs, we need the BC type, BC direction, surface subset of the grid that the BC applies to, and a component name on a solid surface if using C3P for connectivity.

6.1. BuildScripts

The Build scripts approach requires that the user-written script that generates the surface grid will also create a boundary-condition file with the suffix *.ovfi*. Examples of this are given in each of the subdirectories of the *BuildScripts* example.

6.2. Component-Centric

The CGT Script Library provides a mechanism to specify boundary related information including boundary condition. The reader can follow along in the *body.tcl* file. The first piece of information is to name all grids. This is done using the AddGridNames script with the names of the grids associated with a component as argument. AddBCInfo sets the boundary condition for a specific grid. It takes the component name and local grid number along with a list specifying boundary conditions. If multiple boundary conditions are needed for a grid, they are specified with multiple calls to AddBCInfo. The boundary information for a component is written out to a *component.bc.in* file with a call to WriteBCInfo. Thus, one should expect to have the *body.bc.in*, *nozzles.bc.in*, and *fins.bc.in* files in the working directory. Note that all grids in each component must be named so that the final grid count has a name for every grid.

6.3. User Exercise

For each volume mesh you created for section 4.1.4, add boundary condition specification entries in the individual component meshing script.

7. BuildScripts End-to-End Process

The steps used to create the entire grid system in *BuildScripts* is relatively straightforward. Because of this, there is no single script file that generates the grids. It is up to the user to run one of the following sets of commands, depending on which domain connectivity (grid assembly) approach is to be used.

7.1. Pegasus5 Connectivity

The following execution commands give an example generating the grid system using the Pegasus5 connectivity software, and places all of the resulting files needed to run OVERFLOW into a directory called *RunDir*:

```
BuildSurf
BuildVol
BuildPeg5i
cd peg5
mpiexec -np 12 pegasus5mpi -ifile peg.i
echo "grid.in\n3" | peg_plot
cd ..
BuildOveri -skip-xrays
BuildMixsuri
mv overflow.inp mixsur.i peg5/grid.in peg5/XINTOUT RunDir/.
```

This produces all of the input file required by OVERFLOW. Note that some editing of the *overflow.inp* file will be necessary to get OVERFLOW to run properly. This tutorial has included inputs for both the X-ray hole cutting approach, and the Pegasus5 approach. So to get OVERFLOW to run using the Pegasus5 approach, edit the *overflow.inp* file and remove the following namelists:

```
$XRINFO $OMIGLB $DCFGLB
```

To clean the script directories back to the original state, simply issue these two commands:

```
ScriptClean -clobber
/bin/rm -rf peg5 box
```

7.2. DCF+Xray Connectivity in OVERFLOW

Here are example steps that can be used to generate a grid system using OVERFLOW to perform the X-Ray hole cutting and DCF overset interpolation.

```
BuildSurf
BuildVol
BuildPlot -vol
BuildOveri
BuildMixsuri
mv Composite.vol RunDir/grid.in
mv overflow.inp mixsur.i xrays.in RunDir/.
overrunmpi -np 12 overflow
```

7.3. Configuration Control

Here we examine an example of adding or subtracting components in the BuildScripts. Open the *config.tcl* file and find the following two lines:

```
set IncludeFins       1
set IncludeNozzles    1
```

As an exercise, try setting one of these values to 0, and regenerating the entire grid system using one of the sequences of steps above. Don't forget to start with a clean slate by running *ScriptClean* first. Examine the resulting grid system, as well as the Overflow.inp and mixsur.i files and note that with this one simple change, an entirely new grid system and inputs files can easily be generated.

As an advanced exercise, one can try to add a new protuberance onto some location of the vehicle. When adding the new grids into the *config.tcl* file, be sure to add a new variable that can be used to control the inclusion and exclusion of individual parts. This type of control is the primary method used to build-up very complex CFD analysis by many groups who use the Chimera Grid Tools. It allows one to start with a relatively bare version of the vehicle, test the basic setup from end-to-end, and then add parts until the complete configuration has been built. This approach also allows a group of CFD analysts to work in parallel on the grid generation of complex configurations. Each individual can be working on their own parts, which can then be merged together simply by adding the parts to the *config.tcl* file. For very complex configurations with hundreds of parts it is very useful to create another tcl file that defines all of the variables such as *IncludeFins*. The authors typically create a file called *GlobalDefs.tcl* that define the configuration-control variables. This file is sourced by any file that needs it (the *config.tcl* file, the individual grid-script files, etc) using this line:

```
source [GetIfile GlobalDefs.tcl]
```

7.4. BuildPlot

As one progressively builds up a configuration, experience has shown that catching errors early in the process is extremely valuable and reduces the time it takes to go back and fix later problems. The *BuildPlot* script can be used to verify two important groups of inputs: the boundary conditions, and the FOMOCO component families. These can be verified as soon as a set of *.srf* and *.ovfi* files have been created. Given the *plot3d* plotting software, the following procedure can be used to visually check the family inputs:

```
BuildSurf
BuildPlot -srf -family
plot3d < plot_srf.com
```

Note that when the initial *plot3d* window opens one has to hit the return key with the focus in the plot window to display the surface grids. Each of the FOMOCO component families are drawn using the same color. Careful examination of the boundaries between components, such as the collar grids, can be used to spot errors in family specifications. Similarly, this procedure can be used to visually check the boundary conditions:

```
BuildSurf
BuildPlot -srf -bc
plot3d < plot_srf.com
```

8. Component-Centric Master Script

The master script used by the component-centric approach for the JCLV example is called *mkjclv*. This script consists of a few procedures that define certain functions, followed by the main code. The main code begins by setting the directories with a call to *CheckAndSetDir.tcl*. It also pulls in the *inputs.tcl* file so that all the various user settings are available. Finally, it sets the conCode variable which sets the connectivity procedure to be followed.

While all mesh generation and flow solution can be done in the working directory, many cases are setup to run in several directories (e.g. alpha sweep, Mach sweep, grid refinement). To handle this, the user can setup a run directory location. Here, we use the *gdir* variable to set the name of the run directory. The JCLV is setup to use DCF (X-rays), PEGASUS5, or C3P for domain connectivity. So the choice of *gdir* is conditional on the connectivity code used.

If the user wishes to setup dependency checking, this can be performed next. For example, the body is dependent on the *inputs.tcl* and *body.tcl* scripts. However, the nozzles and fins are dependent on the body surface in addition to the *inputs.tcl* and their own component script. The individual component scripts can now be called to generate surface and volume meshes while checking dependencies so as to not duplicate work that has already been done. Dependency checking can be modified by the user as it is coded in the chkDep procedure in the master script. The user may wish to step through that procedure to obtain an understanding of how basic dependency checking is done for the JCLV. Note that the component list is used to setup a loop to generate the surface and volume meshes.

8.1. Bricks or Box Grids

A user has two options depending on the desired path of execution. Option 1 is to allow OVERFLOW to generate all off-body box meshes. We call them boxes because they are generated in a rectangular shape following the Cartesian directions. In OVERFLOW, they are referred to as bricks. Option 2 is to generate these box meshes manually. Either way, the AddBrickInfo script should be called in the master script. For option 2, this script needs to be called with a single argument "off" to indicate that bricks are user-provided, typically using the GenUniformBox or GenStretchedBox scripts. For option 1, the AddBrickInfo script is provided with additional information that defines level-1 mesh spacing (level-1 refers to the grids closest to the near-body grids), and the distance to the far-field boundary. Additional information to change the behavior of how and where the bricks are generated by OVERFLOW can also be specified. See the *scriptlib.html* page along with the OVERFLOW user's manual for additional details.

8.2. Other Types of Off-Body Grids

Other types of off-body grids include meshes to capture a flow feature that is not necessarily near a body. The most prevalent of these are wakes, and engine plumes. As the locations of these can be estimated with reasonable accuracy a-priori, a user may choose to generate such meshes during the mesh generation process. These meshes can be created with the related components, in a separate script, or in a master script.

In the JCLV example, a user may intend to have the nozzles produce a plume to simulate thrust-producing engines. The plume will not be properly resolved with the volume grids generated for the nozzles and they will also not be resolved by the auto-generated brick meshes. Thus, a user may choose to generate a plume mesh. While multiple commands in the scriptlib can be strung together to do this manually, a library procedure called BuildGeneralPlumeGrids is provided to make this process simpler for the specific situation of a plume mesh behind a nozzle. The reader may do this task as an advanced level exercise.

8.3. Mesh Assembly

The first procedure (proc) in the master script is CombineAllGrids. This procedure builds a surface list by simply going through all the components and adding the *component.sur* files to the list. Sometimes, a component is rotated with respect to another component. The rotated components are labeled *.r.sur* instead of *.sur*. Thus, if *.r.sur* exists, it is accepted before considering the *.sur* file in this process. Similarly, the Par(gridlist) variable is used to compile a list of the volume meshes. The CombineGrids script is then called with these lists to put all surface grids and all volume grids in a *grid.sur* file and a *grid.vol* file, respectively.

An additional step in this process is to compute how many grids there are in each component volume grid file. To do this, we call GetNGrid as we gather the list of volume meshes and assign it to Par(componentname,ng). This variable is later used to write input files and is, therefore, required.

8.4. X-ray Assembly

X-ray assembly is done very similarly in the CreateAllXrays proc with the Par(xraylist) variable. However, since we only have cutters in the form of either *.sur* or *.cut* files, we create the X-rays files from the cutters using CreateXrayMap for each entry in the xraylist. These X-ray files are then combined into the *xrays.in* file in the CombineAllXrays proc by creating a list of the X-rays and calling the CombineXrays script.

8.5. Processing Boundary Information

The ProcessBCInfo call is used to process the information stored in the individual component *.bc.in* files. Using the Par(gridlist) variable, the *.bc.in* files are included in the order specified in gridlist to do three things: (1) Assign the grid a global grid number; (2) Name the grid; (3) Assign the boundary conditions associated with each grid. If a group of grids was assigned and written to the *.bc.in* file, the group member names are converted to numbers here. Note that this last step sets a variable called Par(gridname,glist) or Par(groupname,glist), a step that is necessary for the mesh connectivity step to work.

8.6. Mesh Connectivity

Since the Ovr(nxcuti) variable keeps track of the number of cutting instructions with automatic updates by SetCutterCutee script, we begin the CreateOvrHoleCuts procedure by initializing the Ovr(nxcuti) variable to zero. The AddCutterID call is made next to define all the cutters. This is done using the Par(xraylist) variable since the X-ray list already has a list of the cutters. The only thing left to do now is to call the SetCutterCutee script to specify cutting instructions. This call is made with a cutter as the first argument and a list of cutees as the rest of the arguments. Finally, the number of cutting instructions is reported.

8.7. Mega Components

A mega component for force/moments computation is simply a collection of previously defined force/moment components. Using the JCLV fins as an example, each fin has been defined as a component for force/moment computation. However, the user may also need to compute the total forces and moments on all fins. This is done by introducing an “All-Fins” mega component using the AddFomocoMegaComp script. A new mega component name is given as argument along with a list of existing FOMOCO components.

The ProcessFomoInfo command reads the component *.fomo.in* files along with the mega component definitions, and puts it all in temporary memory so that output files can be written next. Grid numbers and subset indices used in defining force/moment components can be checked using a visualization tool in OVERGRID (see the OVERGRID tutorial for an exercise on this feature). Toggle buttons can be used to activate or deactivate display of each named component defined in the FOMOCO input file.

8.8. Writing Inputs Files for OVERFLOW

Before proceeding, the number of total grids set by ProcessBCInfo and stored in the Ovr(ngrid) variable is checked against the number of grids in the volume mesh file (*grid.in*). If the grid file passes this check, WriteOvr2InpFile is called to write the OVERFLOW2 input file followed by a call to WriteFomocoInpFile to write the FOMOCO input file. If C3P is chosen as the connectivity code, a call to WriteC3PInpFile is also made.

8.9. Code Execution

Once all input files are written, it is possible to directly run the necessary codes from within the master script. The RunProg command can be used to run MIXSUR to setup the FOMOCO files so that OVERFLOW will have the necessary input files for computing forces and moments to produce force/moment histories. Additionally, the RunConnectivityCode routine allows the user to run OVERFLOW, PEGASUS5, or C3P to do hole cutting and compute the donor and interpolation information for all fringe points. Finally, the *exec* command can be used to execute *overrunmpi* to run the code and obtain a flow solution.

9. User Exercises

9.1. Additional component

For each surface mesh you created for section 4.1.4 and any volume meshes created, add entries in the master script for mesh connectivity and mega components.

9.2. Connectivity Codes

If you have more than one connectivity code compiled (OVERFLOW, PEGASUS5, C3P), try changing the *concode* variable to use different codes.

9.3. Local Mesh Density

Change a mesh spacing parameter in the *inputs.tcl* file to see how the mesh changes.

9.4. Global Mesh Density

A parameter named Par(ds,scal) is a global spacing scale. The higher the value, the coarser the mesh becomes with 1.0 as its nominal value. Change this parameter while holding all other meshing parameters fixed to see how the mesh density is affected. For the local mesh spacing values in *inputs.tcl*, values for the Par(ds,scal) parameter has been tested between 0.7 and 2.5. Outside these values, we get negative Jacobians when generating volume meshes. An additional exercise would be to change the parameter so that negative Jacobians are obtained, and then change the smoothing parameters for the volume mesh with negative Jacobians to see if you can fix the process so that no negative Jacobians are produced.

10. Overflow Solution Example

Figure 8 presents a plot of an OVERFLOW solution using the grid system from BuildScripts and the Pegasus5 oversetting approach. This solution was run using an input Mach number of 2.2, $\alpha = 4$ deg, and $\beta = 0$. The solution was run for 1200 iterations, and is well converged to a steady state. The contours on the vehicle surface illustrates the pressure coefficient, together with Mach contours in the Y=0 plane.

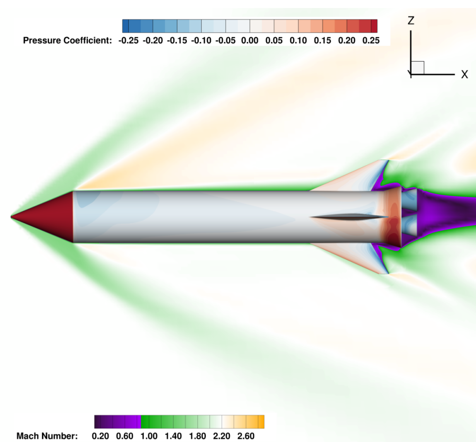


Figure 8. Overflow solution from the BuildScripts grid system using Pegasus5