NASA/TM–20250008940

Loft v4.0: An Updated Automated Mesh Generator for Stiffened-Shell Aerospace Vehicles

Lloyd B. Eldred
Langley Research Center, Hampton, Virginia

September 2025

NASA STI Program Report Series

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

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

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

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

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

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

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

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

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

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

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

- Help desk contact information:

  https://www.sti.nasa.gov/sti-contact-form/ and select the "General" help request type.

NASA/TM–20250008940

Loft v4.0: An Updated Automated Mesh Generator for Stiffened-Shell Aerospace Vehicles

Lloyd B. Eldred
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

September 2025

Available from:

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

Abstract

Loft is an automated parametric mesh generation code that is designed to model stiffened-panel aerospace vehicle structures. After more than 20 years of in-house use and development, the program was released to the public in 2023. Since that release, significant additional features have been added, including new object types, new airfoil shapes, program flow control options, additional options for selecting parts of models, and support for creating boundary conditions, loads, and rigid element stitching from within the code. These additions address previous limitations of the code and significantly expand its application and usefulness.

This memorandum is presented in two parts. The first part contains a technical paper with a brief overview of the code and its applications and a more lengthy discussion of the program's new features. Part two of the memorandum is the updated program user manual. The manual includes in-depth tutorials that use the new features and a complete command reference.

Introduction

Loft is a tool that enables rapid creation of finite element models of aerospace components and vehicles suitable for conceptual and preliminary design studies. Its parametric modeling capabilities enable generation of many variations of a design for exploration of a design space and identification of regions of that space that are worthy of more refined design effort. Over the past 25 years, *Loft* has been used to model parts of a wide variety of aerospace systems, including the Next Generation Launch Technology (NGLT) wing, the payload fairing for the Ares V and the Space Launch System (SLS), the Low Boom Flight Demonstrator (LBFD/X-59), Human Landing System (HLS) reference landers, horizontal takeoff two-stage-to-orbit (TSTO) systems, and planetary atmosphere entry backshells.

Loft's manual was first published in 2011 [1]. The program itself was released for free public use via software.nasa.gov in 2023. Simultaneously, a NASA Technical Memorandum was released [2] with an overview of the code capabilities and a complete updated users' manual. A conference paper and presentation [3] were written to describe applications and scenarios where *Loft* would be useful. Two workshop presentations [4,5] described the uses of the program and the incorporation of *Loft* into a high-fidelity, low-maturity, design environment that is being created at NASA. This memorandum briefly summarizes the previous publications, describes the updated features of the program since its initial release, and includes an updated users' manual for the current features of the code.

The updated, version 4.0, *Loft* program is available for free via a request at software.nasa.gov. It comes packaged with this manual and all of the example files and small utility programs discussed in this document.

Program Overview

Loft is a finite element model (FEM) creation tool that takes a descriptive text file as input and generates a structural mesh in a wide variety of output formats, including NASTRAN, TecPlot, Virtual Reality Modeling Language (VRML, now a subset of the X3D format), and Stereo Lithography (STL). Figure 1 illustrates

an eleven-line input file and two views of the finite element mesh that is produced by *Loft* from that input file.

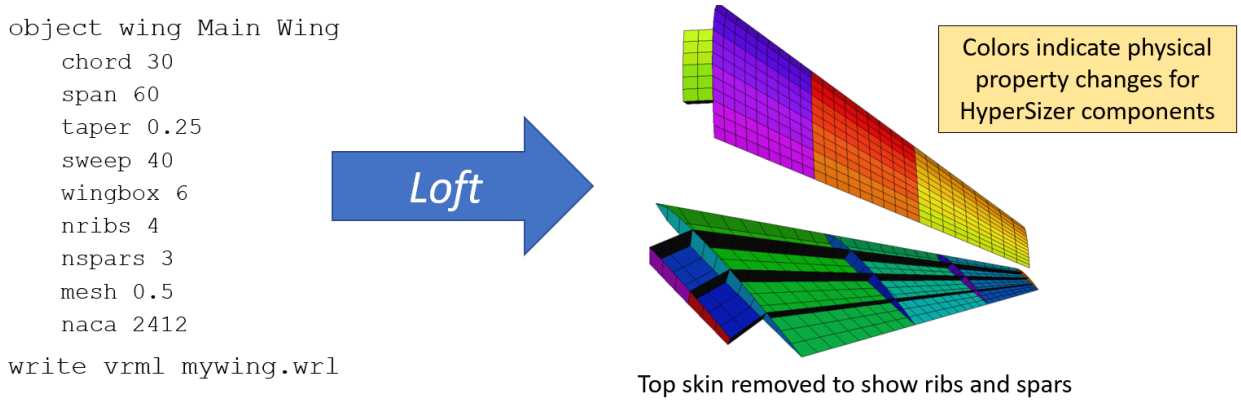


Figure 1. Example Loft input and output

Loft has a powerful variable and math capability that adds to the program's inherent parametric nature. Users can specify meaningful variable names with values. Math support includes addition, subtraction, multiplication, and division as well as trigonometric and hyperbolic functions, log, exponential, square and cube roots, absolute value, and truncation. Figure 2 illustrates the use of variables and math to generate a spherical tank model of a desired volume. Certainly, the calculation for the radius (or the value of pi) could be performed outside of the input file and specified without the use of any variables. But, the variables add clarity and readability to the input and allow the user to quickly generate a family of models with different volumes by changing the value of the volume variable and the output filename.

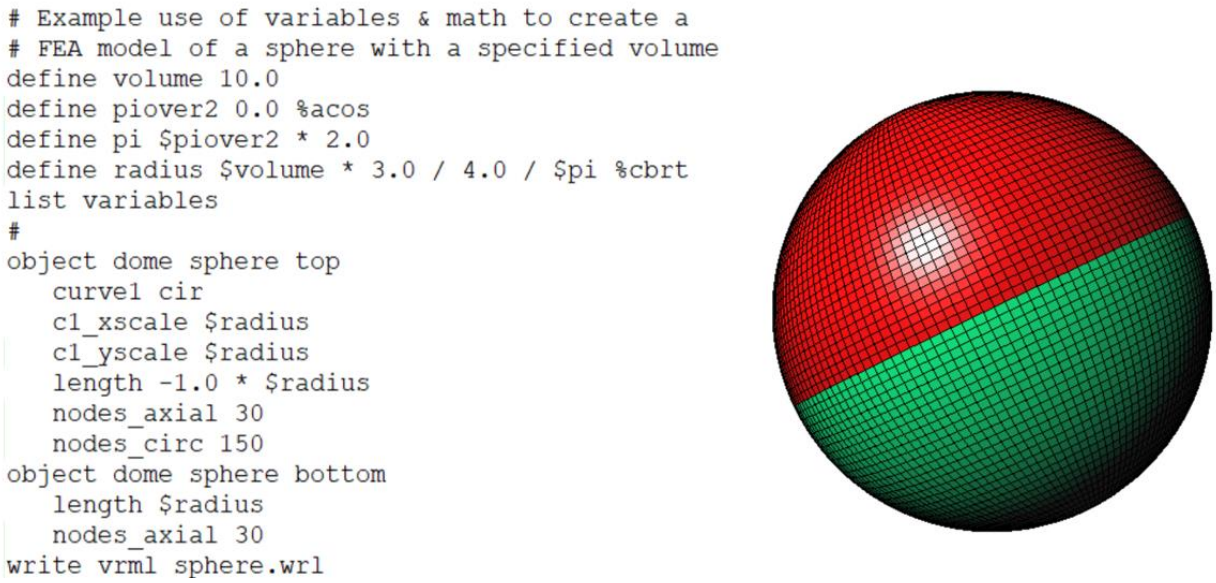


Figure 2. Variables and Math in *Loft*

This math example also demonstrates the inherent parametric nature of the code. Settings that affect the cross-sectional (lateral) shape, dimensions, or mesh counts become the new defaults for later objects and do not need to be specified unless they need to change. Thus, the bottom half of the tank automatically used the shape, radius, and circumferential node counts from the top half. Only the axial values of the length and axial node count needed to be specified. Additionally, the default position for new objects is immediately behind the previous object so that sequentially specified objects are assembled into a stack. The inheritance of lateral settings greatly simplifies the creation of models that are stacks of dome and section objects. Aircraft fuselages and rocket bodies are common components that benefit from this approach. Figure 3 shows an expanded half-model of a TSTO orbiter. This parametric model is explored in full detail in the included users' manual.

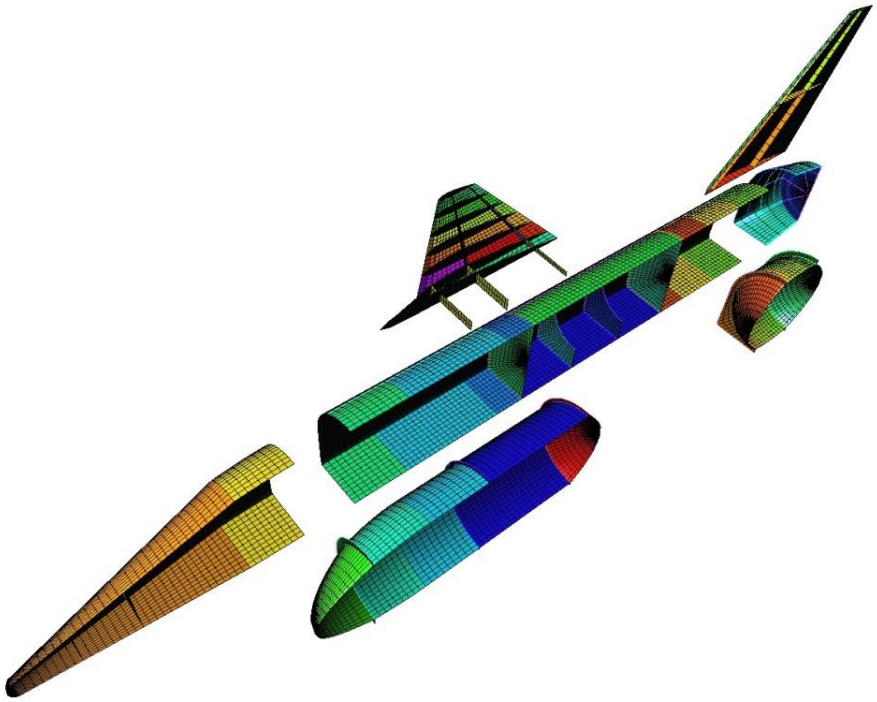


Figure 3. Expanded TSTO orbiter model created in *Loft*

Loft Use Cases

Loft is a general-purpose mesh generation tool. It can be used to create an extremely wide variety of models for multiple applications. But, there are three scenarios where it is a particularly well suited option that should be considered.

The first use case is rapid modeling. A first cut model can be created in minutes with a very small number of input lines. *Loft*'s meaningful default values and intelligent updating of dimensions can substantially reduce the effort needed to make a first cut design. For instance, the cartoon-model of a generic hypersonic aircraft shown in Figure 4 was created with 89 lines of input, 26 of which were comments or variable definitions that made the file easier to read and update. *Loft* also created the STL format file that was used to 3-D print the model.

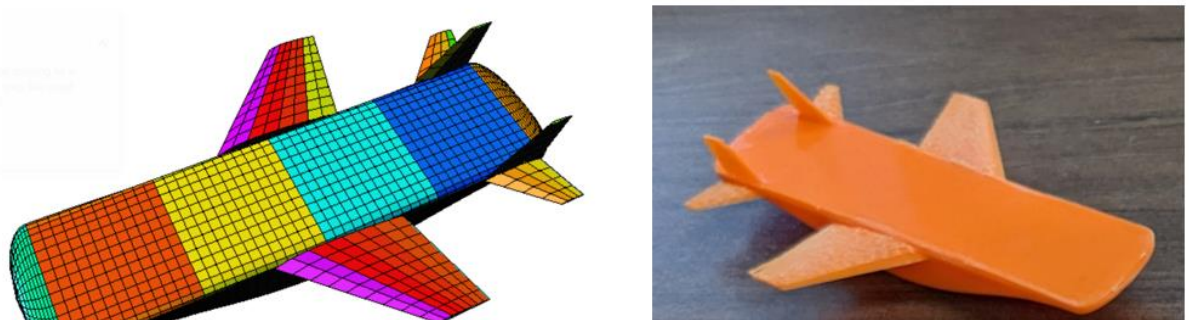


Figure 4. A cartoon hypersonic aircraft FEA model and 3-D print

The second ideal use case for *Loft* is parametric design studies. The ease of changing dimensions or design variables and then writing variant model files enables studies of alternative designs and/or design-of-experiments-style exploration of a design space. As illustrated in the discussion of Figure 2, a family of models can be generated extremely rapidly.

The NASA lunar lander reference vehicle shown in Figure 5 is a more detailed example of this capability. A 656-line input file (not included in this document) specified all of the dimensions of the lander in a few design variables. *Loft*'s math capabilites were then used to compute the location of each component, including the end points of dozens of support struts that are featured in Figure 5. When a global dimension was changed, a new, completely stitched, model could be created in moments.

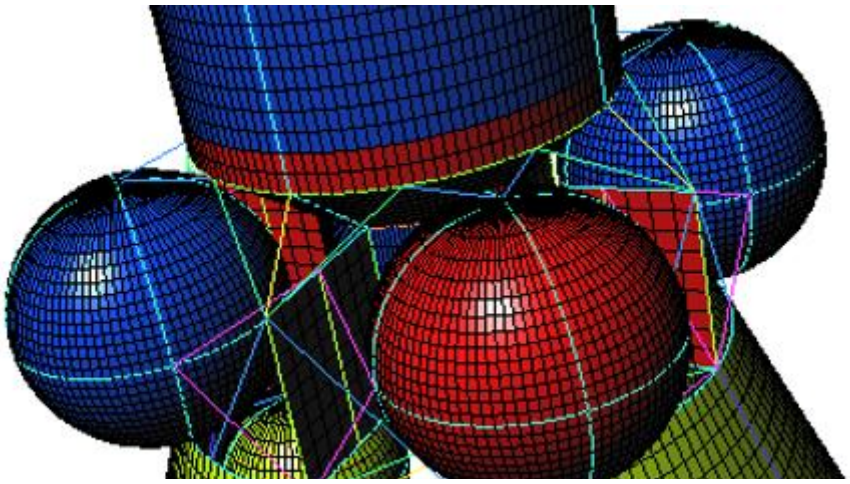


Figure 5. Parametrically defined support struts on a NASA reference lunar lander

The third ideal use case for *Loft* is as part of a multi-code batch analysis system. Its text input, command-line operation, and text file outputs eliminate the need for an application programming interface (API) or graphical user interface (GUI.)

Cerro et al. [6] describe the use of *Loft* as part of a complete conceptual vehicle sizing process. Eldred et al. [7] describe the incorporation of *Loft* into a multidisciplinary system driven by design-of-experiments to perform conceptual design of supersonic aircraft with complex wing and fuselage shapes as illustrated in Figure 6. The wings studied included a potentially large number of spanwise variations of chord lengths, airfoil shape, twist, and sweep angles. The fuselage configurations permitted arbitrary changes in vertical and horizontal diameters and vertical location along the length of the aircraft. These variations were

examined for level of induced sonic boom with the *Loft* generated structural models being used to predict vehicle weight for each configuration. Note that *Loft* generated these wing models as multiple trapezoidal planforms that automatically stitched together to form a single piecewise-trapezoidal model with arbitrary sweep, chord, span, twist, and airfoil shape for each section.

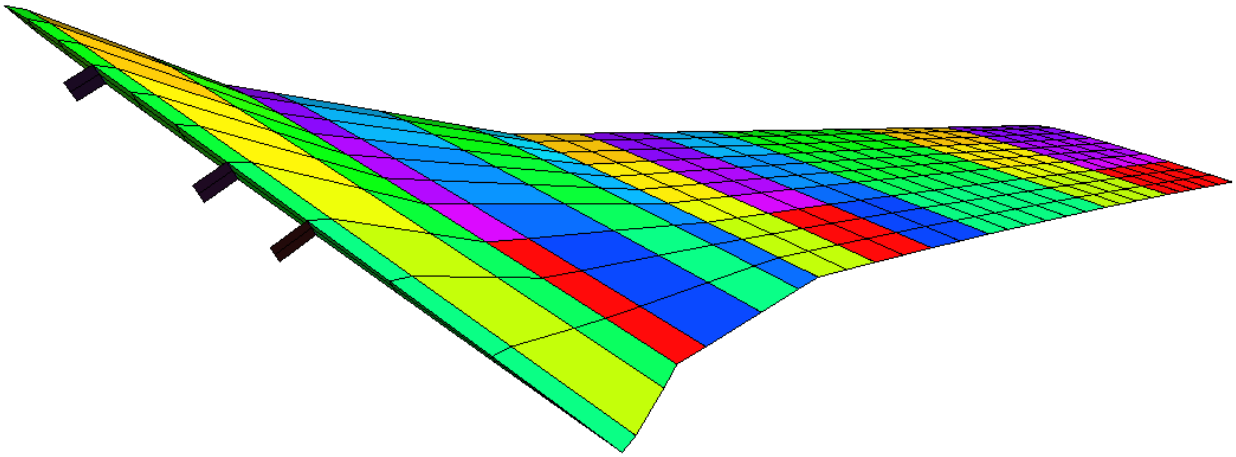


Figure 6. Complex supersonic wing model.

A new low-maturity design capability using *Loft* is currently being built [5]. This system, the Structural Preliminary Analysis for aeRospaCe vehicles (SPARC), will use *Loft* to create conceptual-level finite element analysis (FEA) models of components and vehicles, approximate loading from aerodynamics, thermal, inertial, propulsion, control, and hydrostatic sources, and use the team's existing high-fidelity FEA solution and sizing tools. Other team-developed pre-existing tools to parse NASTRAN input and output, balance loads, map values between dissimilar meshes, etc. will be incorporated to enable rapid conceptual design capturing fluid-thermal-structural-interaction (FTSI).

Loft's parametric modeling and mesh marking make it ideal for this application. *Loft* also features automatic and manual grouping of nodes and elements to enable mapping of loads to only the appropriate portions of the model and to enable other basic tasks such as boundary condition application. For instance, for the wing model illustrated in Figure 1, *Loft* created these 15 groups:

```
Main Wing ROOT NODES
Main Wing TIP NODES
Main Wing ROOT SPAR NODES
Main Wing ROOT RIB NODES
Main Wing CARRYTHR NODES
Main Wing SKIN UP ELEMS
Main Wing SKIN LOW ELEMS
Main Wing SPAR ELEMS
Main Wing RIB ELEMS
Main Wing QUARTER CHORD VECT
Main Wing CT SKIN UP ELEMS
Main Wing CT SKIN LOW ELEMS
Main Wing CT SPAR ELEMS
Main Wing ALL NODES
```

```
Main Wing ALL PANELS
```

Each group created by *Loft* has the user-specified component name (e.g., “Main Wing”) followed by descriptive text. Each group can be reported on by *Loft* on the output screen or to multiple differently formatted text output files. These files can be parsed and modified to create load and boundary condition instructions that are merged with the *Loft* mesh to create a full FEA model. *Loft* can also use these groups to modify or add to the model as shown in the following discussion.

New Features

The updated version 4.0 of *Loft* has many new features, including new and updated objects, input file flow control, and additional math and region operations. Support has been added for creation of NASTRAN boundary conditions, loads, and rigid elements, which improves *Loft* from a finite element mesh generator to a basic finite element analysis input file generator. The previous limitation of *Loft* that required manual stitching of wings, tails, and fins to the main body of the fuselage has been addressed with the ability to programmatically create rigid boundary elements (RBEs) that stitch corners of specified regions together.

New and Updated Objects

The simplest new object added to *Loft* is a node which is created from user specified coordinates. This capability enabled the new support for NASTRAN point masses and force distribution “spider” rigid boundary element (RBE) creation. Figure 7 shows the TSTO orbiter fuselage with four user created nodes. Three are attached to point masses representing the oxidizer, fuel, and payload masses. These are connected by RBEs to the appropriate support bulkheads. The fourth user created node is at the rear of the vehicle representing the thrust of the engines. It is connected by RBE connections to the thrust ring where the engines would be mounted. The generation of these nodes and rigid elements is demonstrated in detail in the annotated TSTO orbiter example included in the users’ manual.

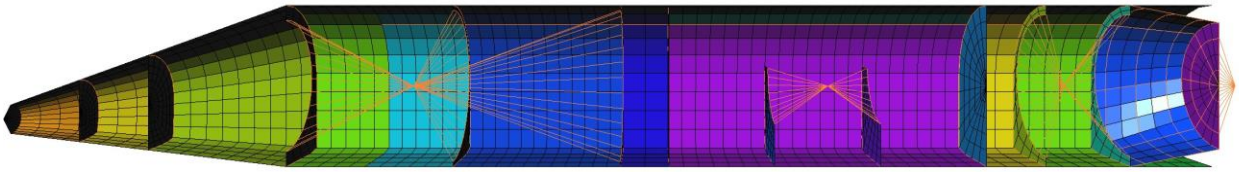


Figure 7. TSTO fuselage with 4 user-created nodes "spidered" to the mesh

The next class of objects are NASTRAN scalar or vector finite element analysis objects. These do not add to the structural mesh, per se, but do add mass, forces, pressures, temperatures, boundary conditions, or rigid boundary element connections to the model. Values can be discrete at a node or smeared across multiple nodes or elements. These objects are currently only written to NASTRAN format output files. However, RBEs are written as linear elements to VRML output files so that they can be visualized. This class of object is also demonstrated in the TSTO orbiter example and documented together in the “BC/RBE/FORCE/MASS/PRESS/TEMP” object description in the manual. Figure 8 shows a *Loft*-generated constant pressure load in red applied to the upper skin of a wing.

A basic truss object was added that can represent a thrust structure or a variety of open interstages on launch vehicles. Figure 9 shows a truss object connecting a circular cross section to a square cross section.

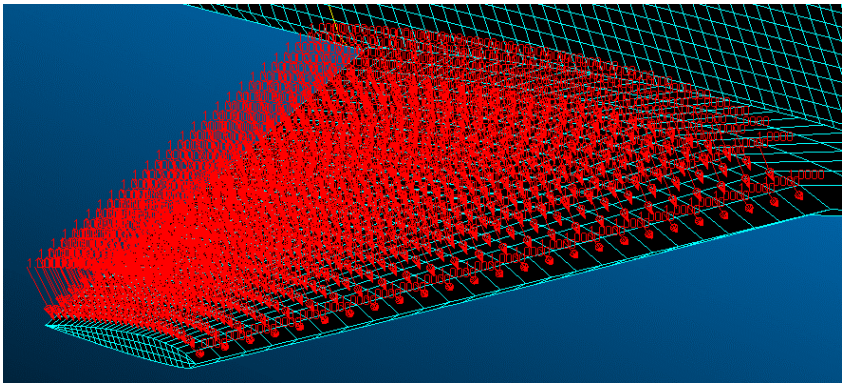


Figure 8. A pressure load applied to the upper wing skin of the TSTO orbiter

A simple trapezoidal block made of solid elements was added to *Loft*. Its primary application is in thermal analyses. Figure 10 shows a block object.

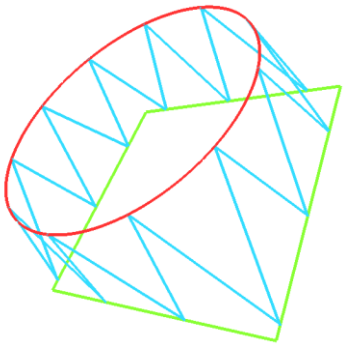


Figure 9. *Loft* Truss Object

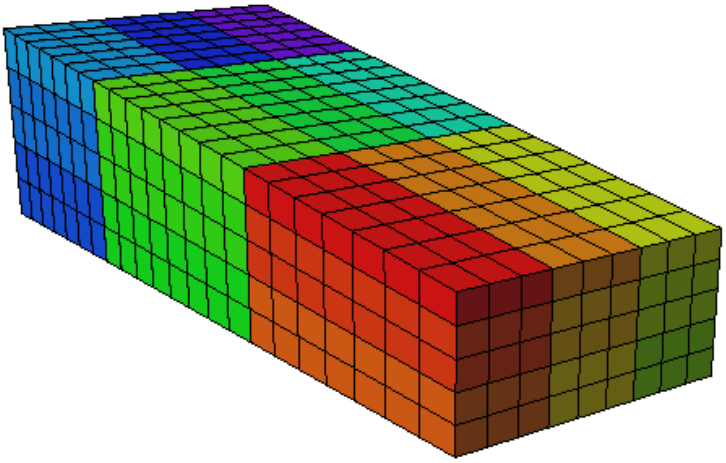


Figure 10. A trapezoidal block made of solid elements

A significant update was made to the wing object type. In addition to the previously supported 4- and 5-digit NACA airfoil cross sections, biconvex and user-defined cross sections are now supported. Also, different airfoil shapes can be used on the top, bottom, root, and tip of a wing. The most common use of this feature is to specify different wing thicknesses at the root and tip of the wing. Figure 11 shows a biconvex airfoil. Figure 12 shows a user-defined diamond cross section with spars, and Figure 13 shows a wing with different upper and lower shapes. Caution should be exercised when combining cross sections with significantly different thicknesses as spars can become sloped when connecting matching percentages along the top and bottom curves.

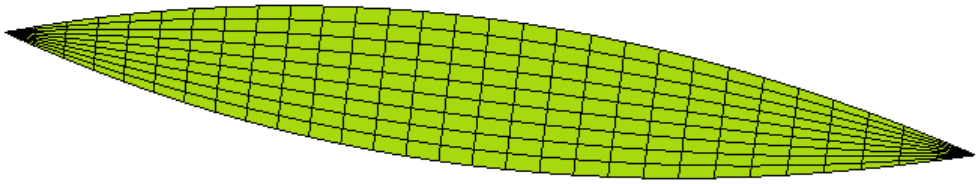


Figure 11. Wing with a biconvex cross section

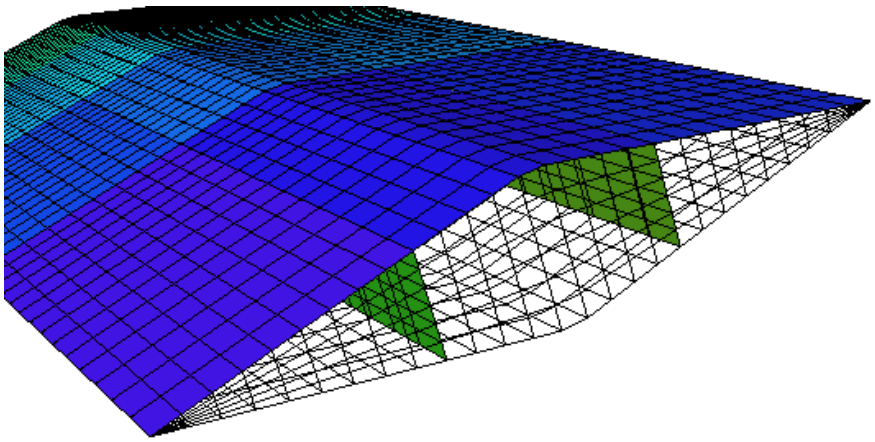


Figure 12. Wing with user-defined diamond cross section

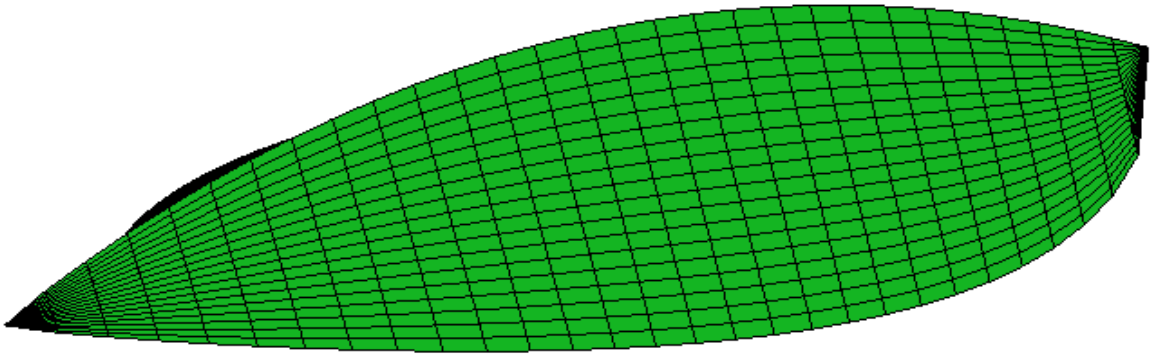


Figure 13. Wing with biconvex upper shape and NACA 2440 lower shape

Figure 14 shows a stiffened box made from a square “airfoil” with ribs and spars.

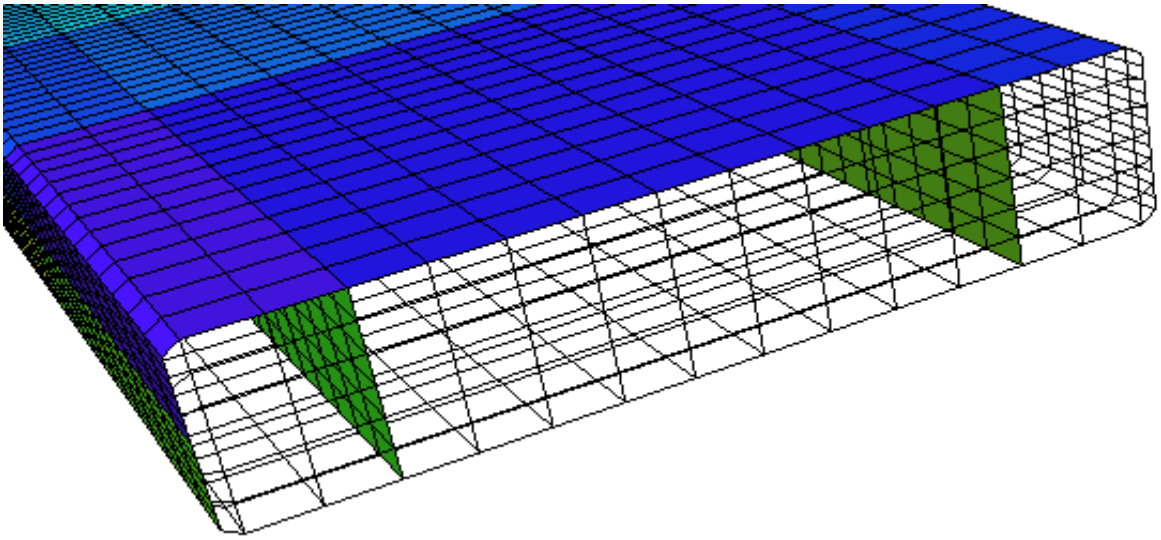


Figure 14. A square "wing" with ribs and spars.

Program Flow Control

Another significant addition to the new version of *Loft* is a variety of tools to control program flow. Three new commands (“linelabel,” “goto,” and “if”) allow loops and conditional execution of input file sections. These new commands are made possible by a change in *Loft* such that it no longer reads and immediately executes each line of the input file. Rather, it reads the entire input file into memory before execution starts. This approach also enables the new “include,” “mirror,” and “clone” commands.

The new “include” command enables the insertion of external input files into the project. These included files could contain common values that are used by multiple projects or could function as a subroutine where input parameters are changed before each call. Figure 15 illustrates a

simple spherical tank case where the same include file operates as a subroutine to generate two different tank models.

```
# main program
define radius 10
include generate_tank.txt
write vrml r10tank.wrl
new
define radius 20
include generate_tank.txt
write vrml r20tank.wrl
```

```
# generate_tank.txt file
object dome tanktop
c1_xscale $radius
c1_yscale $radius
length $radius * -1.0
object dome tankbot
length $radius
```

Figure 15. Main input file and include file used as a subroutine

The new "mirror" and "clone" commands are implemented as macros. The "mirror" or "clone" command is removed from the *Loft* input line stream and replaced by several clipboard (store and recall) and rotation commands that perform the requested operation. The "list input" command can be included after the macro to display the modified input line stream if desired. Figure 16 shows a wing with two clones.

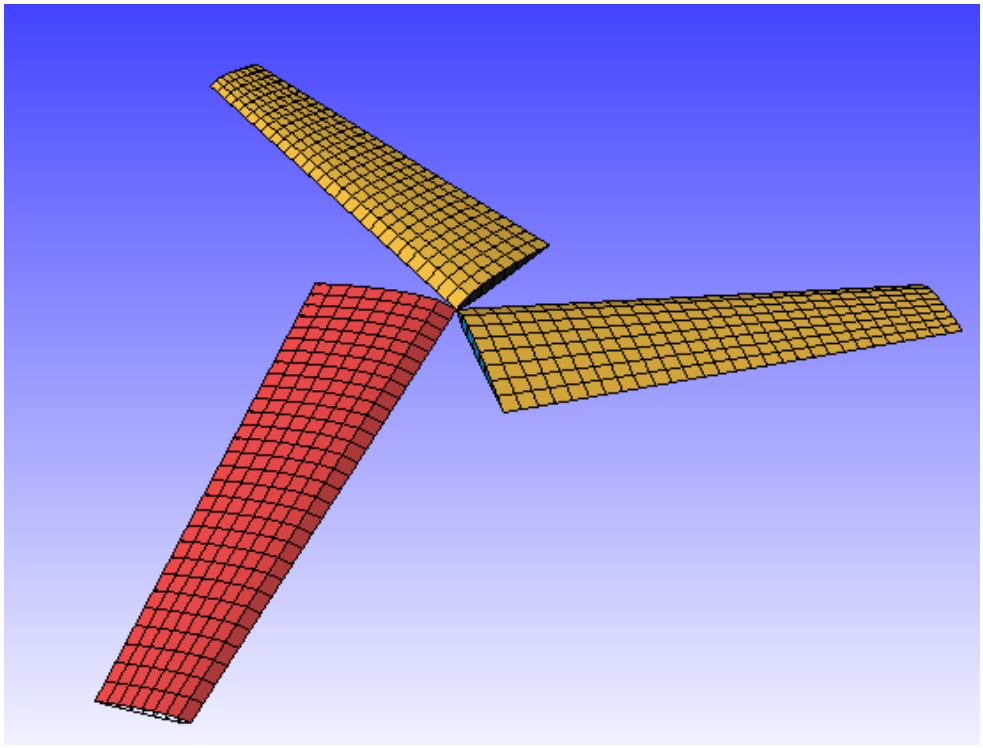


Figure 16. A wing(red) and 2 clones(yellow)

Use of the program flow control commands to execute a loop is illustrated in Figure 17. When creating loops it is important to avoid creating infinite loops. To avoid this, initialize the counter outside of the loop and increment the counter within the loop.

```
define i 0        initialize counter
linelabel 10
<do something we want to repeat>
define i $i + 1   increment counter
if $i < 5
goto 10
<rest of input file>
```

Figure 17. Illustrating a loop in *Loft*

The program control commands can also be used to generate variations of a model in a single input file. The TSTO orbiter input file has a variable "fullvehicle" defined near the top of the file. This is used as a flag to determine if a half or full vehicle is generated when *Loft* is run. In both cases, a half model of the entire vehicle is initially created. Then, if the value of "fullvehicle" is true (i.e., non-zero) that half model will be mirrored to create a full model. If the variable is false (zero) the mirroring operation is skipped and symmetric boundary conditions are created for the plane of symmetry (Figure 18). See the TSTO example for more details.

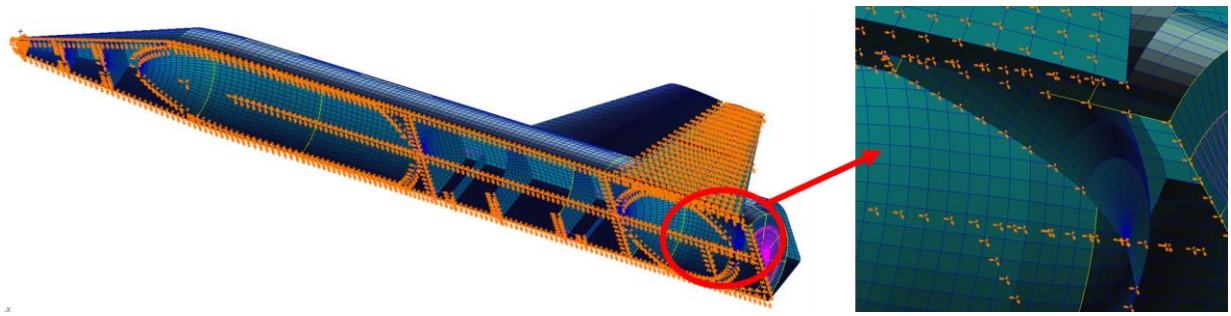


Figure 18. *Loft*-generated symmetric boundary condition (degrees-of-freedom 3, 4, and 5) on centerline of a half vehicle model with closeup detail

Math Updates

Several items were added to the math functionality in *Loft*. Hyperbolic functions (sinh, cosh, tanh, asinh, acosh, and atanh) were added to *Loft*'s function list. Two standard mathematical constants, pi and e can be produced with "@pi" and "@e." The system time and the cpu clock time can be produced with "@time" and "@clock." These only have useful meaning as differences in time:

```
define starttime @time
```

```
define startclock @clock
<stuff happens>
define endtime @time
define endclock @clock
define elapsedtime $endtime - $starttime
define elapsedclock $endclock - $startclock
list variables
```

The ability to compute the minimum and maximum values of coordinates in the current mesh was added. These are invoked as global variables. For instance, "@maxx" and "@minz" will return the maximum x value and the minimum z value, respectively.

Logical inequality operators were added. The operators ">," "<," ">=," "<=," "=," and "!="(not equal) will return a 0 if the comparison is false and 1 if the comparison is true. These are primarily targeted for use with the new "if" program flow control command but can be used in any math operation within *Loft*.

Region Mode Updates

Regions are temporary subsets of the current model. Subsets can be specified by combinations of object name ("Main Wing"), property name ("Main Wing Upper Skin"), labels ("OML") and/or by various geometric location checks. Once created, a region can be written to output, modified, and/or marked to be used later.

The previous release of *Loft* allowed the geometric selection of nodes that fell inside a specified sphere, axially aligned cylinder, or box. The new release adds selection by coordinate comparison. Any of the three coordinates of a node can be compared to a specified value and added/removed if that coordinate is equal, greater than, greater than or equal, less than, or less than or equal to the value. A simplified approach is also available where nodes can be added if a coordinate is positive or negative.

These checks can be combined to bracket a desired portion of the model. For instance, one could create a region comprised of the nodes from the "Main Wing Spar" with X coordinates less than the fuselage width.

A pair of new region operations called "ikeep" and "ekeep" are the inverse of the older "irem" and "erem" operations. In this case, only nodes that meet the specified criteria are retained. Those that do not are removed from the region. The example input below adds all of the nodes on three tank-support bulkhead frames and then keeps only the ones on the inner edge of the frame by using a model variable that contains the radius of the tank. It then removes beam alignment nodes from the region. Finally, the remaining nodes are marked or labeled as "fwd tank support nodes." This label is used later in the model.

```
region
   mkadd fwd fwd ring frame ALL NODES
   mkadd fwd mid ring frame ALL NODES
   mkadd fwd aft ring frame ALL NODES
   ikeep xcyl 0. 0. 0. $tankscale + 1.
   irem xcyl 0. 0. 0. $tankscale / 2. # remove beam alignment nodes
   mark node fwd tank support nodes
```

An optional variation of the "rwrite" region output command was added to the new release. The user can now specify the output format and filename as arguments to the "rwrite" command, making the

syntax the same as the non-region "write" command. If this new option is used, a new file is always created, overwriting any old file of the same name. If appending to an existing file is desired, the new "rappend" command can be used with the same syntax of format and filename as arguments. The previous "rwrite" syntax with no arguments and the format, filename, and overwrite/append choice specified with separate commands remains available.

The last addition to the region mode commands is the "corner" command. This operation identifies the centroid of the nodes in the region and then generates a list of the nodes that are the furthest away from that centroid in each of the eight coordinate quadrants. If no nodes fall in a quadrant, no node is saved. Thus, in most cases a planar region will generate four corners and a non-planar region will generate eight. Some rotation of the region could be required if the desired results are not produced automatically. The argument for the corner command is the name of a group that the nodes are to be added to. Below is an example region command that finds the corner nodes of a portion of a bulkhead below a specified y coordinate and saves them in a group called "Front bulkhead corners."

```
region
   ppadd payload bay fwd bulkhead
   irem yge -80
   corner Front bulkhead corners
```

Automatic Stitching

One of the previous limitations of *Loft* was the requirement to manually stitch wings to a vehicle fuselage. The combination of the new "rbe" object type, the region selection additions, and the region "corner" operation now enable automatic generation of stitching rigid elements. Consider the following example code from the TSTO orbiter model. It uses vehicle parametric dimensions to identify the portions of the wing spar and the fuselage bulkhead that should be connected, uses the "corner" operation to identify the corner nodes on each mesh portion, and then generates rigid boundary elements to connect the two objects. Figure 19 illustrates the new RBEs.

```
region
   mkadd mainwing stiffeners CARRYTHR NODES
   irem xgt $spar1pos + 5.
   irem zlt $fusescale * -1. + 5.
   irem zgt $fusescale - 5.
   corner mainwing fwd corners
region
   ppadd payload bay fwd bulkhead
   irem yge -80.0
   corner Front bulkhead corners
object rbe forward wing rbes
   group1 mainwing fwd corners
   group2 Front bulkhead corners
```

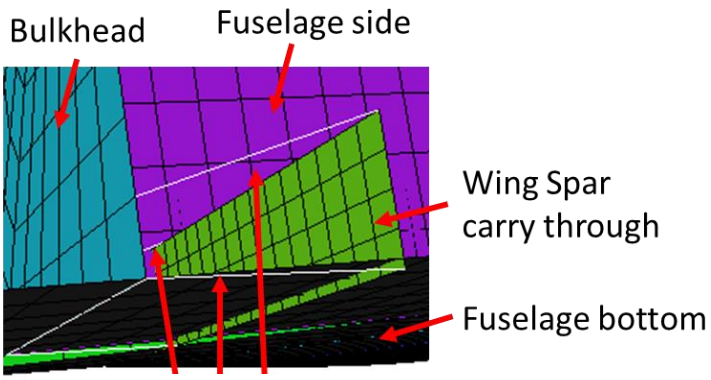


Figure 19. RBEs stitching spar to bulkhead

Other NASTRAN Updates

The new scalar/vector NASTRAN objects can be added to "rbe" type groups, which applies a mark or label to them. These labels can then be used in the region mode to write them out along with the portions of the nodes and elements that have been added to the region.

Four new parameters have been added to the NASTRAN settings in *Loft*. The new parameter "thick" can be used to change the default panel thickness that is written to PSHELL cards. The "sol" parameter can specify a solution type. The "spc" and "load" parameters allow the selection of "setid"s for the boundary conditions or loads. If either or both "spc" and "load" parameters are specified, then a simple case control block will be added to any NASTRAN bdf that is written. That file can then be directly analyzed in NASTRAN with no further editing required.

A very limited support for thermal analysis has also been added. The scalar/vector NASTRAN object can be used to generate nodal initial or boundary condition cards for specified nodes (TEMP or thermal SPC cards.)

Finally, the new "hmcom" and "nohmcom" parameters can be used to toggle on and off (off is the default) comments in the NASTRAN output files that mimic the comments created by HyperMesh. These comments aid in importing models into HyperX and automatically grouping the mesh by the object names specified in the *Loft* input file. The format of these comments has been reverse-engineered to successfully load into HyperX, but updates to HyperMesh and HyperX could break this functionality in the future.

Conclusions

Loft is a powerful finite element mesh generator. Recent updates to the code have increased that power with additional object types and options. Some previous shortcomings, including manual stitching of wings to vehicle bodies, have been addressed with support for automated stitching. Added support for generation of force, pressure, mass, temperatures, boundary conditions, rigid boundary elements, and case control block generation has added basic finite element analysis file creation to the tool.

The updated, version 4.0, *Loft* program is available for free via a request at software.nasa.gov. It comes packaged with this manual and all of the example files and small utility programs discussed in this document.

References

1. Eldred, Lloyd B.; “Loft: An Automated Mesh Generator for Stiffened Shell Aerospace Vehicles,” NASA/TM–2011-217300, November 2011.

2. Eldred, Lloyd B.; “Loft: An Automated Mesh Generator for Stiffened Shell Aerospace Vehicles,” NASA/TM–2023-0001772, July 2023.

3. Eldred, Lloyd B.; “Using Loft: An Automated Parametric Mesh Generator for Stiffened Shell Aerospace Vehicles,” AIAA 2024-2459, AIAA SciTech Forum, Orlando, FL, January 2024.

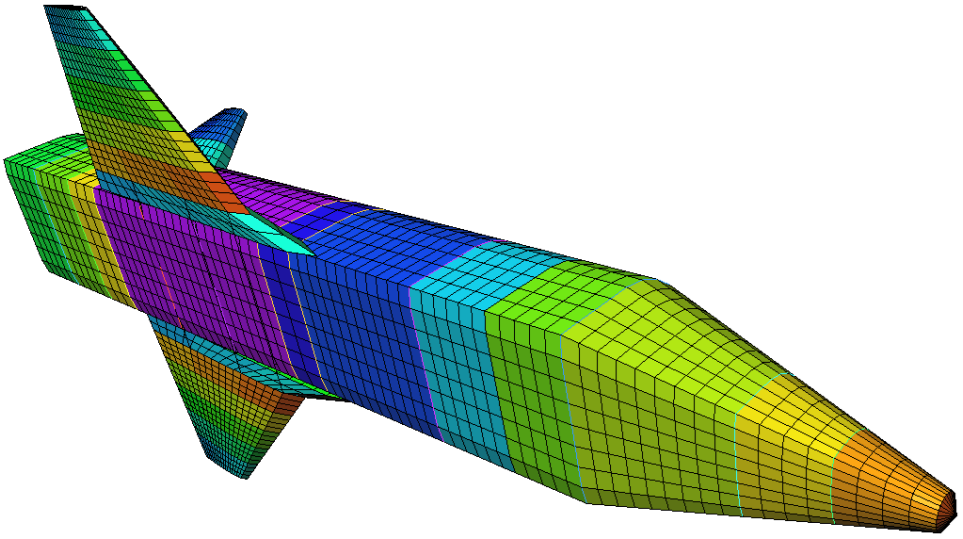
4. Eldred, Lloyd B.; “Two Decades of Aerospace Vehicle Analysis with HyperSizer and HyperX,” HyperX Users Conference, Hampton, VA, June 2023.

5. Eldred, Lloyd B., Seifans, Alexander, M.; “Automated Preliminary Aircraft Sizing with HyperX,” HyperX Users Conference, Hampton, VA, June 2025.

6. Cerro, Jeff; Martinovic, Zoran; Eldred, Lloyd; “Reference Models for Structural Technology Assessment and Weight Estimation,” SAWE Paper No. 3355, 4th International Conference of the Society of Allied Weight Engineers, Inc., Annapolis, Maryland, May, 2005.

7. Eldred, Lloyd B., Padula, Sharon L. and Li, Wu; “Enabling Rapid and Robust Structural Analysis During Conceptual Design,” NASA/TM–2015-218687, February 2015.

Loft
An Automated Mesh Generator
For Stiffened-Shell Aerospace Vehicles
Program Manual



Chapter 1: Introduction

Loft is an automated, parametric, mesh generation code designed for aerospace vehicle structures. Based on user input, it can generate meshes for wings, noses, tanks, fuselage sections, thrust structures, etc. As the mesh is generated, each element is assigned properties that mark what part of the vehicle it is associated with. This property assignment is an extremely powerful feature making possible detailed analysis tasks such as load application and sizing.

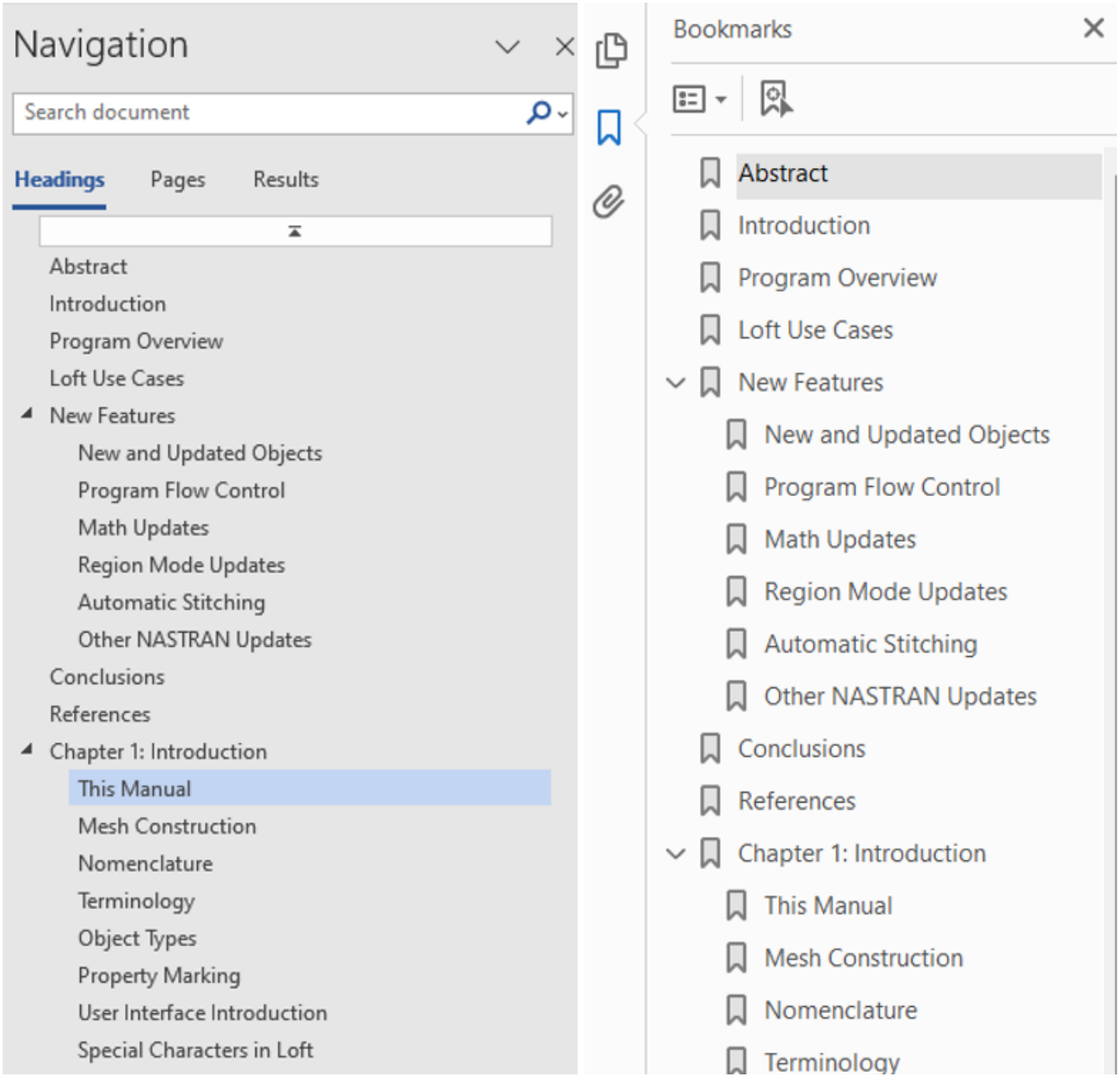
Loft can save meshes in a wide variety of formats, including NASTRAN bulk data file (bdf), EDS' *I-DEAS* Universal File (unv), Abaqus input file (inp), TecPlot, Virtual Reality Modeling Language 2.0 (VRML, now a subset of the X3D standard) and Stereo Lithography (stl) for 3-D printing. The property assignment scheme was designed to make sizing in Collier Research's *HyperSizer* and *HyperX* easy. Support for other mesh storage formats can be added as needed.

This Manual

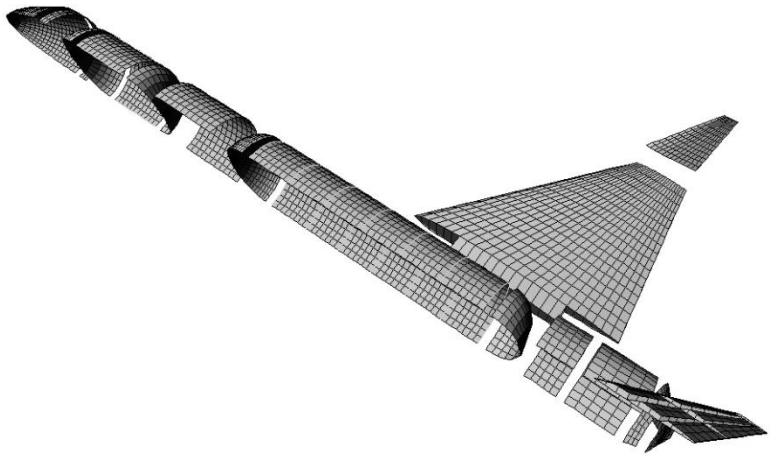
This manual consists of eight parts:

- An introduction and overview of the program and how it works.
- Practical tutorials on constructing a variety of vehicles and components and using many of *Loft*'s features.
- Discussion of the powerful region concept in detail.
- Tips and best practices for the use of *Loft*.
- A technical/programmer's reference describing how the code is written and how to add to it.
- Various external utility programs that have been written for *Loft*.
- A reference guide giving details on all commands and objects.
- Annotated complete example input files.

This manual contains a digital table of contents to aid in navigation. It can be accessed from the Bookmarks option in Adobe Reader or the Navigation Pane in Microsoft Word. Open the table of contents and use it to jump around or search as needed to find the tutorial or command syntax needed for your current project.



Manual Table of Contents in Microsoft Word and Adobe Reader



Mesh Construction

Loft uses very basic finite elements: 4-node quadrilaterals, 3-node triangles, 2-node bars, 8-node solids, and 2-node beams. It uses these simple elements and user input dimensions to build complex full vehicle finite element meshes.

A vehicle is described starting at one end, typically the nose in the case of a fuselage. The user specifies that first component's shape, dimensions, mesh density, and position. The adjacent component is described next, and the process is repeated in axial sequence until the entire structure has been defined. *Loft* copies the dimensions and mesh density from object to object and automatically positions a new object directly behind the previous one, allowing easy construction of a sequential stack of objects. This minimizes user input, with only changes from the default values needing to be specified. In the exploded view above, the example booster object contains eighteen "objects" including ring frames and longerons. Yet it can be built from a 100-line text input file.

Node ordering is set so that element normal vectors point outward. In situations where this is not the desired behavior (such as a concave tank dome), most object types support a `flip` parameter that reverses element node ordering.

Nomenclature

A variety of fonts and styles are used in this manual for distinct purposes. *Italics* are used to introduce new terms and when the *Loft* program itself is named. The `courier font` is used for input file examples, commands, parameters, and references.

Terminology

The lowest level geometric entity used by *Loft* is a *curve*. A *curve* is a two-dimensional object such as a circle, semi-circle, or box. *Loft* includes a library of basic curves and others may be added to the code as needed. Alternatively, *Loft* also features several ways for a user to specify a curve in the input file, including linearly interpolated curves and compound curves built up from any previously defined curves. Curves can be used to create fuselage cross sections as well as for defining wing airfoils beyond the built-in airfoil options.

An *object* is a three-dimensional meshed part made by either extruding one curve or linearly interpolating an extrusion between two curves. (Some objects, such as bulkheads or a ring frame, are actually two-dimensional). Objects include parts such as nose cones, tank domes, tank barrels, bulkheads, etc. Each object is defined separately and has its own name and parameters.

A *stack* is a collection of objects that may make up an entire vehicle. Each object is added to the current stack as it's created, and the full stack is written by the `write` command. The `new` command can be used to start a new stack. The `store` command can be used to assign a name to the current stack, to save it in memory (to a temporary internal clipboard which is lost when the program exits), and to start a new stack. The `recall` command is used to copy a stored stack back into the current stack. `Store` and `recall` can be used to control the scope of object movement, sizing, and distortion commands, as well as to build different configurations of a multi-part vehicle (e.g., Shuttle with external tank (ET) and solid rocket boosters (SRBs), Shuttle with just ET, Shuttle alone).

Object Types

There are a few basic types of objects. Meta-objects are simply macros that combine several of the basic types. Any number and combination of these object types can be created and merged into a single mesh.

Domes are the class of extruded objects taking a single curve to a single nose point. These objects can taper to the nose point in several ways, resulting in elliptical domes, conical domes, parabolic noses, ogive noses, power-law noses or flat bulkheads. Optionally, a droop can be added to a dome to produce simple aircraft nose objects. Domes are meshed with quadrilateral panel elements, except at the nose point where triangular elements are used.

S*ections* are the class of objects that are extruded between two curves. This extrusion is linear and results in parts that can represent tank barrels, fuselage barrels, thrust structures, payload bays, etc. Sections are meshed with quadrilateral panel elements. A *truss* object similarly connects two curves. However, the two curves are meshed with beams or bars and are connected by diagonal struts made of beams or bars.

Frames and *Dframes* are the classes of objects that distribute beam elements along a curve. These can use a single curve as their basis to align with a dome object or be positioned between two curves to align with a panel section. They can run circumferentially or longitudinally (ring frames or longerons). The *frame* object type is used to stiffen a *section* object and the *dframe* object type is used to stiffen a *dome* object.

A *wing* is an extruded surface with internal stiffening (ribs and spars) and optional carry-through. Wings are meshed with quadrilateral panel elements except at the leading edge of each rib where triangular elements are used.

A *tank* is an example of a meta-object macro that combines two dome objects and a section object in a consistent way. It allows for somewhat fewer options than building the tank up from lower-level objects. A *Stifftank* is a meta-object that produces a ring frame stiffened tank.

A *beam* object will generate a beam or a bar element by specifying the location of its endpoints. They are frequently used to specify struts; dome, section, and wing stiffeners are more easily generated with the *frame, dframe,* and *wing* options.

A *node* object specifies a single point in space by supplying its coordinates. Nodes are used when the location of a point mass or force needs to be specified.

A *box* object is a stiffened trapezoidal box created with panels and beams that could represent a mattress tank with flat sides. A *block* object is also trapezoidal but is meshed with solid elements. Blocks are often used for thermal analyses.

A class of objects that specify scalar or vector data that are applied to an element mesh is designed for output to NASTRAN files. These objects can specify scalar values such a constrained degrees of freedom, pressure, and mass or vector values to specify a force's components. See the tutorials on NASTRAN bonus features and automatic stitching as well as the command documentation for the BC/RBE/FORCE/ MASS/PRESS/TEMP object type. These objects do not create nodes or elements but can be labeled/grouped/marked and manipulated with the region mode operations.

Property Marking

One of the powerful features of *Loft* is the labeling of elements corresponding to their location on the model. This is accomplished by assigning dummy properties with descriptive names. (Actual property values are replaced in the analysis or sizing stage). With an I-DEAS output file, each element has a physical and material property reference. Each type of property has a 40-character name available. For NASTRAN, property names are indicated as Patran-compatible comments on the element property and material cards. VRML output files are colored to indicate their property assignments.

For simple domes and sections, the name of the object is placed in the physical property, referenced by all of its elements. The material property is used to indicate where on the object the elements are. The resolution of the material property name is controlled by the "components axial" and "components circumferential" object parameters. A typical material property name could be "Axial 3 Circ 5." Note that these are not element coordinates; there are generally more than one element per component in each direction (but there need not be).

For wing objects and meta-objects like tanks, the physical property name will be more descriptive. It will start with the object name but then add details such as "RIB," "SKIN UPPER," or "DOME AFT." For these kinds of objects, a short object name is recommended so that the full property name will fit in 40 characters. An object name longer than 27 characters will be occasionally truncated. This truncation will be just enough to allow the full inclusion of the detail string.

HyperSizer concatenates the physical and material property names to make component names. Thus, each group of elements with a unique combination of property names will be collected into a component. Typical component names will look like:

"LOX TANK | AXIAL 5 CIRC 2"
"CANARD SKIN LOWER | SB 2 CB 5"

I-DEAS universal files that HyperSizer generates will contain property names that start with "(HSGEN)" and are followed by as much of the component name as will fit in 40 characters.

Loft also generates a variety of groups (also called labels or marks) when running. These groups mark nodes that are on curve endpoints, lines of symmetry, wing attachment points, etc. These groups are named

based on their object name. Thus, for an object called “MyWing,” there will be groups called: “MyWing Root Nodes,” “MyWing Tip Nodes,” “MyWing All Nodes,” etc.

The user can specify additional groups to which an object’s nodes or elements can be added, using the `mark` object parameter. Any number of *marks* can be specified per object and a particular group name can be used by any number of objects. For example, a small nose-cap object might belong to marked groups “Booster Nose Elements” and “Booster OML Elements.”

The `list` command can be used to view the current property and group lists. Use “`list mprops`,” “`list pprops`,” or “`list groups`” to generate a list that is written to the screen.

User Interface Introduction

Loft is controlled by a text file input deck. The user specifies each object that is desired in the model. For each object, geometric data such as diameter, length, and position are supplied. Meshing variables such as the number of elements and the number of sizing components in each direction are also needed. Most input values are optional; default values will be used for any not supplied by the user.

A *Loft* input deck is read line by line. Each line can be a *comment*, *command*, or a *parameter* for the most recent command. Any number of parameter lines can be given (including zero), with a new command line marking the end of the previous command and its parameters. All input is case-insensitive.

Comment lines start with a pound sign, “#,” followed by any amount of text. Comments are ignored by the *Loft* code. Comments can also be placed on a line after a command or parameter by using the pound sign marker.

Command lines cause objects to be created, output to be written, and variables to be set. There is a very short list of legal commands.

Parameters are optional lines that specify details for commands. All parameters are optional and are used when the program default is not what is desired. Some defaults are fixed, but most defaults will change based on previous user input. For instance, the default position for a new object is immediately behind the previous object, and the default curve to extrude is the previous curve. Thus, the defaults will attempt to produce a stack of smoothly connected objects.

To specify a parameter, add a line after the command with the parameter name followed by the new value. Parameter ordering does not matter for `object` parameters; an object is actually generated when the next command is encountered. Parameter ordering *does* matter for the `move` and `region` commands. Indenting of parameters is optional but can improve readability of the file.

Input lines may contain basic mathematical operations, specified in infix notation with equal priority for all operations, e.g., multiplication and division are not given precedence over addition and subtraction. Currently supported operations include addition, subtraction, multiplication, and division along with logical operators greater than, less than, greater than or equal, less than or equal, equal, and not equal.

Loft also supports user-defined variables using the `define` command. These variables may be combined or modified using the basic math operations.

Here is a short example input file.

Comments start with the # symbol, either alone on a line, or after some input.

```
# This creates a circular to breadbox transition
# for a half vehicle
object section MyTransition
   curve1 sc  # semi-circle
   curve2 sbb # semi-breadbox
   length 12
# save
write vrml MyTransition.wrl
```

The three parameter lines for the section object are indented for clarity. This is not required by *Loft*.

Loft is designed to be run from a command line. Windows users may call this a "DOS shell." One way to open a command line interface in Windows is to select "Run…" from the Start Menu, then type "cmd" as the name of the program to be run. Then use the "cd" command to change directories to where the input file and *Loft* executable are located. The input file name is given as an argument when *Loft* is run, such as:

```
loft mytransition.txt
```

On a Windows machine another option is to create a batch file to run *Loft*. Start with creating a text file with the desired command. It's suggested to add a greater than symbol and then the name of a file to capture the output from *Loft*. Your new file text would end up something like:

```
loft inputfile.txt >outputfile.txt
```

Save that text file, then change the extension to ".bat." Now you can double click on the file to execute the stored command or commands. A DOS window will open, show you the command running, and then close. The specified output file can be read to see the run-time output from *Loft*. Other operating systems have similar functionality (Linux/UNIX shell scripts, etc.)

Special Characters in *Loft*

Several symbols are used as flags for *Loft*'s input parsing routines. They indicate that the text following has a special meaning. See the "Variables and Math" tutorial (project 7 in chapter 2) for a more complete discussion of most of the symbols. Here is a current list:

- The number or pound symbol is used to start a comment. It can be used at any point on an input line. Everything after the pound symbol is ignored by *Loft*.
$ - The dollar symbol is used to recall a user variable that has previously been set using the define command.
@ - The at-sign symbol is used to recall a system variable. A list of system variables is provided in Chapter 7's "System Variable List" charts.
% - The percent sign is used to call a math function such as sine or square root. See Chapter 7's "Math Function List" chart.
+, -, *, / - The plus, minus, star, and forward slash symbols are used for their corresponding math function: addition, subtraction, multiplication, and division.

>, <, =, >=, <=, != The less than, greater than, and equal symbols are used for their corresponding logical comparison operation. Less or greater than can be combined with an equal symbol and the exclamation mark can be used with an equal symbol to indicate “not equal.”

Positioning in *Loft*

Each object is automatically positioned by *Loft* in such a way as to produce a single, continuous vehicle. From time to time, this default positioning will need to be overridden. There are a wide variety of positioning, rotation, scaling and warping options available to the user. Most of these operations can be done at both the object and stack levels, with some significant ordering related differences between the two approaches.

The default axes for a vehicle have X as the lateral direction, Y as the vertical direction, and Z as the vehicle axial direction. These axes are aligned in a right-hand rule configuration. Z increases as the stack is built. Another way to state this is that the 2-D curves are defined in the X-Y plane, with Z as the extrusion direction. If, as in the example vehicles included in this manual, the stack starts at the nose, then the positive z direction is aft on the vehicle. Use of the rotation commands prior to saving the mesh can align the mesh as the user prefers. NASA models will typically use X as the vehicle axial direction. Converting to this alignment requires two lines before saving the model:

```
move
roty 90
```

Each object has a local origin that is placed at the current default location. For wings, the local origin is the leading edge root node. For domes, sections, and frames, the local origin is the center point of curve 1.

Most *Loft* vehicles start with an outward dome object (vehicle nose). Consequently, that nose will be specified with a negative length and will be created with most nodes residing on the negative Z-axis. The global origin will be at the rear of the nose (the center of curve 1). A translation must be specified if moving the global origin to the vehicle nose tip is desired.

When a new *section* object is created, the default position for any subsequent objects is moved to the center point of curve 2 (to position it behind that section object). Other object types do not move the default creation point. However, any use of object level or stack level positioning commands (see the heading below) will change the default creation point of all following objects. Note that meta-objects, such as the *tank* type that contain *section*s, will also move the default creation point.

The default positioning for a new object can be set back to the global origin with the `reset` command (which also resets all object dimension defaults to their initial values). A `store` command moves the current stack to an internal clipboard then resets the default position values as well.

Object vs. Stack Level Positioning

To use a positioning parameter at an object level, just add a line specifying the position parameter name (`transx`, `relx`, `rotx`, etc.) and value to the file section describing that object. The ordering of object level parameters does not matter. Once all parameters for the object have been read, the mesh is generated, and then the positioning is performed in the following order: warping, rotations, and then translations.

To position the entire current stack, the `move` command is used. Position parameters that are given, following a `move` command, are acted upon in the order in which they are read.

Translations

There are two types of translation setting options: absolute and relative. The absolute translation parameters `transx`, `transy`, and `transy` override the default position setting and assign an absolute position to the item. The relative translation parameters `relx`, `rely`, and `relz` can only be used at the object level. They add the user-specified value to the default value, rather than just replacing the default. In most cases, using the relative translation parameters is preferable, as a dimension change much earlier in a vehicle stack will not cause inaccurate positioning.

Usage: <parameter> <value>
Example: `relx 2.0`

Rotations

Similarly, there are absolute and relative rotation commands. They are `rotx`, `roty`, `rotz`, `relrotx`, `relroty`, and `relrotz`. As with the translation commands, the relative rotation commands can only be used at the object level.

Usage: <parameter> <value>
Example: `relrotx 2.0`

Scaling

The three scaling commands can only be used at the stack level. They are `scalex`, `scaley`, and `scalez`. (Use the curve `xscale` and `yscale` parameters at the object level to perform a similar function.)

Usage: <parameter> <value>
Example: `scalex 2.0`

Warping

Warping allows the distortion of part of a mesh. The warp commands use a coordinate axis as the dividing line between parts of the mesh that are modified and parts that are not. The last two letters of the parameter specify the side of the axis (p for positive, n for negative) and the axis to use as the division. For instance, the `warppx` parameter will distort all nodes that start with positive x coordinates.

There are two types of warping available: constant and gradient. Constant warps (`warppx`, `warppy`, `warppz`, `warpnx`, `warpny`, and `warpnz`) will scale all nodes in the specified zone by the given values. Gradient warps (`gwarppx`, `gwarppy`, `gwarppz`, `gwarpnx`, `gwarpny`, and `gwarpnz`) increase the distortion the further the node is from the given axis. The user-supplied value is the scaling applied for nodes that start one unit away from the axis. Nodes that start two units away from the axis are distorted twice as much, and so on.

Each of the warp parameters takes three arguments: the amounts to scale the x, y, and z coordinates of affected nodes. For example, the parameter "`gwarpny 1.0 1.0 2.0`" will scale the z coordinates of any node that starts with a y coordinate less than zero. A node that starts at y = –1 will have its z coordinate doubled, if it starts at y = –1.5 it will have its z coordinate tripled, etc.

Only one warp operation can be specified at the object level per object (the last one read will be the one that is performed.) A warp operation combined with a scale operation can produce the effect of two warp operations. Any number of warp operations can be performed at the stack level. Interleaving warp parameters with translation parameters can give a very fine control over the nodes being distorted.

These commands can significantly change element aspect ratios and lead to poorly formed elements. Use with care and verify that the desired effect is being obtained before proceeding.

Usage: <*parameter*> <*x scale*> <*y scale*> <*z scale*>
Example: `warpnx 0.1 2.0 5.2`

Flipping

By default, node ordering for elements is chosen such that element normals will point outward. The `flip` parameter can be used to reverse this ordering. It is valid for both objects and the full stack. Only panel node ordering is affected. A quad that started with nodes 1-2-3-4 will be flipped by reordering its nodes to 2-1-4-3.

Usage: `flip`

Turning

This option is valid only at the stack level. A `turn` parameter reorders the nodes with the intention of changing the material orientation vector to be parallel to a different element axis. A quad that started with nodes 1-2-3-4 when turned will be connected 2-3-4-1. The actual interpretation of this operation will depend on the FEA package used.

Usage: `turn`

User Specified Curves

Loft supports three ways of defining new curves in the input file. Once defined, a user-defined curve can be used in the same ways that a curve from the built-in curve library is used. As part of the definition process, the user specifies a mnemonic for the new curve. Whenever a curve mnemonic is encountered after that point, *Loft* will search the list of user-defined curves, then its internal curve mnemonics. This makes it possible to override the definition of a built-in curve.

Interpolated curves are built from user-specified x, y coordinates pairs. Currently, only linear interpolation between the user's points is supported; options for curved interpolation may be added in the future.

Compound curves are built by tracing the outside of sequentially listed curves until the next curve is encountered, then tracing its outside until it intersects with the next curve, etc. This curve option can be used to define the shape of multi-lobe tanks, etc.

Lofted curves are curves created by blending two parent curves. These curves are temporarily created in most mesh creation processes that *Loft* performs where the cross section of the object is changing along its length from the curve specified at one end to the curve specified at the other end. The user-defined lofted curves allow the user to store and use these blended shapes. One application of the lofted curve type is to create a bulkhead in the middle of a section.

Curves are defined by using the `curve` command, followed by the type (`interpolated`, `compound`, `lofted`, etc.) and a user supplied name. Parameter lists for the curve command are discussed in reference chapter 7, and tutorials on using all types of user-defined curves are in tutorials 3 and 4 in chapter 2.

Chapter 2: Tutorials

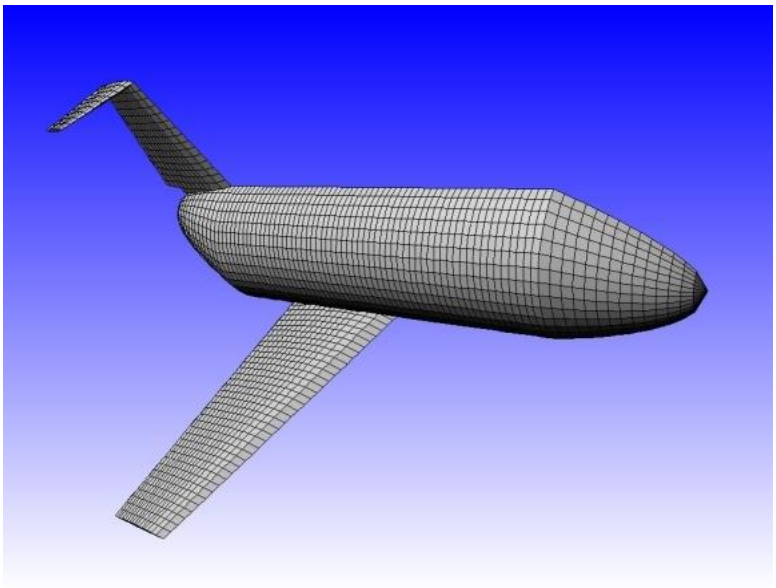
Introduction

Loft is an easy-to-use program that takes very simple finite elements and builds detailed finite element meshes. A user controls *Loft* by creating a text input deck with their favorite editor such as notepad in Windows and vi or emacs in Unix/Linux.

The input files developed in these tutorials are all available in their finished forms in the "tutorials" subdirectory. They are named "project1.txt," etc. and will produce output files named "project1.wrl," etc.

List of Tutorials

Project 1: A Simple Commuter Jet



The examples in these tutorials will consist mostly of symmetric or half models, where only one side of the vehicle is generated. This is done so that internal details of the meshes can be viewed easily. Project 2 will show how to modify the input file to produce a full vehicle model.

A good practice is to start the file with a few comment lines describing the file. The tutorial projects will also use comments throughout the files being created for ease of reading and to explain what is going on. These are completely optional. So, the input file starts:

```
# Loft Tutorials: Project 1
# A Simple Airliner
# Created 4/16/03 by N. Jineer
```

Generally, a user will want to describe a vehicle starting at one end and moving sequentially from major component to major component. This example starts with the nose:

```
# The nose
object dome Nose
```

`Object` is a *Loft command*. As might be inferred from its name, it creates a new object. That's all that is needed, assuming the desired result is a spherical dome that is one unit in radius and one unit in length. But, let's change from the default values. To do that, *parameters* are supplied for the object command. All parameters are optional. It's only when the default values need to be overridden or when the user wants clarity that they are needed. For instance, the initial default value for the `curve1` parameter (as found in the `dome` object documentation in Chapter 7) is `sc`, so the first new line below isn't actually necessary at this stage.

```
curve1 sc
```

```
length -15.0
c1_xscale 10.0
c1_yscale 10.0
```

The curve library section of chapter 7 shows the various curve shapes that *Loft* currently supports and the mnemonics by which a user references them. The `sc` mnemonic produces a semi-circle. The `length` parameter controls how long the dome is. Since the positive axial direction for *Loft* is aft, and the nose should be generated in the other direction, a negative value is given. The next two lines change the radius of the circle in the horizontal (x) and vertical (y) directions. Here both scale factors, `c1_xscale` and `c1_yscale`, are set to be the same value of 10.0.

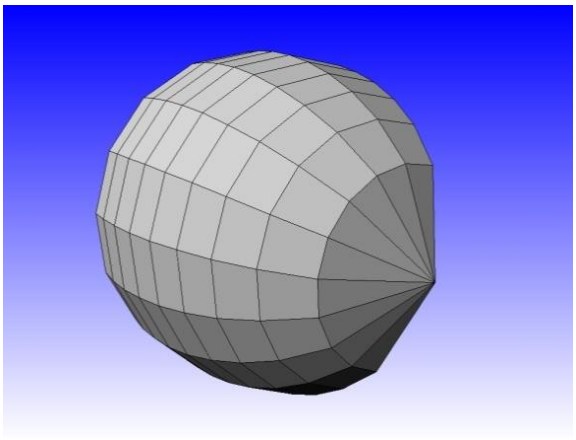
Now, let's see the result. To do that, an output command is added to the file:

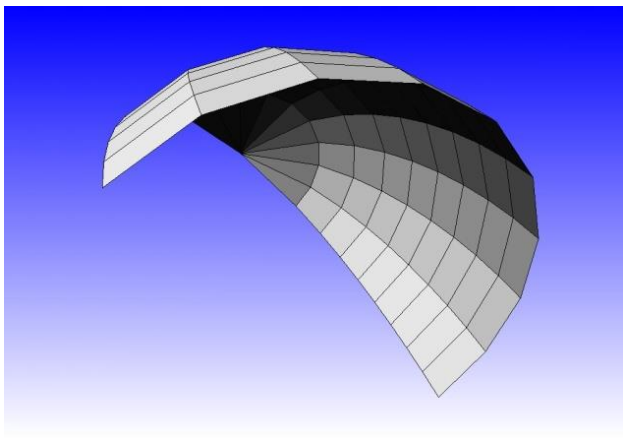
```
# Save and exit
write vrml project1.wrl
end
```

The `write` command tells *Loft* to write the current mesh to a data file, in a variety of possible formats (see the command reference in chapter 7 for supported formats). The `end` command is optional; *Loft* will exit when it runs out of input. Save the file, then run *Loft* at a command line prompt (under Windows open a MS-DOS Shell window)

```
loft project1.txt
```

Loft will produce a variety of text output describing what it is doing. If all went well, *Loft* created a new VRML 2.0 file called "project1.wrl." Open this file in an appropriate viewer (one is not included with *Loft*) and rotate the model to see it from various perspectives:





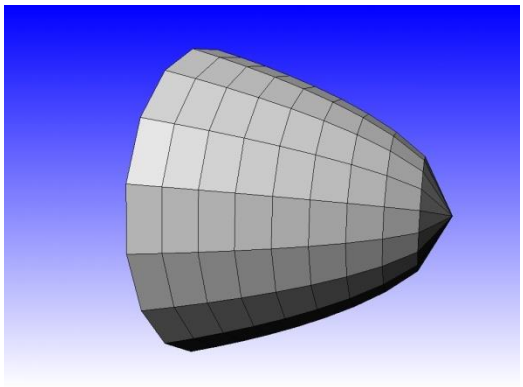
Obviously, the model could use some improvements. Open the input file in the editor again.

More parameters will be added to the end of the nose object definition, so move the cursor above the "`# Save and exit`" line. From now on save, run *Loft*, and view the current object whenever desired to see how things are going. Note that write commands can be added wherever desired in the input file, so "`write vrml project1-nose.wrl`" could be added after all the nose object parameters and "`write vrml`

`project1-nose-and-body.wrl`" after the body is added, etc. Remember, however, that all parameters for a command (such as the object command currently being edited) need to follow that command directly; once another command is encountered (i.e., a `write` command) the previous command is finished.

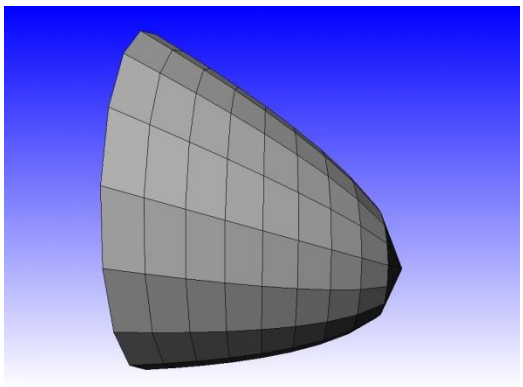
The first thing to change is the curvature of the nose. Referring to the "taper library" section of chapter 7, there are illustrations of differently shaped dome objects and the mnemonics necessary to use them. Change from the default spherical tapered dome to a parabolic tapered one.

```
taper para
```



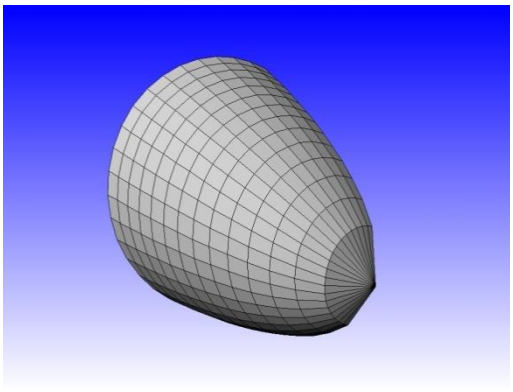
Now, drop the nose tip down a little so the pilots can see out.

```
zdroop 4.0
```



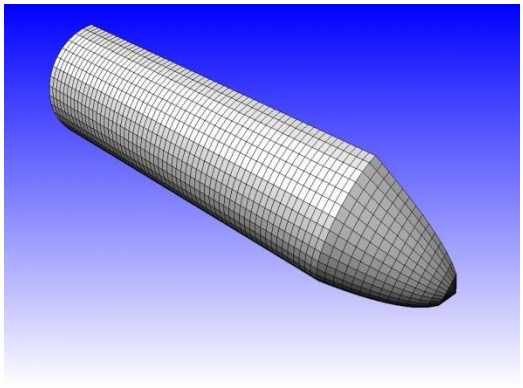
And make the mesh a little denser.

```
nodes_circ 21
nodes_axial 15
```



Now, create a fuselage body. That requires a new section object.

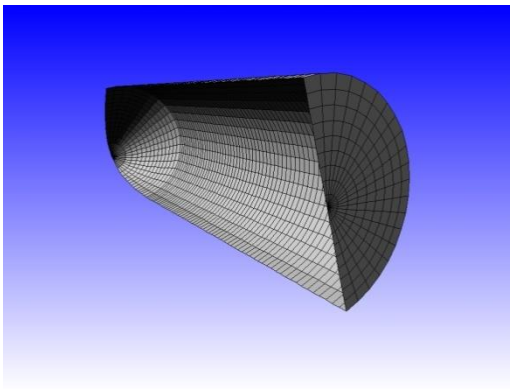
```
# Fuselage
object section Fuselage
length 50
nodes_axial 60
```

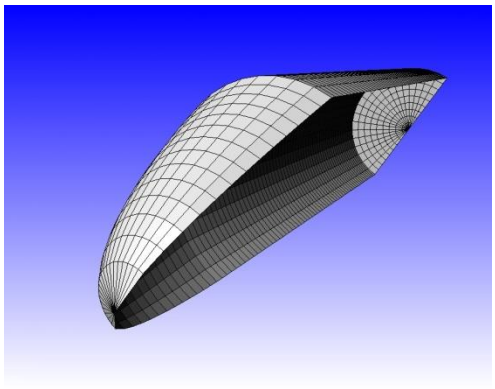


Notice that significantly fewer parameters are needed compared to the nose. Most of the nose shape parameters are now the default for the next object. Only those that change need to be specified.

Next, add a flat bulkhead to show a little bit of internal detail. Note that a bulkhead is created by making a dome object and specifying another taper schedule. A parabolic taper was used for the nose; here a bulkhead taper is used.

```
# Bulkhead
object dome Bulkhead
taper bulk
nodes_axial 10
```

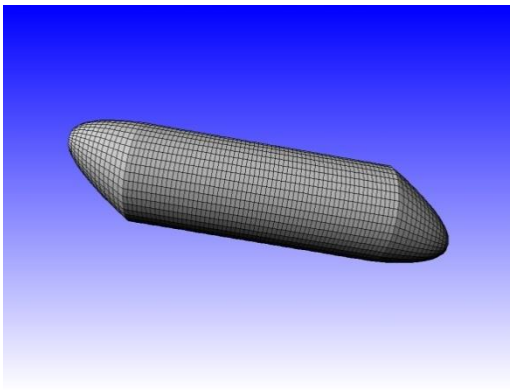




Each new object is automatically positioned behind the previous object: the fuselage is behind the nose, and the bulkhead is behind the fuselage. This makes building sequential structures like this very simple. Manually positioning objects will be covered shortly.

Next, add the rear part of the fuselage. In this case, it will look very much like the nose, but drooping in the opposite direction.

```
# Rear Cap
object dome Rear cap
taper para
length 15.0
zdroop -4.5
nodes_circ 21
nodes_axial 15
```

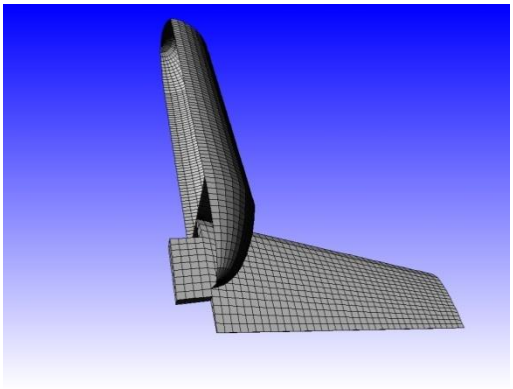


Next, move onto the wing.

```
# Main Wing
object wing Main Wing
span 40
chord 20
taper 0.5
```

```
sweep 20
mesh 1
rootnaca 3412
tipnaca 3410
sparpos 10
sparpos 25
sparpos 75
ribpos 33
ribpos 66
wingbox 5
boxfront 2
```

That's a lot of parameters, but the meaning of most of them should be obvious (refer to `wing` object documentation in chapter 7 if needed). Spars are positioned at 10, 25, and 75 percent of chord and ribs at 33 and 66 percent of the span (ribs are automatically created at 0 and 100 percent). The last two lines ask for *Loft* to create a wingbox carry-through. The default behavior is to extrude the front most and rear most spars to make this box, but the `boxfront` parameter here says to use the second front-most spar instead (thus extruding from the 25 and 75 percent spars, not the 10 percent.) The resulting model looks like this:

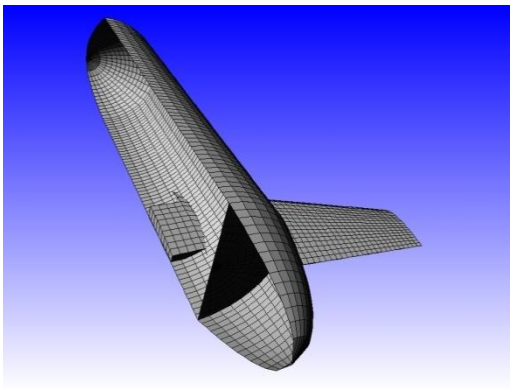


The wing shape is correct, but it's in the wrong place. Why is that? First, dome objects' lengths do not alter the default starting point of the next object. And the origin of a wing object is at its leading edge root. So, the leading edge root point of the wing is sitting at the rear center point of the fuselage section.

There are a couple of ways to move the wing. It is possible to specify the exact position of the leading edge root point with the `transx`, `transy`, and `transz` parameters. There are cases when this is the way to go, but in most cases, the relative translation parameters `relx`, etc. are better. These values are translations relative to the default position. Doing things this way will result in the wing staying in the same spot at the rear of the fuselage even if the fuselage length is later changed.

```
relx 5
rely -9.5
relz -25
```

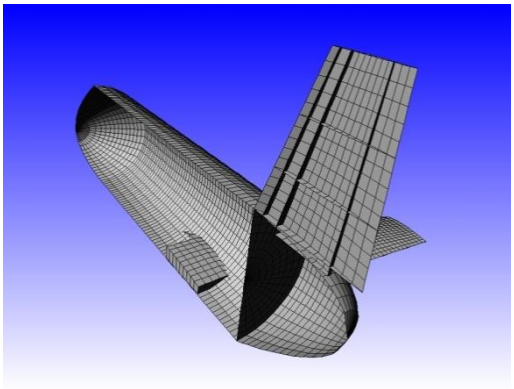
The x translation moves the carry-through to the centerline. The y translation moves the wing down to the bottom of the fuselage, and the z translation moves the wing forward.



Now, add a vertical tail to the top of the rear cap.

```
# Vertical Tail
object wing Vertical Tail
span 18
chord 15
rootnaca 0412
tipnaca 0410
halfwing bottom
wingbox 1
rotz 90
rely 19.5
relz 25
relx -5
```

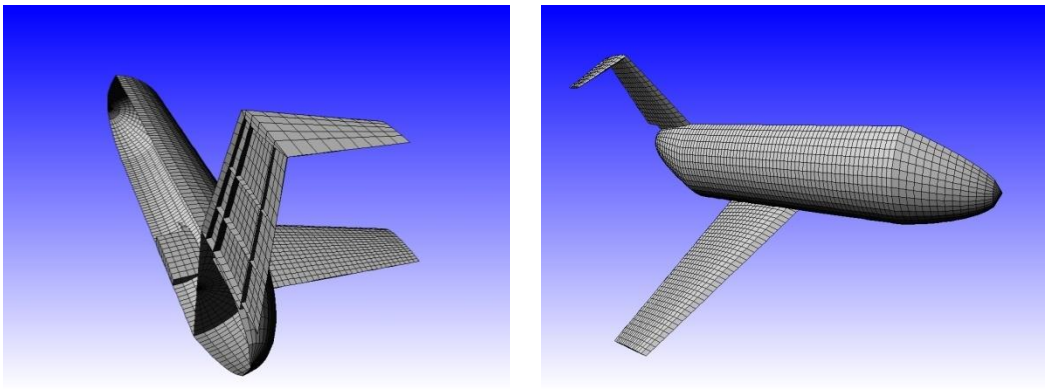
Here symmetric airfoil sections were chosen, and since the tail is on the line of symmetry, only half of it was generated by specifying the `halfwing` parameter. The default position for the tail object is at the leading edge root point of the main wing, so the x translation moves the origin (leading edge root) of the tail back to the centerline, the y translation moves it to the top of the fuselage, and the z translation moves it back to the end of the fuselage section object. The rotation command spins the tail to be vertical. With the `halfwing` option, it's possible to see the internal spars and ribs on the tail, which are in the same



position as on the main wing (since no change was specified). Finally, add a high horizontal tail to the top of the vertical tail:

```
# Horizontal Tail
object wing Horizontal Tail
chord 7.5
span 11.0
rely 18
relz 6.551
rotz 0
```

The `rotz` parameter needs to be reset back to zero, from its new default of 90. Notice, however, that the `halfwing` parameter did not have to be turned off – as seen in the `wing` object definition in chapter 7 it always defaults to `off`. The chord length and y and z translations are chosen to position the horizontal tail aligned with the top of the swept vertical tail.



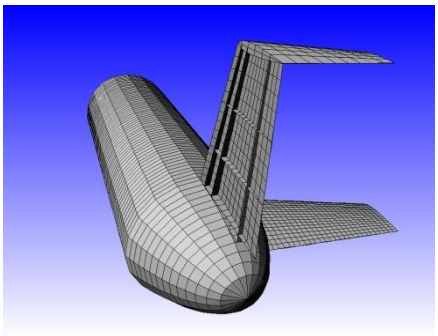
Note that the various wing objects are not actually connected (in a finite element sense) to the fuselage or each other at this stage. Before using this model to perform an analysis, some work should be done with the mesh density on the horizontal tail (to make it match that on the vertical tail), and some ring frames should probably be added where the wing and tail connect to the fuselage to provide stronger attachment points.

Project 2: Converting Project 1 Mesh to a Full Vehicle

There are three different ways to accomplish this task. Each will be demonstrated in this tutorial. The choice as to which option is better depends on the situation. The first approach is to modify a few lines in the input deck to change the half pieces to full ones and to make portside wing surfaces. The second approach is to use *Loft*'s internal clipboard to clone and mirror the half vehicle. The third approach uses a single macro command to perform the mirroring operation. It produces the same mesh as the second approach. The first option is better if only a full model is desired. The second/third options are convenient if both models are needed for different reasons.

Approach 1: Change from half objects to full

Copy project1.txt file to project2a.txt. Open the new file in the editor and move down to the second non-comment line: "`curve1 sc.`" Change the `sc` to `cir`. Running *Loft* on this modified file produces:

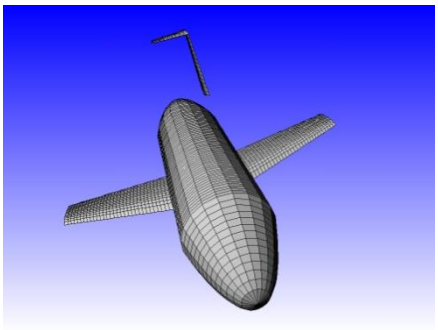


The new full circle curve1 parameter becomes the default for the rest of the fuselage objects by only changing the one line at the beginning of the file. You may also want to double the circumferential node density so that the spacing is the same as before: "`nodes_circ 41.`" Now, fix the wings.

After the Main Wing object (which could be renamed as Starboard Wing), add the following:

```
object wing Port Wing
wingside port
wingbox 5
relx -10
```

This can be short because all of the Main Wing geometric parameters have become the default for any subsequent `wing` object. The `wingbox` parameter, however, always defaults to zero (see the `wing` object documentation in chapter 7) so it needs to be set again. And other than the two parameters specified in the new lines above, that's exactly what is wanted.



Why has the vertical tail moved? This is one of the hazards of using relative position parameters: the vertical tail is now 5 units to the port of the origin of the port wing (leading edge root), rather than the origin of the starboard wing. Instead of changing the tail's "`relx -5`" parameter to "`relx 5`," change it to:

```
transx 0.0
```

Also, delete the tail's `halfwing` parameter. Finally, create a port horizontal tail object by adding these two lines after the starboard horizontal tail object:

```
object wing P Horizontal Tail
wingside port
```

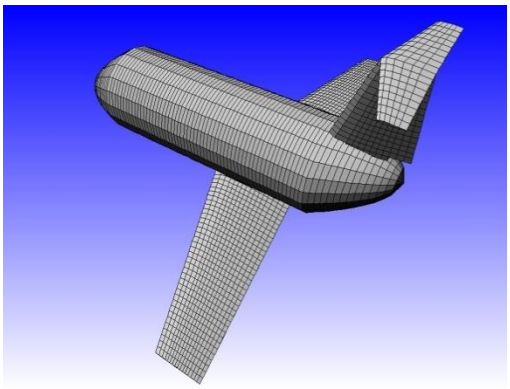
With all of the edits, the final input deck is:

```
# Loft Tutorials: Project 2a
# A Simple Airliner
# Created 4/16/03 by N. Jineer
# The nose
object dome Nose
     curve1 cir
     length -15.0
     c1_xscale 10.0
     c1_yscale 10.0
     taper para
     zdroop 4.0
     nodes_circ 41
     nodes_axial 15
# Fuselage
object section Fuselage
     length 50
     nodes_axial 60
# Bulkhead
object dome Bulkhead
     taper bulk
     nodes_axial 10
# Rear Cap
object dome Rear cap
```

```
     taper para
     length 15.0
     zdroop -4.5
     nodes_circ 21
     nodes_axial 15
# Main Wing
object wing Starboard Wing
     span 40
     chord 20
     taper 0.5
     sweep 20
     mesh 1
     rootnaca 3412
     tipnaca 3410
     sparpos 10
     sparpos 25
     sparpos 75
     ribpos 33
     ribpos 66
     wingbox 5
     boxfront 2
     relx 5
     rely -9.5
     relz -25
object wing Port Wing
     wingside port
     wingbox 5
     relx -10
# Vertical Tail
object wing Vertical Tail
     span 18
     chord 15
     rootnaca 0412
     tipnaca 0410
     wingbox 1
     rotz 90
     rely 19.5
     relz 25
     transx 0.0
# Horizontal Tail
object wing SB Horizontal Tail
     chord 7.5
     span 11.0
     rely 18
     relz 6.551
```

```
     rotz 0
object wing P Horizontal Tail
     wingside port
# Save and exit
write vrml project2a.wrl
end
```

which produces the complete model shown below. As with the half model, stitching of the wing surfaces to each other and the fuselage would be necessary prior to any finite element analysis.

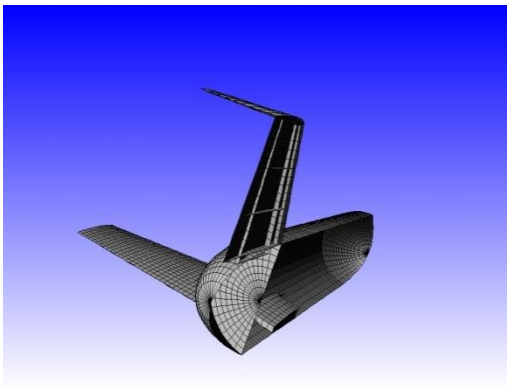


Approach 2: Clone the half model into a full model

This part of the tutorial will create a very similar mesh another way. Start by copying the original project1.txt file to project2b.txt. Open the file and move the cursor down past all the object commands and parameters and before the "`# Save and exit`" line. Add the following lines:

```
# Store the starboard half model
store SB
# Recall and mirror it
recall SB
move
     scalex -1.0
     flip
```

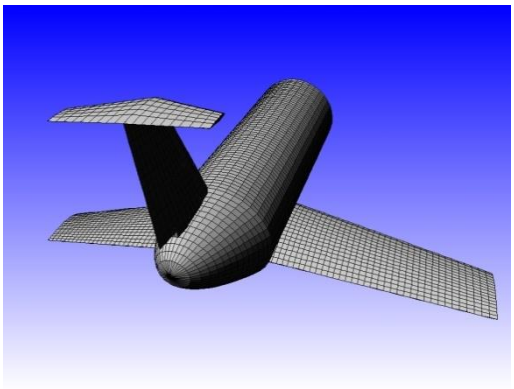
These commands start by moving the half model to the internal clipboard and naming it "SB." The `store` command clears and resets the active workspace. So, the next command `recall`s it back into active memory. The next three lines perform two stack level `move` operations. The "`scalex -1.0`" parameter changes the sign of all nodes' x coordinates. This mirrors the mesh, but also has the undesired effect of causing all the element normal vectors to point inward rather than outward. The `flip` parameter reverses all the normal vectors. At this stage, the model looks exactly like before, but mirrored onto the port side:



Now, to get the original starboard mesh recalled and merged, just add:

```
# Recall it again
recall SB
```

The merge part of the operation, which is performed automatically, can be a little slow, particularly when the same object is being combined. The final mesh looks like:



The meshes produced by these two approaches are in many ways identical. The nodes and the elements are in the same places (the cloned approach may have extra nodes and elements in the vertical tail due to being created as two half wings). The real differences are subtle. If FEA mesh files were created, the differences could be located. In the first case, the two wing and the two horizontal tail meshes each have differently named properties and groups associated with them. With the second approach, the two wings share properties and groups, and the two horizontal tails do as well.

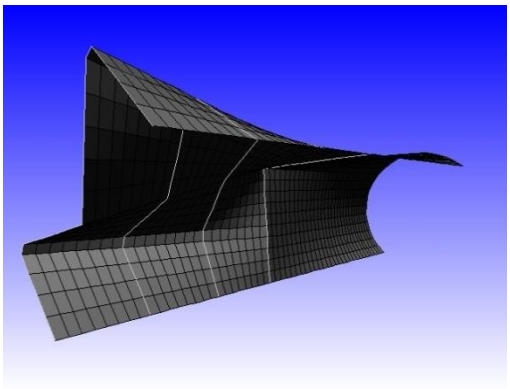
Approach 3: Clone the half model into a full model using a macro command

Adding the single line below to the initial project instead of the 9 lines in the previous example will accomplish the steps in approach 2 with the same resulting mesh.

```
mirror x
```

Project 3: Creating and Using User-defined Curves

Part A: Interpolated Curves

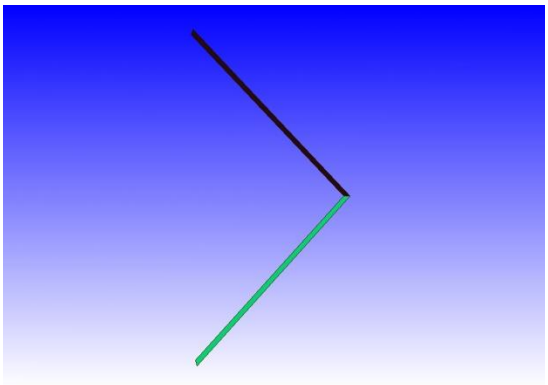


Loft's curve library covers the basic shapes used for many aerospace vehicle components. But, the library can't contain everything. This project explains how to use the interpolated curve definition capability to create user-defined shapes.

Defining an interpolated curve is easy. Just provide a sequential list of nodes that define the corners of the shape. Start at the top of the curve (12 o'clock) and define nodes in a clockwise fashion.

In general, try to define your curve with a nominal radius of 1.0. The user then defines an object's size with the `xscale` and `yscale` parameters. Alternatively, give full-scale coordinates for the curve's definition points and keep the object scale parameters close to 1.0.

The figure above is generated using the built-in semi-circle shape on the right end and two user-defined interpolated curves at the center and left end. The center shape is a half diamond. The cross section looks like:



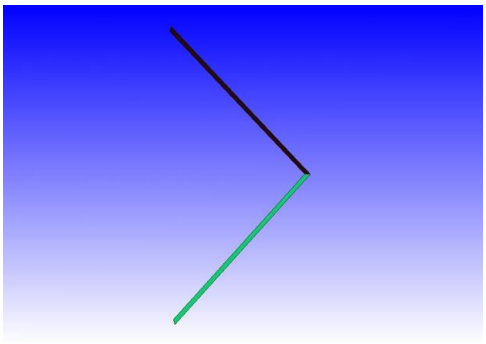
To define this shape to fit in a unit circle, start at the top. The coordinates are x=0, y=1. The midpoint of the shape is at x=1, y=0, and the bottom point is at x=0, y=-1. The command and parameters to specify these coordinates as a *Loft* interpolated curve named "sd" are:

```
# half diamond shape
curve interpolated sd
  start 0.0 1.0
  line 1.0 0.0
  line 0.0 -1.0
```

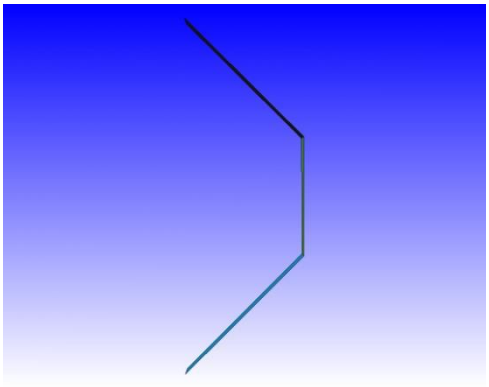
Once defined, the "sd" mnemonic can be used in any subsequent objects as if it were a curve in the library.

The user should keep in mind that due to the sampling scheme used by *Loft* to distribute nodes, the points given when defining the shape may or may not appear exactly in the final meshed objects that use the curve. When the user has finished defining a curve, *Loft* will compute the lengths of each segment and the total length of the curve. Then, when the curve is used it will evenly distribute the meshed points along the total length of the curve.

For example, if the user specifies the above "sd" curve and has a `nodes_circ` parameter of three, *Loft* will generate nodes at 0, 50, and 100 percent along the curve, and by coincidence, create the exact inputted shape:



But, if the user instead had a `nodes_circ` parameter of four, *Loft* would generate nodes at 0, 33, 66, and 100 percent along the curve, giving a cross sectional shape that looks like:

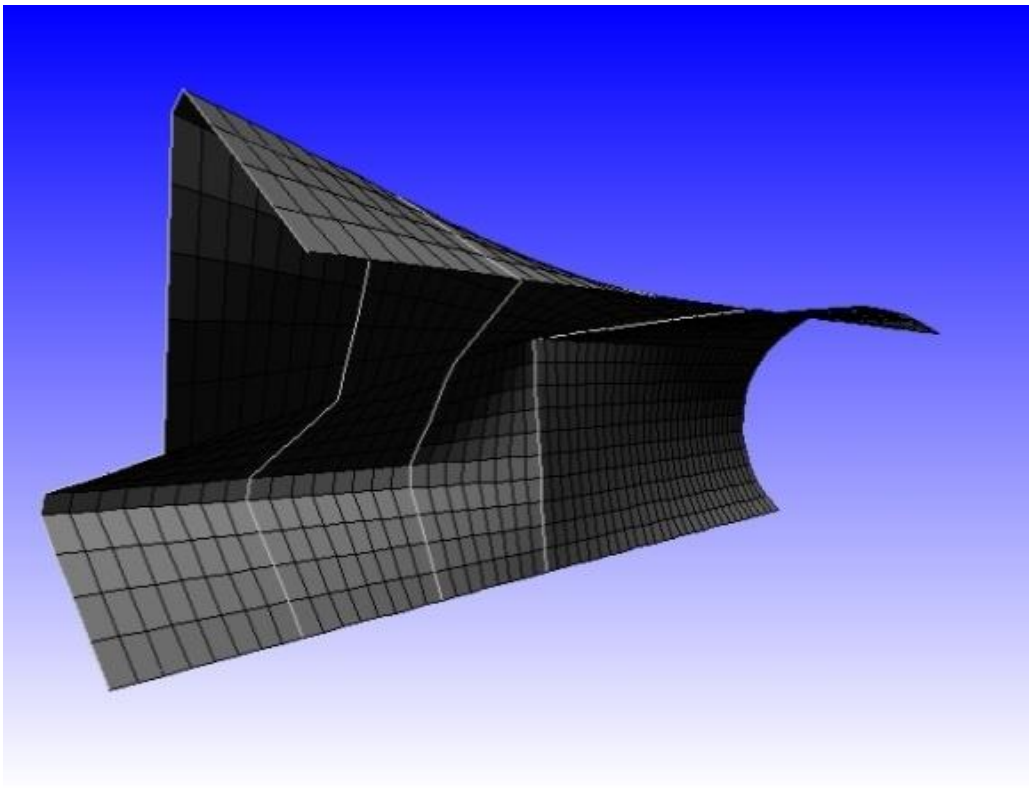


By the way, *Loft* will show this same corner-rounding behavior when using library curves and the other types of user-defined curves. The user may need to experiment with the number of nodes specified if hitting the corners exactly is important. See project 5 for some additional ways to address this issue.

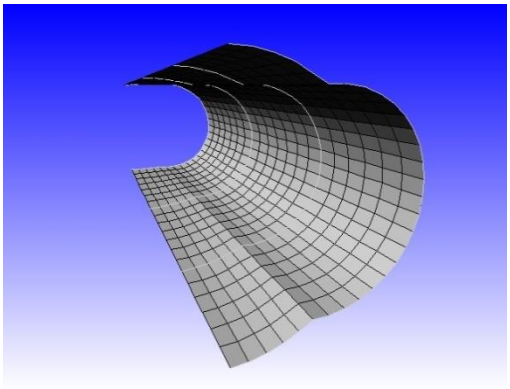
To finish this project, define a second interpolated curve (the M-shaped left side of the original figure) and then use both curves:

```
curve interpolated toothout
  start 0.0 1.0
  line 1.0 1.0
  line 0.25 0.0
  line 1.0 -1.0
  line 0.0 -1.0
object section Barrel
  curve1 sc
  curve2 sd
  c1_xscale 15.589
  c1_yscale 15.589
  c2_xscale 15.589
  c2_yscale 15.589
  nodes_circ 21
  length 50
  nodes_axial 30
  components_axial 6
object section Barrel2
  curve2 toothout
  length 40
  nodes_axial 25
object frame Ring Frames
# save
write vrml project3a.wrl
end
```

The complete file specifies two user-defined curves and then builds two sections. The first section blends a semi-circle to the user's semi-diamond shape. The second section blends the semi-diamond to the letter "M" shaped "toothout" curve. Note that in the finished mesh the corner of the "sd" curve is sampled exactly, as is the middle corner of the "toothout," but the two intermediate corners are slightly rounded. Finally, a frame object is added to the second section. The white lines in the figure show the circumferential beam elements that make up the frame. They align precisely with the section mesh at each station.



Part B: User-defined Compound Curves



A more powerful option for user-defined curves is the compound curve. As the name implies, compound curves are combinations of previously defined curves. In fact, *any* previously defined curve can be used as a `child` curve to build up a more complex parent compound curve. Any library curve, as well as any previously defined interpolated, compound, or lofted (see project 3C below) curve, can be used.

Loft is currently unable to compute the intersections of two arbitrary curves, so the user must tell the code where to stop using one child curve and where to start using the next. *Loft can* locate the intersection points of circles and semi-circles with other circles or semi-circles. However, any other curve combination will need user intervention to specify intersection locations.

The Compound Curve Concept

To picture the basic idea of a compound curve, imagine a sheet of rolled dough and a handful of interestingly shaped cookie cutters. Imagine selecting a cutter and making an impression in the dough with it but not removing the cookie. Then, select another (or perhaps the same) cutter and make another impression that intersects the first. Continue this process as long as desired. Now imagine using a finger to re-blend all of the internal lines leaving only the outer-most indention. This could produce a very strange shape. That's basically what the compound curve type allows one to do.

The "s" Parameter

Internally, *Loft*'s curves are generated based on fractional location along their perimeter. This perimeter coordinate is called "s" and varies between zero and one. If the user generates a barrel object with three nodes in the circumferential direction, *Loft* will generate nodes at s = 0.0, s = 0.5, and s = 1.0 on each curve and linearly connect them.

The library curve subroutines' only function is to accept an "s" value as input and to return the two-dimensional coordinates of the point at that fraction along the curve. All library curves are defined with s = 0 at the 12 o'clock position, and s increasing as one moves clockwise around the curve to s = 1 at its end.

This is the semi-circle subroutine:

```
angle = (90.0 - 180.0 * s) * pi / 180.0;
x = cos(angle);
```

```
y = sin(angle);
```

The full circle routine uses instead:

```
angle = (90.0 - 360.0 * s) * pi / 180.0;
```

Looking at these two code snippets confirms that s = 0 generates the (x, y) coordinates of a node on the curve at 12 o'clock and a nominal radius of 1.0. Any other value for s generates the coordinates for that fractional location along the curve.

Of Parents, Children, and Arcs

Return to the dough and cookie cutter metaphor above. Each time a cookie cutter was used a `child` curve was created. Now picture the outer-most "parent" boundary line. Each portion of that curve contributed by a new `child` is called an "arc."

The task when defining a compound curve is to sequentially specify the `child` curves necessary to generate each arc of the final curve. In many cases, a particular child will be specified more than once since it may contribute to more than one section of the parent curve.

For each `child` curve, specify the mnemonic for the `child` curve, its center coordinates, and its radius. The next step is to specify what portion of the `child` will contribute to the parent curve. This is done with the `sstart` and `sstop` parameters. These are the "s" coordinates of the `child` curve that mark the endpoints of the arc being specified. Optionally, *Loft* can automatically compute these parameters when two circle or semi-circle children intersect.

For proper extruding and connection of panels, the final compound curve should start on the horizontal centerline at the 12 o'clock position and trace clockwise around to the end of the curve. Typically, the end will be either at 6 o'clock or back at 12 o'clock. Put some planning into the radius values used for the `child` curves. Ideally, the resulting parent curve should have a nominal unit radius. This will make later use of the compound curve and selection of x and y scale values consistent with the scale values used with the library curves. Alternatively, the compound curve can be specified with actual dimensions. In such a case, the x and y scale values for objects using those curves will be near unity. Just keep in mind that the radii and center points specified when defining the curves will be scaled later by the meshing routines.

How Loft Uses a Compound Curve

Once a compound curve has been defined, *Loft* calculates the circumference of each arc (by a piecewise-linear approximation for non-circular arcs) and sums them to compute the total circumference for the compound curve. Each child's contribution to the total circumference is used to determine what range of the parent's "s" coordinate for which it is responsible. When the compound curve code is asked for an (x,y) coordinate based on a particular "s," *Loft* will figure out which child is responsible for that location and where on that child's arc the point is. This information is used to compute a "local s" parameter for the child curve. The coordinates returned by the child are scaled and translated to generate the coordinate of that spot on the parent curve.

A Compound Curve Example

The first example project is a half-model of a three-lobe tank cross section. Looking at the picture above imagine making the shape by combining a semi-circle on the left with a full circle on the right.

Start with the user specified curve command, specify `compound` as the type of user curve, and supply a curve name:

```
curve compound half3lobe
```

From the picture above, there are three "arcs" that make up the full compound `curve`. So, three `child` blocks must be specified to define the curve. In this case the first and the last arc are made from the same child, but this is not necessarily always the case. For this first project the semi-circle and circle library curves are used. Since they are circular shapes, *Loft* can compute the intersection points rather than requiring the user to specify the endpoints of each arc with the `sstop` and `sstart` parameters.

So, the first child is a semi-circle centered at (0,0) with a radius of 5:

```
child sc
x 0.0
y 0.0
radius 5.0
```

Then, the next arc uses the full circle library curve:

```
child cir
x 3.5
y 0.0
radius 4.0
```

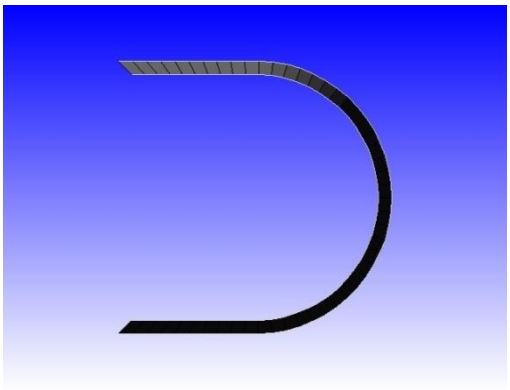
The last arc is part of the first curve, so that block is copied here:

```
child sc
x 0.0
y 0.0
radius 5.0
```

Finally, to generate the picture above, create a very short section object using the new compound curve

```
object section Barrel
  curve1 half3lobe
  curve2 half3lobe
  length 1
  nodes_circ 51
  nodes_axial 2
# save
write vrml project3b1.wrl
end
```

The next step is to generate a different compound curve. This time, using a half square and a circle to generate a shape like this:



First, start a new compound curve:

```
curve compound roundbox
```

The mnemonic for a half (or semi) square is `ss`. The compound curve parameter `radius` can be used for any child curve to scale it up from the default nominal unit radius. The two corners of the square occur

at 25 and 75 percent along the curve. For the first arc only the top edge of the curve is needed, so the arc goes from s = 0.0 to s = 0.25. Since `sstart` = 0.0 is the default, it does not have to be specified.

```
child ss
x 0.0
y 0.0
radius 3.0
sstop 0.25
```

Next, a full circle is specified with the same radius and an `sstop` parameter of 0.5:

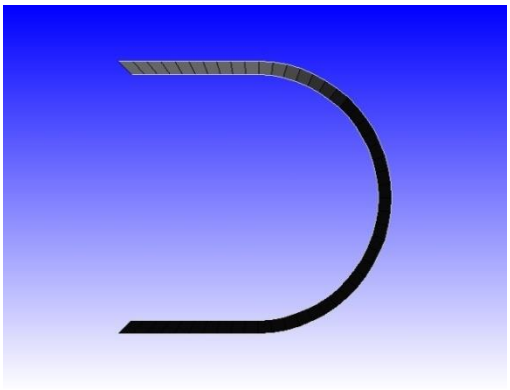
```
child cir
x 3.0
y 0.0
radius 3.0
sstop 0.5
```

(Yes, a semi-circle could have been used here with no `sstop` parameter necessary.) Finally, to specify the bottom flat arc, return to the semi-square and specify portion between s = 0.75 and 1.0.

```
child ss
x 0.0
y 0.0
radius 3.0
sstart 0.75
```

To generate a sample representation of the new compound curve just add:

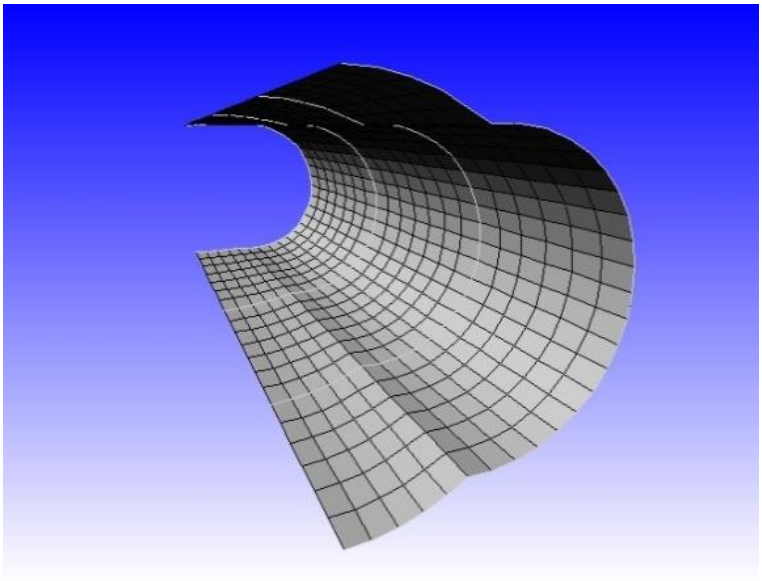
```
object section Barrel
     curve1 roundbox
     curve2 roundbox
     length 1
     nodes_circ 51
     nodes_axial 2
object frame Ring Frames
# save
write vrml project3b2.wrl
end
```



Finally, create the picture at the top of this tutorial by combining the two compound curves in a file that contains the two curve specifications. The ring frame object is optional but demonstrates that beams can be created that will follow the interpolated shape between the two user-defined compound curves (they are the white lines at either end and the center in the figure).

```
object section Barrel2
     curve1 roundbox
     curve2 half3lobe
     c2_xscale 1.0
     c2_yscale 1.0
     length 20
     nodes_axial 21
     nodes_circ 31
     components_axial 3
object frame Ring Frames
# save
write vrml project3b3.wrl
```

```
end
```

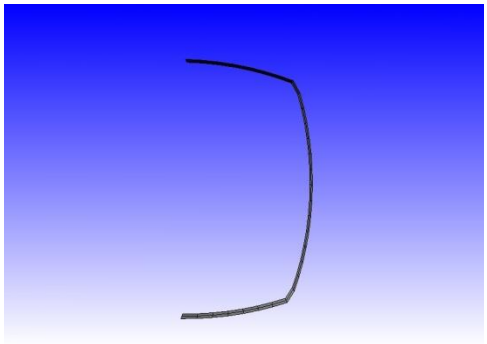


Part C: User-defined Lofted curves

The third type of user-defined curve is the "lofted" curve. *Loft* generates, but does not save, curves automatically when building a section object. At each station along the section object the program computes the intermediate cross section as it transitions from the `curve1` end to the `curve2` end. The lofted curve type allows the user to do the same thing, with or without actually creating a corresponding section object. Another way to look at these curves is that they create a cross-sectional slice shape from a (possibly virtual) section.

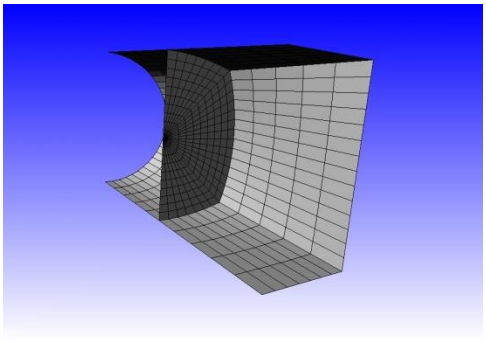
To create a lofted curve, the user specifies the curves at that are to be blended to form the new cross section. As with the compound curve, any type of curve including user-defined curves can be used as the end shapes. The user then specifies the fractional position along the transition from `curve1` to `curve2` with the `station` parameter. A `station` value of 0.0 would result in a curve exactly matching `curve1`. A value of 1.0 would match `curve2`. The example below uses 0.5, which is 50% along the transition from 1 to 2 and results in the cross section shown.

```
curve lofted lcurve1
     curve1 sc
     curve2 ss
     station 0.5
object section test-section
     curve1 lcurve1
     curve2 lcurve1
     length 0.1
     nodes_axial 3
     nodes_circ 30
write vrml project3c1.wrl
end
```



One use of this curve type is to generate mid-section bulkheads:

```
# test of mid-section bulkheads
curve lofted lcurve1
     curve1 sc
     curve2 ss
     station 0.5
object section test-section
     curve1 sc
     curve2 ss
     length 4.
     nodes_axial 11
     nodes_circ 29
object dome bulkhead
     taper bulk
     curve1 lcurve1
     relz -2
write vrml project3c2.wrl
end
```



Care should be taken if node-stitching is desired to make sure that the bulkhead is positioned at a spot on the section object with nodes. In the above example, an odd number of nodes was used axially to ensure that a node line existed at the 50% axial station on the section. The lofted curve was defined as a 50% blend of the two end curves. And the created bulkhead, which by default would have been positioned at the rear (square end) of the section, had a `relz` of –2 applied to position it at the midpoint of a 4 unit long section.

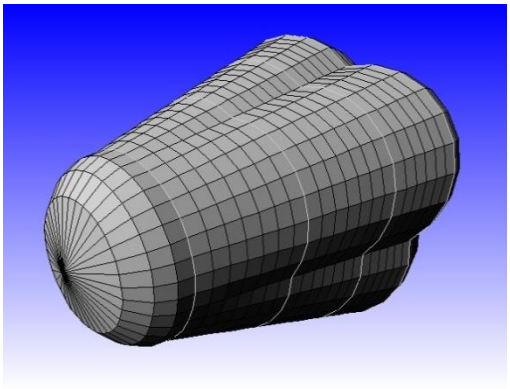
If the desired position of the bulkhead is not at an easy-to-align position (e.g., 46.4543% of the section length), then the best approach will be to create the lofted curve and use it to create a forward section (`curve1` to the bulkhead), the bulkhead, and the aft section (bulkhead to `curve2`) as three objects rather than two. This approach allows for easy and exact positioning and node-stitching at completely arbitrary axial stations. The following input file generates the same result as before, but creates three objects:

```
curve lofted lcurve1
     curve1 sc
     curve2 ss
     station 0.5
object section forward
     curve1 sc
     curve2 lcurve1
     length 2.
     nodes_axial 6
     nodes_circ 29
object dome bulkhead
     taper bulk
object section aft
```

```
      curve2 ss
      length 2.
      nodes_axial 6
write vrml project3c3.wrl
end
```

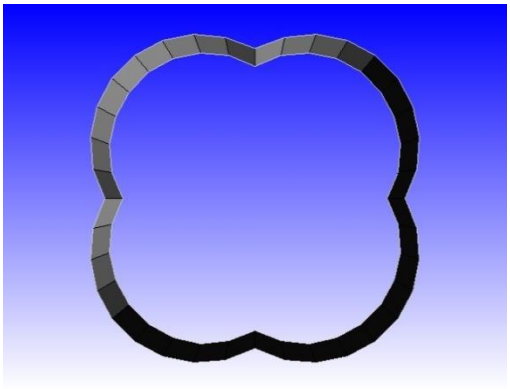
A very similar approach can be used to create a bulkhead that supports an internal structure such as a tank. The bulkhead would be constructed using a zero-length section object with one end curve defined as a lofted curve extracted from the desired position along the fuselage section and the other end as a lofted curve extracted from the tank object. See the annotated TSTO orbiter example at the end of this manual for a demonstration of this process.

Project 4: A Tapered Four-Lobe Tank



This project represents a tank that might be used in a vehicle nose cone if very tight packaging were necessary.

The first step to building this tank is to define our compound four-lobe curve.



Remember, our task is to define this curve in a clockwise fashion starting at 12 o'clock. Thus, we start with the upper-right circle:

```
curve compound 4lobe
child cir
x 1.0
y 1.0
radius 2.0
```

The default for any child curve is to start at s = 0. This is not what we need here. Some trigonometry will show that the 12 o'clock point is at (0.0, 1.732). This corresponds to 30° counter-clockwise from vertical, or 330° clockwise. Using the full circle formula from project 4, we get:

```
sstart 0.916666667
```

We don't need to specify `sstop` since *Loft* can automatically calculate it for the intersection of two circles. So, we can just specify our remaining three lobes:

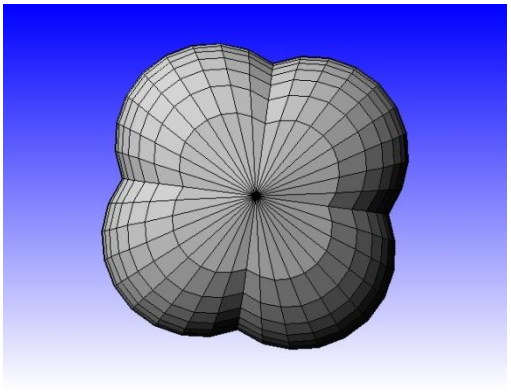
```
child cir
x 1.0
y -1.0
radius 2.0
child cir
x -1.0
y -1.0
radius 2.0
child cir
x -1.0
y 1.0
radius 2.0
```

Since we're not specifying any further child curves, we again need to do some math to find that the point (0, 1.732) is 30° clockwise from curve four's start, resulting in:

```
sstop 0.083333333
```

To generate the rest of the pictured tank you can add:

```
object dome front
     curve1 cir
     c1_xscale 1.5
     c1_yscale 1.5
     nodes_circ 37
     length -1
     nodes_axial 5
object section Barrel
     curve2 4lobe
     c2_xscale 1.0
     c2_yscale 1.0
     length 5
     nodes_axial 21
     components_axial 3
object frame Ring Frames
     object dome back
     length 3
     nodes_axial 13
# save
write vrml project4.wrl
end
```



Project 5: Controlling Circumferential Node Distribution

By default, *Loft* distributes nodes spaced evenly along a curve's circumference (with a couple of minor exceptions – see the breadbox and filleted square curves descriptions in the chapter 7). This is the best general approach for producing a smooth finite element mesh, but it may fail to capture details in some cases. This "sampling error" was discussed briefly in tutorial project 3 on creating interpolated curves.

This project discusses several advanced approaches to addressing problems with the circumferential node distribution. Some are rather involved.

Approach 1: Change the Node Count

By far the easiest technique to address a sampling problem is to change the value of the `nodes_circ` parameter. Generally, increasing this value will do a better job of accurately capturing any particular curve's shape.

But, if you have insight into where a particular feature occurs along a curve, choosing a value of this parameter that places a node that percentage along the shape can also improve the modeling of that feature. This may mean decreasing the `nodes_circ` value. The interpolated curve tutorial showed an example where a value of 3 did a better job of catching a sharp point than a value of 4.

The annotated TSTO orbiter example demonstrates a case in which ensuring a node is placed at 40% of the circumference of a curve is necessary for successful node merging. This leads to a requirement to use a multiple of five plus one as the circumferential node count. A count of 6 will generate nodes at 0, 20, **40**, 60, 80, and 100%. A count of 11 will generate nodes at 0, 10, 20, 30, **40**, 50, 60, 70, 80, 90, and 100%, etc. The "plus one" part of this formula is necessary because of the node at 0%.

Approach 2: Local s-distribution

A relatively easy way to address sampling problems with user-defined curves is to switch to local rather than global s-distribution. Each child-arc of a user-defined curve contributes some fraction of the total circumference of the parent curve. That fraction of the total nodes in the circumferential direction will be used to sample that curve. In the default global s-distribution approach, the nodes are spaced evenly along the parent curve.

The local s-distribution option moves the nodes that model each child-arc to be evenly spaced along the child-arc. This has the effect of forcing a node to be generated at most junctions between child-arcs. If a child-arc is too short to qualify for a node in the global approach, it won't get one in the local approach either. If the detail from that short child-arc is important, the user will need to resort to one of the other approaches in this section to capture that detail.

The s-distribution approach is controlled by the parameters `c1_s` and `c2_s`. Thus, you can use different approaches for each end of a section object. Valid values for the parameter are `global` (the default), `local`, and `copy`.

The `copy` option indicates that the curve is to use the same s-distribution as used for the other end of the section. This can produce less twisted elements if the local distribution on the other end of the section

has significantly moved nodes. The use of the copy option only has practical effect if the other end is set to `local`. (If both ends are set to `copy`, the global approach will be used on both ends).

Like all circumferential parameters, the settings of these two parameters are used to change the defaults for all subsequent objects. Be sure to reset their values when they are no longer needed. Be careful using these parameters when adjacent objects are expected to stitch together. Nodes that have different spacing are unlikely to be merged accurately. The `copy` option is particularly likely to create these kinds of problems, as it may copy its s-distribution from a completely different curve than the adjacent object.

Approach 3: Sub-Curves

A rather involved approach that gives much more control is to create a user-defined curve, then use *Loft*'s debug output to break the curve back into "sub-curves" that are used to generate partial objects. This is a lot more work but allows the user to specify exactly how many nodes are to be used to represent each child-arc of the original parent.

If you look at the program debug output that is generated when using the "roundbox" compound curve created in the previous tutorial, you'll see this summary of the calculations that *Loft* made to use the curve. For each child-arc, the output lists its circumference, the local "s" start and end points of the arc, and the global "s" start and stop points:

```
finish_ccurve: Summary of Compound Curve roundbox
    child   circ   local_sstart local_sstop   global_sstart global_sstop
       0 1.000000    0.000000     0.250000      0.000000      0.194305
       1 3.141560    0.000000     0.500000      0.194305      0.804724
       2 1.005000    0.750000     1.000000      0.804724      1.000000
    End of Summary for Compound curve roundbox
```

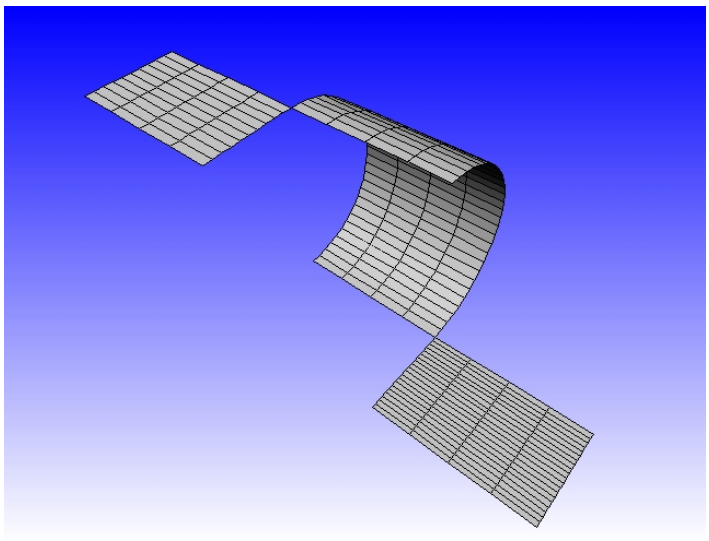
The global "s" start and stop points indicate what portions of the parent curve are contributed by each child. We can use those values to extract just those contributions into new compound curves:

```
curve compound rb-arc1
child roundbox
sstart 0.0
sstop 0.194305
curve compound rb-arc2
child roundbox
sstart 0.194305
sstop 0.804724
curve compound rb-arc3
child roundbox
sstart 0.804724
sstop 1.0
```

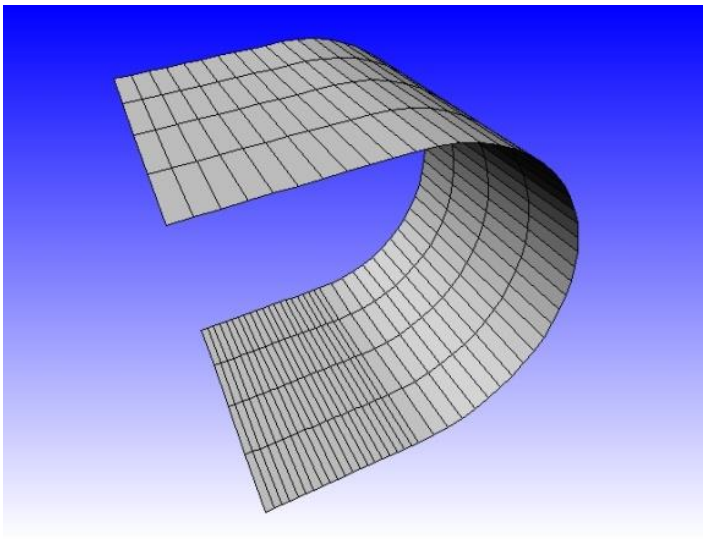
(Remember that the "roundbox" curve definition needs to be copied into this new input file – user-defined curves are not added to *Loft*'s internal library permanently.)

Now, each of these new sub-curves can be used to create partial objects with much more control over node density on each arc. Here's an example creating an extruded "roundbox" object with varying mesh densities.

```
object section arc1
     curve1 rb-arc1
     curve2 rb-arc1
     length 5
     nodes_circ 11
     nodes_axial 5
object section arc2
     curve1 rb-arc2
     curve2 rb-arc2
     nodes_circ 31
object section arc3
     curve1 rb-arc3
     curve2 rb-arc3
     nodes_circ 21
```



This figure shows the three new curves separately. The bottom section does have twice the mesh density of the other two sections, and nodes are created exactly at the junction points of the arcs. But, the automatic positioning in *Loft* is putting each new section object immediately behind the previous one. To fix that, add a "`relz -5`" parameter to both "`arc2`" and "`arc3.`" Notice that no positioning is needed in the x or y directions, since the new curves are already positioned correctly in x and y. Once that is done, the result is:



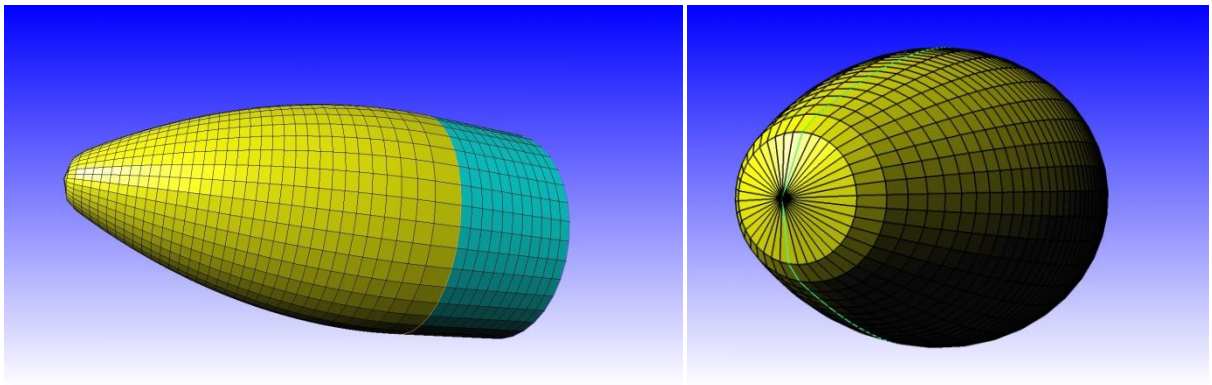
This sub-curve technique gives the user a lot of additional control on mesh density and locating important nodes, but it is a lot more effort than the other approaches. The main drawback in this approach is the difficulty in obtaining compatibility with meshes generated without sub-curves. Generally, objects generated from sub-curves can only be effectively attached to other sub-curve-based objects without a lot of additional work.

Finally, note that if the goal of this sub-curve project was only to double the mesh-density on the bottom plate of the curve, the same result could have been accomplished with just two sub-curves. The first would be the top plate and round section (from s = 0.0 to 0.804724) and the second would be the bottom plate. The sub-curve approach can be used to grab any portion of another curve.

Project 6: Introduction to Regions

The *Loft* command `region` contains a powerful set of tools to allow the user to query, modify, and/or mark portions of the current stack. This tutorial illustrates a small portion of these capabilities. Chapter 3 of this manual documents the full set of region definition and operation parameters.

Start with an ogive-shaped nose cone with a short barrel. The colors on the picture indicate the two property sets used in the model. Also note the beams running the length of the model that represent the separation joint for the shroud.

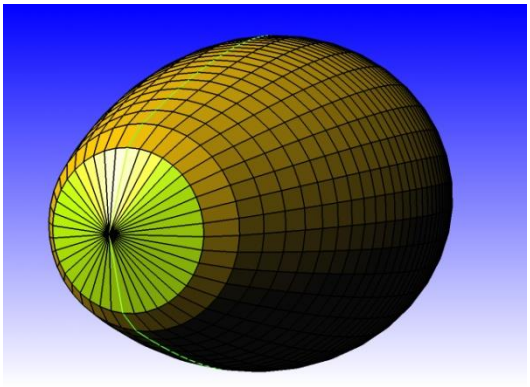


```
object dome Nose
     curve1 cir
     c1_xscale 1.0
     c1_yscale 1.0
     length -550.000
     nodes_circ 41
     nodes_axial 35
     components_circ 1
     components_axial 1
     taper ogive
     param1 55.
     param2 983.230
     param3 198.0
     zdist 0.73
     transz 618.0
object dframe Sep Joints
     count 3
     align axial
#
object section Barrel
     length 200.0
     c1_xscale 198.0
     c1_yscale 198.0
     c2_xscale 196.0
```

```
      c2_yscale 198.0
      nodes_axial 12
      components_axial 1
object frame Bottom Ring
      count 1
      position 1.0
object frame Top Ring
      count 1
      position 0.0
object frame Sep Joints
      count 3
      align axial
# rotate so that x is aft
move
roty 90
```

Next, use the region mode to specify a volume and change the element property settings within that volume. Here, the goal is to make the elements on the very tip of the nose into a different component for later sizing purposes:

```
# Nose Cap
region
      iadd xcyl 0.0 0.0 0.0 30.
      pprem Nose Sep
      setpp Nose Cap
```



There are two parts of defining this region. The *inclusive add* parameter `iadd` adds all elements that have any nodes within the specified cylindrical area. In this case, the beam elements that represent the separation joint should not be updated. So, the *remove by physical property name* parameter `pprem` is used to delete those elements from the region specification (but not from the stack!). Finally, the remaining elements are changed to a new physical property name using the `setpp` parameter.

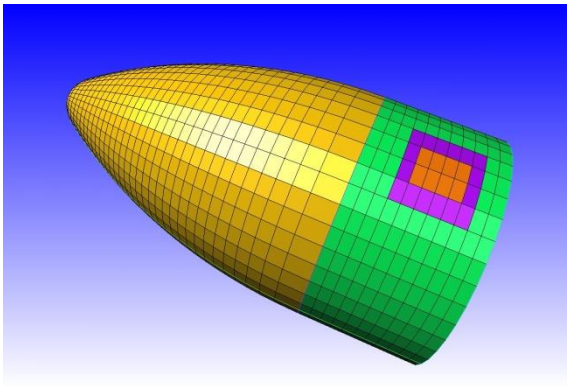
The first two parameters are “passive” parameters. They have not changed the stored stack data in any way. The last parameter, `setpp`, changed the stored stack data. This is an example of an “active” region

parameter. Any number of passive parameters may be performed to set up and query a region. But for the sake of clarity, only one active parameter is allowed per region definition.

The next step is to stencil out a door on one side of the barrel. This is very similar to the previous example. However, we'll go one step further and specify a doorframe of panel elements around the door itself. This requires two region parameters to perform the two active operations.

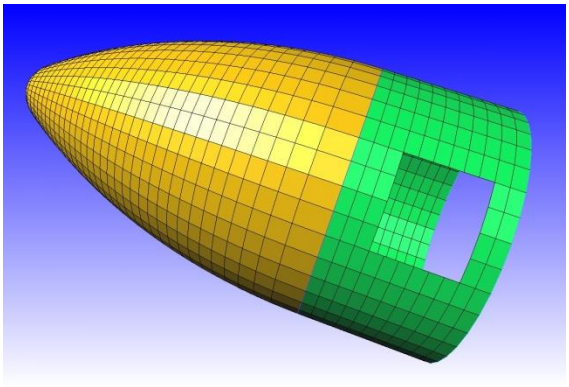
```
# Cut out a door with frame border
region
     iadd box 732. 0. 198. 85. 72. 120
     setpp Large Door Frame
region
     eadd box 732. 0. 198. 85. 72. 120
     setpp Large Door
```

Note that the two add parameters use exactly the same coordinates and dimensions. The difference is that the second operation uses the *exclusive add parameter* `eadd` rather than the *inclusive add parameter* `iadd`. The `eadd` parameter requires that all nodes for an element fall in the specified volume while the `iadd` parameter requires only one node to be in the volume. This difference makes building these border frames easy. Note that it is possible for the volume to exactly intersect a line of nodes and produce identical results along an edge for the two parameters.



The region command can also be used to produce partial models. The following code creates an output file that does not contain the door or door frame:

```
#
region
     ppadd Large Door Frame
     ppadd Large Door
     inverse
rwrite vrml project6a.wrl
```



The additional input lines add the door and frame to the region, then invert the region membership. Finally, an output file containing just the elements in the region is written. These elements will have the same indices and properties as they do in the full model. Thus, this approach can be used to generate models for tasks such as mapping aerodynamic loads to the exterior elements of a model. The resulting load data can then be applied to the full model (with interior elements) with no element renumbering required.

For a more complex model with many more objects, the object level `mark` command can be used to arbitrarily apply labels to each component such as “OML” or “LH2.” Objects can have any number of marks. Then the region-mode parameterss `mkadd` and `mkrem` can be used to add/remove groups of components by these labels.

Project 7: Variables and Math

Loft supports two types of variables: "user-defined" and "system." This capability greatly expands the parametric power of the program by allowing critical dimensions or values to be set once and then used repeatedly. If a requirement changes, only that single value must be updated. The basic math support in the *Loft* input file reader adds even more flexibility. Named variables can also significantly improve input file clarity and reduce the chance of errors.

Input Line Math

Loft supports simple math operations on an input line. These operations are addition, subtraction, multiplication, and division. The corresponding operation symbols are the normal "+," "-," "*," and "/." A space must be used on either side of the operation symbol. Any number of operations can be performed on a line. All math calculations are performed left to right, **with no preference given to multiplication or division**. Parentheses are not supported. Multiple variables defined on multiple lines can be used to perform separate parts of a complex computation where order must be controlled.

Since computation of math operations is performed left to right, the expression "50 + 10 * 3" evaluates sequentially as:

50 + 10 * 3 = 60 * 3 = 180

User-defined variables

A variable can be defined in a *Loft* input file by using the `define` command. Any desired name (with no spaces) can be used for the variable name. To reference a user variable, the dollar symbol, "$," is placed before the variable name. These variables can be used in any *Loft* input command or parameter as needed.

Here are some examples:

```
define var1 50.0
define var2 10.0
define var3 $var1 + $var2 * 3.0
define var1 40.0
```

The user variable `var3` is computed using the previously defined `var1` and `var2` variables. It has the value of 180.0 (see discussion of input line math above). The last example redefines `var1`. Any later references to that variable will use the new value.

System Variables

System variables are the collection of *Loft's* current default values for object parameters. These values are continuously updated as the user specifies parameters. Thus, there is no `define` command, per se, to set these values. Rather, they are set through the normal use of *Loft*.

System variables are referred to by a specific name (see a chart of all available variables chapter 7 of this manual). To reference a system variable an "at" symbol, "@," is placed before the variable name.

Examples:

```
object wing demo
span 10.0
chord @wing.span / 2.0
```

Math Functions

Loft supports some standard math functions including trigonometry, roots, etc. See the math function chart later in chapter 7 of this manual for a full list of supported functions. Math functions are called by using the percent symbol and the function mnemonic. They must be placed at the end of a line after any variable or arithmetic. Multiple functions can be used on a single line. Each function will be applied to the preceding number in the order read. Note that the `@pi` system variable could also have been used rather than being defined as a user variable.

Examples:

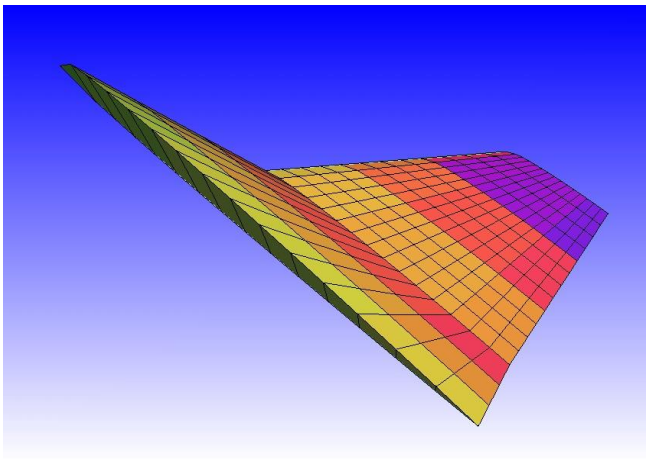
```
define pi 3.14159265359
define four 4.
define two $four %sqrt
define zero $pi %sin
define negone $pi %cos
define zeroagain $pi * $pi %sqrt %sin
```

Logical Operations

Loft supports six logical operations ">," ">=," "<," "<=," "=," and "!=" (greater than, greater than or equal, less than, less than or equal, equal, and not equal). When they are encountered, the values will be compared and a 1.0 will be returned if the equality/inequality is true and a 0.0 will be returned if it is false. Remember the left to right sequential operation of the math preprocessor. If complex comparisons are desired, use multiple lines and variables to construct the desired result. These operations are most commonly used with the `if` flow control command.

Example: A Compound Wing

Loft supports only trapezoidal wing planforms. More complex shapes can be built up from multiple trapezoids and the math and variables capability of *Loft* can be used to make this assembly easier. For this example, we'll construct a swept wing with a large root strake.



Math is used first to calculate the strake's taper ratio directly from the root and tip chords rather than requiring the file creator to do the calculation. Then, the chordwise mesh density of the outboard section is computed using the system variables that contain the outboard section's root chord and the strake's taper ratio.

```
object wing strake
  chord 900.
  span 80.
# Use math to calculate tip/root = 0.48
  taper 432. / 900.
  sweep 80.0
  rootnaca 2212
  tipnaca 2208
  sparpos reset
  sparpos 10.
  sparpos 36.
  sparpos 80.
  ribpos reset
  ribpos 33.
  ribpos 66.
  notip 1
  meshchord 0.02
  meshspan 0.06
  meshthick 0.02
#
object wing mainwing
  chord 432.
  span 251.
# to match strake, divide its mesh value by its taper ratio = 0.0416
  meshchord @wing.mesh_chord / @wing.taper
  taper 0.37037
  sweep 45.0
  naca 2208
```

```
relx 80.
relz 453.70255
```

An Important Caveat

The math and variable support described in this project is implemented as a preprocessor that immediately replaces all the variables with their corresponding values and performs all the requested calculations before handing the now conventional input line to the main *Loft* user interface. Objects are only actually created when a new command is read and *Loft* determines that the user is therefore done with specifying parameters for that object. Finally, the positioning system variables (`@transx`, etc.) are only updated after an object has been created and merged into the current stack.

The combination of these three factors can lead to some confusion. Consider the following code example, which will result in different values assigned to the two user variables `var1` and `var2`.

```
object section fuselage
   length 10
define var1 @transz
define var2 @transz
```

Loft will read these lines in order. It will start a new section object and define its length to be 10. Then it will read the first `define` command and the preprocessor will replace the `@transz` system variable with the value of 0. Then, the main *Loft* code will determine that a new command has been specified and thus the user is done with the previous object. The section object will be created and the `@transz` system variable will be assigned a new value of 10. Next, *Loft* will actually create the `var1` variable and assign it the value of 0 that the preprocessor had already placed on the input line. Finally, the last `define` command will be read. The preprocessor will replace the variable `@transz` with the value 10 and then the main code will assign that value to `var2`. Thus, for very subtle reasons, the values of `var1` and `var2` will be different.

A work around for this issue is to put another command between the last object parameter and the first `define` command. That command will trigger the generation of the object and the updating of the `@transz` system variable before the definition command is read and handed to the preprocessor. For instance, just adding the command `null` before the `var1` definition would result in both variables have the same, expected, value of 10.

Project 8: Bodies of Revolution, Toroids, and Helixes

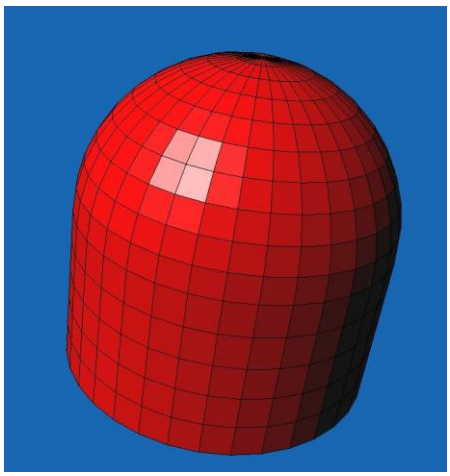
Any curve type can be used to create a body of revolution in *Loft*. Five parameters in the section object type can be used to create bodies of revolution, toroids, and helixes. These parameters are `radius`, `c1_rotation`, `c2_rotation`, `c1_yoffset`, and `c2_yoffset`.

Some caveats for these objects: These meshes will not stack well in a sequential object generation (like the full examples at the end of this manual). Currently frames won't generate on the rotated object except at the initial curve1 position. Finally, the model will not be aligned with the center of rotation at zero; it will need to be moved if that is desired (use `transx -1 * <radius>`).

The parameter `radius` is used to specify the desired distance from the y-axis aligned rotation axis to the x=0 point on the curve being extruded. On half curves in the built-in library, x=0 on the line of symmetry of the curve (where the missing mirror half would start). For full curves, x=0 on the centerline of the curve.

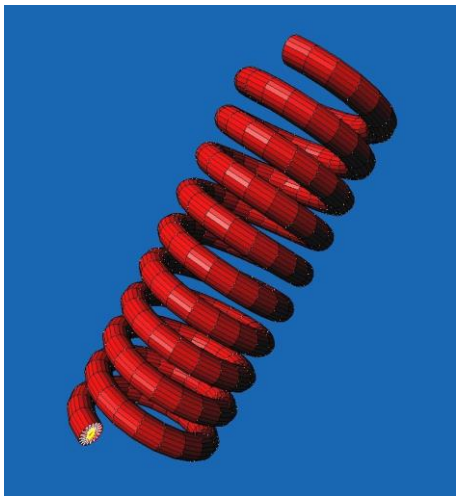
The simplest body of rotation using a half curve is illustrated below where a semi-breadbox ("sbb") library curve (square bottom half, circular top half) is rotated 360 degrees. The *Loft* input file to generate this mesh is:

```
# Body of revolution
object section bor
   curve1 sbb
   curve2 sbb
   nodes_axial 36
   nodes_circ 21
   length 0
   radius 0
   c1_rotation 0
   c2_rotation 360
# save
write vrml bor.wrl
end
```



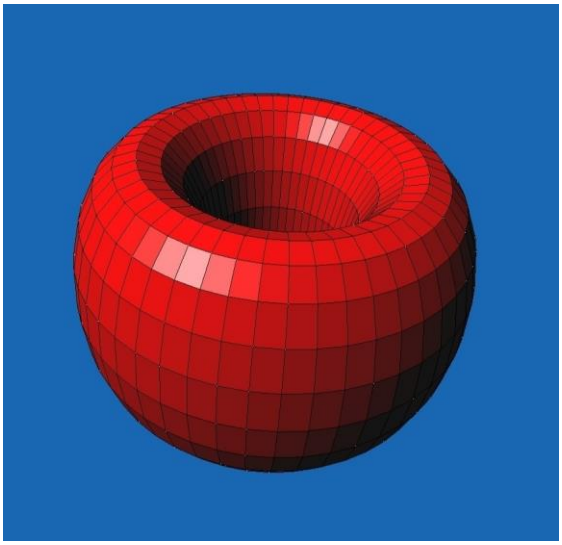
The two `rotation` parameters are used to specify the arc in degrees that the corresponding end is rotated. Using 0 and 360 will produce a full body of revolution.

By using a much higher value for the rotation, such as 3600 degrees (10 revolutions), and a `yoffset` at one end, a helix can be produced.

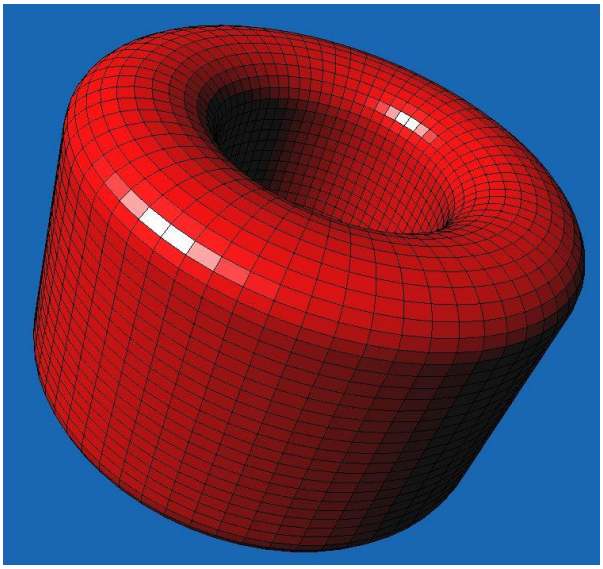


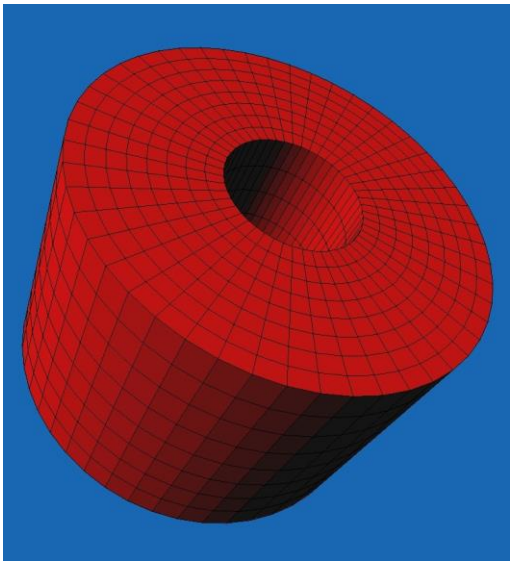
```
# Helix
object section Spring
   curve1 cir
   curve2 cir
   nodes_axial 360
   nodes_circ 10
   length 0
   radius 2
   c1_rotation 0
   c2_rotation 3600
   c2_yoffset 30
move
   transx -2
# save
write vrml spring.wrl
end
```

Any *Loft* curve type can be used including user-defined curves. `Curve1` and `curve2` can even be different, which will work fine for a helix or partial body of revolution but will not stitch well in a full 360 body. The examples below used a variety of cross section curves (`cir`, `fillet`, and `squ`), 360-degree rotation, a `radius` value higher than one, and zero `yoffset`. Note the `move` command that aligns the x=0 axis with the center of the finished object. Only one input file is shown.



```
# Simple Toroid
define myrad 3.0
object section tank1
   curve1 cir
   curve2 cir
   c1_yscale 1.5
   c2_yscale 1.5
   nodes_axial 36
   nodes_circ 20
   length 0
   radius $myrad
   c1_rotation 0
   c2_rotation 360
move
   transx -1 * $myrad
# save
write vrml toroid.wrl
end
```





Project 9: Program Flow Control

Program flow control is the ability of the user to direct *Loft* to read/operate on its input in a way other than sequential. This can be a jump to a different section of the input file, a loop that repeats a block of input, a conditional execution of some input, or a subroutine that can be called with a different input. These capabilities are implemented with a very minimal but functional set of commands.

Flow control requires the use of *Loft*'s variable and math functionality (see previous tutorial). The logical operators (`>`, `<`, `>=`, `<=`, `=`, `!=`) are particularly useful. The logical operators return 1.0 when the comparison is true and 0.0 when it is false.

The `linelabel` command is the base for most of the program flow control capabilities. It does not perform any action other than marking a location in the input file with an identification line number. The argument for the command is a unique value. If duplicate line numbers are assigned, *Loft* will use the earliest position in the input where that number is encountered and ignore later occurrences. Variables can be used to specify line numbers but exercise care with changing those values.

The `goto` command is the next program flow control command. Its argument is a line number. When read, *Loft*'s execution will jump to the specified `linelabel` line. This can be before or after the current execution location. Line numbers can be used in external input files that are inserted with the `include` command, but *Loft* will not be able to jump forward into an external file that has not yet been read; it can only jump backwards into a previously included file.

`If` is the conditional program flow command. Its argument is most commonly a logical comparison (e.g., "`if $i > 5`") but can be anything that produces a value. If the argument value is non-zero, the result is treated as true and the next input line is read and executed. If the value is near zero (defined as between +/- 1.0E-4 to allow for round off errors), the result is treated as false and the next input line is skipped.

These three commands enable powerful control of the flow of *Loft*'s execution. They can be combined to create loops, branching execution, functionality like the switch/case logic in the C programming language or the "on/goto" logic in the BASIC programming language, etc.

The `if` command can be used to support multiple configurations of a model in a single input file. The TSTO orbiter example included at the end of this manual defines a variable called `fullvehicle` near the top of the file. If the value is set to zero (false), then a half vehicle is generated. If the value is set to one (true), a full vehicle is generated. See the annotated example for more details.

To create a loop, start by initializing a counter variable. Then, add a line number that is the beginning of the loop. Now, write the input that you want to repeat several times. At the end of that input, increment the counter variable. Now, do a comparison to see if the counter is more than the desired number of loop executions and either loop back to the top or exit the loop and continue with the rest of the file. Here is an example of doing that:

```
define i 0
linelabel 10
   <do something we want to repeat>
   define i $i + 1
if $i <= 5
```

```
goto 10
<rest of input file>
```

Care should be taken to avoid infinite loops. Ensure that the counter initialization is outside the loop and that the counter incrementing is inside the loop. This example will perform the commands within the loop five times. One application of loops could be to make an array of identical tanks. The user would need to make sure that each new tank was rotated or translated to a different location inside the loop to avoid duplicates being automatically merged.

To have *Loft* select between multiple potential blocks of code, create a destination line number variable and math calculations to set that variable to one of several possible values.

```
define destination $length * 10.
goto $destination
```

This approach can produce program logic that is similar to "switch/case" in the C programming language or "on/goto" in the BASIC language. Just ensure that line numbers are set up for all possible values of the destination variable.

The `include` command is used to insert lines from an external file into *Loft*'s input stream. It can be used for many reasons, including file readability, defining a set of variables that is used by multiple projects and needs to be consistent, or as a subroutine. A simple example of this subroutine approach is shown below:

```
# main program
define radius 10
include generate_tank.txt
write vrml r10tank.wrl
new
define radius 20
include generate_tank.txt
write vrml r20tank.wrl
```

And the generate_tank.txt file that is being included:

```
# generate_tank.txt file
object dome tanktop
   c1_xscale $radius
   c1_yscale $radius
   length $radius * -1.0
object dome tankbot
   length $radius
```

Here the included file enables the creation of two spherical tanks with different radii without having to create duplicate code. Note that the output files have different names and the `write` command is not in the subroutine.

Project 10: NASTRAN bonus features

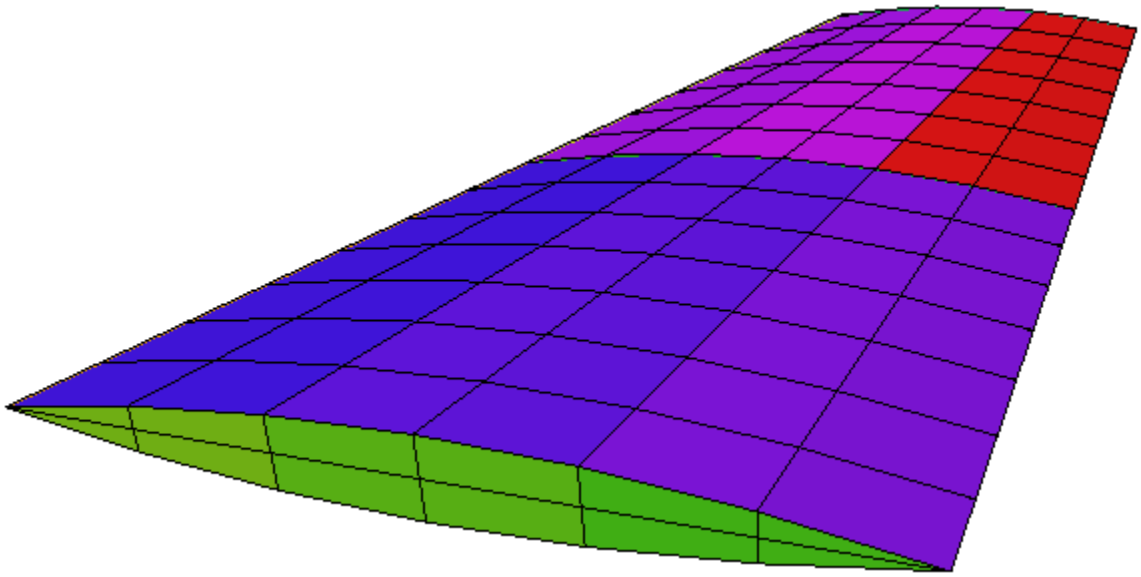
A collection of *Loft* features was designed to add analysis capabilities beyond mesh generation. These features are currently only fully applicable to NASTRAN output files. A few features are written to VRML files. Output support for other formats may be added in the future.

These features make use of *Loft*'s region mode capabilities that can mark nodes and elements based on their object names (e.g., "main wing"), portion of an object (e.g., "upper skin"), or geometric location. Combinations of these factors can be used to focus in on only a few nodes and/or elements for which applying analysis entities is desired.

These analysis entities include forces, pressures, temperatures, boundary conditions, rigid boundary elements (RBEs), and point masses. A single NASTRAN case control block that references the loads and boundary conditions can also be generated and written.

The annotated TSTO orbiter example included at the end of the manual makes extensive use of each of these capabilities. This tutorial shows the use of a few representative features necessary to generate a wing model, apply a uniform pressure to the upper skin, constrain the model at the root-spar nodes, and write a ready-to-run NASTRAN deck for the loaded model.

First, define the wing. Since we'll be constraining the nodes on the spar root, we need to request that spars be generated. This example uses a biconvex airfoil with the default 10% thickness to chord ratio.



```
object wing demo wing
   span 30
   chord 20
   taper 0.5
   sweep 20
   nspars 2
   nribs 3
   naca bicon
```

```
   meshchord 0.5
   meshspan 0.5
   meshthick 0.5
```

Next, let's list some of the groups and properties that were automatically created by *Loft* to go with this wing.

```
list groups
list pprops
```

These commands are optional. They produce output to the screen (shown in text box below) that will be helpful in creating the load and boundary condition sets that we'll do next.

```
Read line: list groups

 Group list for mesh demo wing:

        0: demo wing ROOT NODES                    members      4, type nodes, id 0
        1: demo wing TIP NODES                     members      4, type nodes, id 1
        2: demo wing ROOT SPAR NODES               members      6, type nodes, id 2
        3: demo wing ROOT RIB NODES                members     25, type nodes, id 3
        4: demo wing CARRYTHR NODES                members      0, type nodes, id 4
        5: demo wing SKIN UP ELEMS                 members     84, type elems, id 5
        6: demo wing SKIN LOW ELEMS                members     84, type elems, id 6
        7: demo wing SPAR ELEMS                    members     56, type elems, id 7
        8: demo wing RIB ELEMS                     members     36, type elems, id 8
        9: demo wing QUARTER CHORD VECT            members      0, type nodes, id 9
       10: demo wing ALL NODES                     members    219, type nodes, id 10
       11: demo wing ALL PANELS                    members    260, type elems, id 11

Read line: list pprops

  List of physical properties:
 0    demo wing RIB
 1    demo wing SPAR
 2    demo wing SKIN UPPER
 3    demo wing SKIN LOWER

  End of physical property list.
```

Now, we can use this information to create a pressure load set and a root-spar-node boundary condition set. For the pressure, we specify the upper skin element group, a value of 1.0 for a unit load up, and a NASTRAN setid of 100.

```
object press upper skin lift
   group1 demo wing skin up elems
   value 1.0
   setid 100
```

For the root boundary conditions we specify that they are to be applied to the root spar nodes and are to constrain the translation in all three directions (degrees of freedom 1, 2, and 3) and use a setid of 200.

```
object bc root bc
   group1 demo wing root spar nodes
   doflist 123
   setid 200
```

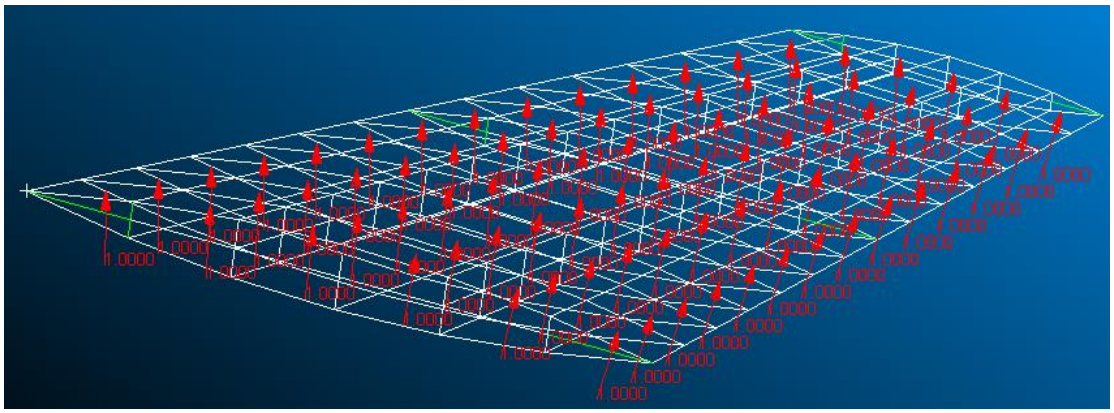
Then, we set our NASTRAN parameters to refer to the two previous setids and to use NASTRAN solution 101:

```
nastran sol 101
nastran loadset 100
nastran spc 200
```

And finally, we write out the generated file in two formats:

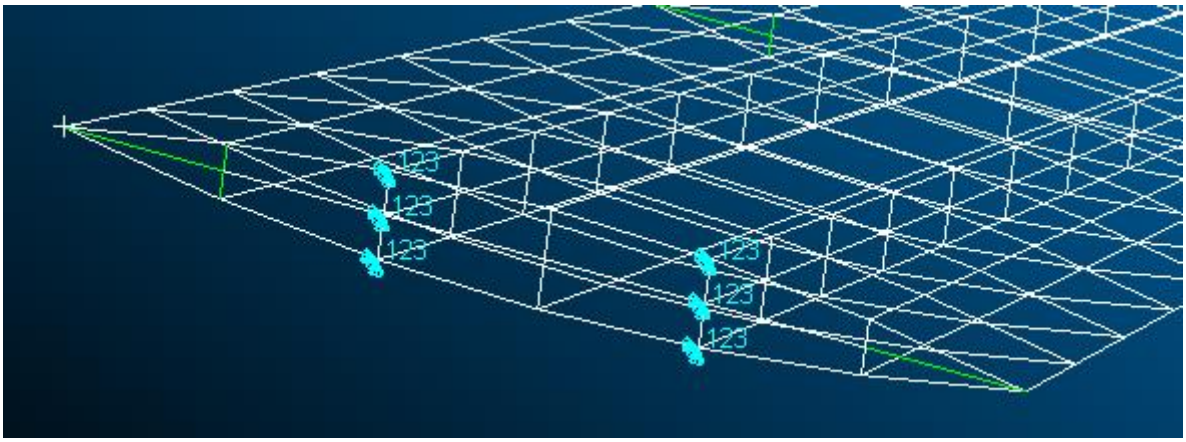
```
write vrml project10.wrl
write nastran project10.bdf
```

After *Loft* is run using this input file, the resulting bdf file can be imported into Patran to inspect the load and boundary conditions and run in NASTRAN to produce stress and deflection results.



Upward unit pressure load created by *Loft*

Read through the annotated TSTO orbiter example at the end of this manual for more detailed and complex generation of NASTRAN analysis entities including forces, point masses, and rigid boundary elements (RBEs).



Root spar translation boundary conditions

Project 11: Automatic Stitching

Loft will automatically merge nodes and elements that are coincident. Stacks of objects that form a fuselage will generally align and automatically merge as long as node count and distribution are the same.

The meshes on wing roots and fuselage sidewalls are inherently different; there are not coincident nodes to automatically merge. Aerodynamically, a fairing is often used to blend the wing to the fuselage. But, the structural loads are generally designed to be carried by wing spars connected to fuselage bulkheads and/or ring frames. Since *Loft* is a structurally focused program, the second approach is the recommended technique and the one that will be demonstrated in this project.

The annotated TSTO orbiter model in the examples at the end of this manual uses the techniques that this project will demonstrate to attach both its wing and its vertical tail. It does this parametrically so that if vehicle dimensions change or the user switches between half and full models, the stitching still works. This example demonstrates the approach with a much simpler configuration.

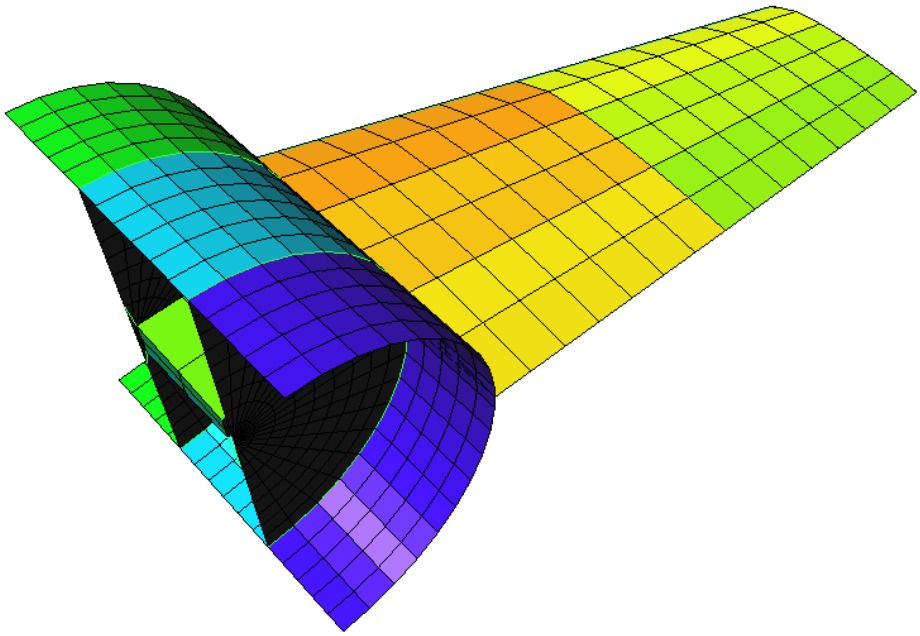
The steps necessary for this approach are:

1. When building the model consider the structural load paths between the wing and vehicle body. Position bulkheads and/or ring frames near the wing spars.
2. Isolate and mark the portions of the spars and bulkheads that are going to be connected and use the region "corner" operation to identify the corners of the marked areas.
3. Create rigid boundary elements (RBEs) that connect the identified corners.

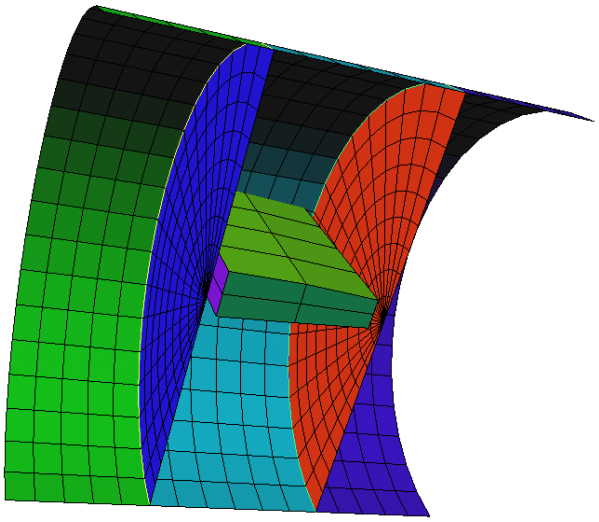
For this simplified case, we're going to model a wing and only the section of the fuselage that is beside the wing. Two spars and two bulkheads will be positioned at one-third and two-thirds of the chord. These will then be marked and connected to each other. For visualization purposes, an axial offset will be applied to the bulkheads so that they do not perfectly align with the spars. This will allow us to better see the generated RBE elements. This offset can be modified to zero if desired.

```
define chord 20
define offset 3
define meshdens 1.0
# derived dimensions
define fuserad $chord / 2.
define length1 33.3333 - $offset * $chord / 100.
define length2 66.6666 + $offset * $chord / 100. - $length1
define length3 $chord - $length1 - $length2
list variables
# make fuselage in 3 sections with a bulkhead between each
object section fuse1
   length $length1
   c1_xscale $fuserad
   c1_yscale $fuserad
   c2_xscale $fuserad
   c2_yscale $fuserad
   nodes_axial $length1 * $meshdens
   nodes_circ 20 * $meshdens
object dome bulkhead1
   taper bulk
   nodes_axial 10 * $meshdens
```

```
object dframe ring1
object section fuse2
   length $length2
   nodes_axial $length2 * $meshdens
object dome bulkhead2
   taper bulk
   nodes_axial 10 * $meshdens
object dframe ring2
object section fuse3
   length $length3
   nodes_axial $length3 * $meshdens
# make wing
object wing mywing
   span $chord * 1.5
   chord $chord
   taper 0.5
   sweep 20
   nspars 2
   nribs 3
   wingbox $fuserad
   meshchord 0.5 * $meshdens
   meshspan 0.5 * $meshdens
   meshthick 0.5 * $meshdens
   transz 0.0
   transx $fuserad
```



The wingbox and bulkheads that we want to stitch together:



Now that the model is built with the spars and bulkheads positioned close to each other (step 1 in our process) we can prepare to perform step 2 which is identifying the nodes that we want to connect. To help with marking the desired stitching areas, a list of automatically generated groups is requested:

```
list groups
```

The output from this command is written to the screen or piped to a file using the windows ">" pipe command (e.g., `loft project11.txt > project11.out`). It lists several groups for each of the objects in the model. We are particularly interested in the nodes on both bulkheads and on the carry-through spars. Looking at the group list, we can see the groups we want:

```
8:  bulkhead1 ALL NODES              members     91, type nodes, id 8
17: bulkhead2 ALL NODES              members     91, type nodes, id 17
34: mywing CT SPAR ELEMS             members     16, type elems, id 34
```

Another option is to use the carry-through nodes.

```
26: mywing CARRYTHR NODES            members     99, type nodes, id 26
```

It would be a little more work to isolate the desired spar nodes from the carry-through nodes group. Your groups could have different id numbers, but the names are what we need for the next step and they should not change. This listing of the groups is not required for the stitching operation but is included as part of this tutorial project to clarify where the names used in the next step were obtained.

The next set of lines performs step 2 for the forward bulkhead. All of the nodes on the bulkhead are added to a region, then only the nodes that are vertically within 1/5 of the radius from the center are retained using two `ikeep` operations. The division by 5 in this step could be changed to another value if a broader or tighter area were desired. The bulkhead mesh density will also have an impact on this choice because at very low mesh densities nodes may or may not exist in the specified region if it is very narrow. A broader area (dividing by 3 or 4 instead of 5) would be more robust but perhaps less realistic.

```
region
```

```
   mkadd bulkhead1 ALL NODES
   ikeep yle $fuserad / 5.
   ikeep yge -1.0 * $fuserad / 5.
   corner bulk1attach
```

The corner operation identifies the four nodes that are still in the region and that are the most distant from the centroid of the region in each coordinate quadrant. The nodes are added to a group named "`bulk1attach`" and will be used as connection points. The carry-through spar nodes are marked in a similar process. Here the `ikeep` operation keeps the spar nodes that are in the front half of the model.

```
region
   mkadd mywing CT SPAR ELEMS
   ikeep zlt $chord / 2.
   corner wing1attach
```

The connections for the rear spar/bulkhead pair are marked similarly using the second bulkhead and a "zgt" inequality rather than "zlt" to select the rear half of the model.

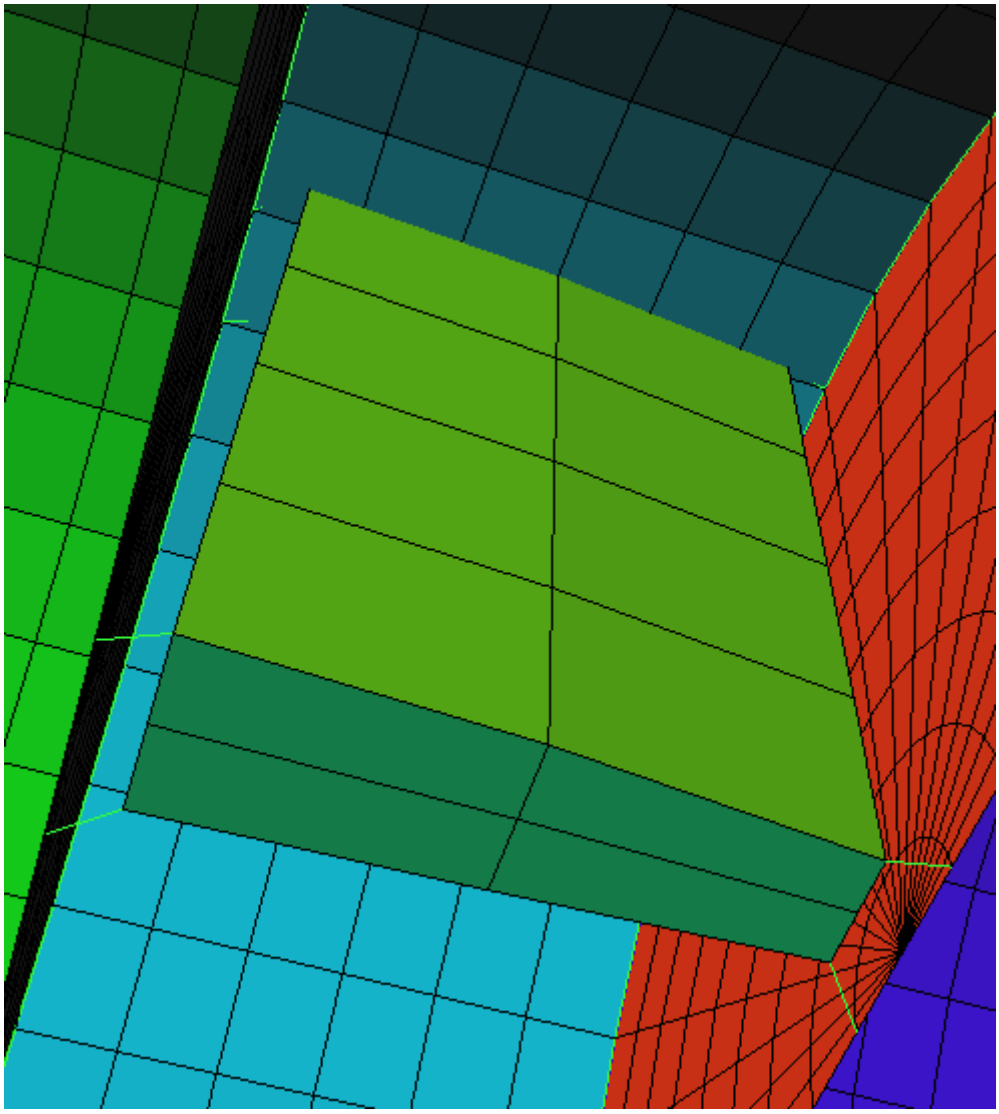
```
region
   mkadd bulkhead2 ALL NODES
   ikeep yle $fuserad / 5.
   ikeep yge -1.0 * $fuserad / 5.
   corner bulk2attach
region
   mkadd mywing CT SPAR ELEMS
   ikeep zgt $chord / 2.
   corner wing2attach
```

We have completed step 2 of our stitching process. Step 3 is performed easily by referencing the four groups of corner nodes that we have created:

```
object rbe forward attach
   group1 bulk1attach
   group2 wing1attach
object rbe aft attach
   group1 bulk2attach
   group2 wing2attach
```

The very last step of the project is to save the model. The new RBEs will be visible on the VRML model as lines. The spar nodes at the wing root are just outside the fuselage, so the outboard RBEs are only partially visible from the inside. Resetting the `offset` variable to zero to make the spars and bulkheads coplanar would make the RBEs very short and difficult to see but is what would probably work best for analysis.

```
write vrml project11.wrl
write nastran project11.bdf
```



Diagonal green lines are the new RBEs connecting the wing to the bulkheads

Chapter 3: Regions

The region tool set is a feature of *Loft* that allows the user to query or modify a section of the current stack. Regions are inherently temporary constructs, but their effects may include permanent changes to the mesh by deleting parts, changing property assignments, etc. Regions can also be used to query statistics on the mesh and produce reports.

There are two parts of the region process. The first is to specify what nodes and elements make up the region. The second is to perform the desired task(s) on those nodes and elements.

Defining a Region

There are multiple ways to identify nodes and elements to add to a region. A control volume such as a box or sphere can be specified. A coordinate inequality can be used to keep or remove everything on one side of a cutting plane. A material property, physical property, or automatically generated group can be used. A name previously used in a *Loft* `mark` command can be accessed to add those elements to a region. Multiple combinations of these options can be strung together.

For instance, one could define a region as all elements marked as "OML" that do not have "main wing" as their physical property. While exact syntax will be discussed later in this chapter, the logic of this operation would be "add all elements marked oml" followed by "remove elements with physical property main wing."

Acting on a Region

There are two classes of actions that can be performed on a region. Passive actions are actions such as queries that do not change the mesh data. Active actions modify the mesh data in the region by changing properties, deleting nodes or elements, etc. **Only one active action can be performed in any particular use of the region command**, as the node and element lists that *Loft* uses to define that region will become stale. A new region command can be started to perform additional active operations.

Like the stack-level `move` command operations, the `region` parameters are acted upon sequentially. Thus, one could add some elements, do a (passive) query, add some more elements, do another query, remove some elements, query, and then perform an (active) cut action to complete the current region command.

Region Commands

Region mode is entered by issuing the *Loft* command `region`. Any number of region-mode operations can be specified in sequence until another *Loft* command is encountered. After the first active operation, any further operations will be ignored and a warning to that effect issued. A new `region` command must be started for each additional active operation that the user wishes to perform. All `region` commands reset the initial list of selected nodes and elements in the region to be empty.

Mesh Selection Parameters

These parameters add or remove elements and nodes from the current selection list. They are all passive.

The *volumetric* selection parameters identify nodes that fall in the specified volume. *Loft* then adds all elements that use those nodes to its selection list as well. This element addition can be "inclusive," resulting in the addition of any element that has at least one of its nodes in the specified volume, or it can be "exclusive," where all element nodes must be in the volume for that element to get added to the selection list.

The *property* selection parameters identify elements that have the specified material property, physical property, or *Loft* `mark`. In turn each node that those elements use is also added to the selection list.

Volumetric Selection Parameters

`iadd` – Inclusive node addition. Adds all nodes that fall within a specified volume of space. Any elements that use any of these nodes will be added as well. Volumes are specified by use of simple three-dimensional shapes including spheres, cylinders, and boxes. Cylinders are aligned with an axis and are infinite in length. *Warning*: Any beams whose alignment nodes fall in the specified volume, even if the beam end points themselves do not, will also be added. The type "all" will add all nodes (and thus all elements) in the current stack. No dimensions are required for the "all" type.

For the coordinate comparisons (xeq, xgt, xge, xlt, xge, etc.) a single value is specified and the node is added if its coordinate meets the criteria. The planar division options (px, nx, etc.) represent positive or negative coordinates and do not require a value. They are just a shortcut; "px" is treated as "xgt 0."

Usage: **`iadd`** <*volume type*> <*center of volume*> <*dimensions of volume*>
Or: **`iadd`** <*coordinate comparison type*> <*value*>
Or: **`iadd`** <*planar division comparison type*>
Examples:
```
iadd sphere 10. 20. 25. 5.
iadd yge -4.3
iadd nx
```
Volume Type = "all," "sphere," "xcyl," "ycyl," "zcyl," "box"
Coordinate comparison type = "xeq," "yeq," "zeq," "xgt," "ygt," "zgt," "zge," "yge," "zge," "xlt," "ylt," "zlt," "xle," "yle," "zle"
Planar Division comparison type = "px," "py," "pz," "nx," "ny," "nz"
Center = x, y, z coordinate of center of volume
Dimensions = radius for sphere and cylinders,
= xlength, ylength, zlength for box.
Value = coordinate to be compared to

`irem` – Inclusive node removal. Removes from the selection list all nodes that fall within a specified volume of space. Any elements that use any of these nodes will be removed as well. **This operation does not delete anything from the mesh, it just removes the specified items from the region selection list.** *Warning*: Any beams whose alignment nodes fall in the specified volume, even if the beam end points themselves do not, will also be removed. The type "all" will remove all nodes (and thus all elements) in the current stack. Arguments and usage are the same as the `iadd` parameter .

`eadd` – Exclusive node addition. Adds all nodes that fall within a specified volume of space. Any elements *with all of their nodes in the selection list* will be added as well. Arguments and usage are the same as the `iadd` parameter .

erem – Exclusive node removal. Removes from the selection list all nodes that fall within a specified volume of space. Any elements *with all of their nodes in the volume* will be removed as well. Arguments and usage are the same as the `iadd` parameter .

ikeep/ekeep – Inclusive/exclusive node removals that are the inverse of the `irem/erem` commands. In other words, nodes/elements that fall within the specified volume are retained and those that do not are removed. Thus, "`ikeep all`" has no effect. Arguments and usage are the same as the `iadd` parameter .

Property Selection Parameters

mpadd – Add elements to the selected list based on their material property name. The material property name is used to indicate where on the component the elements reside and vary based on the `components_axial` and `components_circ` object variables. All nodes used by the elements are also added to the selected list. Use the "`list mprops`" command to see current project material properties.

Usage: **mpadd** <*material property name*>
Example: `mpadd SB 0 CB 0`

mprem – Remove elements from the selected list based on their material property name. The material property name is used to indicate where on the component the elements reside and vary based on the `components_axial` and `components_circ` object variables. All nodes used by the elements are also removed from the selected list. If some of those nodes are used by other elements that are still selected, an `update` operation may be desired.

Usage: **mprem** <*material property name*>
Example: `mprem SB 0 CB 0`

ppadd – Add elements to the selected list based on their physical property name. The physical property name is in most cases the object name given by the user. All nodes used by the elements are also added to the selected list. Use the "`list pprops`" command to see current project physical properties.

Usage: **ppadd** <*physical property name*>
Example: `ppadd lox tank`

pprem – Remove elements from the selected list based on their physical property name. The physical property name is in most cases the object name given by the user. All nodes used by the elements are also removed from the selected list. If some of those nodes are used by other elements that are still selected, an `update` operation may be desired.

Usage: **pprem** <*physical property name*>
Example: `pprem lox tank`

mkadd – Add elements to the selected list based on their marks. Marks are set using the `mark` parameter during object creation. An object can have any number of marks. By default, it will have one that contains its object name. In preparation for the use of this command the user can assign marks such as "OML," "fuselage," "tankage," "bulkheads," "wings," etc. and then add and remove multiple objects based on the chosen marks. All nodes used by the elements are also added to the selected list. Use the "`list groups`" command to see current project marks/groups/labels.

Usage: **mkadd** <*mark name*>
Example: `mkadd OML`

mkrem – Remove elements from the selected list based on their marks. Marks are set using the `mark` parameter during object creation. An object can have any number of marks. By default, it will have none. In preparation for the use of this command, the user can assign marks such as "OML," "fuselage," "tankage," "bulkheads," "wings," etc. and then add and remove multiple objects based on the chosen marks. All nodes used by the elements are also removed from the selected list. If some of those nodes are used by other elements that are still selected, an `update` operation may be desired.

Usage: **mkrem** <*mark name*>
Example: `mkrem OML`

Passive Operation Parameters

Passive operations can be used to change membership of a region or list information about the current nodes or elements that are in the selected list. By default, the output is printed to the screen and the user has the option of piping the output to a file using the command line. Alternatively, the user can specify an output filename for the query results to be sent to. The user can also specify that the data is to be formatted as FEA file data lines (e.g., the node list could be in NASTRAN GRID cards) or (by default) in a more human readable format. Some query results will not have an appropriate FEA format to be printed in and will only be reported in the *Loft* native style.

inverse/invert – Change all items in the selection list to not-selected and all not-selected items to selected. Both spellings have the same effect.

Usage: **Inverse**
Example: `inverse`

update – Re-add all nodes used by elements in the selection list to the node selection list. Depending on the order of addition and removal operations and the choice of exclusive or inclusive, the two lists may not be completely synced. If syncing is desired, this will force an update.

Usage: **update**
Example: `update`

fileout/fileappend – Specify an output file to send `query` and `rwrite` outputs to. By default, this output is printed to the screen. All output is appended to the end of a (possibly) pre-existing file. Either command name may be used.

Usage: **fileout** <*filename*>
Example: `fileout region1.wrl`

filenew/filewrite – Specify an output file to send `query` and `rwrite` outputs to. By default, this output is printed to the screen. This variant creates a new file (overwriting any existing file of the same name) rather than appending to a possibly pre-existing file as `fileout` does.

Usage: **filenew** <*filename*>

Example: `filenew region1.wrl`

`format` – Specify the format for the query outputs. The *Loft* default is a human readable chart format. Other options are "nastran," "abaqus," "stl," and "vrml." Some queries may produce output not suitable for the requested format in which case that output will be presented in the *Loft* format. This value will be reset to the default when a new region is created.

Usage: **`format`** <*filetype*>
Filetype = "loft," "nastran," "abaqus," "stl," "vrml." (loft is the default)
Example: `format vrml`

`query` – Request various reports on the items in the selected list. Specifying "nodes" will list the selected node numbers and each node's coordinates. "Elements" will list the element numbers, their nodes, their properties, and (as supported by the chosen format) any marks on the elements. "rbes" will list all of the rbe/bc/force/mass/press objects in NASTRAN format. "Matprop" will list the material properties used and "physprop" will list the physical properties used. "Properties" will list both the material and the physical properties used by the selected elements.

Usage: **query** <*type*>
Type = "nodes," "elements," "properties," "matprop," "physprop"
Example: `query elements`

`mark` – Add a label to all of the nodes or elements in the region. Items can have as many different labels as desired. Marks have limited uses. They can use used to sort elements in the region command and will be output as groups when an I-DEAS output file is created. Support for NASTRAN SET grouping can be enabled by removing a comment in "nastran.c." The `mark` parameter takes two arguments: the group type (node, element, or rbe) and the group name. A marked group can contain either nodes or elements, but not both. If the type parameter is not present, the "element" type is used.

Usage: **`mark`** <*type*> <*name*>
Example: `mark element OML`

`comment` – Write a commented line of text to current output in the current format.

Usage: **`comment`** <*text of comment*>
Example: `comment These elements are all marked OML`

`rappend` – Write the selected items as if they were a complete mesh. The output is appended to, rather than overwriting, the specified file This command ignores the `fileout`, `filenew`, and `format` settings, rather matching the syntax of the non-region `write` command and the alternative form of the `rwrite` command, requiring the format and filename be supplied with the command.

Usage: **`rappend`** <*format*> <*filename*>
Alternate example: `rappend nastran region.bdf`

`rwrite` – Write the selected items as if they were a complete mesh. Uses the values set by the `format` and `fileout` or `filenew` commands. There is an alternate form of `rwrite` where the format and filename are specified along with the command (as is the case with the non-region `write` command) and the

`format`, `fileout`, and `filenew` commands are ignored. In the alternate form, a new file is always created. If appending to an existing file is desired either use the `rappend` command or the non-alternate form of the `rwrite` command with a previous `fileout` specification.

Usage: **`rwrite`**
Example: `rwrite`
Alternate usage: **`rwrite`** <*format*> <*filename*>
Alternate example: `rwrite nastran region.bdf`

`corner` – Identifies up to 8 nodes that are most distant from the centroid of the region's nodes, one in each of 8 coordinate quadrants relative to the centroid. These nodes are added to the named group for later use. If the object is not aligned with the coordinate axes, some rotation to align may be desired to correctly identify the corners.

Usage: **`corner`** <*name*>
Example: `corner Main Wing spar corners`

Active Operation Parameters

Active operations attempt to change the selected region's mesh in some way. This can be a property change, deletion, rotation, flipping of elements, etc. Again, once one active operation has been performed on the specified region, the selection list is marked as being "stale" (since nodes and elements it points to may no longer exist or may no longer meet the region selection criteria) and no further operations are permitted on the region.

`cut` – Remove selected elements and nodes. This operation has two modes. The "element" mode will remove only the elements in the current region. No nodes will be deleted. The "node" mode will remove both the marked elements and the marked nodes. Additionally, non-selected elements may be deleted depending on the number of their nodes that remain after node deletion. Panels that end up with three nodes are converted to triangles. Panels with two or fewer nodes are deleted. Bars or beams that lose any nodes (including their alignment node) will also be deleted. The node version of this operation is similar, but not identical, to the (non-region) subtract command.

Usage: **`cut`** <*type*>
Type = "element," "node"
Example: `cut element`

`setmp` – Change elements to use the specified material property. If the property name does not exist, it will be created.

Usage: **`setmp`** <*name*>
Example: `setmp door cutout`

`setpp` – Change elements to use the specified physical property. If the property name does not exist it will be created.

Usage: **`setpp`** <*name*>
Example: `setpp nose cap`

flip – Reorder element nodes to reverse normal vector direction.

Usage: **flip**
Example: `flip`

rotate – Reorder element nodes to rotate element orientation. The original node 2 becomes node 1, the original node 3 is now node 2, etc., and the original node 1 becomes node N.

Usage: **rotate**
Example: `rotate`

beamalign – Re-align orientation of any beams to use a new alignment node. Coordinates of the node can be specified or the first node in a specified group of nodes will be used. This operation does not modify bars (that start with no alignment node since they have only an axial degree of freedom.)

Usage: **beamalign** <*x*> <*y*> <*z*> or **beamalign** <*node group name*>
Examples:
```
beamalign 0.1 0.2 0.3
beamalign a group with nodes
```

baralign – Convert bars to beams, adding an alignment node. The alignment of any existing beams will not be changed, use `beamalign` for that operation.

Usage: **baralign** <*x*> <*y*> <*z*> or **baralign** <*node group name*>
Examples:
```
baralign 0.1 0.2 0.3
baralign a group with nodes
```

beam2bar – Convert beams to bars, removing the alignment node.

Usage: **beam2bar**
Examples: `beam2bar`

Chapter 4: Tips and Best Practices

Names

Loft automatically creates a lot of names. It creates lots of groups and enables easy manual creation of even more. It is important to choose good names so that *Loft* and any external analysis packages that use the names (e.g., HyperX) can be used effectively.

For instance, an object named "bulkhead" is fine if it is the only one. If there are several in the model, then "bulkhead 1," "bulkhead 2," etc. will work. But, "fwd lox support bulkhead" is even clearer. It not only provides better understanding, it also permits adding more bulkheads as your design matures so that you don't end up with "bulkhead 1.5." Be clear with your naming logic so that you don't confuse "fwd tank aft bulkhead" with "aft tank fwd bulkhead."

Similarly, have a clear plan for your other named items like variables and labels to improve readability and reduce the chance of using the wrong item.

Comments

As with any other type of coding, make substantial use of comments. The pound symbol, "#," is used to indicate the start of a comment. A comment can either be on a line by itself or placed at the end of a line after a command or parameter.

It is good practice to start an input file with several lines of comments that give information about the file itself, including what is being modeled, who created the file, what date it was created and/or last modified, what units the dimensions are in, etc.

Variables

An excellent modeling approach is to have a section at the beginning of your input file with your main driving dimensions defined using good variable names. This makes the input file easier to read, reduces chances of typos, makes it easier to update the model, and makes it easier to validate that your model is correct. See the annotated TSTO orbiter example model. It has a very long section defining nearly every dimension on the vehicle. It is easy to understand and to change as the design evolves.

Even if this level of parameterizing is not needed, using variables when a value needs to be referenced multiple times reduces errors. And using meaningful names for your variables can make it possible for someone else to understand your model (perhaps that someone else will be you a few years later.)

One very common and recommended variable is a global mesh density variable. This can be included in every object definition with an appropriate multiplier to produce a clean mesh. Adjust the multipliers as you develop the model so that you have a consistent, low aspect ratio, mesh size throughout the model. A low value of the mesh density variable can be used for very rapid model generation while the model is being created and debugged. Once the layout is correct, increasing the mesh density variable at the beginning of the file will generate a denser mesh suitable for analysis.

Use the `list variables` command to list all of the user defined variables and their values. Check carefully for any variables with unexpected zero values. That generally indicates that there was a spelling

error in defining the name of that particular value variable. *Loft* will return a value of zero for any undefined variable that is used and generate a warning that an undefined variable was referenced.

Piping output

Loft is very verbose with its output. If everything runs correctly, you probably won't need to read any of it. But, while you are creating the model and things are not working as desired, the screen output can be very useful. The amount of output can make it frustrating to scroll back through to find the problem.

An approach to address this is to pipe the output to a text file that you can then open in an editor to more easily scroll and search through all of the text. In Windows, the greater than sign, ">," instructs the system to write the program output to a file. Two greater than signs, ">>," will append the output to an existing file if you want to check the output from multiple models. This approach is used when a validation run of a new version of *Loft* is performed on all of the tutorial and example files in this manual.

```
loft inputfile.txt > outputfile.txt
```

In Linux or UNIX the pipe symbol is the vertical bar "|":

```
loft inputfile.txt | outputfile.txt
```

Open the output file in notepad, vi, emacs, or other favorite text editor. You can then scroll through the file to find the section of the output that you are currently working on by searching for the text of an input line since they are echoed to the screen as they are executed. Or you could search the file for the words "warning" or "error" to see if *Loft* identified a problem.

Debugging with the list command

Loft's `list` command is a powerful validation tool to make sure that you are creating what you intend. Make heavy use of it and pipe the program output to a text file (see previous tip) so that the listed data can be examined more easily.

The `list` command has a large number of parameters to select which data is listed:

- `ccurves, icurves, lcurves` – user defined curves (compound, interpolated, lofted)
- `stacks` – models saved with the `store` command
- `variables` - names and values
- `groups,marks` – synonyms for labeled node/element lists
- `mprops, pprops` – property lists
- `ribs, spars` – wing rib/spar locations
- `mesh` - gives various data counts
- `rbes` – NASTRAN bonus data: forces, pressures, sbcs, rbes
- `input` – the current *Loft* input stream as modified by program flow control operations
- `all` – stand back, that's a lot of information (but not actually all of the above)!

A Quality Bonus Tip

Another tip is to suggest the use of the `quality` command. This will do a number of basic checks on the quality of the mesh, including looking for high aspect ratios, degenerate objects (where a node is used more than once), non-planar panels, etc. Addressing identified issues may be as simple as adjusting your object mesh densities or merge tolerance or may take some redesign of your modeling approach.

Chapter 5: Programmer's Guide and Reference

Introduction

This portion of the *Loft* user's manual can be used to gain a deeper insight into how *Loft* functions. But it is really intended for someone who wants to add new object types or functions to the program. The chapter starts with a conceptual description of how the program works, followed by an overview of the code structure. Finally, there are sections that describe how to add objects, commands, new output types, and new curve types to the program.

As program operations are described, the C file and/or subroutine that performs the function will be listed in the form "`subroutine.c/function-name.`"

Geometries and Meshes

A *Loft* input file contains a user's definition of a vehicle's geometry. The user's specified object types, dimensions, and meshing parameters are called the "abstract geometry." *Loft*'s main function is to read this abstract geometry and turn it into a concrete mesh made of nodes, elements, and a wide collection of elemental properties.

Loft does not internally store the abstract geometry of a vehicle. It has a "master" abstract geometry that consists of one object of each supported type. This master geometry is populated at program start with the default values described in the object descriptions in chapter 7. (`interface.c/initial_defaults`). As the program reads the user's geometry parameters, this master geometry is updated with the user's specified values (`interface.c/generate_object`). When an object definition is completed, a mesh is generated for the object and the master geometry is updated by copying appropriate changes to the other object types and by resetting other parameters to their initial values.

Loft works with two mesh data structures at a time. Both start with no data. The "stack" is a mesh containing all the previously generated objects' nodes, elements, and elemental properties. The "mesh" is the structure containing the current object. Both data structures are stored in the exact same way. An object generation subroutine is passed an empty mesh for which it allocates memory, populates with nodes and elements, and returns. When the mesh is completed, it is immediately merged with the stack and then erased by freeing its allocated memory. (The `store` command works very much like the "cut" command on a word processor. A pointer to the current stack is stored, and then a new empty working stack is created. Similarly, a `recall` command is like a "paste" command. The same routine that combines the main stack and a new mesh (`util.c/merge_sections`) combines the current working stack with the specified stored stack. In this case, the stored stack is not erased.)

Code Overview

Data structure/Constant definitions
- `loft.h`
- `loft-const.h`

Mesh storage and manipulation
- `util.c`
- `modify.c`

Mesh generation
- `loft.c`
- `wing.c`

Curve definitions
- `curves.c`

Region operations
- `region.c`

Output routines
- `abaqus.c`
- `ideas.c`
- `nastran.c`
- `vrml.c`
- `stl-ascii.c`
- `tecplot.c`
- `custom.c`

User input/Program control
- `interface.c`
- `variables.c`

Adding a New Object Type to *Loft*

The first step in adding a new object type to *Loft* is design. Determine the parameters that the user must set to define the abstract geometry of the new object and select default values for those parameters. Then, work out the logic of using those parameters to generate nodes, elements, and properties.

Now that there is a plan, it's time to start coding. In broad terms, there are two parts to writing the code: writing the meshing routine itself and adding support for the new object to the user interface. Both are somewhat involved.

Both parts of the coding will rely heavily on the object definition in "`loft.h`." Edit this file and move down to the abstract geometry object definitions section. Add a new structure here that defines the abstract geometry's parameters for your new object. Be sure to include structure members to define the object name, position, alignment, and a marklist. Finally, add your geometry structure to the "master_geom" structure near the end of the file.

The New Meshing Routine

You can add your meshing routine to "`loft.c`" or start a new source file. Your choice should be made based on the length and complexity of the meshing code. For instance, the various wing related meshing routines were created in a separate "`wing.c`" file. If you create a new file, remember to update the makefile so that it will be compiled and linked. Take a look at the various existing meshing routines for a feel of how they are written. The basic outline of each of these codes is as follows:

1. Based on geometry input parameters, make a conservative estimate of the number of nodes, elements, material properties, and physical properties needed by the new mesh. It is okay to allocate a little more space than is actually used if an exact calculation is difficult.
2. Call `malloc_mesh` to allocate memory for that data.

3. Create appropriate loops to generate the mesh data. As it is generated, store each piece of data by using the data storage routines from "`util.c`," e.g., `storenode, storequad, storetri, storegroup, addgroupmember, createproperty`, etc.
4. Update the mesh node (mesh-> nnodes) and panel (mesh->npanels) counts with the actual numbers of objects created.
5. Warp, rotate, and move the mesh.
6. Call `group_all_nodes` and `group_all_elem`.

If you look at the wing generation code, you'll note that it intentionally creates many duplicate nodes. It is okay to do this as long as space is allocated for them in the call to `malloc_mesh`. Just add a call to `merge_points` to the end of your routine to consolidate these duplicates.

Integrating Your New Object into the User Interface

The first step is to edit "`loft-const.h`" and create a new constant for your object type in the section that starts with "`#define OBJ_NONE 0`." Use the next available integer after the ones that are currently in use. For illustration purposes, let's say the new routine is used to create a wheel object and that the last object type used was number 12. Add "`#define OBJ_WHEEL 13`" at the end of the block.

Next, there is a lot of work to be done in "`interface.c`." Here we're going to create a new routine to parse the parameters for your new object, and then add support for the new object to the "parse_input," "parse_new_object," "generate_object," and "initial_defaults," routines.

The parameter parsing routine created should be similar to "`interface.c/ parse_section_param`." This routine will receive each line of text that is a parameter for the object. It should parse the parameter name and values from that line and assign them to appropriate data blocks in the abstract geometry structure. Finally, it should issue a warning if it was unable to do anything with the parameter it was given.

Remember to add a prototype for the new parsing routine to the top of the interface file.

The next step is to add the object to the "`parse_input`" routine. There are only two parts to this. First, add a malloc call at the top of the routine to make space to store your abstract geometry data. Be sure to add your new structure to the section that checks that the malloc succeeded. Then, scroll down to the line "`case CMD_NONE`" and add a line to the end of the parsing routines. It should be something like:

```
if(current_object == OBJ_WHEEL)
     parse_wheel_param(line,master.wheel);
```

Now, move down to the "`generate_object`" routine. Add a pointer variable for the abstract geometry and extract that pointer from the master geometry. Then, add a block that calls the new meshing routine if the object is of your new type, i.e.,

```
if(type == OBJ_WHEEL){
     printf("  Calling make_wheel\n");
     make_wheel(*wheel_geom,mesh);
}
```

After the new mesh is generated, we need to update the defaults of any abstract geometry types that need it. In most cases, you'll want to leave the current object's parameters as the defaults for the next object of the same type, but in some cases, you'll want to set them back to the defined default every time. You can update the defaults for any other geometry types as well. Add lines to your version of the block above in "`generate_object`" to update the desired defaults.

Scroll down to the "`initial_defaults`" routine. As with the previous routine, the first step is to add and extract a pointer variable for your abstract geometry. The other task here is to add a block that populates every data item in your geometry structure with its default value. Your defaults should be chosen such that if the user specifies no parameters, the meshing code will still generate a valid mesh.

Finally, scroll down to the "`parse_new_object`" routine. Again, add and extract a pointer variable to your abstract geometry. Next, add a block that tests for an object type name of your new type, sets the object name, and sets the `current_object` variable to your new type if it's found. For example:

```
if(strncmp(type,"wheel",5)==0){
     sprintf(wheel_geom->name,"%s",objectname);
     *current_object=OBJ_WHEEL;
     return;
  }
```

Now, compile, test, and debug your new object.

Adding a New Command to *Loft*

Adding a new command is a very similar process to adding a new object. As before, there are two steps: creating the routine to perform the new operation and integrating the command into the interface. It's difficult to be more specific since new commands could do anything and be logically integrated in many different places. You will probably want to add a new command number to "`loft-const.h`" and a "`case`" statement to the main loop in "`interface.c/parse_input.`"

Adding a New Output Type

Loft currently supports six types of mesh outputs. With accurate documentation of the new desired output format, it should be straightforward to use one of the existing output types as a basis for the new type and then edit the "`interface.c/output_stack`" routine to add a new block for your output routine.

A special case is the "custom" output type. This was created to make it easier for the user to modify the output to be exactly as they desire. No editing of the interface code is required; modify "custom.c" to produce the desired output and recompile. Typically, this approach has been used to make a short-term modification to one of the existing output types. For example, one could copy the NASTRAN output routines into `custom.c`, rename the functions, and then make small changes that might a) specify a non-structural mass for some elements, b) change the order that elements are written, or c) reduce the number of properties that the elements use. By making these types of changes to the custom output type, no hard to remove changes are made to the core output routines.

Adding a New Curve Type

The curve primitive routines are all located in the "`curves.c`" file. Scroll down to look at the semi-circle routine. The variable "s" is an input variable that ranges between 0.0 and 1.0. It represents the fractional position along the curve from its start (0.0) to end (1.0) for which coordinates are desired. The variables "x" and "y" are output values used to return the coordinates. If you're creating a curve family like the filleted curve, then "x" is also used as an input variable giving the family shape parameter.

The first step is to write a generation routine for your new curve type similar to the others in the file. Remember when modifying the variables "x" and "y" that their pointers are being passed rather than the variables themselves. Thus, your routine needs to set "*x" to the computed x coordinate.

Next, to add the new curve to the interface, return to the top of the "`curves.c`" file. Add a prototype for your generation routine. Now, scroll down a little and add a block for your new curve type and generation routine to the "`curves.c/curvefunctionptr`" routine. Note that there are different sections for non-family curves, family curves, and user-defined curves.

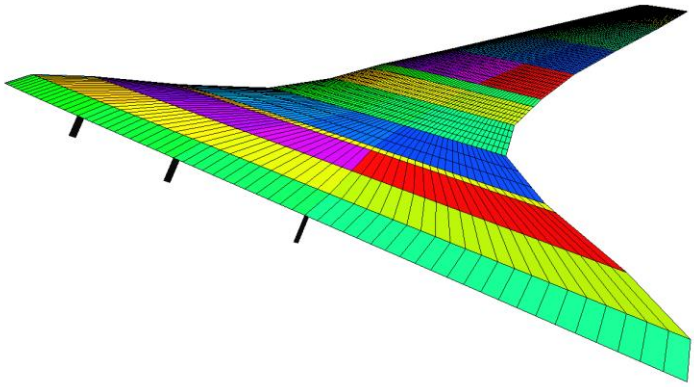
Be careful when selecting your curve's mnemonic to avoid collisions with other curves. For instance, if you want to use the mnemonic "`ssquiggle`," you need to add your check to `curvefunctionptr` before the check for the semi-square curve, since that check compares the first two characters of the curve name to "`ss`." It might be clearer if you chose "`semisq`" for your mnemonic instead. (You can see in the current routine that the check for the semi-circle "`sc`" mnemonic occurs after the check for the semi-cosine-wiggle "`sccw`.")

Now, save, compile, and test your curve. It should be usable from any object that uses curve primitives. There is no need to modify any of the meshing routines or user interface routines.

Chapter 6: External Utility Programs

In order to integrate *Loft* into a variety of multidisciplinary analysis systems, several utility programs have been written. These were created with general utility in mind and are therefore included in the *Loft* distribution and documentation. These programs can be used in a batch mode or can be used to speed up a manual model generation. They create normal *Loft* input files that can be modified as desired.

WingCoords2Loft



WingCoords2Loft is a utility that reads a file containing wing cross section data at various stations along the span of the wing and generates a *Loft* input file to create that wing. The resultant model can be viewed as piecewise trapezoidal.

WingCoords2Loft reads two input files. "hrm2wingcoords.out" contains the wing cross section data. "wingcoords2loft.in" is an optional input file that specifies structural details such as rib and spar locations and mesh density.

It creates multiple output files. "wingcoords2loft.out" contains a *Loft* input file for the wing. "wingcoords2loft.spars" contains the x (axial) coordinate of the spar roots in feet.

When that input file is run, *Loft* creates NASTRAN, VRML, and Tecplot versions of the FEA model. *Loft*'s region mode is used to create additional files that are used to automate analysis of the model. "upperskinelems.txt" contains a list of elements on the wing upper skin. It also contains the total wing planform area. If the `weight` parameter is used in wingcoords2loft.in, then a smeared pressure value is printed that will produce 25% of that weight as lift when applied to the listed skin elements. "lowerskinelems.txt" is a similar file containing the lower skin elements. "rootnodes.txt" contains a list of nodes at the centerline. It is intended to be used to automate boundary condition application. "rootprops.txt" contains a list of the NASTRAN physical and material properties used on the root spars.

hrm2wingcoords.out

This file contains the wing cross section data. Note that for the purposes of this program, the normal NASA coordinate system is used: x is axial (chordwise), y is lateral (spanwise), and z is vertical. This is different than the base coordinates used for *Loft*. Also, the interleaving text lines shown in the example file are required to be present although they are not required to contain anything specific. Input units are feet. The models created by *Loft* are scaled to be in inches.

`double x0,y0,z0` = coordinates of wing reference location (leading edge root). Will be added to section x,y,z values below to produce true positions of wing nodes. Should be 0,0,0 if section x,y,z's are absolute positions. Units are feet.

`double Wfuse` = half of average or maximum (selection based on AVG|MAX flag in input) width of vehicle fuselage along wing. This value is used to create non-skinned carry-through.

`integer N` = supplied number of wing sections. This could be slightly different than the N requested in the input due to curvature awareness, but the relationship between *N*input and *N*output should be monotonic; a lower value of *N*input should produce a lower value of *N*output.

N lines of `double Xle,Yle,Zle,Xte,Yte,Zte,Tmax` where

`Xle` = x location (axial) of section leading edge

`Xte` = x location of section trailing edge

`Yle,Ytz` = y location (span) of leading/trailing edge section nodes

`Zle,Zte` = Z locations (height) of leading/trailing edge nodes. This will affect *Loft* positioning and wing twist

`Tmax` = Maximum thickness of wing section in inches. *WingCoords2Loft* will convert this to percent thickness to generate an approximate NACA airfoil section.

Example hrm2wingcoords.out file

```
Wing Reference Coordinates
50.0 0.0 3.0
Maximum Fuselage Half Width
0.75
Number of Sections
4
Section Details (X,Y,Z)le,(X,Y,Z)te, Tmax
0.0 0.75 0.0 5.0 0.75 0.0 0.2
3.0 15.0 0.0 8.0 15.0 0.0 0.4
6.0 25.0 0.0 12.0 25.0 0.0 0.2
10.0 40.0 0.0 15.0 40.0 0.0 0.2
```

wingcoords2loft.in

This file contains the information on desired structural details for the model. It is optional. If it is not present, default values are used. As with *Loft* itself, all parameters in this file are also optional. Again, default values will be used for any non-specified parameters.

As with *Loft*, the user can either specify a rib/spar count or give exact positions but not both. Giving a rib/spar count will result in that many evenly distributed ribs or spars. (e.g., an input of "`nspars 2`" will give the exact same result as "`sparpos 0.3333`" and "`sparpos 0.66666.`") Rib and spar positions are specified in percentages of span and chord. The two styles of rib/spar specification should not be mixed. Using both won't break things for either code but may result in unexpected outcomes. In both codes only the last style of specification will be used by the code. Earlier parameters will have no effect. Unlike the default behavior of *Loft*, ribs are not automatically created at 0 and 100% span; they will need to be specified in this file (using an `nribs` value of 2 will create just the 0 and 100% ribs.).

Parameter List (can be specified in any order):

`Nribs` (default 2): number of evenly spaced ribs to create

`Nspars` (default 0): number of evenly spaced spars to create
`ribpos` (no default): percent span (0-100) location to create a rib
`sparpos` (no default): percent chord (0-100) location to create a spar
`mesh` : Finite element mesh density per unit length (higher values produce a denser mesh) for all three mesh directions. When used, the three specific parameters `meshthick`, `meshspan`, and `meshchord` are reset to this value.
`meshchord` (default 3.0): Finite element mesh density per unit length in the chordwise direction (higher values produce a denser mesh). Note that tapering of chord length and thickness across the span of the wing will not cause a change in mesh counts; there will be the same number of nodes along the tip rib as on the root rib. Example: a setting of 5 on a wing with a 5 unit long chord setting will result in approximately 25 nodes in the chordwise direction on both the top and bottom skin (the exact node count will depend on spar positions and integer math truncations). This is a real number not an integer and can be less than one if desired. This parameter changes the chordwise mesh distribution for the skins and ribs.
`meshspan` (default 3.0): Finite element mesh density per unit length in the spanwise direction. (See discussion above.) This parameter changes the spanwise mesh distribution on the skins and spars.
`meshthick` (default 3.0): Finite element mesh density per unit length in the thickness direction. (See discussion above.) This parameter changes the vertical mesh density of the ribs and spars. It has no effect on the wing skins.
`rotx` (default 0.0): specifies a desired rotation about the x (axial) axis (dihedral) of completed wing
`roty` (default 0.0): specifies a desired rotation about the y (spanwise) axis (angle of attack) of completed wing.
`rotz` (default 0.0): specifies a desired rotation about the z (vertical) axis of the completed wing.
`weight` (default 0.0): specifies vehicle weight. Used to compute pressure required on wing to support this weight. A line of text specifying that pressure is added to the upper and lower skin element output files. This pressure is sufficient to support one quarter of the specified weight on each of the upper and lower wing surfaces.
`mergetol` (default 0.02) specifies tolerance for *Loft*'s node equivalence operation. Any nodes that are at the specified value or closer will be merged together.
`minthick` (default 1) integer value specifying minimum percent thickness for wing sections.
`naca` (default "00XX"): specifies the NACA 4 or 5 digit airfoil series to use for the wing. The last two digits represent the wing thicknesses and are replaced at each section by the value derived from the geometry information.
`halfwing`(default: off): Flag to turn on generation of just the top or bottom half of the wing. Used primarily for vertical tails on the symmetry lines of a half vehicle. Values are "off," "on," "bottom," and "top." ("top" and "on" are the same).
`wingside` (default: starboard) Flag to control which side of the vehicle to build the wing for. Values are "starboard," "port," "right," and "left" (starboard = right, port = left).

Example wingcoords2loft.in file:

```
sparpos 25.
sparpos 45.
sparpos 65.
ribpos 0.
ribpos 30.
ribpos 60.
ribpos 100.
mesh 0.8
naca 00XX
```

```
weight 25000.
```

wingcoords2loft.out

Running *WingCoords2Loft* will produce this output file. This file is a *Loft* input file for the specified wing. Running it with *Loft* will produce FEA models of the wing.

upperskinelems.txt

This file contains a list of elements on the wing upper skin. It also contains the total wing planform area. If the `weight` parameter is used in wingcoords2loft.in, then a smeared pressure value is printed that will produce 25% of that weight as lift when applied to the listed skin elements.

Example partial upperskinelems.txt file:

```
These are upper skin elements.

Planform area is 36666.497808 square inches.

Constant pressure for 5250.000000 of lift is -0.143182 psi.
Region Element Listing
   i   node1 node2 node3 node4 matprop physprop
    13     2     9     8     1        1        3
    14     4    11     9     2        1        3
    15     6    13    11     4        1        3
    16    15    21    13     6        2        3
```

lowerskinelems.txt

This file contains a list of elements on the wing lower skin. It also contains the total wing planform area. If the `weight` parameter is used in wingcoords2loft.in, then a smeared pressure value is printed that will produce 25% of that weight as lift when applied to the listed skin elements.

rootnodes.txt

This file contains a list of nodes at the centerline. It is intended to be used to automate boundary condition application.

Example rootnodes.txt file:

```
$ These are the wing centerline nodes to have BC applied.
GRID          49        6.3728E2-9.6E-134.0013E1
GRID          50        6.3725E2-9.4E-135.2924E1
GRID          51        8.0024E2-1.2E-124.1210E1
GRID          52        8.0024E2-1.2E-125.2714E1
GRID          53        9.6320E2-1.5E-124.1476E1
GRID          54        9.6324E2-1.5E-124.8343E1
```

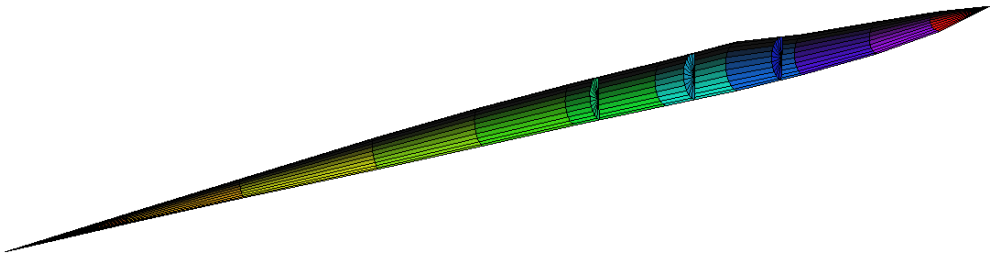
wingcoords2loft.spars

This file contains the x (axial) coordinate of the spar roots in inches.

rootprops.txt

This file contains a listing of the properties used in the spar root.

FuseCoords2Loft



FuseCoords2Loft is a similar utility program that generates a *Loft* input file for a vehicle fuselage. The input files describe the cross-sectional dimensions of the fuselage at various stations and optionally the location of desired bulkheads.

FuseCoords2Loft reads two input files. "hrm2fusecoords.out" contains the cross-sectional dimensions at various stations. "fusecoords2loft.in" is an optional file that contains structural model details such as bulkhead locations and mesh density.

The program writes a *Loft* input file to "fusecoords2loft.out."

When the *Loft* input file is run, *Loft* creates NASTRAN, VRML, and Tecplot versions of the FEA model. Region mode commands are included that create a list of the requested bulkheads and their NASTRAN property IDs in "bulkprops.txt." This information is used to tell NASTRAN how to "glue" the wing spars to the appropriately positioned bulkheads.

hrm2fusecoords.out

This file contains the fuselage cross section data. Note that for the purposes of this program, the normal NASA coordinate system is used: x is axial (chordwise), y is lateral (spanwise), and z is vertical. This is different than the base coordinates used for *Loft*. Also, the interleaving text lines shown in the example file are required to be present although they are not required to contain anything specific. Input units are feet. The models created by *Loft* are scaled to be in inches.

`double x0,y0,z0` = coordinates of fuselage reference location (nose).

`integer N` = supplied number of fuselage sections. This could be slightly different than the N requested in the input due to curvature awareness, but the relationship between *N*input and *N*output should be monotonic; a lower value of *N*input should produce a lower value of *N*output.

N lines of `double x,z,Rhorz,Rvert`, where

- x = axial station of section
- z = vertical station of the section center
- Rhorz = horizontal radius of fuselage at that station
- Rvert = vertical radius

Example hrm2fusecoords.out file:

```
Fuselage Reference Coordinates
50.0 0.0 3.0
Number of Sections
6
```

```
Section Details x,z,rhorz,rvert
0.0 0.1 0.0 0.0
0.5 0.0 .08 .08
2.0 0.1 .1 .12
4.0 -0.1 .1 .1
5.5 -0.01 .08 .08
6.0 0.1 0.0 0.0
```

fusecoords2loft.in

This file contains the information on desired structural details for the model. It is optional. If it is not present, default values are used. As with *Loft* itself, all parameters in this file are also optional. Again, default values will be used for any non-specified parameters.

Parameter List (can be specified in any order):

`Mesh` : Finite element mesh density per unit length (higher values produce a denser mesh) for both mesh directions. When used, the two specific parameters `meshcirc` and `meshaxial` are reset to this value.
`MeshAxial` (default 3.0): Finite element mesh density per unit length (higher values produce a denser mesh) in the axial direction.
`MeshCirc` (default 3.0): Finite element mesh density per unit length (higher values produce a denser mesh) in the circumferential direction.
`bulkhead` (default none): Specifies the name and absolute axial position of a requested bulkhead. A corresponding entry listing its assigned property id will be written to *bulkheadlist.txt*. Bulkheads can be specified in any order. *FuseCoords2Loft* will sort them and create them.

Example: bulkhead mainwing 28.75

`rotx` (default 0.0): specifies a desired rotation about the x (axial) axis of completed fuselage. (roll)
`roty` (default 0.0): specifies a desired rotation about the y (spanwise) axis of completed fuselage. (pitch)
`rotz` (default 0.0): specifies a desired rotation about the z (vertical) axis of completed fuselage. (yaw)
`curve` (default `sc`): specifies a *Loft* curve name to be used for the fuselage.

Example fusecoords2loft.in file:

```
meshaxial .1
meshcirc  1.
bulkhead glue1 51.588333
bulkhead glue2 62.863333
bulkhead glue3 74.138333
```

fusecoords2loft.out

Running *FuseCoords2Loft* will produce this output file. This is a *Loft* input file that generates the specified fuselage. Running it with *Loft* will produce the FEA models.

bulkprops.txt

This output file contains the NASTRAN property IDs for the requested bulkheads.

Example bulkprops.txt file:

```
$ These are the fuselage bulkheads and their propeties.
$ Loft physical property 100006 is mapped to the following Nastran p- cards
$ Pset: "glue1" will be imported as: "pshell.170000"
PSHELL    170000  100000  1.0000  100000          100000
```

```
$ Loft physical property 100008 is mapped to the following Nastran p- cards
$ Pset: "glue2" will be imported as: "pshell.190000"
PSHELL    190000  100000  1.0000  100000          100000
$ Loft physical property 100010 is mapped to the following Nastran p- cards
$ Pset: "glue3" will be imported as: "pshell.210000"
PSHELL    210000  100000  1.0000  100000          100000
```

Chapter 7: Command & Object Reference

Alphabetical Command List

`Clone` – Create one or more duplicates of the current stack. These are positioned evenly rotated around the specified axis. For example, "`clone x 1`" will produce one duplicate rotated 180 degrees around the x axis from the original. See similar `mirror` command for a slightly different result. Note: this operation creates stored stacks called `clonetempN` as part of its functionality, which could overwrite a user stored stack if the same name were used.

usage: **`clone`** <*axis*> <*number*>
defaults: <axis>=x, <number>=1
example: `clone y 2`

`Curve` – Define a user curve

usage: **`curve`** <*type*> <*mnemonic*>
type = "interpolated," "compound," "lofted"
mnemonic = name for the curve
example: `curve compound 3lt`

`Define` – Define a variable

This command allows the user to define a named variable to be used later in the input deck. The dollar symbol, "$," is used to invoke a variable and tell *Loft* to replace the text with the previously specified value.

usage: **`define`** <*name*> <*value*>
variable usage example: `length $mydimension`
example: `define mydimension 5.6`

`End` – End current *Loft* run (optional). Note that if used in an `include` file, the program will still stop and not return to reading the previous input file.

usage: **`End`**

`GoTo` – Move input file reading to another point in the file. Argument is a unique integer that matches the number used in a `LineLabel` line. This labeled line can either be anywhere in the main input file or in a current or previously read `include` file. A line in an `include` file that is referenced after the `goto` command cannot be located. If a text label is desired, use a previously defined variable with a descriptive text name. Note that if variables/math are used for any labels that are later in the file, those math calculations will be performed using the present values of the variables. Therefore, define these variables before use as a label or in a `goto` and generally avoid the use of any variable that changes (e.g., system variables like `@transz`.)

usage: **`GoTo`** <*unique integer*>
examples:
```
goto 100
goto $tank_generation_section
```

Ideas – Indicate I-Deas version for output

This command only affects which datasets are used in any I-DEAS universal files that are written after the command is used. It does not affect *Loft*'s internal data. Thus, it is possible to write different output files with different I-DEAS versions for the same data.

usage: **Ideas** <*version*>
version = 8 or 9
default: 9
example: `ideas 8`

If – Conditional program control command. If the argument is zero or approximately zero (absolute value < 0.0001), the test is false and the next input line is skipped. If the argument is non-zero, the test is true and the next input line will be executed. The logical operators "<," ">," "=," ">=," "<=," and "!=" may be used, or any other combination of *Loft* variables and math that will produce a true or false result (non-zero or zero). When used in combination with the `goto` command and a counter variable, a loop functionality can be created.

usage: **If** <*test*>
example: `if $i > 5`

Include – Read input from another file and then return to the previous file once the second file is completed. Note that since this is a command, any object that is in the process of being defined will be generated before the new file is read. Multiple levels of include are allowed. *Loft* will stop execution if the specified file is not found.

usage: **Include** <filename>
example: `include moreloftstuff.txt`

LineLabel – Assign a label to a location in the input file that *Loft* can seek to using the `goto` command. The command's argument is a unique integer that is not required to be sequential with other line labels. If a text label is desired, a previously defined variable with a descriptive name can be used. Note that if variables/math are used for any labels that are later in the file, those math calculations will be performed using the present values of the variables. Therefore, define these variables before use as a label or in a `goto` and generally avoid the use of any variable that changes (e.g., system variables like `@transz`.)

usage: **LineLabel** <*unique integer*>
examples:
```
linelabel 100
linelabel $tank_generation_section
```

List – Output various lists to the screen. This command is intended for model debugging purposes. The options "groups" and "marks" are synonymous. The "input" option will output the current *Loft* input lines as modified by any `include` commands already read and any commands that act as macros like `mirror` and `clone`. The "input" option does not get included when "all" is chosen.

usage: **List** <*type*>

type = “ccurves,” “icurves,” “lcurves,” “stacks,” “variables,” “groups,” “input,” “marks,” “mprops” (material properties), “pprops” (physical properties), “ribs,” “spars,” “mesh” (gives various data counts), “rbes” (lists RBE, BC, MASS, TEMP, FORCE, and PRESS objects) or “all”
default: (none)
example: `list stacks`

`MergeTol` – Distance for considering nodes to be identical. These nodes are merged by removing higher numbered duplicates and replacing references to them with references to the lower numbered, remaining, node. This merging is done at various points in wing generation as well as when adding new objects to the current stack.

usage: **`MergeTol`** <*distance*>
default: 0.001
example: `mergetol 0.01`

`Mirror` – Create a duplicate, but reversed, object on the other side of a specified axis. See similar `clone` command for a slightly different result. The mirror command can be used to convert from a half-vehicle model to a full vehicle. Note: this operation creates a stored stack called “mirrortemp” as part of its functionality, which could overwrite a user stored stack if the same name were used.

usage: **`mirror`** <*axis*>
default: x
example: `mirror x`

`Move` – Rotate, translate, scale, warp, split and/or flip the full stack
Note that, unlike the rotation and translation parameters for an individual object, results of this command *do* depend on the order of the parameters – each operation is executed following each parameter.

Rotation and translation values are set with the `rotx, roty, rotz, transx, transy`, and `transz` parameters just like those allowed for single objects. (Note that these are absolute translations and rotations, not relative to any previous settings.) In addition, the `scalex, scaley`, and `scalez` parameters can be used to adjust the size of the current stack.

There are also six “warp” parameters that distort part of the stack. The six parameters are `warppx, warpnx, warppy, warpny, warppz,` and `warpnz`. The two letters after the “warp” prefix indicate the region of action of the warping. Thus, `warppx` will scale the parts of the stack that are in the positive x region and leave the nodes where x<=0 alone. These six parameters all take three values that are the amount to scale that region in the x, y, and z directions. So, a `move` parameter that said “`warpnz 1.0 2.0 1.0`” would double the y coordinates of all nodes that started with z less than 0. Use of the rotation and translation parameters before and after a warp operation allows fine-tuning of the area to be affected. The `warp` options are intended to be used to make shapes such as the fuselage for a lifting body. Care should be taken with the scale factors and the object mesh options to keep element aspect ratios reasonable.

Gradient warps are also possible with the six gwarp parameters. These are `gwarppx, gwarpnx, gwarppy, gwarpny, gwarppz,` and `gwarpnz`. They work identically to the constant `warp` parameters above, but the distortion increases linearly from zero distortion at the axis to the specified values at a unit distance from the axis and higher further away from the axis. So, a parameter like “`gwarppy 2.0`

`1.0 1.0`" would double the x coordinates of any node at y equals 1 and quadruple the x coordinate of any node at y equals 2.

The `flip` parameter reverses the node ordering for panel elements, thus changing the direction of their normal vectors. It takes no arguments.

The `split` parameter breaks each quadrilateral element into two triangular elements with node ordering going from 1-2-3-4 to 1-2-4 and 3-4-2.

usage: **`Move`**
example:
```
Move
Scalex 0.5
Scaley 0.2
Transx 30.5
Roty 33.3
Warpnz 1.0 2.0 1.0
Gwarppy 2.0 1.0 1.0
Flip
Split
```

`Nastran` – Controls NASTRAN format output options. A minimal case control block is written to every full NASTRAN format file. If either or both "spc" or "load" setids are given, then a simple subcase is added to the control block using those ids.

usage: **`Nastran`** <*parameter*> <*value*>
examples: `nastran grid 8, nastran cylx, nastran spc 1000`

List of Nastran command parameters:
Grid = number of columns used in grid cards. Values are 8 or 16. Default is 8.
Cylx, cyly, cylz = flag to turn on cylindrical coordinate output. Last letter indicates the non-transformed axis (axial direction). Coordinates are converted on the fly as the NASTRAN file is written; the internal *Loft* coordinates are not transformed.
Cart = flag to restore Cartesian coordinate output, which is the default setting.
hmcom/nohmcom = flags to turn on/off (off by default) a limited set of HyperMesh style comments to allow model importation into HyperX
spc = setid for boundary condition cards in a simple case control subcase
load = setid for load cards in a simple case control subcase
sol = solution number for the NASTRAN run (default = 101)
subcase = subcase number for NASTRAN run (default = 1)
thick = dummy panel thickness written to PSHELL cards (default = 0.1)

`New` – Deletes current stack from memory

By default each new object's mesh is added to the previous meshes - creating a *stack*. This command starts a new stack (presumably after issuing a `store` or `write` command to save the previous one.) All defaults are reset to their initial values.

usage: **`New`**

Null – No effect command. The main use of this command is to force the completion of the current object and update all of the system variables to reflect its creation.

usage: **Null**

Object – Create a meshed object

usage: **Object** <*type*> <*name*>
type = type of object to create, e.g., dome, section, wing, tank, etc.
name = descriptive name of the object, 40 characters or less, used to mark elements
example: `object dome LOX Tank Aft Dome`

Offset – Define index offset for written meshes. This value can be set by output file type or globally. If no type is specified, all the offset for all types are set. Note that the offset value is not used for VRML or Tecplot output as nodes receive their index implicitly by their order in the output file.

usage: **Offset** <*type*> <*value*> or **Offset** <*value*>
type = output file type effected. Valid types are "nastran," "ideas," "abaqus," and "region." The "region" type affects the output of the query parameter in region mode.
v*alue* = amount to offset the indices. *Loft* internal indices start from 0. NASTRAN, for instance, does not support an element or node numbered as 0, so a value greater than zero should be specified. Default for ideas and abaqus is 1. Default for nastran and region is 100000.
examples: `offset nastran 100000`
`offset 50`

Quality – Performs mesh quality checks on the current stack and prints a report.

usage: **Quality**

Read – Reads a supported format mesh into *Loft* as a new object

This command allows the import of a variety of externally generated meshes into *Loft*. This is an extremely simplified process focusing on capturing nodes and connectivity. All property information is lost. All elements are converted to simple 4-node rectangles, 3-node triangles, or 2-node bars. Unusual element types are very likely to fail.

usage: **Read** <*file type*> <*file name*>
file type = type of file to read: vrml, abaqus, or nastran
file name = Name of file to be read
example: `read nastran myinput.bdf`

Recall – Copies a clipboard stack into the active stack

This command copies a previously stored stack (see **store** command) from the temporary stack clipboard back into active memory. The copy on the clipboard is not deleted and can be recalled any number of times. Multiple recalls of the same complex object can take some time to accomplish, as the various merging operations for items with the same name can be slow. A `recall` operation does not change any default geometric values.

usage: **Recall** <*name*>
example: `recall External Tank`

Region – Enter region mode.

The `region` tool set is a powerful feature of *Loft* that allows the user to query or modify a section of the current stack. Regions are inherently temporary constructs, but their effects may include permanent changes to the mesh by deleting parts, changing property assignments, etc. Regions can also be used to query statistics on the mesh and produce reports. The `region` mode has a long list of parameters that are described in chapter 3 of this manual. These abilities partially overlap the **list** and **subtract** commands.

usage: **Region**

Reset – Reset defaults to initial values, without deleting the current stack.

usage: **Reset**

Store – Move the current stack to a temporary clipboard and start over, **reset**-ing all default values.

The current stack is assigned the supplied name and stored in memory. The active stack that commands operate on is cleared and values are set back to the initial defaults. Any number of stacks can be simultaneously copied to the clipboard.

usage: **Store** <*name*>
example: `store External Tank`

Subtract – Delete all nodes that fall within a specified volume of space. Any elements that use these nodes will be deleted as well. Quads (4-node elements) that lose one node will be converted to triangles. Volumes are specified by use of simple three dimensional shapes including spheres, cylinders, and boxes. Cylinders are aligned with an axis and are infinite in length. *Warning*: Any beams whose alignment nodes fall in the specified volume, even if the beam end points themselves do not, will also be deleted. A similar, but not identical, effect can be produced by the region mode "cut" operation.

usage: **Subtract** <*type*> <*center of volume*> <*dimensions of volume*>
type = "sphere," "xcyl," "ycyl," "zcyl," "box"
center = x, y, z coordinate of center of volume
dimensions = radius for sphere and cylinders,
= xlength, ylength, zlength for box.
example: `subtract sphere 10. 20. 25. 5.`

Units – Specify unit set. (default = inch)

Loft is unit-less. For NASTRAN output this command affects the magnitude of the values used on property or material cards. For I-DEAS universal file output, this command just changes which units are indicated for any files written after the command.

usage: **Units** <*length unit*>

length unit = “foot,” “feet,” “inch,” “cm,” “meter”
example: `units meter`

`Vrml` – Control vrml color output

Selects if the vrml output mesh contains color information and if so, which color pallet to use. Options listed below in parenthesis are synonyms of each other. The forward option produces a more red/blue picture. The backward option produces more yellow/pink.

usage: **`Vrml`** <*option*>
option = (“off,” “no”), (“forward,” “on”), (“reverse,” “backward”), “rainbow,” “primary”
default: primary
example: `vrml reverse`

`Write` – Write current mesh to an output file.

usage: **`Write`** <*file type*> <*file name*>
file type = type of file to save: “custom,” “vrml,” “unv,” “abaqus,” “tecplot,” “stl,” or “nastran”
file name = Name of output file
example: `write vrml rocket.wrl`

VRML and Tecplot files containing small portions of a large model will still be large files. This is due to the way nodes are implicitly indexed in the files based on their definition order rather than an explicit index number. This means that every node needs to be included in the file even if there are only a small number of panels in the partial model.

STL (STereo Lithography) is a 3D printing file format. *Loft* will output a readable mesh for all triangles and quads in the model, but that model will not necessarily be manifold/watertight (in fact none of the models in this manual are). Some additional effort with adding endcaps or suppressing internal detail can produce a printable model. Alternatively, some third-party tools (for example, Microsoft’s 3D-Builder) may be able to make the model watertight and printable.

Object Types and Parameters

Common Parameters

All object types except the individual beam and rbe objects use these parameters. They control positioning, rotation, distortion, alignment, and group marking.

`rotx` – angle to rotate object about its origin's x axis in degrees (absolute)
default = 0, or last value specified

`roty` – angle to rotate object about its origin's y axis in degrees (absolute)
default = 0, or last value specified

`rotz` – angle to rotate object about its origin's z axis in degrees (absolute)
default = 0, or last value specified

`transx` – distance to translate object's origin from the global origin in the x direction
default = 0, or endpoint of previous section (domes do not update this default)

`transy`– distance to translate object's origin from the global origin in the y direction
default = 0, or endpoint of previous section (domes do not update this default)

`transz`– distance to translate object's origin from the global origin in the z direction
default = 0, or endpoint of previous section (domes do not update this default)

`relrotx` – angle to rotate object from its default position about the x axis in degrees.
default = 0

`relroty` – angle to rotate object from its default position about the y axis in degrees.
default = 0

`relrotz` – angle to rotate object from its default position about the z axis in degrees.
default = 0

`relx` – distance to translate object's origin from its default position in the x direction.
default = 0

`rely` – distance to translate object's origin from its default position in the y direction.
default = 0

`relz` – distance to translate object's origin from its default position in the z direction.
default = 0

`flip` – change the element normal direction to point inward rather than outward. This parameter takes no argument. It must be specified for each object where flipping is desired (it does not change the default orientation). This parameter is not available for the block object type.

`warppx`, `warppy`, `warppz`, `warpnx`, `warpny`, `warpnz` – distort the part of the object in the region specified by the last two letters (p means positive, n means negative, and x, y, and z, are the coordinate axes) by the specified three values. Only one `warp` or `gwarp` parameter may be specified per object.

default: (no warp)

`gwarppx`, `gwarppy`, `gwarppz`, `gwarpnx`, `gwarpny`, `gwarpnz` – distort the part of the object in the region specified by the last two letters (p means positive, n means negative, and x, y and z, are the coordinate axes) by the specified three values. Scaling of the original coordinates varies linearly with the node's original distance from the specified axis. Only one `warp` or `gwarp` parameter may be specified per object.

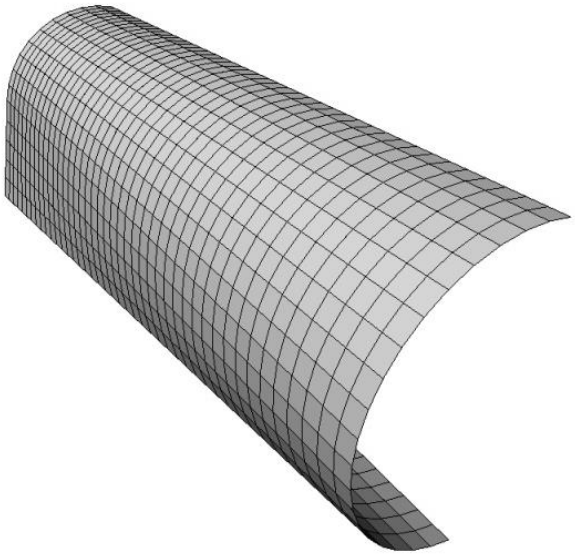
default: (no warp)

`mark` – add a label to a group of nodes or elements. Items can have as many different labels as desired. Marks have limited uses. They can be used to sort elements in the region command and will be output as groups when an I-DEAS output file is created. Support for NASTRAN SET grouping can be enabled by removing a comment in "`nastran.c`." The Mark parameter takes two arguments: the group type (node, element or rbe) and the group name. A marked group can contain either nodes, elements, or rbe class objects.

Example: `mark element OML`

default: none

Section/Truss



A **section** is a 3-D object made by interpolating between two 2-D curves. Curved transitions may be generated using the taper parameter. The origin of the object is the center point of curve 1 (which for semi-curves is on the axis of symmetry).

A **truss** is a standalone object made of bars or beams that generates ring frames at the 0 and 100 percent ends of the object and diagonal cross supports between them. The number of truss nodes used is set with the `tnodes` parameter. Strut endpoints are evenly distributed but attached to the closest existing node on the 0/100 end rings. Thus, a higher value or very careful selection of `nodes_circ` may produce a cleaner truss. Also, a value of 2 for `nodes_axial` is recommended, but not required. A value higher than 2 may produce unsupported degrees of freedom as well as potentially curved struts due to the interpolation between the two end shapes.

Parameter List

Note that most axial direction defaults do not change to match earlier inputted values (the `transx` parameter is an exception).

`curve1` – mnemonic for first curve (see curve library)
default = sc, or last curve used

`curve2` – mnemonic for second curve (see curve library)
default = sc, or last curve used

`c1_xscale` – factor to scale x dimensions of curve 1 by
default = 1, or last x scale

`c1_yscale` – factor to scale y dimensions of curve 1 by
default = 1 or last y scale

`c2_xscale` – factor to scale x dimensions of curve 2 by

default = 1, or last x scale

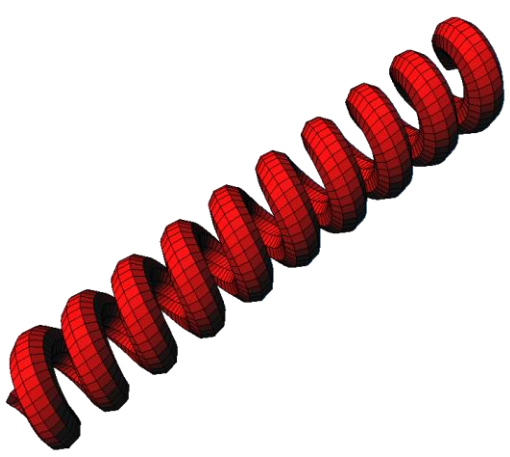
`c2_yscale` – factor to scale y dimensions of curve 2 by
default = 1, or last y scale

`c1_xoffset` – distance to horizontally translate curve 1
default = 0, or last x offset

`c1_yoffset` – distance to vertically translate curve 1
default = 0, or last y offset

`c2_xoffset` – distance to horizontally translate curve 2
default = 0, or last x offset

`c2_yoffset` – distance to vertically translate curve 2
default = 0, or last y offset



`c1_rotation` – angle in degrees to rotate end 1 about the y axis. This parameter is intended to make toroidal or helixical shapes. For instance, setting one end to zero, the other to 3600 (ten 360 rotatations) and the yoffset on an end to something greater than 10 times the yscale will produce a 10 revolution spring.
default = 0, or last rotation

`c2_rotation` – angle in degrees to rotate end 2 about the y axis. This parameter is intended to make toroidal or helixical shapes.
default = 0, or last rotation

`c1_s` – scheme to use to distribute nodes circumferentially along `curve1`. Values may be "global," "local," or "copy." A "global" distribution spaces nodes evenly along the circumference of the un-scaled curve. A "local" distribution spaces nodes evenly along each arc of a user-defined piecewise curve (interpolated or compound). This has the effect of positioning nodes at each joint between child arcs. A "copy" distribution uses the node spacing of the other end of the section in order to produce less twisted elements. If both ends of the section are set to "copy," a "global" distribution will be used.
default = "global," or previous `c2_s`

`c2_s` – scheme to use to distribute nodes circumferentially along curve 2. See discussion of `c1_s` above.
default = "global," or previous `c2_s`

`length` – length of section
default = 1

`radius` – rotation radius used when `c1_rotation` or `c2_rotation` are non zero.
default = 1, or last radius

`nodes_circ` – number of finite element nodes to use in the circumferential direction
default = 10, or last value specified

`nodes_axial` – number of finite element nodes to use in the axial direction
default = 10

`components_circ` – number of different material props to use in circumferential direction. Use of this parameter overrides the `circ_cpos` list of component edge positions and creates evenly distributed component edges (e.g., specifying 3 components will produce edges at 33 and 67 percent of circumference)
default = 1, or last value specified

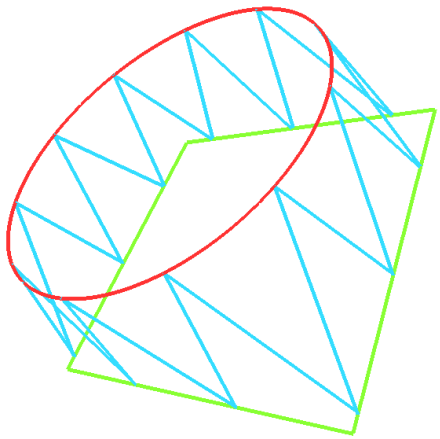
`components_axial` – number of different material properties to use in axial direction. Use of this parameter overrides the `axial_cpos` list of component edge positions and creates evenly distributed component edges (e.g., specifying 3 components will produce edges at 33 and 67 percent of length)
default = 1

`axial_cpos` – position of one axial component edge in percent. Values can be the word "reset" to remove the current list of positions, or between 0 and 100 to set the percentage where elements created after that location will be in a new component. Multiple positions can be set. Use of this parameter overrides the `components_axial` setting and vice versa.

`circ_cpos` – position of one circumferential component edge in percent. Values can be the word "reset" to remove the current list of positions, or between 0 and 100 to set the percentage where elements created after that location will be in a new component. Multiple positions can be set. Use of this parameter overrides the `components_circ` setting and vice versa.

`taper` – This setting controls how quickly curve1 transitions to curve2. This taper option will have significant effect only if the scales and/or offsets of the two end curves are significantly different. Pictures of these taper types are shown in the library section at the end of the chapter 7. Those pictures show a section that transitions between two semi-circles of different size and offset.
For the `linear` option, value has no effect. For the `cosine` option, value is the number of half waves. For the `power` option, value is the exponent of the interpolation curve (1.0 gives linear).

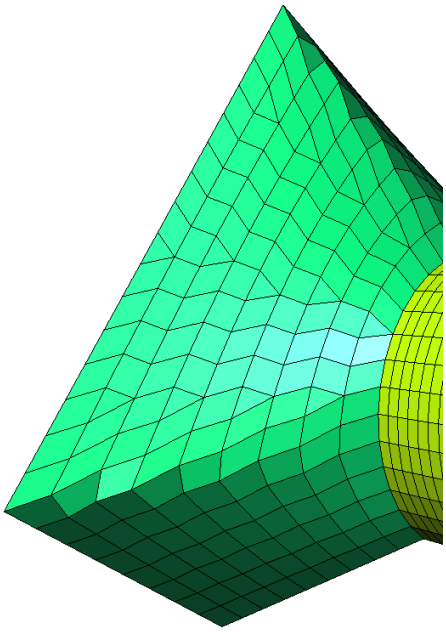
Usage: `taper` <*type*> <*value*>
Type = "linear," "power," "cosine"
Defaults: type = linear
value = 1.0



`tnodes` – number of truss endpoints to use

`type` – kind of 1-D object to generate for a truss object. Should be `beam`, `rod`, or `bar` (rod and bar are the same).
Default: beam

TSection



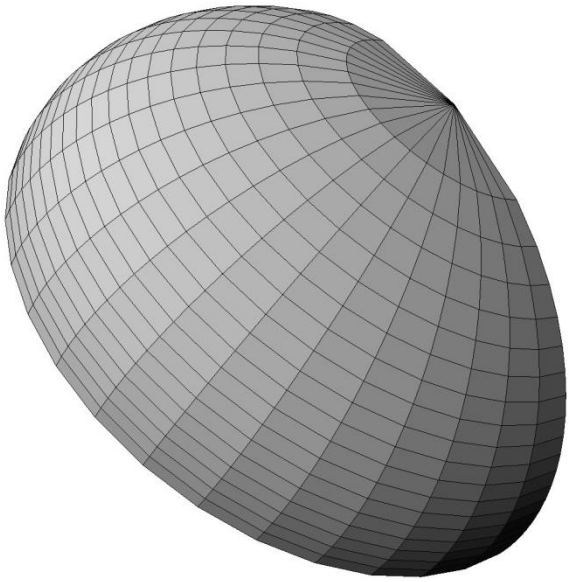
A TS**ection** is an under-development variation on the Section object type. This object allows the user to specify a different value of `nodes_circ` at each end of the section. This results in a number of triangular elements being created to gradually change from one node count to the other.

No TFrame object has been created to allow ring frames and longerons to attach to a TSection. A conventional Frame object may be used. It should stitch well along edges of the section but will generally not attach properly across the middle of a TSection. If such a mid-frame is desired use multiple base objects to force straight element edges at the desired location.

The TSection object uses the same parameters as the Section object with one addition:

Additional Parameter List
`nodes_circ2` – number of finite element nodes to use in the circumferential direction at the second end of the section.
default = 10, or last value specified

Dome



A D**ome** is a 3-D object made by extruding a single 2-D curve to a single nose point. The origin of the object is the center point of curve 1 (which for semi-curves is on the axis of symmetry). Adding a dome object does not change the default position of the next object (unless a translation/rotation parameter is specified).

Parameter List

`curve1` – mnemonic for first curve (see curve library)
default = sc, or last curve used

`c1_xscale` – factor to scale x dimensions of curve 1 by
default = 1, or last x scale

`c1_yscale` - factor to scale y dimensions of curve 1 by
default = 1 or last y scale

`c1_xoffset` – distance to horizontally translate curve 1
default = 0, or last x offset

`c1_yoffset` – distance to vertically translate curve 1
default = 0, or last y offset

`c1_s` – scheme to use to distribute nodes circumferentially along curve 1. Values may be “global,” “local,” or “copy.” A “global” distribution spaces nodes evenly along the circumference of the un-scaled curve. A “local” distribution spaces nodes evenly along each arc of a user-defined piecewise curve (interpolated or compound). This has the effect of positioning nodes at each joint between child arcs. A “copy” distribution uses the node spacing of the other end of the section in order to produce less twisted elements. If both ends of the section are set to “copy,” a “global” distribution will be used.
default = “global,” or previous scheme

`length` – length of section
default = 1

`nodes_circ` – number of finite element nodes to use in the circumferential direction
default = 10, or last value specified

`nodes_axial` – number of finite element nodes to use in the axial direction
default = 10

`components_circ` – number of different material props to use in circumferential direction. Use of this parameter overrides the `circ_cpos` list of component edge positions and creates evenly distributed component edges (e.g., specifying 3 components will produce edges at 33 and 67 percent of circumference)
default = 1, or last value specified

`components_axial` - number of different material properties to use in axial direction. Use of this parameter overrides the `axial_cpos` list of component edge positions and creates evenly distributed component edges (e.g., specifying 3 components will produce edges at 33 and 67 percent of length)
default = 1

`axial_cpos` – position of one axial component edge in percent. Values can be the word "reset" to remove the current list of positions, or between 0 and 100 to set the percentage where elements created after that location will be in a new component. Multiple positions can be set. Use of this parameter overrides the `components_axial` setting and vise-versa.

`circ_cpos` – position of one circumferential component edge in percent. Values can be the word "reset" to remove the current list of positions, or between 0 and 100 to set the percentage where elements created after that location will be in a new component. Multiple positions can be set. Use of this parameter overrides the `components_circ` setting and vise-versa.

`taper` – mnemonic for taper schedule (see taper library)
default = elli

`droop` – mnemonic for droop schedule (see droop library)
default = line

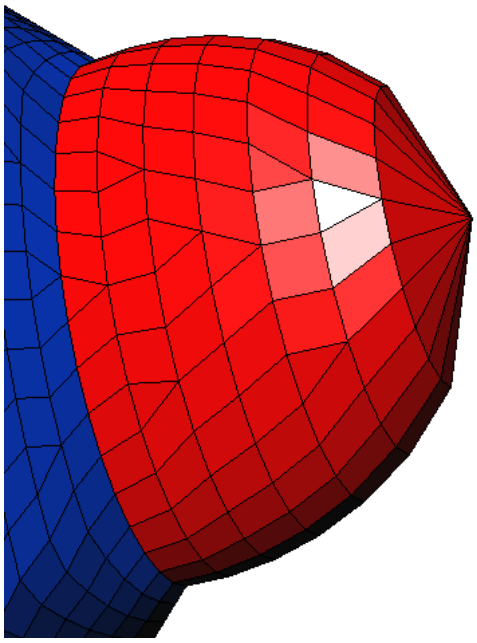
`zdist` – controls distribution of nodes axially. The value must be greater than zero and less than or equal to one. The lower the value specified the more the nodes are biased toward the dome nose. A value of one (the default) results in nodes being distributed linearly in the z direction. A value of 0.5 results in nodes spaced in such a way as to produce equal radial spacing when viewed from nose on.
The actual equation used is: $z_i = \texttt{length} * (\texttt{i/nodes\_axial})^{\texttt{zdist}}$
default = 1.0

`zdroop` – distance to droop nose point from centerline
default = 0

`param1, param2, param3` – additional parameters whose meanings vary depending on the value of the `taper` option chosen. Since the meaning may change from an exponent expected to be between zero and one to a radius that may be hundreds of inches, exercise care in the use of these values. These values

are reset to -1.0 after use. This indicates to *Loft* that the default value should be used. Thus, any desired parameters need to be set for each dome created. (see taper library at end of chapter 7).

TDome



A T**Dome** is an under-development variation on the Dome object type. This object allows the user to specify a different value of `nodes_circ` at each end of the section. This results in a number of triangular elements being created to gradually change from one node count to the other.

No TDframe object has been created to allow ring frames and longerons to attach to a TDome. A conventional DFrame object may be used. It should stitch well along edges of the section but will generally not attach properly across the middle of a TDome. If such a mid-frame is desired use multiple base objects to force straight element edges at the desired location.

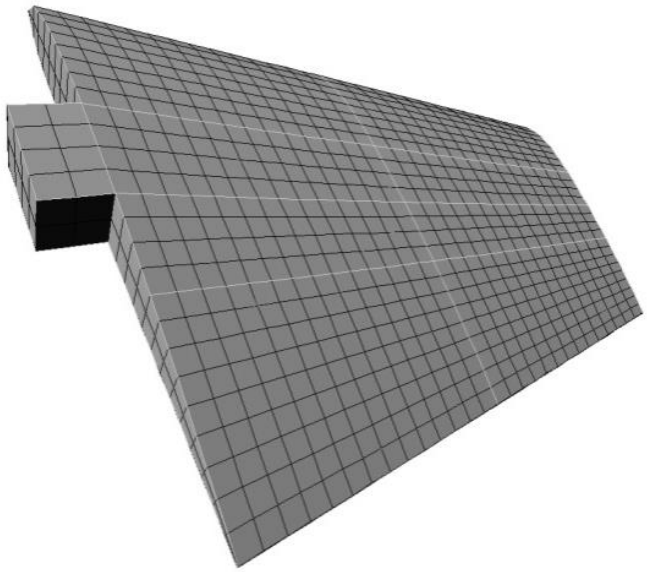
The TDome object uses the same parameters as the Dome object with one addition:

Additional Parameter List

`nodes_circ2` – number of finite element nodes to use in the circumferential direction at the nose end of the dome (the tip is a single node, and this value is the count of nodes in the row just before the nose).

default = 10, or last value specified

Wing



A **Wing** object is a 3-D object composed of panels that represent a lifting surface's skin, ribs, and spars. This object creates one trapezoidal planform lifting surface (a right wing, a tail, a winglet) per call. It allows the user to specify spar and rib positions and which spars to extrude to form the wingbox carry-through. Other optional settings allow wing twist, different airfoil shapes at the root, tip, top, or bottom and beam/bar stiffening of the ribs and spars. Partial generation of the wing in the chordwise direction (to support things like control surfaces) is also supported.

Beam stiffening is only partially implemented at this time. The beams are connected properly, but their alignment is not properly set. (They are all aligned with node 1.)

The object local origin is the leading edge root node.

The wing object supports two types of parameters: specific and generic. Generic parameters change one or more specific parameters. For instance, the generic `naca` parameter will change the values of the specific parameters `rootnaca`, `tipnaca`, `nacatop`, and `nacabot`. The main parameter list contains just the specific parameters. A separate list of generic parameters is given at the end of this object section. The effect of the two parameter types is read-order specific. Specifying "`naca 2015`" followed by "`root-naca 2212`" will result in the root using a 2212 airfoil and the tip using a 2015. If the `rootnaca` parameter was specified before the `naca` parameter then both the root and tip would use a 2015 airfoil. If the user desires to be more specific, the top and bottom shapes can be specified separately using the root-nacatop, rootnacabot, tipnacatop, and tipnacabot parameters. Also, in addition to specifying a 4- or 5-digit NACA airfoil shape, the user may also specify a biconvex airfoil with a desired thickness to chord ratio or any defined Loft curve (built-in, interpolated, or compound).

Historic note: *Loft* has had a large collection of different wing object types. To reduce confusion these have all been collected into one wing type using the same parameters and generation code. For the short term, the additional wing object types are still available to be used but are not documented and will eventually be eliminated. This has the advantage of only having to maintain one wing generation routine. (Most of the generic parameters are from the older, less powerful, wing object types.)

Parameter List (Specific and more specific)
`chord` – root chord length
default: 1

`span` – single wing span
default: 1

`taper` – ratio of tip chord length to root chord length
default: 1

`sweep` – leading edge sweep angle in degrees
default: 0

`rootnaca, tipnaca` – specific airfoil NACA designation (contains camber and thickness data) for wing root/tip. May be a NACA 4- or 5-digit airfoil specification, a biconvex airfoil with a desired thickness to chord ratio (default 0.1), or the name of any valid *Loft* curve (built-in, interpolated, or compound). For *Loft* curve use, since the normal definition of curves starts at (0,1), the x and y values of the curve will be swapped so that the curve is horizonal rather than vertical. The specified thickness to chord will be used to scale the now vertical x values of the *Loft* curve. Both the top and bottom surface of the root or tip are set by these parameters.
defaults: 2410
thickness to chord 0.1
Examples: `rootnaca 2030`
`rootnaca biconvex 0.20`
`rootnaca sc 0.10`

`nacatop, nacabot` – specific airfoil designation for the top/bottom of the wing. Each sets both the root and tip of either the top or bottom to that specification.

`rootnacatop, rootnacabot, tipnacatop, tipnacabot` – more specific airfoil designation parameters for the wing end surfaces separately.

`rootaoa` – root twist angle in degrees. Wing half-chord is the rotation axis, positive twist produces a higher section angle of attack (root up).
default: 0

`tipaoa` – tip twist angle in degrees. Wing half-chord is the rotation axis, positive twist produces a higher section angle of attack (tip up).
default: 0

`twist` – synonym for `tipaoa` parameter.

`rootvert` – vertical offset of wing root. Positive is up.
default: 0

`tipvert` – vertical offset of wing tip. Positive is up. Can be used to produce wing dihedral.
default: 0

`wingbox` – carry-through length. May be zero. At least 2 spars must be specified if a carry-through is desired. This value is always reset to zero after object generation, so any desired non-zero values must be set for each new object.
default: 0

`sparpos` – percentage of chord to place a spar. These can be specified in any order; the program automatically sorts them as they are read. If either of the words "reset" or "clear" is specified rather than a percentage, the current list of spars is deleted and the `boxfront` and `boxrear` parameters are reset to their default values. This reset option is needed because the lists of spars and ribs are kept as the default from one wing to the next.

`ribpos` – percentage of span to place a rib. Automatic ribs are created at 0 and 100 percent span and do not need to be specified by the user. These can be specified in any order; the program automatically sorts them as they are read. If either of the words "reset" or "clear" is specified rather than a percentage, the current list of ribs is deleted (with the 0 and 100 percent automatic ribs being immediately re-added). See the `notip` parameter if suppression of the tip rib is desired.

`boxfront` – spar number to extrude to make wingbox carry-through front (used only if the wingbox parameter is > 0). Numbering is based on proximity to the wing leading edge, not on the order that the `sparpos` parameters occur. This value is reset to the default if the list of spar positions is cleared.
default: 1

`boxrear` – spar number to extrude to make wingbox carry-through back (used only if wingbox parameter is > 0). Numbering is based on proximity to the wing leading edge, not on the order that the `sparpos` parameters occur. This value is reset to the default if the list of spar positions is cleared.
default: (last spar)

`meshchord` – finite element mesh density per unit length in the chordwise direction (higher values produce a denser mesh). Note that tapering of chord length and thickness across the span of the wing will not cause a change in mesh counts; there will be the same number of nodes along the tip rib as on the root rib. Example: a setting of 5 on a wing with a 5 unit long `chord` setting will result in approximately 25 nodes in the chordwise direction on both the top and bottom skin (the exact node count will depend on spar positions and integer math truncations). This is a real number, not an integer, and can be less than one if desired. This parameter changes the chordwise mesh distribution for the skins and ribs.
default: 3.0

`meshspan` – finite element mesh density per unit length in the spanwise direction. (See discussion above.) This parameter changes the spanwise mesh distribution on the skins and spars.
default: 3.0

`meshthick` – finite element mesh density per unit length in the thickness direction. (See discussion above.) This parameter changes the vertical mesh density of the ribs and spars. It has no effect on the wing skins.
default: 3.0

`sparstiff` – flag to turn on generation of stiffening bars/rods or beams at the top and bottom of the spars. Values are "off," "on," "beam," "bar," and "rod." ("on," "bar," and "rod" are all equivalent).
default: off

`ribstiff` – flag to turn on generation of stiffening bars/rods or beams at the top and bottom of the ribs. Values are “off,” “on,” “beam,” “bar,” and “rod.” (“on,” “bar,” and “rod” are all equivalent).
default: off

`halfwing` – flag to turn on generation of just the top or bottom half of the wing. Used primarily for vertical tails on the symmetry lines of a half vehicle. Values are “off,” “on,” “bottom,” and “top.” (“top” and “on” are the same).
default: off

`wingside` – flag to control which side of the vehicle to build the wing for. Values are “starboard,” “port,” “right,” and “left.” (starboard = right, port = left).
default: starboard

`notip` – flag to control generation of outboard (100% span) rib. This is useful when you are building up a compound wing of multiple trapezoidal sections and do not want a double rib at the junction. Values of “1,” “on,” or “true” will disable the wingtip rib generation. Values of “0,” “off,” or “false” will re-enable it. This flag is always reset to off after each wing generation.
default: off (wingtip rib is generated)

`nowbrib` – flag to control generation of the rib at the end of the wingbox carry-through. Generally this rib would fall on the centerline of the vehicle. Values of “1,” “on,” or “true” will disable the wingbox rib generation. Values of “0,” “off,” or “false” will re-enable it. This flag is always reset to off after each wing generation.
default: off (wingbox rib is generated)

`start` – percentage of chord length to start generating the object. Any spars that are specified at lower positions than this value are ignored. The start and stop parameters are used to generate partial wing objects (e.g., control surfaces).
default: 0

`stop` – percentage of chord length to stop generating the object. Any spars that are specified at higher positions than this value are ignored. The start and stop parameters are used to generate partial wing objects (e.g., control surfaces).
default: 100

`gen_up_skin` – flag to control the creation of the wing upper skin. Values are “on” and “off.” This flag is always reset to “on” after an object has been created.
default: on

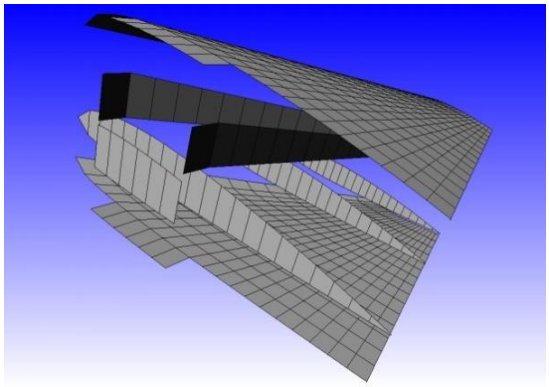
`gen_low_skin` – flag to control the creation of the wing lower skin. Values are “on” and “off.” This flag is always reset to “on” after an object has been created.
default: on

`gen_spars` – flag to control the creation of the wing spars. Values are “on” and “off.” Even when off, the other wing elements will be positioned to align with the spars that are specified in the object geometry. Thus, each part of the wing could be generated separately and merged to create the same mesh as if they were created together. This flag is always reset to “on” after an object has been created.

default: on

gen_ribs – flag to control the creation of the wing ribs. Values are “on” and “off.” Even when off, the other wing elements will be positioned to align with the ribs that are specified in the object geometry. Thus, each part of the wing could be generated separately and merged to create the same mesh as if they were created together. This flag is always reset to “on” after an object has been created.

default: on



Expanded view of Wing parts created by sequential use of each of the gen_XXX flags

Parameter list (Generic)

mesh – finite element mesh density per unit length (higher values produce a denser mesh). This is a global setting for the entire object. When used, the three specific parameters meshthick, meshspan, and meshchord are reset to this value.

naca – airfoil NACA designation (contains camber and thickness data). When used, the more specific rootnacatop, rootnacabot, tipnacatop, and tipnacabot are reset to this value.

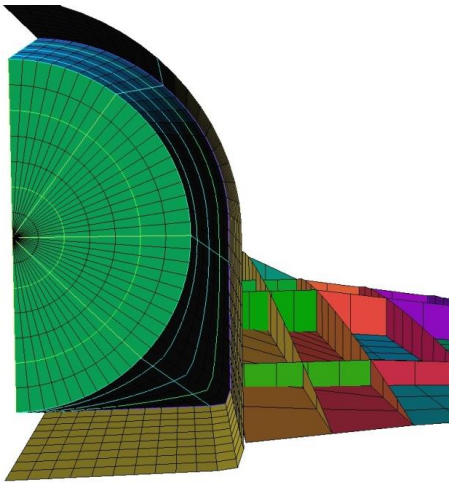
nribs – number of wing ribs, including root and tip. Must be greater than or equal to 2. When used, the current ribpos parameter settings are erased and the specified number of new evenly spaced ribs are placed in the ribpos list.

nspars – number of wing spars. When used, the current sparpos parameter settings are erased and the specified number of new evenly spaced spars are placed in the sparpos list.

nodeschordwise – approximate number of finite element nodes to use along each chord line (the top surface and the bottom surface will each have this many nodes.) This will reset the meshchord value to (specified value)/(current chord). The actual number of nodes may vary due to integer math and positioning of nodes exactly at spar positions.

elemperspanbay – approximate number of finite elements to use between each rib. This parameter will reset the meshspan parameter to (specified value) * (current number of ribs) / (current span).

Frame/DFrame



A **Frame** is an object made of beam elements distributed between two curves. Frame objects are based on the last S**ection** object – taking their shape and dimensions from that section.

A **DFrame** is also a frame type object but is based on/attached to the previous **Dome** object. It has the same parameters as the frame object.

For both object types, the `align` parameter can be used to select axial or circumferential alignment. If a single line of beams is desired, the `count` variable can be set to one, and the `position` parameter can be used to specify the position along the curve. A frame object does not change the default position of the next object. All beams are by default aligned with a node set at x = 0, y = 0, z = <beam start point z>. This may not be what is desired in all cases, so the `x3`, `y3`, and `z3` parameters can be used to override this setting. The bright lines in the figure above are thrust structure stiffening beams created using both frames and dframes. *Loft* will detect and remove duplicate beam/bar elements created at the junction points of two adjacent sections.

Parameter List

`align` – direction of beam elements: "axial" or "circ"
default: circ

`count` – number of frames to make (integer)
default: components setting of parent section/dome +1 in direction specified. The frames will be positioned at the same component edge locations that are used in the parent object, whether set by count (`components_axial`) or by explicit location (`axial_cpos`). Overriding the count will lose this location paring and result in even spacing of the specified number of frames.

`position` – location of a single frame, in fraction of the direction specified, must be between zero and one. Ignored if count does not equal 1.
default: 0

`type` – kind of 1-D object to generate. Should be "beam," "rod," or "bar" (rod and bar are the same).
default: beam

`x3,y3,z3` – location of beam alignment node
default: x3 = 0, y3 = 0, z3 = beam start coordinate

Beam

A **Beam** is a one-dimensional object where the user specifies the absolute position of the end points. This object type can generate either a beam (has axial and bending stiffness) or a rod/bar (has only axial stiffness). The parameters specified for this object do not change the defaults for the other object types (but are remembered for other beam objects). None of the general object parameters (`move`, `rotate`, `scale`, `warp`, `flip`) are supported at the object level.

Parameter List

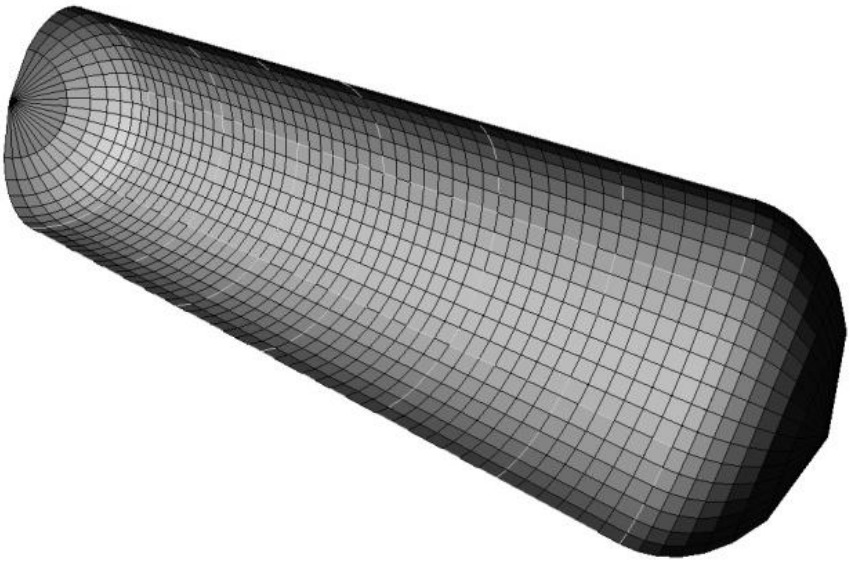
`type` – kind of 1-D object to generate. Should be "beam," "rod," or "bar" (rod and bar are the same).
default: beam

`x1,y1,z1` – end point coordinates
default: 0,0,0, or previous settings

`x2,y2,z2` – end point coordinates
default: 1,1,1, or previous settings

`x3,y3,z3` – beam alignment node coordinates
default: 0,1,0 or previous settings

Tank



A **Tank** is a meta-object composed of three objects: an elliptical dome of negative length, a tank barrel section, and an elliptical dome with positive length (the same as the negative length). The three objects will be named based on the supplied name for the tank meta-object but will have " FD," " B," or " AD" (for "forward dome," "barrel," and "aft dome") added. The tank object shares the section object parameters and defaults, with one additional parameter: dome length.

The tank local origin point is the center point of curve 1 (the center of the front of the barrel section). Use of a tank object does update the global default creation point to the center of curve 2.

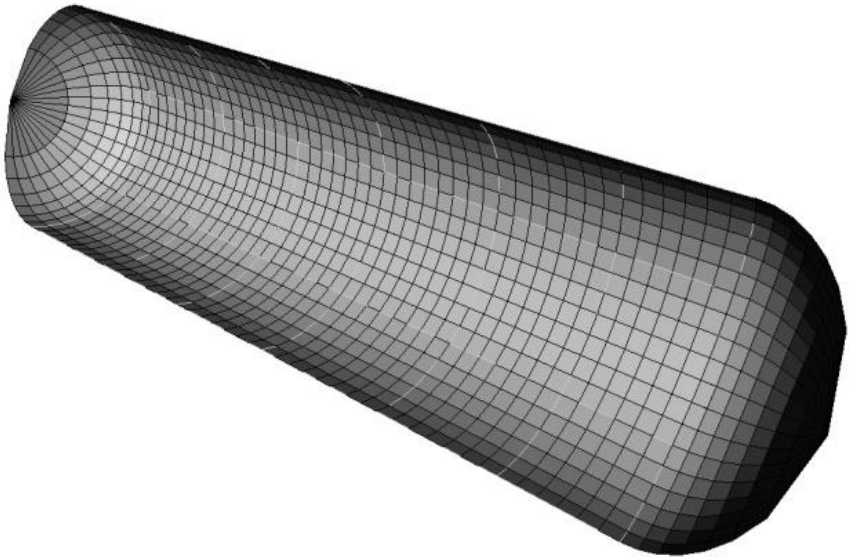
Additional Parameter List

See **Section** object type above for a base list of parameters.

`domelength` – length of the elliptical domes

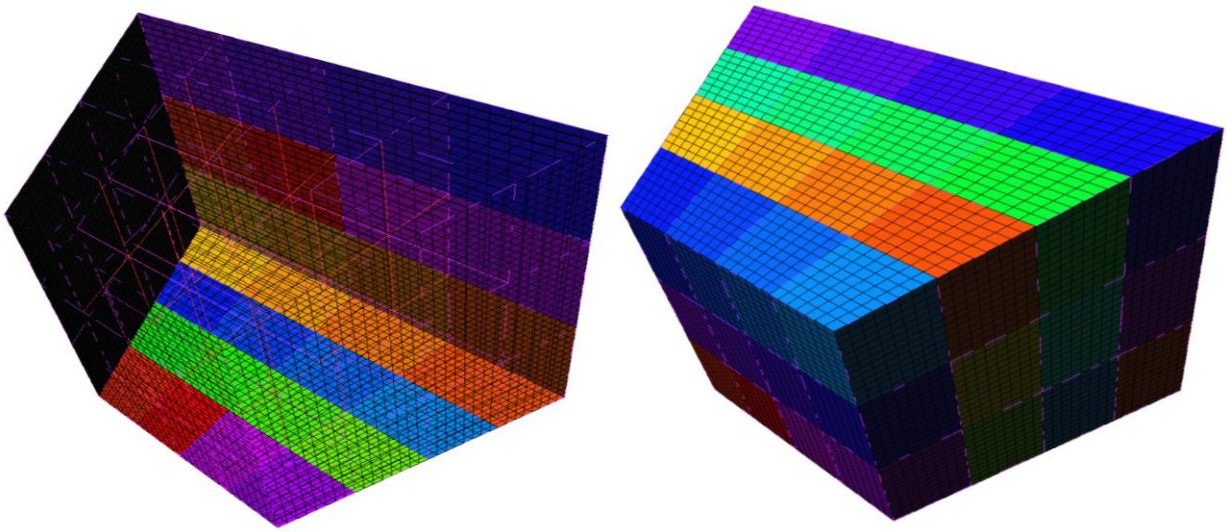
default: 0.707 * Average of corresponding section end's `scale_x`,`scale_y`

StiffTank



A **StiffTank** is a ring frame stiffened tank meta-object. It is constructed the same as the tank meta-object with the addition of circumferential ring frames being added along the edge of each barrel component (as controlled by the `components_axial` parameter). The string " R" is added to the object name for the frame object. See the tank and section objects for its parameters. No stiffening is added to the domes.

Box



A **Box** is a trapezoidal flat faced object with the front and back surfaces parallel. Stiffeners may optionally be placed along face component edges and/or through the volume of the box using the `stiff_skin_X` and `stiff_vol_X` parameters detailed below. There are no parameters to specify cross sectional shape—a square is always used. Note that like the wing object this object will not generally automatically stitch properly to an adjacent section or dome object as the node distribution will be different.

Parameter List

`c1_xscale` – factor to scale horizonal dimension of front end by
default = 1

`c1_yscale` – factor to scale vertical dimension of front end by
default = 1

`c1_xoffset` – horizontal distance to move front end
default = 0

`c1_yoffset` – vertical distance to move front end
default = 0

`c2_xscale` – factor to scale horizonal dimension of aft end by
default = 1

`c2_yscale` – factor to scale vertical dimension of aft end by
default = 1

`c2_xoffset` – horizontal distance to move aft end
default = 0

`c2_yoffset` – vertical distance to move aft end
default = 0

`length` – axial length of box
default = 1

`nodes_vert` – number of nodes in the vertical direction
default = 10

`nodes_horz` – number of nodes in the horizontal direction
default = 10

`nodes_axial` – number of nodes in the axial direction
default = 10

`components_vert` – number of components in the vertical direction
default = 3

`components_horz` – number of components in the horizontal direction
default = 3

`components_axial` – number of components in the axial direction
default = 3

`stiff_skin_vert` – controls the creation of stiffeners in the vertical direction on the front, back, left, and right skin panels. Values of "1," "on," or "true" will enable the stiffeners. Values of "0," "off," or "false" will disable them.
default = off

`stiff_skin_horz` – controls the creation of stiffeners in the horizontal direction on the front, back, top, and bottom skin panels. Values of "1," "on," or "true" will enable the stiffeners. Values of "0," "off," or "false" will disable them.
default = off

`stiff_skin_axial` – controls the creation of stiffeners in the axial direction on the top, bottom, left and right skin panels. Values of "1," "on," or "true" will enable the stiffeners. Values of "0," "off," or "false" will disable them.
default = off

`stiff_skin_all` – toggles all three `stiff_skin_X` settings to the specified value. Values of "1," "on," or "true" will enable the stiffeners. Values of "0," "off," or "false" will disable them.
default = off

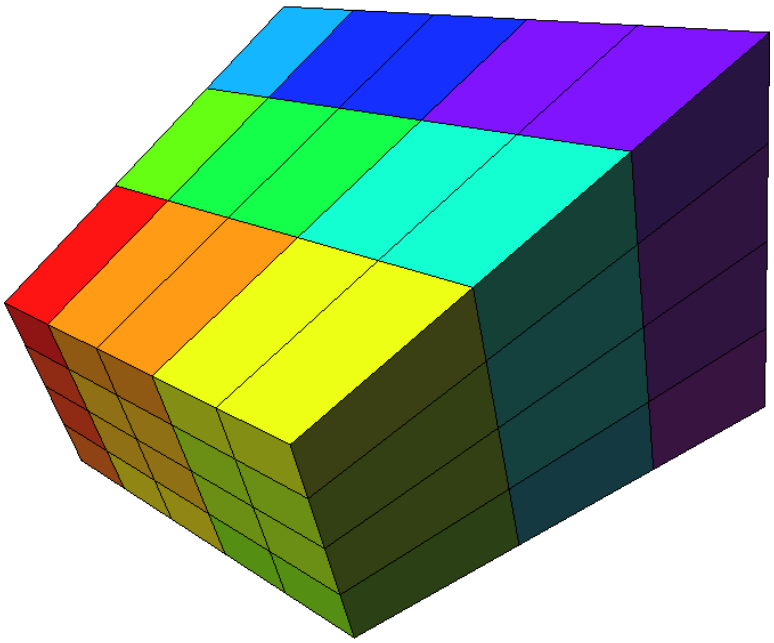
`stiff_vol_vert` – controls the creation of stiffeners in the vertical direction in the box internal volume. Values of "1," "on," or "true" will enable the stiffeners. Values of "0," "off," or "false" will disable them.
default = off

`stiff_vol_horz` – controls the creation of stiffeners in the horizontal direction in the box internal volume. Values of "1," "on," or "true" will enable the stiffeners. Values of "0," "off," or "false" will disable them.
default = off

`stiff_vol_axial` – controls the creation of stiffeners in the axial direction in the box internal volume. Values of "1," "on," or "true" will enable the stiffeners. Values of "0," "off," or "false" will disable them.
default = off

`stiff_vol_all` – toggles all three `stiff_vol_X` settings to the specified value. Values of "1," "on," or "true" will enable the stiffeners. Values of "0," "off," or "false" will disable them.
default = off

Block



The **Block** object creates a three-dimensional, solid-element-based trapezoidal prism object. It has two rectangular ends with variable scales, horizonal offsets, and vertical offsets supported at each end. Output in NASTRAN will produce 8-node CHEXA elements, and in VRML each element face will be written as a flat panel. Other output formats are not currently supported for this object. All of the rotation, translation, and warping parameters (in the common parameters list) are supported except for the `flip` parameter.

Parameter List

`c1_xscale` – factor to scale horizonal dimension of front end by
default = 1

`c1_yscale` – factor to scale vertical dimension of front end by
default = 1

`c1_xoffset` – horizontal distance to move front end
default = 0

`c1_yoffset` – vertical distance to move front end
default = 0

`c2_xscale` – factor to scale horizonal dimension of aft end by
default = 1

`c2_yscale` – factor to scale vertical dimension of aft end by
default = 1

`c2_xoffset` – horizontal distance to move aft end
default = 0

`c2_yoffset` – vertical distance to move aft end
default = 0

`length` – axial length of block
default = 1

`nodes_vert` – number of nodes in the vertical direction
default = 10

`nodes_horz` – number of nodes in the horizontal direction
default = 10

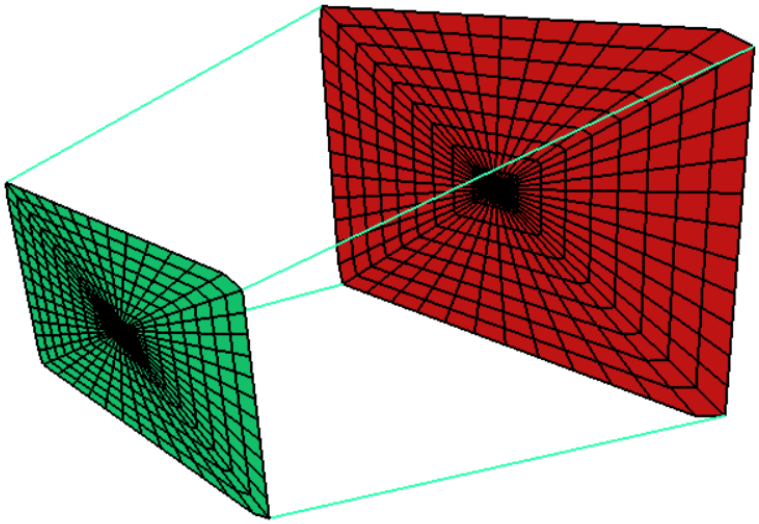
`nodes_axial` – number of nodes in the axial direction
default = 10

`components_vert` – number of components in the vertical direction
default = 3

`components_horz` – number of components in the horizontal direction
default = 3

`components_axial` – number of components in the axial direction
default = 3

BC/RBE/FORCE/MASS/PRESS/TEMP



The **BC/RBE/FORCE/MASS/PRESS/TEMP** object types map user supplied values to the project mesh. Currently, they will be reflected only in NASTRAN format output files. An exception is that an RBE element will be depicted in VRML as a 1-D line.

The BC object creates boundary conditions (NASTRAN SPC cards) with the user specified degrees of freedom (dof) constrained. The RBE object creates rigid body elements that connect degrees of freedom for two or more nodes. The FORCE object type applies a specified force vector (NASTRAN FORCE card) to a group of nodes. The MASS object type applies a concentrated mass (NASTRAN CONM2 card) to a group of nodes. Both FORCE and MASS values can be given as a value to be applied to every specified node or as a value to be smeared across the nodes producing the value as a total. The PRESS object type applies a specified pressure (NASTRAN PLOAD2 card) to a group of elements. The TEMP object applies a specified initial or boundary condition temperature (TEMP or SPC card) to a group of nodes.

For RBE objects, if the `node1` and `node2` parameters are used then those nodes are connected and the `approach` parameter is ignored. If `node1` and `group2` parameters are used and `approach` is given as `spider`, then `node1` is connected to every node in `group2`. Otherwise `group1` is connected to `group2` using the specified approach.

For non-RBE object types, `node1`, `group1`, and `group2` can all be used to specify nodes (or elements in the PRESS case) that are to have the specified values mapped to them. `Node2` is a synonym for `setid` for these cases.

The group membership used for these objects is frequently created using the `corner` parameter in the `region` command mode. Please refer to that section of the manual for additional insight.

Parameter List

`node1` – index of first node to use
default = -1

node2 – index of second node use. (Note that this shares storage with the setid parameter, so each will overwrite the other.)
default = -1

group1 – name of first group of nodes (or elements for a PRESS type)
default = null

group2 – name of second group of nodes (or elements for a PRESS type). Group2 is optional for BC, FORCE, MASS, and PRESS objects but if used will produce the same type of cards for the second group of nodes/elements. For RBE objects it is used as a list of nodes to connect to the nodes in group1 using the specified approach.
default = null

doflist – list of degree of freedom numbers to use for a boundary condition or an RBE object. This value is ignored for FORCE, MASS, and PRESS objects. These are given in a NASTRAN style adjacent list containing the digits 1,2,3,4,5, and/or 6.
default = 123456

type – choice between "rbe2" or "rbe3," "bc," "force," "mass," or "press." A rbe can be specified as either a four character string or a single digit ("rbe2" or "2") or ("rbe3" or "3"). This type parameter is automatically set when using the corresponding name to create the object. In other words, if an object is created using "object force thrust" then the type is set to force.
default = rbe2 for RBE object, object type for other objects.

approach – method used, in RBE object, to connect multiple nodes, or in MASS or FORCE object to compute the applied value to each node. Only the first three letters of the chosen approach need to be given.
"sequential" – the first node of group1 will be connected to the first node of group2, second node to second, etc. until one group runs out of nodes. Remaining nodes are not connected.
"reverse" – the first node of group1 will be connected to the last node (N) of group2, the second node to the last but one, etc. until one group runs out of nodes. Remaining nodes are not connected.
"closest" – each node in group1 is connected to the closest node in group2. This is not necessarily a unique pairing.
"shortest" – (not currently implemented) the shortest total length of unique connections between group1 and group2 are created, nodes are not reused so uneven groups will result in unused nodes
"spider" – node specified in node1 parameter or the first node in group1 is connected to every node in group2.
"smeared" – if specified for a mass or force object, the specified values (value1-3) will be divided by the number of nodes that have the value applied so that the total value applied is the specified amount. If "smeared" is not specified, then every node will have the unscaled value applied.
"tbc" – "thermal boundary condition," if specified for a thermal object, the specified value will be applied as a boundary condition and generate a thermal SPC card. (thermal default)
"tic" – "thermal initial condition," if specified for a thermal object, the specified value will be treated as an initial temperature and generate a TEMP card.
default = sequential

mark – add new elements to specified group

`setid` – index for boundary condition, force, or pressure card numbers. (Note that this shares storage with the `node2` parameter, so each will overwrite the other.) This index then needs to be referenced in the case control section of a NASTRAN analysis file in order to be used.

`value/value1` – floating point value used for PRESS object pressure, MASS object mass, or the x component of the applied force on a FORCE object. "`value`" with no number after it will also be interpreted as "`value1`."

`value2` – floating point value used for the y component on a FORCE object

`value3` – floating point value used for the z component on a FORCE object

Example object commands and parameters:

```
object rbe forward wing rbes
   group1 mainwing fwd corners
   group2 Front bulkhead corners
object bc symmetry bc
   group1 centerline
   setid 999
   dof 345
object force thrust
   group1 engine ring
   setid 1000
   value1 -100.0
   value2 0.0
   value3 0.0
object press lift
   group1 mainwing skin SKIN UP ELEMS
   setid 1001
   value1 -1.0
object mass ballast
   group1 nosecap ALL NODES
   value 1000
   approach smeared
object temp root temp
   group1 wing root nodes
   value 100.0
   approach tbc
   setid 100
```

Node

A **Node** object is used to create a single node in the mesh. One application is to use variables to parametrically position a node that will be used as the hub of a rigid body element (RBE) spider. This spider can be used to distribute a load over a collection of connected nodes.

Parameter List

`x, y, z` – specify the x, y, or z coordinate of the node
default = -1.0

`mark` – label to apply to the node. Note that a group called "<object name> ALL NODES" is automatically created. Any group can then be used, for instance, in a RBE object creation process. Creating a group of elements is not supported in the node creation object.
default: none
example: `mark node My node group`

User Curve Types and Parameters

The internal library curves are all defined such that they have a nominal radius of one. For instance, a square is two units long on an edge. This allows the use of object level curve scaling parameters to reflect the actual dimensions desired for the mesh. This approach is recommended, but not required, for user-defined curves. For proper alignment of normal vectors, curves should be defined sequentially in a clock-wise fashion.

Mnemonics for user-defined curves can be chosen such that they override internally defined curves (i.e., a user-defined "`sc`" curve would replace the internal one). Defining a second user-defined curve with the same name generally will not override the previous shape. When data from a curve is needed, *Loft* scans through the curve libraries in the following order and stops when it gets a match: 1) Interpolated curves, in the order they were defined, 2) Compound curves, in the order they were defined, 3) Lofted curves, in the order they were defined, and 4) Internal curves. If no match is found, *Loft* will use a semi-circle.

Interpolated Curves

Interpolated curves are defined by specifying x and y coordinates of points along the curve. Point order is important. Various interpolation options may be available in the future, but currently only linear interpolation is supported. "y" is the vertical coordinate and "x" is the horizontal.

Parameter List

`start` – initial point coordinates
Example: `start 0.0 1.0`

`line` – coordinates of new point to be connected to the previous point by a line.
`Example: line 1.0 1.0`

Compound Curves

Compound curves are curves built up by combining previously defined curves. Any curve type (built-in, interpolated, lofted or previous compound) can be used. Only circles and semi-circles have modules that will automatically compute their intersection points with each other. If an intersection is not between two circle or semi-cirle `child` curves, then the user will need to specify the portions of each curve that is to be used. See the project 3 tutorial in chapter 2 of the manual for a more complete explanation of this process.

Parameter List

`child` – name of child curve. This starts a new child curve definition. All parameters that follow will refer to this child until a new child starts or the entire compound curve definition is finished by another command.

`x` – x coordinate to use for center of child curve
default 0.0

`y` – y coordinate to use for center of child curve
default 0.0

`radius` – scale factor for curves
default 1.0

`sstart, sstop` – fraction along a curve's circumference to start/stop (defaults 0.0, 1.0). For circle/sc curves these values are overwritten when the curve intersection code is called: e.g., curve 3's `sstop` value is reset when curve 4 is specified. Thus, `sstart` will have an effect only on the first specified circle/sc curve and `sstop` will have an effect only on the last circle/sc specified curve. For curve types where intersection calculation code has not been written (i.e., anything other than circle or semi-circle), these values will not be overwritten and in fact are the only way to use these types of curves in a compound curve.

Lofted Curves

Loft inherently creates a "lofted" curve whenever it creates a dome or a section and is creating nodes at a station between the two ends of the object. The "lofted" user-defined curve type allows the user to extract one of these intermediate shapes for later use. Applications include creation of mid-section bulkheads. Any curve types can be used as the end curves.

Parameter List

`curve1` – name of first source curve.
default = sc

`curve2` – name of second source curve.
default = sc

`station` – fractional position between the two curves used to create the new user curve.
0.0 = `curve1`, 1.0 = `curve2`
default = 0.5

Example:

```
curve lofted midbarrel
  curve1 sc
  curve2 ss
  station 0.3
```

Libraries

Curve Library

This is a list of the currently coded curves and their mnemonics. All curves have a nominal radius of one.

Curve *families* allow the user to tack a single parameter onto the name of the curve to affect the final shape generated. No space is left between the mnemonic and the parameter, e.g., `fillet0.44` or `sccw3.2`. The parameter is optional.

Most curves are available in both a full, 360-degree version and a semi, 180-degree version. When using a full curve, *Loft* will use the `nodes_circ` parameter to generate the curve, but the first and last nodes (at 0 and 360 degrees) will be merged and the mesh will have one fewer node in that direction than was specified by the user. Keep this in mind and increase the value of the parameter if necessary.

Simple Curves

Circle – "cir" – unit radius full circle.

Semicircle – "sc" – unit radius half circle.

Square – "squ" – full square of width and height 2.

Semi-square – "ss" – half square of dimension 2 (encloses radius 1 circle exactly)

Breadbox – "bb" – circular on top, square on the bottom. (**Note:** for compatibility with the other library curves, the breadbox curve has s=0.25 and 0.75 at the junctions of the circle and the square. These are not 25% and 75% along its circumference.)

Semi-breadbox – "sbb" – half section with top half circular and bottom half square. (**Note:** for compatibility with the other library curves, the semi-breadbox curve has s = 0.50 at the junction of the circle and the square. This is not 50% along its circumference.)

Line – "line" – vertical line from +1 to –1, for webs and longitudinal bulkheads

Horizontal line – "hline" – horizontal line from +1 to –1

Curve Families

Semi-circle-cosine-wiggle – "sccw" – cosine wiggle shape
parameter meaning – number of full cosine waves to generate
default=2.5

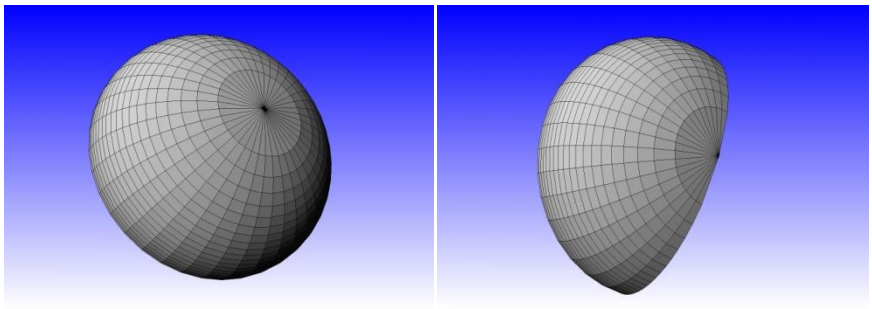
Filleted box – "fillet" – square with rounded corners (**Note:** the distribution of s along the filleted box is not exactly by circumference.)
parameter meaning – radius of fillet, between 0 and 1
default=0.25

Semi-Filleted box – “sfillet” – half section square with rounded corners. (**Note:** the distribution of s along the semi-filleted box is not uniform in circumferential distance.)

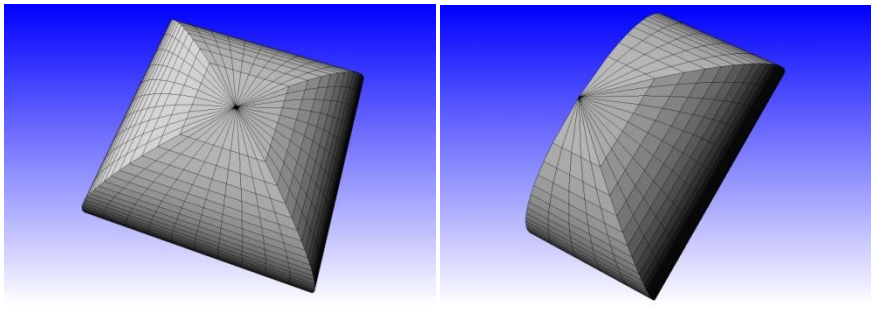
parameter meaning – radius of fillet, between 0 and 1

default=0.25

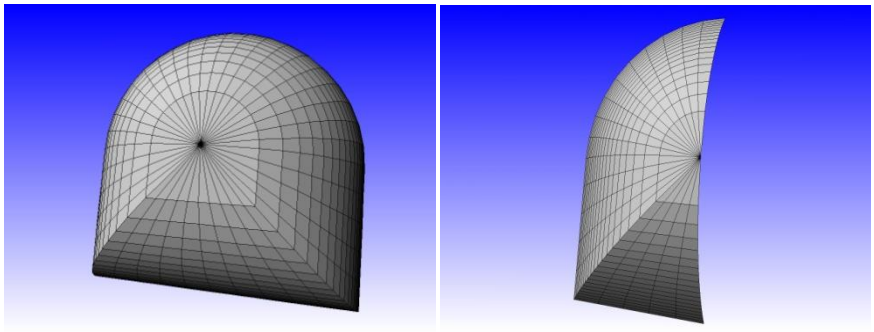
Library Curves illustrated with Dome Objects



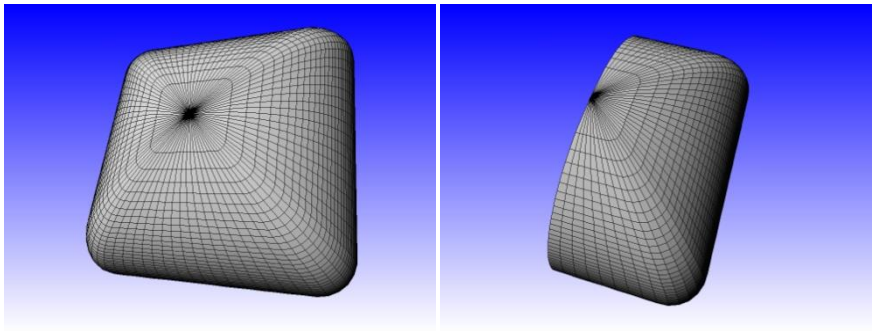
Circle Semi-Circle



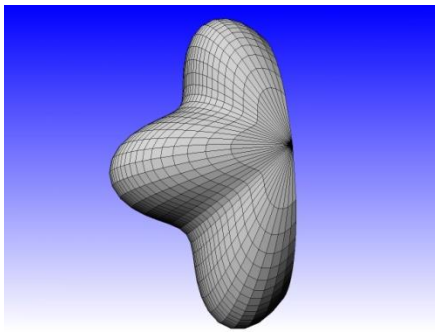
Square Semi-Square

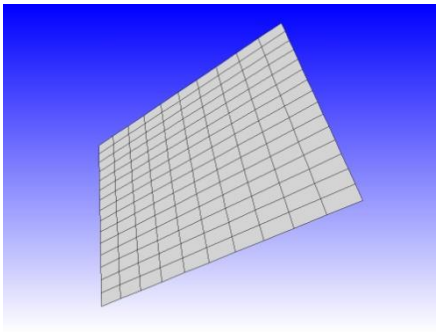


Breadbox Semi-breadbox



Fillet Semi-Fillet





Cosine Wiggle Line or HLine

Dome Taper Library

This is a list of the currently coded dome taper schedules and the meaning of the *paramN* options.

Bulkhead – “bulk” – planar (zero length) bulkhead

Linear – “line” – linear taper (cone shaped)

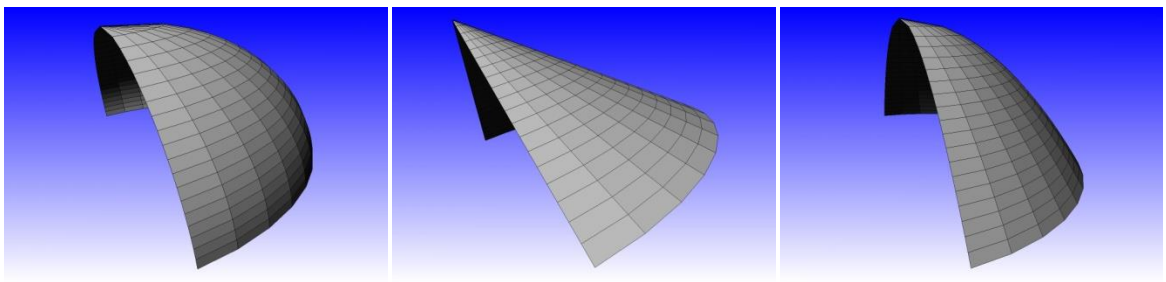
Parabolic – “para” – power law nose shape
param1 = exponent of taper schedule.
default = 0.5 = true parabola

Elliptical – “elli” – elliptical taper for tank domes

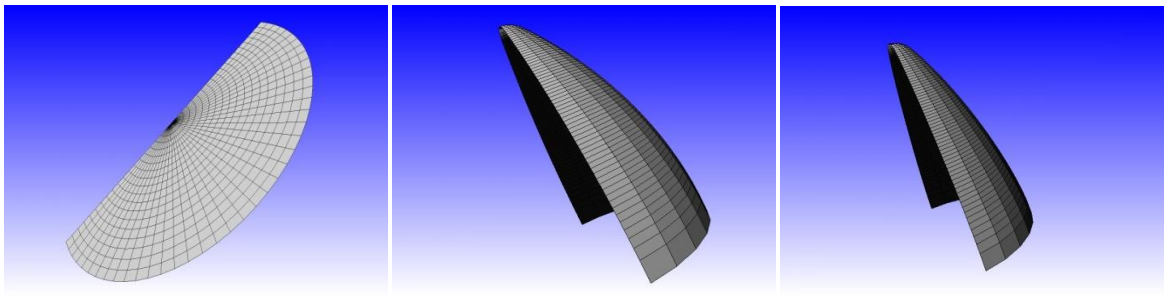
Ogive – “ogive” – tangent ogive nose with spherical nose cap
param1 = nose cap radius. default = 1.0
param2 = radius of main section curve. default = 0.0
param3 = radius of nose base. default = 1.0

Haack – “haack” – LD-Haack nose shape with optional spherical blunt cap
param1 = length of nose without blunt cap. default = dome length
param2 = nose cap radius. default = 1.0
param3 = nose cap length. default = 0.0

Dome Taper Library Examples



Elliptical Linear Parabolic



Bulkhead Ogive Blunt LD-Haack

Section Taper Library

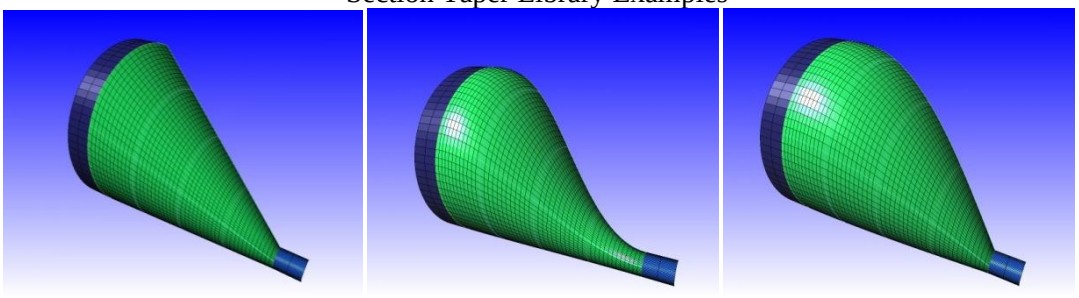
This is a list of the currently coded section taper schedules and the meaning of the `value` options. The pictures show a section object that interpolates between one semi-circle and a larger, offset semi-circle. Circumferential and axial frames are added.

Linear – "line" - linear taper

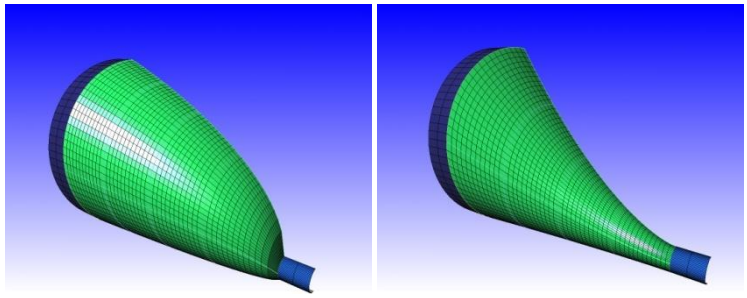
Power – "power" – power curve taper
value = exponent of taper schedule.
default = 1.0 = linear

Cosine – "cosine" – cosine schedule, offers tangency possibilities
value = number of cosine half waves.
default = 1.0

Section Taper Library Examples



Linear Cosine 1.0 Cosine 0.5



Power 0.5 Power 1.5

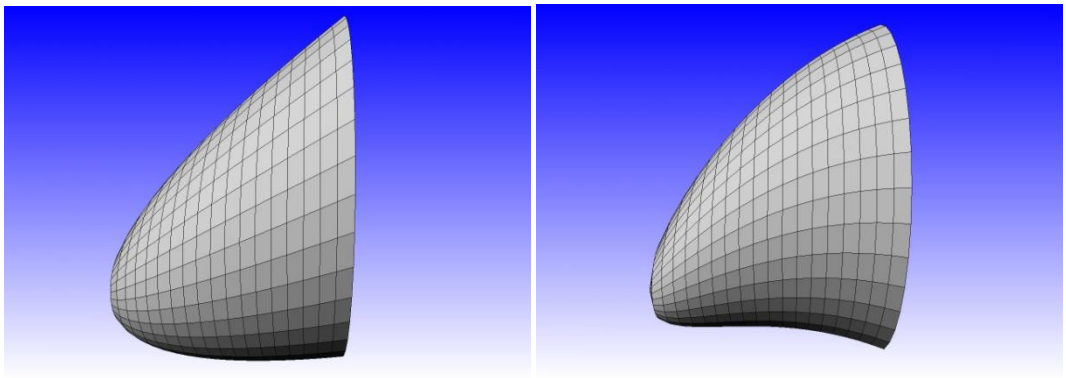
Droop Library

This is a list of the currently coded dome droop schedules.

Linear – “line” – nose centerline descends linearly

Parabolic – “para” – nose centerline descent smoothly increases

Droop Library Examples



Linear Parabolic

System Variable List

This is a list of the system variables available for use in a *Loft* input file. They correspond to the object parameters set by the user in the input file and will return the current values of those variable.

Global Variables

Variable	**Invoked by**
transx – x coordinate for next object	@transx
transy – y coordinate for next object	@transy
transz – z coordinate for next object	@transz
rotx – x rotation for next object	@rotx
roty – y rotation for next object	@roty
rotz – z rotation for next object	@rotz
components_circ – components in circumferential direction	@components_circ
nodes_circ – nodes in circumferential direction	@nodes_circ
maxM, minM – highest or lowest value of coordinate M in current stack	@maxx, @maxy, @maxz, @minx, @miny, @minz
time – system time in seconds	@time
clock – CPU time in seconds	@clock
pi – ratio of a circle's circumference to its diameter. ~3.14152965	@pi
e – Euler's constant, base of natural log and exponential function. ~2.71828	@e

Section Variables

Variable	**Invoked by**
length – length of section object	@section.length
taper – taper value of section object	@section.taper
components_axial – components in axial direction	@section.components_axial
nodes_axial – nodes in axial direction	@section.nodes_axial

Dome Variables

Variable	**Invoked by**
length – length of dome object	@dome.length
zdist – axial node distribution	@dome.zdist
droop – droop value of dome object	@dome.droop
param1 – parameter 1	@dome.param1
param2 – parameter 2	@dome.param2
param3 – parameter 3	@dome.param3
components_axial – components in axial direction	@dome.components_axial
nodes_axial – nodes in axial direction	@dome.nodes_axial

Wing Variables

Variable	**Invoked by**
chord	@wing.chord
span	@wing.span
taper	@wing.taper
sweep	@wing.sweep
twist	@wing.twist
wingbox – wingbox length	@wing.wingbox
mesh_chord	@wing.mesh_chord
mesh_span	@wing.mesh.span
mesh_thick	@wing.mesh_thick
wing transx – x position of next wing (wing objects do not update the global transx/y/z variables. Instead, the default position of a new wing object is the same as the previous wing object.)	@wing.transx
wing transy – y position of next wing	@wing.transy
wing transz – z position of next wing	@wing.transz

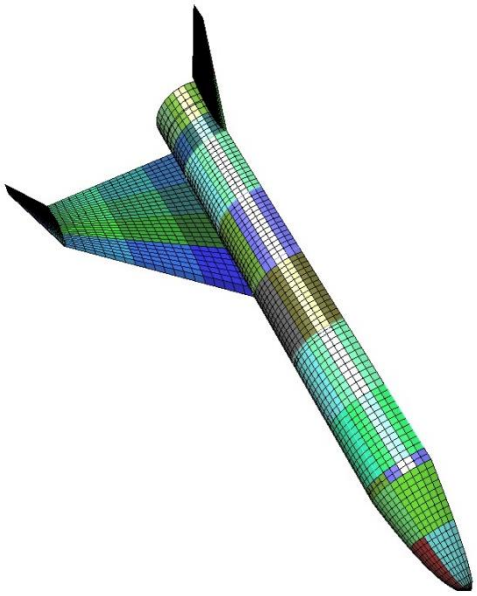
Math Function List

Function	Invoked by
Sine	%sin
Cosine	%cos
Tangent	%tan
Arcsine	%asin
Arccosine	%acos
Arctangent	%atan
Hyperbolic Sine	%sinh
Hyperbolic Cosine	%cosh
Hyperbolic Tangent	%tanh
Hyperbolic Arcsine	%asinh
Hyperbolic Arccosine	%acosh
Hyperbolic Arctangent	%atanh
Exp	%exp
Log	%log
Square root	%sqrt
Cube root	%cbrt
Absolute value	%abs
Integer or truncation	%int

For these functions, all angles are in radians.

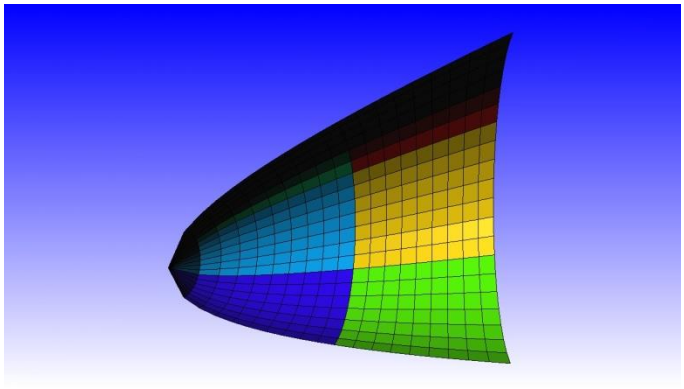
Chapter 8: Example Input Files

Loft Example Input File #1

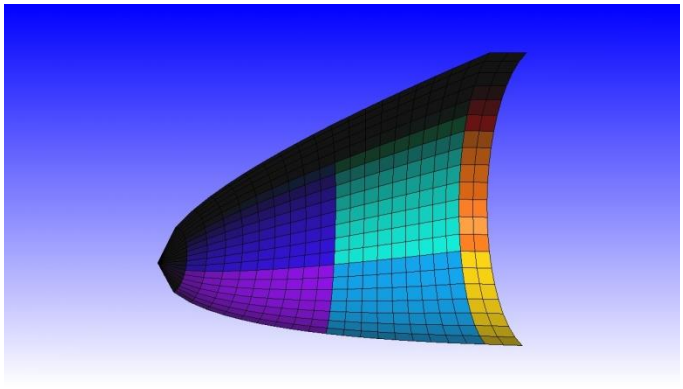


The first full example *Loft* input file builds a simple conceptual level finite element model of a Two Stage To Orbit (TSTO) booster vehicle. A lot of the design details of the vehicle, such as stiffeners, are very notional and the wing carry-through passes through the aft tank. It contains approximately 100 lines of basic *Loft* commands and parameters. It does not make use of math, variables, user-defined curves, the region mode, or perform any store/recall operations.

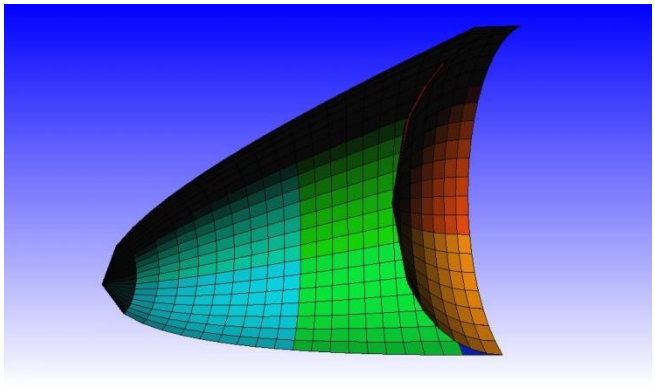
```
# Testing full vehicle based vaguely on
# ISAT Reference vehicle Mach 3.4 TSTO Vehicle
# Booster
# Our nose
object dome BST Nose
curve1 sc
c1_xscale 15.589
c1_yscale 15.589
length -36
taper para
nodes_circ 21
nodes_axial 20
droop line
zdroop 8
components_axial 2
```



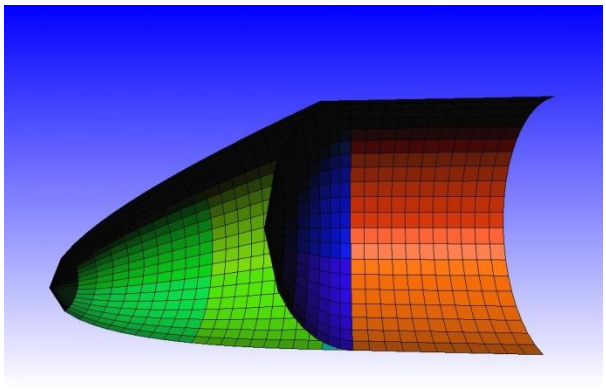
```
# Short fuselage extension to get nose
# not to impinge on forward tank
object section BST Nose Barrel
length 3.885
nodes_axial 3
components_axial 1
```



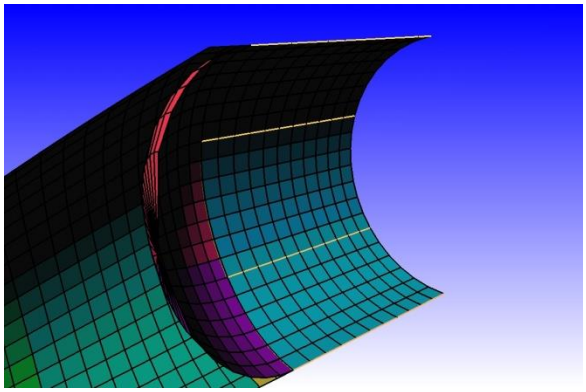
```
# Forward LOX Tank
object dome BST LOX FW Dome
length -11.02
taper elli
nodes_axial 8
components_axial 1
```



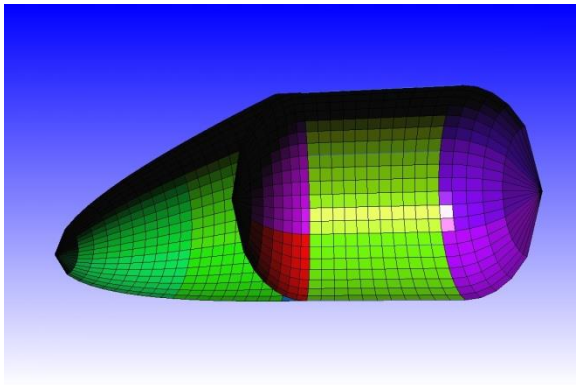
```
object section BST LOX Barrel
length 23.205
nodes_axial 12
components_axial 1
```



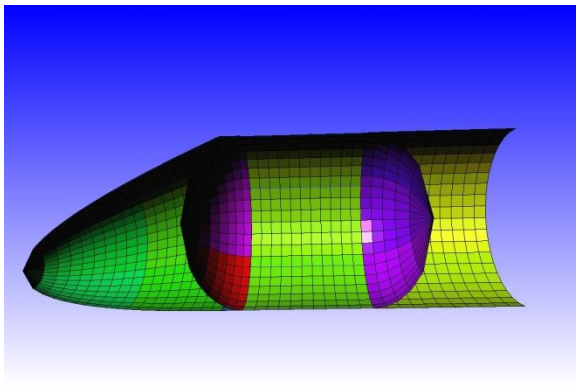
```
object frame BST LOX Frame
align axial
```



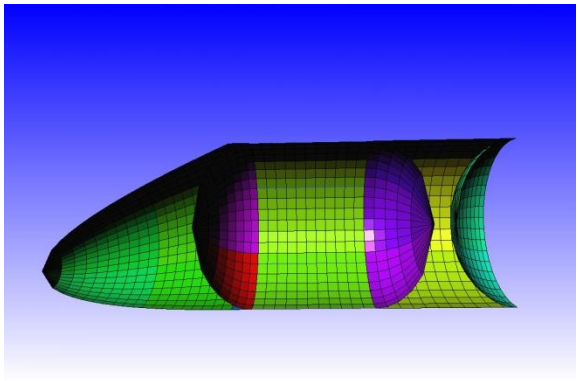
```
object dome BST LOX AFT Dome
length 11.02
taper elli
nodes_axial 6
components_axial 1
```



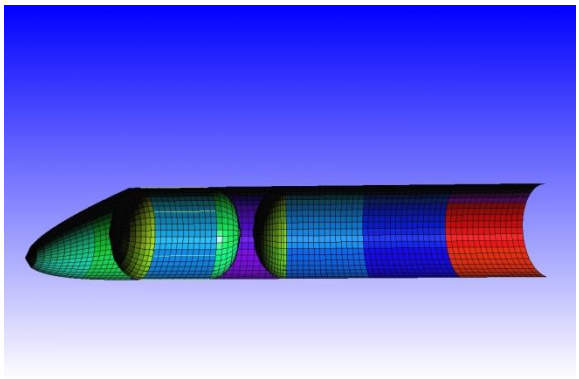
```
# Intertank adaptor
object section BST ITA
length 26.04
nodes_axial 12
components_axial 1
```



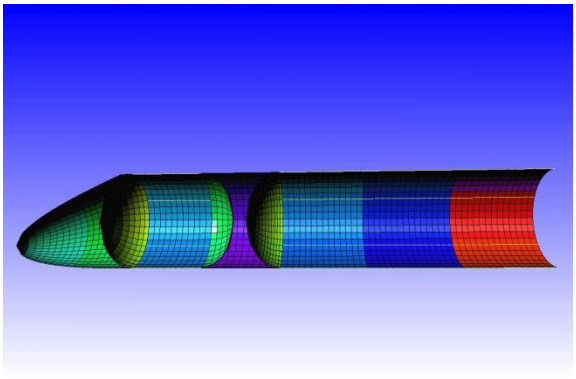
```
# LH2 Tank
object dome BST FW Dome
length -11.02
taper elli
nodes_axial 12
components_axial 1
```



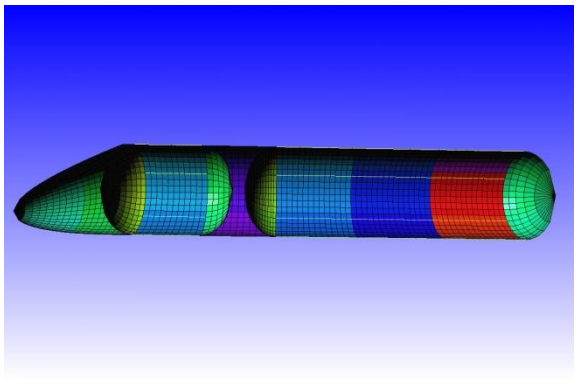
```
object section BST LH2 Barrel
length 87.35
nodes_axial 44
components_axial 3
```



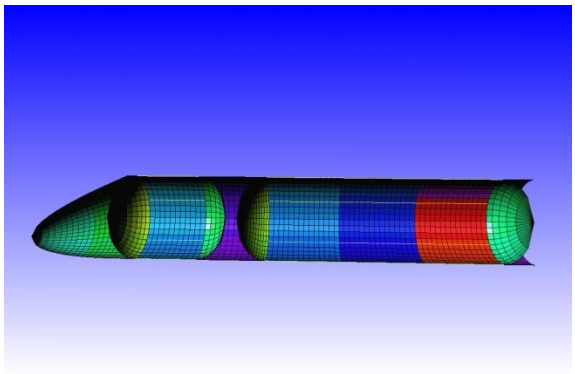
```
object frame BST LH2 frame
```



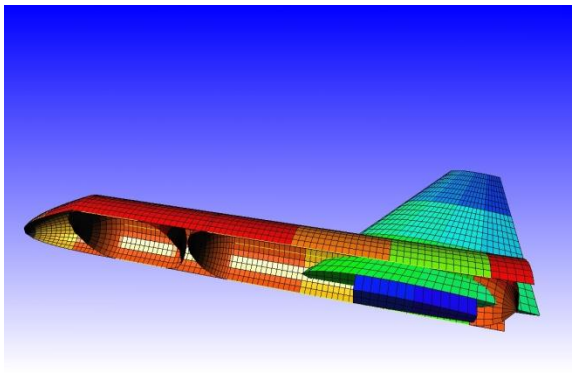
```
object dome BST LH2 AFT Dome
length 11.02
taper elli
nodes_axial 6
components_axial 1
```



```
# Tank shroud
object section BST Tank Shroud
length 11.02
nodes_axial 6
components_axial 1
```

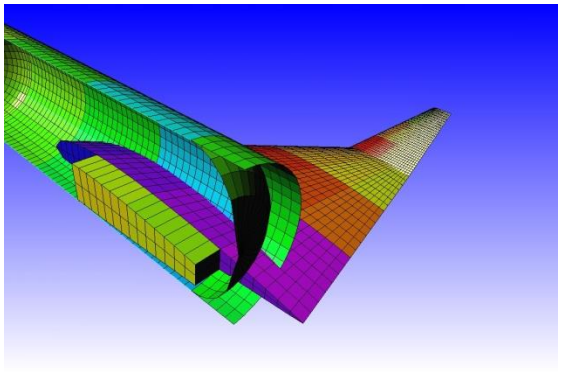


```
# Wing
object wing Main Wing
chord 80
span 60
taper 0.25
sweep 40
wingbox 6
transx 6
relz -70
rely -12
nribs 4
nspars 3
meshchord .4
meshspan .4
meshthick .4
naca 2412
```

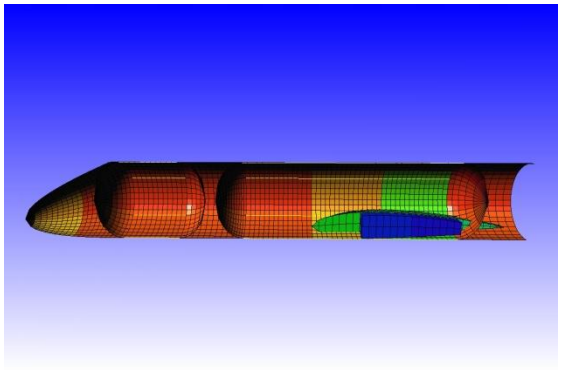


```
# Tip fin
object wing Winglet
chord 20
span 20
wingbox 0
transx 66
relz 50.35
rotz 50
```

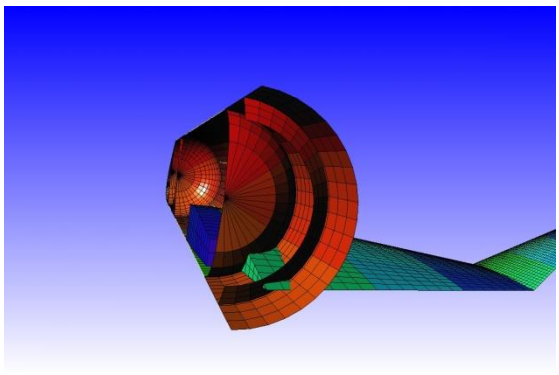
```
meshchord 1.6
meshspan 1.6
meshthick 1.6
```



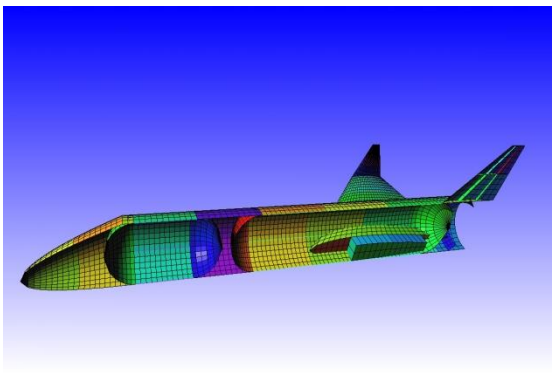
```
# Thrust structure shroud
object section BST TS Shroud
length 16.5
nodes_axial 6
components_axial 1
```



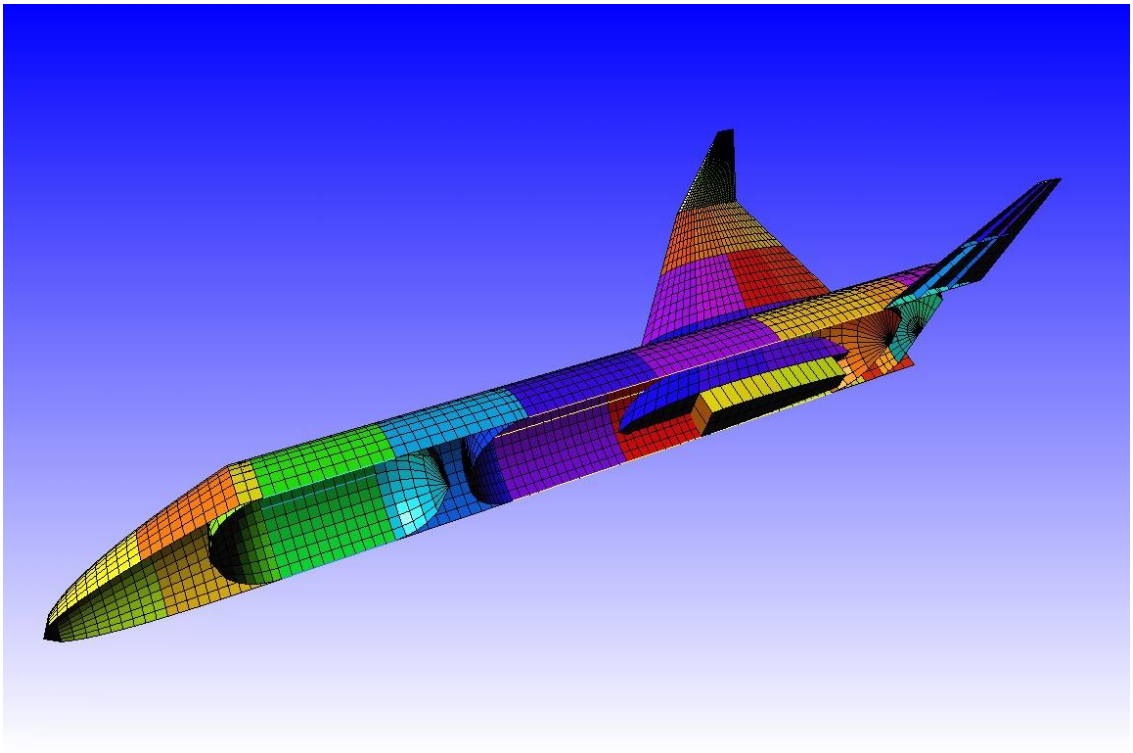
```
# Put a chopped off cone inside the shroud
# to represent the thrust structure
# note the relz parameter's use
object section BST Thrust Structure
length 3
c2_xscale 12
c2_yscale 12
relz -10.5
nodes_axial 4
components_axial 1
```



```
# Vertical tail on line of symmetry
object wing Tail
naca 0612
nribs 3
nspars 2
halfwing bottom
chord 30
span 30
transy 15.589
rotz 90
relz -20
mesh .4
```

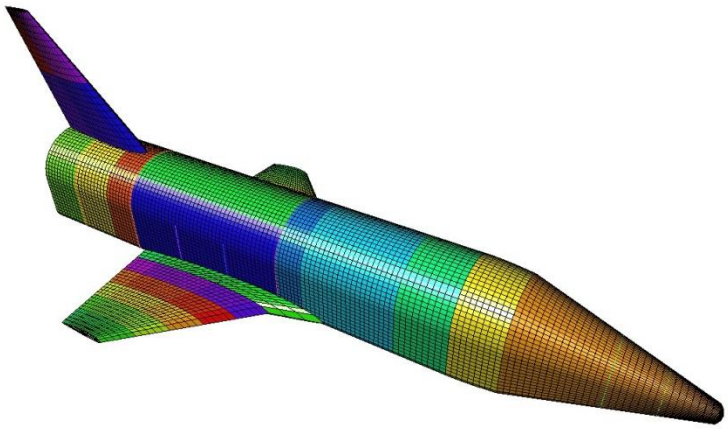


```
# bulkhead to close off thrust structure
object dome BST Thrust Bulkhead
taper bulk
components_axial 1
# save
write vrml full-color.wrl
end
```



Obvious issues with this model include the wing carry-through passing through the aft tank and the various wing surfaces needing to be stitched together. But this model is sufficient for some early configuration studies and to produce images for presentation and discussion. The next example produces a much more realistic model of a very similar vehicle.

Loft Example Input File #2



The second full example *Loft* input file builds a significantly more complex finite element model of a similar Two Stage To Orbit (TSTO) orbiter vehicle suitable for advanced conceptual analysis. Most of the neglected design details in the first example have been addressed in this model with carefully positioned stiffeners and wings. This input file uses approximately 1100 lines of *Loft* commands and parameters.

This example has been significantly updated from the version included in earlier editions of this manual. It is now completely parametric; all dimensions are specified as variables and positions are computed from those dimensions. The wing and tail are automatically stitched to the fuselage bulkheads using rigid boundary elements (rbes). Many other recently updated program features are also used, such as the mirror command, include file capability, generation of point masses and spider rbe connections for force and mass distribution, and program flow control with logical operators, if statements, and gotos.

Significant use is made of user-defined curves to define the fuselage shape at various stations. The region mode is used to change the property assignments needed to create the payload bay door and to create partial models for loads mapping. The store/recall capability is used extensively to position major components and to create presentation figures that focus on particular components. Substantial use is also made of user variables and command line math.

The input file starts with some comments describing the model:

```
# Loft input deck to generate
# LaRC TSTO-2009-2A Orbiter
# Revised model with RBE stitching
#
# Units are in inches
#
```

The first command is a variable used to specify if the user desires a full model or a half model. The model is created as a half vehicle. A later `if` command will check this variable and either execute or skip over a

mirror operation. Also, symmetric boundary conditions will be applied to the centerline nodes if a half model is desired but will not be created for a full model.

```
# select which version of model to create
# if fullvehicle = 0, a half model
#                = 1, a full model
define fullvehicle 0
```

The next block of commands defines the mesh density and derives number of nodes used circumferentially on the fuselage. It is necessary to use a variable to store this value because the use of the `store` command resets all default values including the `nodes_circ` setting. Variables are not reset by the `store` command. The `$meshdensft` variable is only used once (on the next line) so could be removed and the value could be directly set on the `$meshdens` variable.

```
# Global mesh controls
# Orbiter is 138' long from nose tip to tip of vertical tail
# meshdensft is approximately nodes/foot but will get
# modified by integer math on each section
# 0.5 is probably a good initial analysis density.
# 0.25 or 0.125 can be used for faster runtimes while
# testing the model
define meshdensft 0.125
define meshdens $meshdensft / 2.
# for proper payload bulkhead stitching circnodes needs to be a
multiple of 5 plus 1
define circnodes 55 * $meshdens %int
define circnodes $circnodes * 5 + 1
```

Note the need for the `$circnodes` variable to be a multiple of 5 in order for the payload bay bulkheads to align and stitch. We'll see the reason for that when we define the curves to build those bulkheads. The plus one requirement is because there is a first and last node as well as all the nodes between them.

As a demonstration of *Loft*'s math functionality, pi is computed next using $2*\cos^{-1}(0)$. It could also have been just defined as a value or by using the `@pi` system variable.

```
# pi
define piover2 0.0 %acos
define pi $piover2 * 2.
```

The next long section defines dimensions for all of the vehicle components. Any of these values can be changed by the user and the vehicle model should adjust and be generated successfully. Variable names are chosen to make their use clear, but some additional comments are added before or inline with a few of them for clarity.

```
# Vehicle dimensions & stations
# nominal cross section radius of main fuselage
define fusescale 102.
# Nose
define noselength 450.54
```

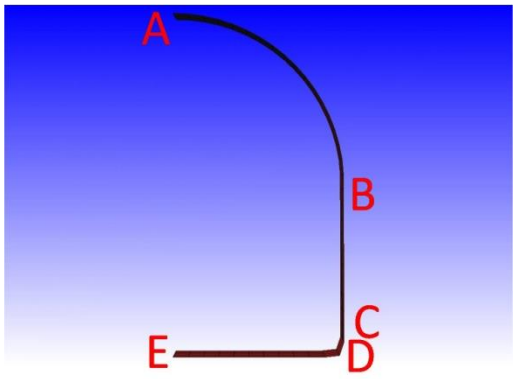
```
define noseoffset -42.0
define capradius 18.
define caplength -1. * $capradius / 2.
define bulletbulk 100. # distance forward of const fuselage to
place bulkhead (fwd LH2 support)
define bulkhead1 $noselength - $bulletbulk / 4.
define bulkhead2 $noselength - $bulletbulk / 2.
define bulkhead3 $noselength - $bulletbulk
# Tanks
define fwd_tank 325.
define aft_tank 43.
define aft_dome 96
define tankscale 96.
define aft_support $aft_dome / 3.
# Skirts over domes
define fwd_tank_skirt 62  # used only at aft of front tank
define aft_tank_skirt 76  # used front and aft of aft tank
define aft_skirt 103.
# Fuselage
define fuselength 1013.
define longeron_pos 0.18
define  fuse_center_bay  $fuselength  -  $fwd_tank  -  $aft_tank  -
$aft_skirt - $fwd_tank_skirt - $aft_tank_skirt
define half_lh2_nose 200. / 2.
define mid_bulk $fwd_tank + $half_lh2_nose / 2
# Payload bay
define plb_start $fwd_tank + $fwd_tank_skirt
define plb_length $fuse_center_bay
define plb_half $plb_length / 2
define plb_third $plb_length / 3
define plb_scale 72.
# Wing
define strakechord 498.196
define strakespan 31.
define straketipchord 377.777
define straketaper $straketipchord / $strakechord
define strakesweep 75.179
define straketangent $strakesweep * $pi / 180. %tan
define mainchord $straketipchord
define mainspan 233.
define maintipchord 113.235
define maintaper $maintipchord / $mainchord
define mainsweep 45.854
define spar1 10.
define spar2 36.
```

```
define spar3 72.
# Tail
define tailchord 260.337
define tailspan 281.5
define tailtipchord 77.955
define tailsweep 47.
# Thrust Cone
define thrustconelength 80
define thrustconelength $aft_skirt + 10.
```

The next command references a second input file that contains commands that define all of the user-defined curves needed to construct the vehicle. Use of an include file is not required. In this case, it is done as a demonstration and to make the parent input file easier to read and edit.

```
# User defined curves
include tsto-curves.txt
```

The first user-defined curve is the half-slice-of-bread cross sectional shape of the fuselage. The final shape is made of two circular portions: one at the top and one at the bottom outside corner, and two linear portions: the flat bottom and a five degree sloped sidewall. The internal circle shapes can be used for the circular portions, but the linear portions must be defined as interpolated curves. Then a compound curve named "body" is defined that combines the four children into one curve.

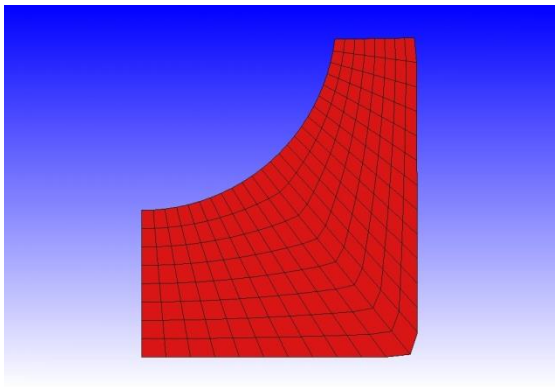


The next 88 lines of input are from "tsto-curves.txt."

```
# Define child curves of unit half cross section
# (cross sectional shape fits in -1 to 1 square space)
# point definition:
#     A = top (centerline) of curve
#     B = intersection of circ top & 5deg side
#     C = intersection of 5deg side and 1/17 fillet
#     D = intersection of 1/17 fillet and flat bottom
#     E = bottom (centerline) of curve
# line B-C
curve interpolated mylineBC
```

```
  start 0.996195  0.0871557
  line  0.999776 -0.9360497
# line D-E
curve interpolated mylineDE
  start 0.9411765 -1.0
  line 0.0 -1.0
# combine into full cross section
curve compound body
  child sc
  sstop 0.47222222222
  child mylineBC
  child sc
  sstart 0.47222222222
  sstop 1.0
  radius 0.0588235
  x 0.941176
  y -0.94117647
  child mylineDE
```

The next user-defined curves to create are those that define the mid-payload-bay support bulkheads. These have circular cross sections at the top/inboard and match the just-defined fuselage cross section at the bottom/outboard. The values of the `sstart` parameters were arrived at through trial and error. Note that the actual bulkhead is not created here, just the curves that are used later when the payload bay is created. Also note that variables defined in the parent input file are used here; *Loft* treats the include file input lines as if they were inserted into the parent.



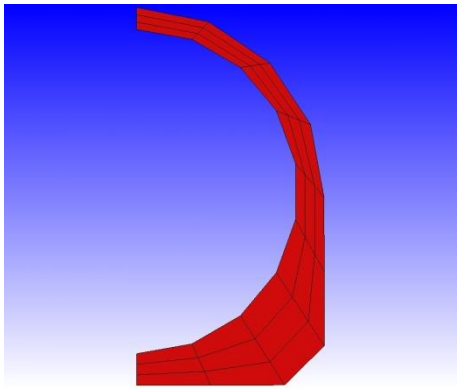
```
# Payload Bay Support bulkhead curves
# plb1 = semi-circle bay shape
# plb2 = sidewall & floor shape
curve compound plb1
  child sc
  sstart 0.54
  radius $plb_scale / $fusescale
  x 0.0
  y 24.0 / 102.
```

```
curve compound plb2
  child body
  sstart 0.400
```

The fuselage side of the bulkhead curve definition specifies "`child body`" and "`sstart 0.400`." This copies the fuselage curve shape starting at 40% along its circumference. Recall that near the start of the project we had a requirement that the `$circnodes` variable be a multiple of 5 plus 1. That requirement is to ensure that a fuselage node is always located at that 40% location for the bulkhead to connect to.

If the "plus 1" requirement is unclear, recall the "sd" semi-diamond shape in the user defined curve tutorial (Chapter 3, Project 3A). To accurately sample the curve we needed a node at the 50% point. Having a count of 3 placed nodes at 0, 50, and 100%. Having a count of 4 placed nodes at 0, 33, 66, and 100%. Having 5 would place nodes at 0, 25, 50, 75, and 100%. Any even count would miss placing a node at 50% and any odd count would hit 50%. Another way to state this is: to ensure a node occurs at 50% our node count needs to be a multiple of 2 plus 1. For this payload bay example, we want to ensure a node at 40%, which can also be written as two fifths. So, any multiple of 5 plus 1 will have nodes at zero, one, two, three, four and five fifths of the circumference.

The orbiter nose starts with a small circular cap that transitions to the body cross section defined earlier. The forward tank has a bullet shaped dome that projects a significant distance into the nose, making a support bulkhead necessary in this region. Two curves are defined to support the tank dome at 50 percent of its length: "forebullet" is the outer curve of the bulkhead which captures the fuselage nose shape at the desired position, and "dome50" is the tank dome shape at the same station. Two additional lofted curves are defined to allow the construction of full bulkheads in the nose designed to bracket the forward landing gear location: "fore25," and "fore50."



```
# Pieces of forebody bulkhead
define nl2 $noselength - $bulletbulk
curve lofted forebullet
  curve1 sc
  curve2 body
  c1_xscale $capradius
  c1_yscale $capradius
  c1_yoffset $noseoffset
  c2_xscale $fusescale
  c2_yscale $fusescale
```

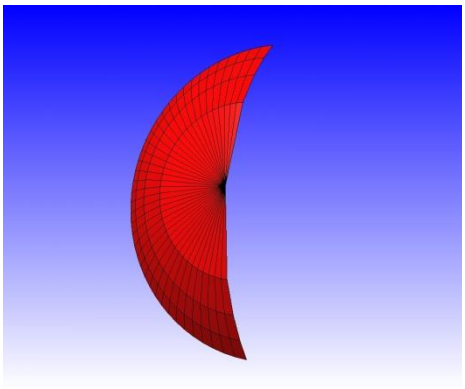
```
  c2_yoffset 0.0
  station $nl2 / $nl2
curve lofted forebulk1
  curve1 sc
  curve2 forebullet
  c1_xscale $capradius
  c1_yscale $capradius
  c1_yoffset $noseoffset
  c2_xscale 1
  c2_yscale 1
  c2_yoffset 0.0
  station $bulkhead1 / $nl2
curve lofted forebulk2
  curve1 sc
  curve2 forebullet
  c1_xscale $capradius
  c1_yscale $capradius
  c1_yoffset $noseoffset
  c2_xscale 1
  c2_yscale 1
  c2_yoffset 0.0
  station $bulkhead2 / $nl2
curve lofted dome50
  curve1 sc
  c1_xscale $tankscale
  c1_yscale $tankscale
  taper elli
  station 0.50
curve lofted aftdome
  curve1 sc
  station 1 / 3
  taper elli
  c1_xscale $tankscale
  c1_yscale $tankscale
list ccurves
list lcurves
```

Following the completion of the curve definition section, the `list` debugging command is used to confirm the creation of all of the desired curves. This step is optional. In the text output from *Loft*, these commands produce:

```
Current List of Compound Curves

 0   body
 1   plb1
 2   plb2

Read line: list lcurves

Current List of Lofted Curves

 0   forebullet
 1   fore25
 2   fore50
 3   dome50
```

Having finished reading the curve-defining include file, *Loft* returns to reading the main input file at the previous location. The input then starts defining the vehicle, starting at the nose. Note the use of the previously defined `$circnodes` variable. Also notice that all external components are given the "OML" mark and that both the external skin and the structural bulkheads are given the "fuselage" mark.

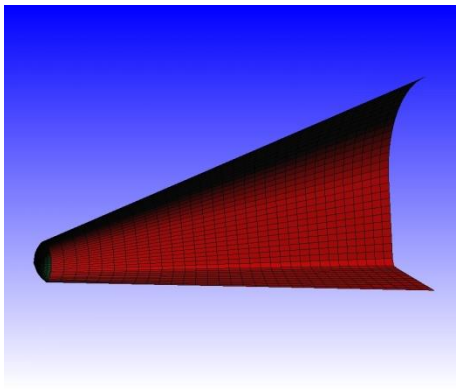
```
#
# Build vehicle
#
# =================== Nose ==========================
object dome nosecap
  curve1 sc
  c1_xscale $capradius
  c1_yscale $capradius
  c1_yoffset $noseoffset
  length $caplength
  nodes_circ $circnodes
  nodes_axial $caplength * $meshdens * -1. * 2.
  taper para
  components_axial 1
  components_circ 1
  mark element OML
  mark element fuselage
  transz $caplength
```



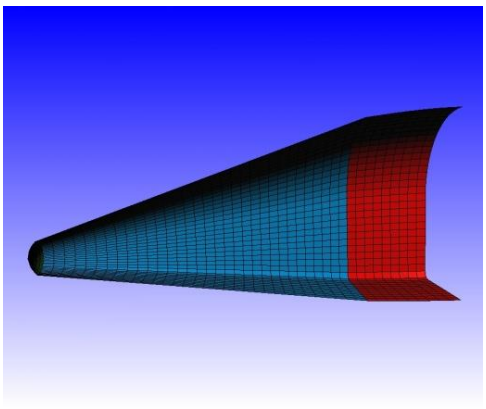
The forebody section is created in multiple segments to force edges at 25 and 50% of its length. The nose-gear bulkheads will be placed at these positions and will stitch to the fuselage correctly.

```
object section forebody1
  curve2 forebulk1
  c2_xscale 1.0
  c2_yscale 1.0
  c2_yoffset 0.0
  length $bulkhead1
  nodes_axial $bulkhead1 * $meshdens
  components_axial 1
  mark element OML
  mark element fuselage
object section forebody2
  curve2 forebulk2
  c2_xscale 1.0
  c2_yscale 1.0
  c2_yoffset 0.0
  length $bulkhead2 - $bulkhead1
  nodes_axial $bulkhead2 - $bulkhead1 * $meshdens
  components_axial 1
  mark element OML
  mark element fuselage
object section forebody3
  curve2 forebullet
  c2_xscale 1.0
  c2_yscale 1.0
  c2_yoffset 0.0
  length $bulkhead3 - $bulkhead2
  nodes_axial $bulkhead3 - $bulkhead2 * $meshdens
  components_axial 1
  mark element OML
```

```
  mark element fuselage
```



```
object section forebody4
  curve2 body
  c2_xscale $fusescale
  c2_yscale $fusescale
  c2_yoffset 0.0
  length $bulletbulk
  nodes_axial $bulletbulk * $meshdens
  components_axial 1
  mark element OML
  mark element fuselage
```



The `null` command below does not actually do anything. But, it does force *Loft* to generate the "fore-body4" object and update the `@transz` system variable to reflect the new object. The `$noseend` variable is used later when the full vehicle is assembled from major components. Beams are also created along the bulkhead/nose intersection. The `zdroop` parameters on the two bulkheads are used to move the center node of the bulkhead down from the vehicle centerline to the object center.

```
null
define noseend @transz
# create 2 bulkheads using lofted curves
```

```
object dome Nose Front Bulk
  curve1 forebulk1
  c1_xscale 1.0
  c1_yscale 1.0
  zdroop $noseoffset * -0.71
  length -0.0001
  transz $bulkhead1 + $caplength
  nodes_axial 100 * $meshdens
  zdist 0.6
  components_axial 1
  mark element fuselage
object dframe nose fwd ring frame
  count 1
  mark element fuselage
object dome Nose Mid Bulkhead
  curve1 forebulk2
  zdroop $noseoffset * -0.48
  transz $bulkhead2 + $caplength
  length -0.0001
  nodes_axial 100 * $meshdens
  zdist 0.7
  components_axial 1
  mark element fuselage
object dframe nose mid ring frame
  count 1
  mark element fuselage
```

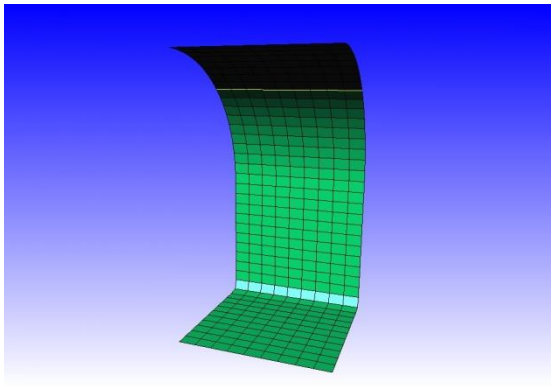
Finally, the completed nose is moved to the *Loft* internal clipboard with the `store` command. Remember that the `store` command resets all object defaults and starts a new stack with no nodes or elements.

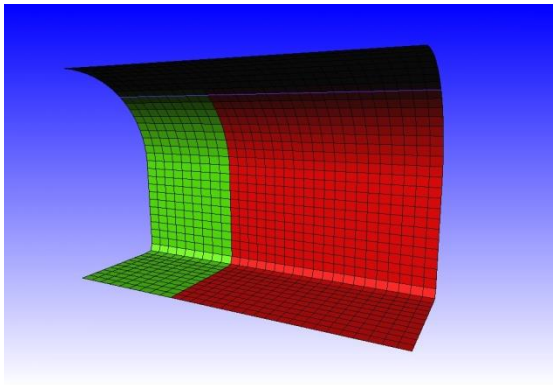
```
store nose
```

The constant cross-section portion of the fuselage is defined in several sections. These cuts were made to force the creation of nodes at axial stations that will later have bulkheads. Each fuselage portion also has a longeron created at 18 percent around the curve. The longeron runs the length of the rest of the vehicle, including along the edge of the payload bay door and onto the thrust structure.

```
# ===================== Fuselage ==========================
# Along fwd tank barrel
object section fuselage1
  curve1 body
  curve2 body
  c1_xscale $fusescale
  c1_yscale $fusescale
  c2_xscale $fusescale
  c2_yscale $fusescale
  length $mid_bulk - $half_lh2_nose
  nodes_axial $mid_bulk - $half_lh2_nose * $meshdens
  nodes_circ $circnodes
```

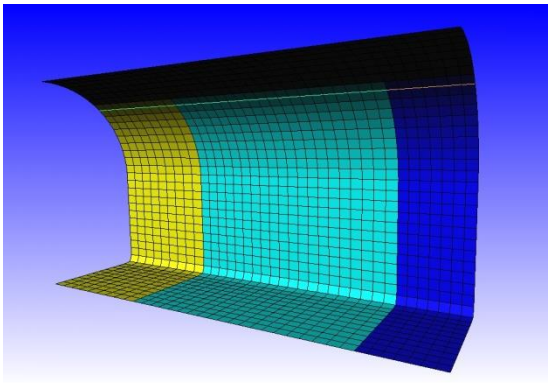
```
  components_axial 1
  components_circ 1
  mark element OML
  mark element fuselage
object frame longeron1
  count 1
  align axial
  position $longeron_pos
  mark element fuselage
object section fuselage1.5
  curve1 body
  curve2 body
  c1_xscale $fusescale
  c1_yscale $fusescale
  c2_xscale $fusescale
  c2_yscale $fusescale
  length $mid_bulk
  nodes_axial $mid_bulk * $meshdens
  nodes_circ $circnodes
  components_axial 1
  components_circ 1
  mark element OML
  mark element fuselage
object frame longeron1
  count 1
  align axial
  position $longeron_pos
  mark element fuselage
```





```
# Along fwd tank aft dome
object section fuselage2
  length $fwd_tank_skirt
  nodes_axial $fwd_tank_skirt * $meshdens
```

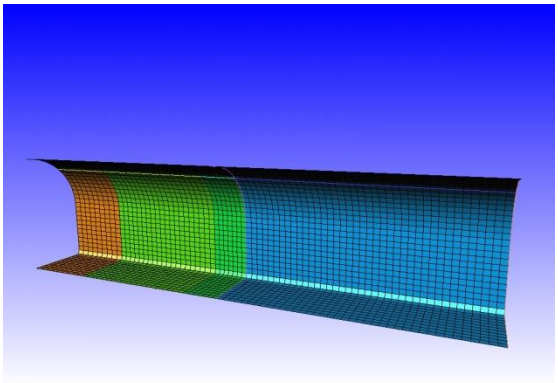
```
  nodes_circ $circnodes
  components_axial 1
  mark element OML
  mark element fuselage
object frame longeron2
  count 1
  align axial
  position $longeron_pos
  mark element fuselage
null
define plb_start @transz
define plb_center $plb_start + $plb_half
```



In this case, the `null` command is not strictly necessary to force `@transz` to have the desired value; the longeron object definition caused the generation of the "fuselage2" object and the updating of the `@transz` system variable.

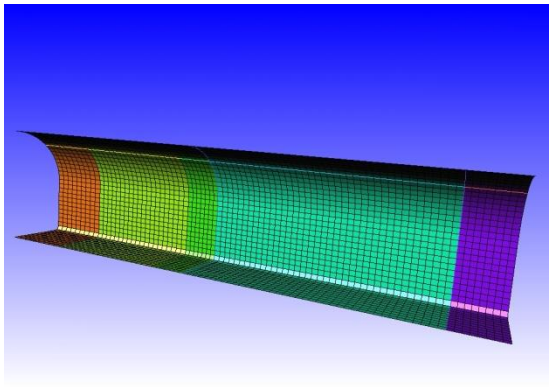
```
# Payload Bay fuselage split into thirds for supports at 1/3 and
2/3
object section fuselage_center_bay
  length $fuse_center_bay / 3.
  nodes_axial $fuse_center_bay / 3. * $meshdens
  nodes_circ $circnodes
  components_axial 1
  mark element OML
  mark element fuselage
object frame longeron3
  count 1
  align axial
  position $longeron_pos
  mark element fuselage
object frame forward pl ring
  count 1
  align circ
  position 0.0
  mark element fuselage
object section fuselage_center_bay
  length $fuse_center_bay / 3.
  nodes_axial $fuse_center_bay / 3. * $meshdens
  nodes_circ $circnodes
  components_axial 1
  mark element OML
  mark element fuselage
object frame longeron3
  count 1
```

```
  align axial
  position $longeron_pos
  mark element fuselage
object section fuselage_center_bay
  length $fuse_center_bay / 3.
  nodes_axial $fuse_center_bay / 3. * $meshdens
  nodes_circ $circnodes
  components_axial 1
  mark element OML
  mark element fuselage
object frame longeron3
  count 1
  align axial
  position $longeron_pos
  mark element fuselage
object frame aft pl ring
  count 1
  align circ
  position 1.0
  mark element fuselage
```

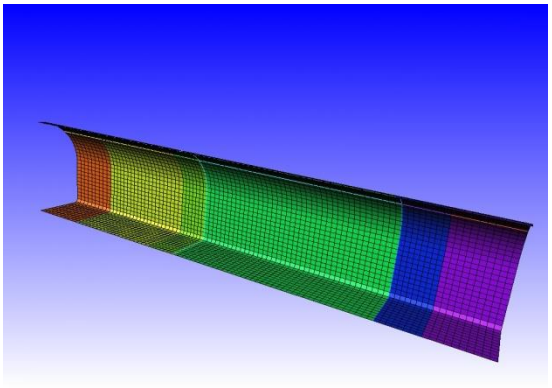


```
# Fuselage along Aft tank fwd skirt
object section fuselage4
  length $aft_tank_skirt
  nodes_axial $aft_tank_skirt * $meshdens
  nodes_circ $circnodes
  components_axial 1
  mark element OML
  mark element fuselage
object frame longeron4
  count 1
  align axial
  position $longeron_pos
```

```
  mark element fuselage
```

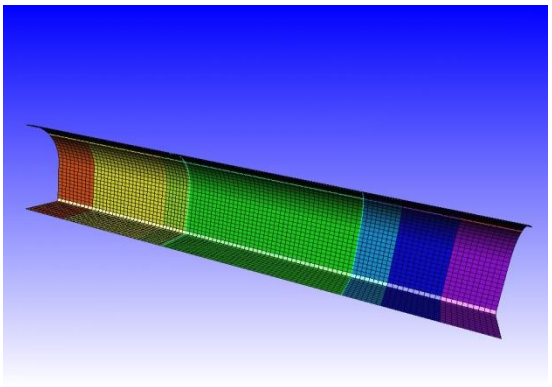


```
# Fuselage along Aft tank barrel
object section fuselage5
  length $aft_tank + 64
  nodes_axial $aft_tank + 64 * $meshdens
  nodes_circ $circnodes
  components_axial 1
  mark element OML
  mark element fuselage
object frame longeron5
  count 1
  align axial
  position $longeron_pos
  mark element fuselage
```



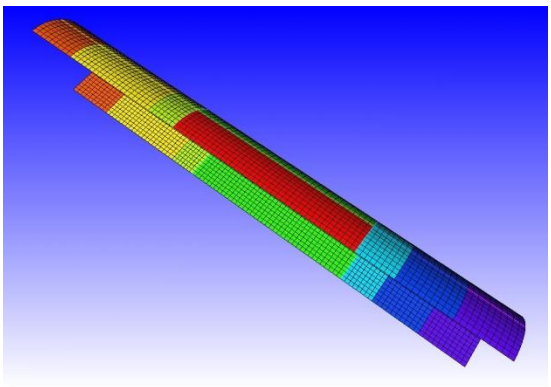
```
# Fuselage along Aft tank aft skirt
object section fuselage6
  length $aft_skirt
  nodes_axial $aft_skirt * $meshdens
  nodes_circ $circnodes
  components_axial 1
  mark element OML
  mark element fuselage
```

```
object frame longeron6
  count 1
  align axial
  position $longeron_pos
  mark element fuselage
define fuseend @transz + $noseend
```



The next step is to add some detail to the payload bay. The region command is used to modify the physical property assignment of elements along the upper section of fuselage object "fuselage3." These updated elements represent the payload bay doors.

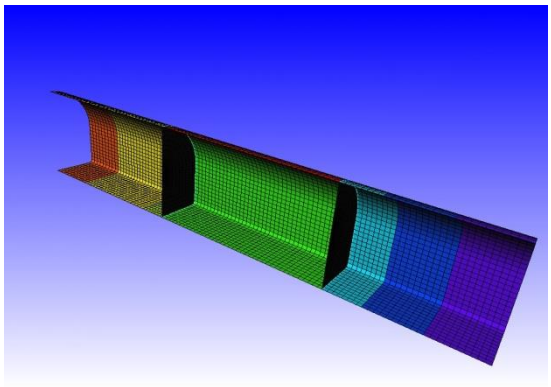
```
# ================== Payload bay ======================
region
  iadd box 0. 102. $plb_center 130. 130. $plb_length
  pprem fuselage2
  pprem fuselage4
  setpp payload doors
```



Then full bulkheads are added at the front and rear of the payload bay and partial, support, bulkheads are added at the 1/3 and 2/3 positions in the bay.

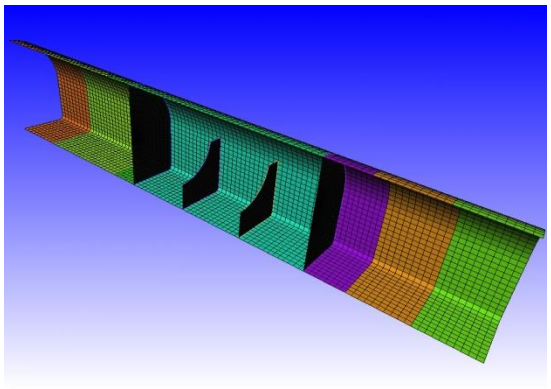
```
object dome payload bay fwd bulkhead
```

```
  curve1 body
  c1_xscale $fusescale
  c1_yscale $fusescale
  taper bulk
  transz $plb_start
  transy 0.0
  transx 0.0
  nodes_circ $circnodes
  components_axial 1
  mark element fuselage
object dome payload bay aft bulkhead
  curve1 body
  taper bulk
  relz $plb_length
  transy 0.0
  transx 0.0
  components_axial 1
  mark element fuselage
```



```
object section payload bay fwd support
  curve1 plb1
  curve2 plb2
  length 0.0
  transz $fwd_tank + $fwd_tank_skirt + $plb_third
  components_axial 1
  components_circ 1
  nodes_axial 100 * $meshdens
  nodes_circ $circnodes * 0.6 + 1
  mark element fuselage
object frame fwd plb support frame
  count 1
  align axial
  position 0.0
  mark element fuselage
```

```
object frame fwd plb support frame
  count 2
  align circ
  mark element fuselage
object section payload bay aft support
  curve1 plb1
  curve2 plb2
  length 0.0
  relz $plb_third
  components_axial 1
  components_circ 1
  nodes_axial 100 * $meshdens
  nodes_circ $circnodes * 0.6 + 1
  mark element fuselage
object frame aft plb support frame
  count 1
  align axial
  position 0.0
  mark element fuselage
object frame aft plb support frame
  count 2
  align circ
  mark element fuselage
```



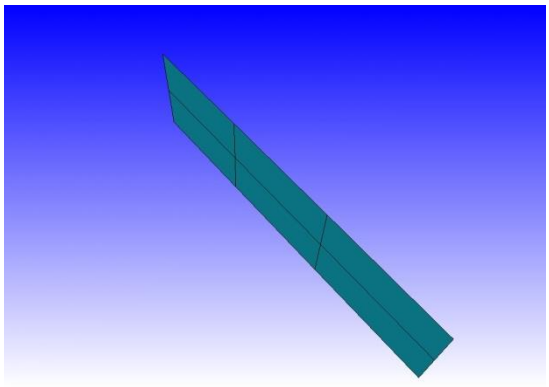
Finally, the completed fuselage component is moved so that it is immediately aft of the nose using the previously created `$noseend` variable. Then, the new end location is stored. Finally, the full stack is moved onto *Loft's* internal clipboard and a new stack is started.

```
move
  transz $noseend
define plb_end @transz
store fuselage
```

The next major component created in the input deck is the wing. The wing has two trapezoidal sections: a narrow, inboard, strake and a wider outboard main section. The strake has one spar, positioned at the 10

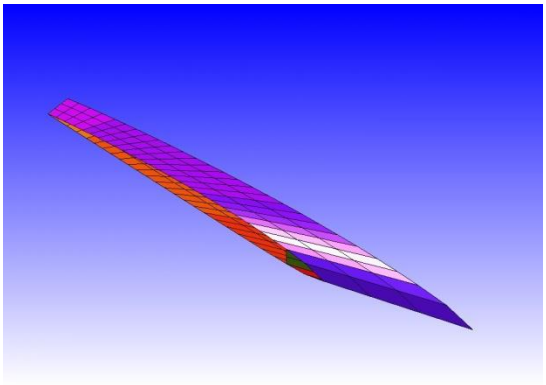
percent chord location. The strake is generated first. When the strake skin is created, it is created as if there were additional spars at the 36 and 82 percent chord locations. This forces a line of nodes to be created along the phantom spars and allows correct stitching with the main wing which does have spars at all three positions. Note the extensive use of the `gen_XXX` flags and the use of the `mark` command to mark only the wing skin as "OML."

```
# ======================== Wing ==========================
define wingoffset $noseend + $plb_start - 115.87
object wing strake spar
  chord $strakechord
  span $strakespan
  taper $straketaper
  sweep $strakesweep
  rootnaca 2407
  tipnaca 2408
  sparpos $spar1
  ribpos reset
  notip 1
  meshchord $strakechord * $meshdens / 700.
  meshspan $strakespan * $meshdens / 20.
  meshthick $meshdens / 2.
  transz $wingoffset
  relx $fusescale
  rely $fusescale * -.9314
  gen_up_skin off
  gen_low_skin off
  gen_ribs off
  mark element wing
```



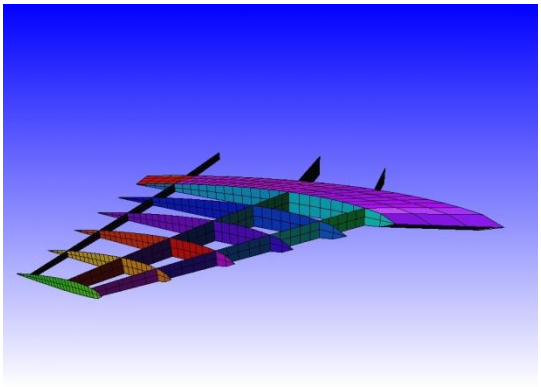
```
#
# Generate the rest of the strake
# Position spars so that the skin aligns with the main wing
#   but do not actually generate the spar elements
object wing strake
  sparpos reset
  sparpos $spar1
  sparpos $spar2
```

```
  sparpos $spar3
  notip 1
  gen_spars off
  mark element OML
  mark element wing
```



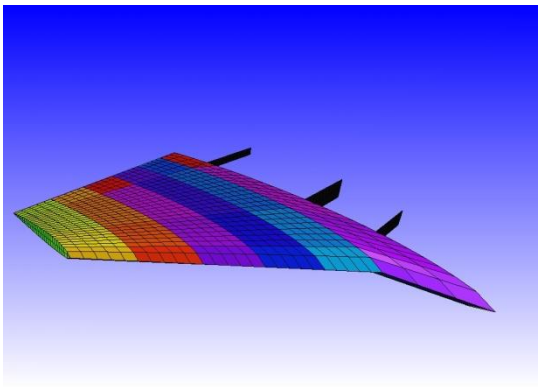
The main wing is also specified as two objects. The reason for this is to apply the “OML” mark to only the wing skin.

```
object wing mainwing stiffeners
  chord $mainchord
  span $mainspan
  meshchord @wing.mesh_chord / @wing.taper
  taper $maintaper
  sweep $mainsweep
  rootnaca 2408
  tipnaca 2313
  ribpos reset
  ribpos 20.
  ribpos 40.
  ribpos 60.
  ribpos 80.
  relx $strakespan
  relz $mainoffset
  wingbox $fusescale + $strakespan
  gen_up_skin off
  gen_low_skin off
  nowbrib 1
  mark element wing
```



A careful examination of the crank area between the strake and the main wing will show that the strake is properly stitched to the main wing along the rib at the crank location. The strake skin is also attached to its leading edge (10 percent) spar but is not attached to any of the carry-through spars. Depending on element flexibility, some manual stitching could be required to connect the strake root rib to the carry-through spars.

```
object wing mainwing skin
  wingbox 0.0
  notip 1
  gen_ribs off
  gen_spars off
  mark element OML
  mark element wing
```



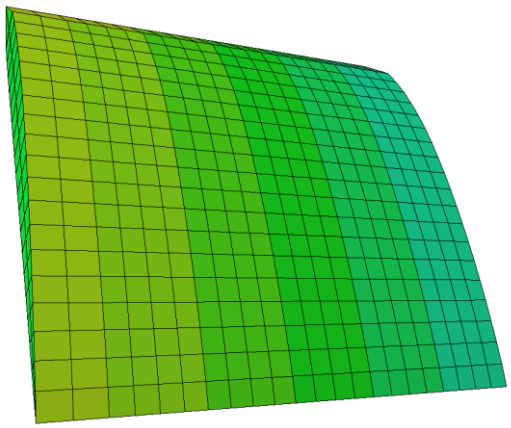
Depending on your use of the OML group it may or may not matter that we've added the skin but missed adding the tip rib to the OML group. So, to fix that issue we use the group with the ribs and retain only the outermost one. Then we write out the model.

```
# Add tip rib to OML
region
   mkadd mainwing stiffeners RIB ELEMS
   ikeep xgt $fusescale + $strakespan + $mainspan - 1.0
   mark element OML
write vrml orb-wing.wrl
store mainwing
list stacks
```

The `list stacks` debug command is optional and lists the stacks that have been stored on the internal clipboard.

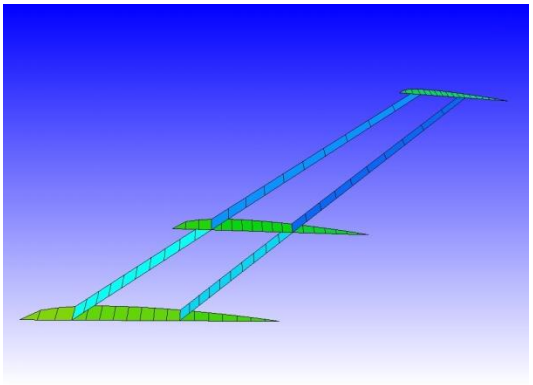
The next section generates an alternate "surrogate" version of the wing that has the same number of nodes and elements but is scaled to a square shape with coordinates ranging from zero to one. The coordinates represent the percent chord and percent span of the original wing. It does this by eliminating sweep and taper for the wing and then scaling the final model by the total span or root chord dimensions. This alternate model is used by an external code to generate pressure loads that are functions of wing position. This block can be deleted if it is unneeded.

```
# ===== Surrogate model of wing: [0,1] square with same nodes
define totalspan $strakespan + $mainspan
object wing surrogatewing
  chord $strakechord
  span $strakespan
  taper 1.0
  sweep 0.0
  naca 2407
  sparpos $spar1
  ribpos reset
  notip 1
  meshchord $strakechord * $meshdens / 700.
  meshspan $strakespan * $meshdens / 20.
  meshthick $meshdens / 2.
  transx 0.0
  transy 0.0
  transz 0.0
  mark element surrogatewing
object wing surrogatewing
  chord $strakechord
  span $mainspan
  taper 1.0
  sweep 0.0
  ribpos reset
  ribpos 20.
  ribpos 40.
  ribpos 60.
  ribpos 80.
  relx $strakespan
  mark element surrogatewing
move
  scalex 1. / $totalspan
  scaley 1. / $totalspan
  scalez 1. / $strakechord
  list variables
  write vrml orb-surwing.wrl
  store surrogatewing
```



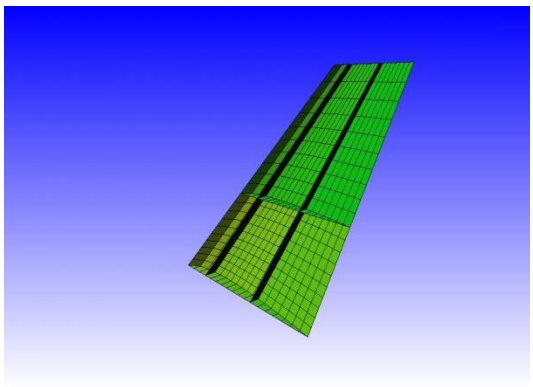
Next, the tail will be created as a new stack. As with the main wing components, it is created as two objects so that the skin can be marked as "OML."

```
# ===================== Tail ========================
object wing tail stiffeners
  chord $tailchord
  span $tailspan
  taper $tailtipchord / $tailchord
  sweep $tailsweep
  rootnaca 0613
  tipnaca 0618
  sparpos reset
  sparpos 19
  sparpos 60
  halfwing bottom
  ribpos reset
  ribpos 50
  wingbox 0.
  meshchord $tailchord * $meshdens / 300.
  meshspan $tailspan * $meshdens / 450.
  meshthick 0.02
  transz $fuseend - $tailchord
  rely $fusescale
  transx 0
  rotz 90
  gen_up_skin off
  gen_low_skin off
  mark element tail
```



```
object wing tail skin
  halfwing bottom
  gen_ribs off
  gen_spars off
  gen_up_skin on
```

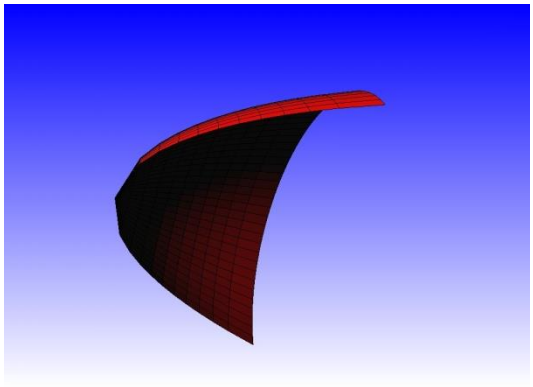
```
  gen_low_skin on
  mark element OML
  mark element tail
write vrml orb-tail.wrl
store tail
```



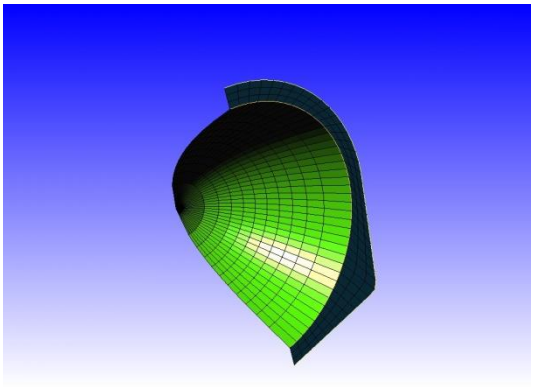
Then, the input deck specifies the forward tank. Two of the user-defined lofted curves created at the beginning of the file are used here to create the support bulkhead on the bullet shaped nose of the tank. Support bulkheads are given the "fuselage" mark and tank walls are all given the mark "LH2." These marks will be used later to extract just these elements from the full model.

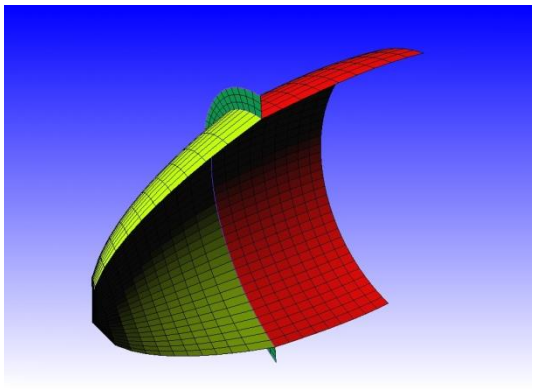
```
# ======================== Fwd Tank =====================
define fwd_tank_start $noseend - $bulletbulk
object dome fwd tank fwd dome
  curve1 dome50
  c1_xscale 1.
  c1_yscale 1.
  length -1 * $half_lh2_nose
  transx 0.0
  transy 0.0
  zdist 0.7
  transz $fwd_tank_start
  nodes_axial $half_lh2_nose * $meshdens
  nodes_circ $circnodes
  components_axial 1
  components_circ 1
  taper para
```

```
  mark element LH2
```

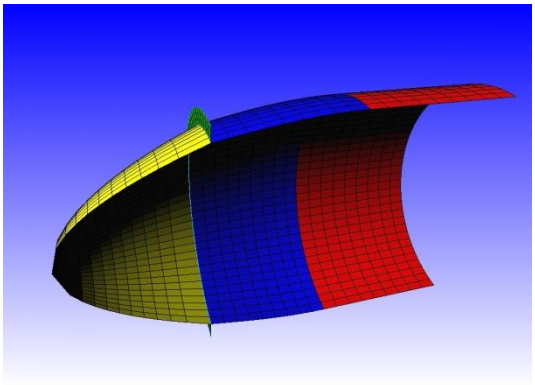


```
object section fwd tank fwd bulk
  curve2 forebullet
  length 0.0
  components_axial 1
  nodes_axial 100 * $meshdens
  mark element bulk
  mark element fuselage
object frame fwd fwd ring frame
  count 2
  mark element fuselage
object section fwd tank dome2
  curve1 dome50
  curve2 sc
  length $half_lh2_nose
  c1_xscale 1.
  c1_yscale 1.
  c2_xscale $tankscale
  c2_yscale $tankscale
  nodes_axial $half_lh2_nose * $meshdens
  components_axial 1
  taper cosine 0.5
  mark element LH2
```

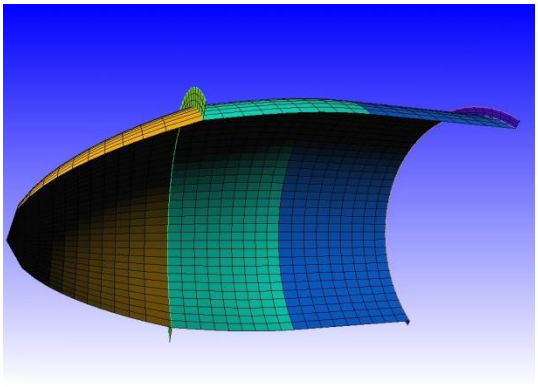




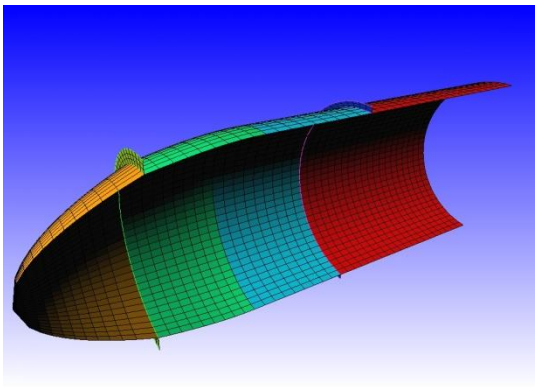
```
object section fwd tank barrel pt 1
  length $mid_bulk - $half_lh2_nose
  nodes_axial $mid_bulk - $half_lh2_nose * $meshdens
  components_axial 1
  mark element LH2
```



```
object section fwd tank mid bulk
  curve1 body
  curve2 sc
  c1_xscale $fusescale
  c1_yscale $fusescale
  length 0.0
  components_axial 1
  nodes_axial 100 * $meshdens
  mark element bulk
  mark element fuselage
object frame fwd mid ring frame
  count 2
  mark element fuselage
```

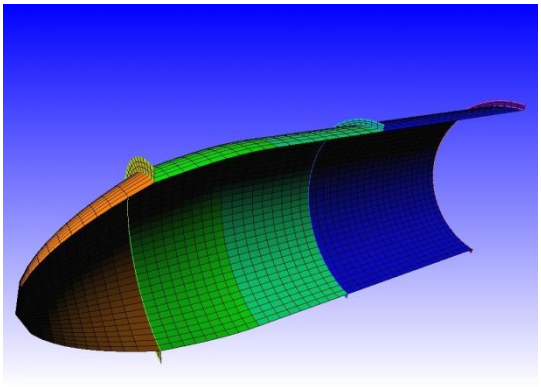


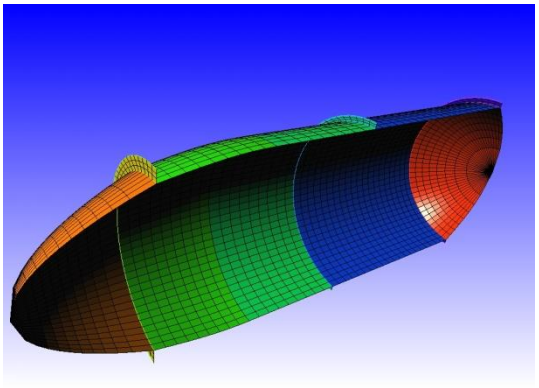
```
object section fwd tank barrel pt 2
  length $mid_bulk
  nodes_axial $mid_bulk * $meshdens
  components_axial 1
  mark element LH2
```



```
object section fwd tank aft bulk
```

```
  curve1 body
  curve2 sc
  c1_xscale $fusescale
  c1_yscale $fusescale
  length 0.0
  components_axial 1
  nodes_axial 100 * $meshdens
  mark element bulk
  mark element fuselage
object frame fwd aft ring frame
  count 2
  mark element fuselage
object dome fwd tank aft dome
  length 50
  nodes_axial 50 * $meshdens
  components_axial 1
  mark element LH2
write vrml orb-lh2.wrl
store fwd_tank
```

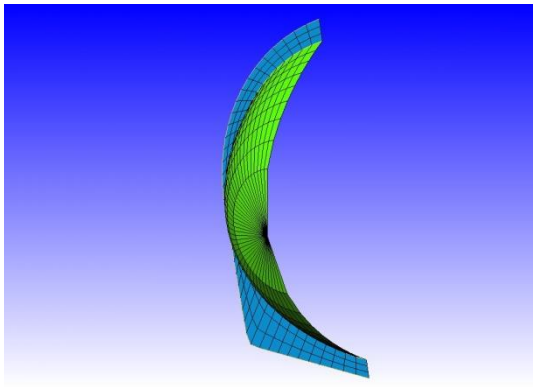




The aft tank is built in a similar process to the forward tank. It is shorter but still has mid-dome bulkheads like on the front of the forward tank.

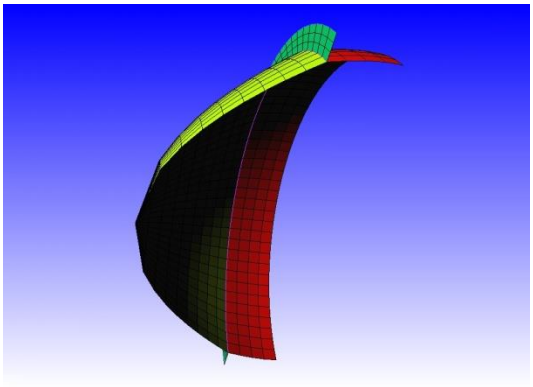
```
# ====================== Aft Tank ========================
object dome aft tank fwd dome
  curve1 aftdome
  length $aft_support - $aft_dome
  c1_xscale 1.
  c1_yscale 1.
  nodes_axial $aft_support - $aft_dome * $meshdens * -1.
  nodes_circ $circnodes
  components_axial 1
  components_circ 1
  taper para
```

```
  mark element LOX
object section aft tank fwd bulk
  curve1 body
  curve2 aftdome
  c1_xscale $fusescale
  c1_yscale $fusescale
  c2_xscale 1.
  c2_yscale 1.
  length 0.0
  components_axial 1
  nodes_axial 50 * $meshdens
  mark element bulk
  mark element fuselage
null
define aftfwdbulk @transz
object frame aft fwd ring frame
  count 2
  mark element fuselage
```

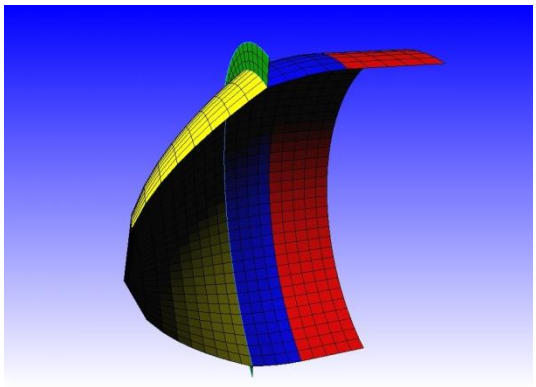


```
object section aft tank fwd curve
  curve2 sc
  c2_xscale $tankscale
  c2_yscale $tankscale
  length $aft_support
  taper cosine 0.5
  mark element LOX
  components_axial 1
```

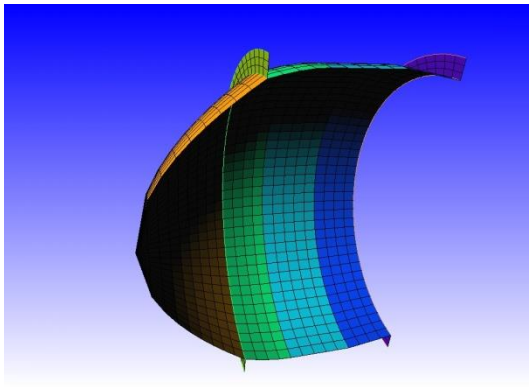
```
  nodes_axial $aft_support * $meshdens
```



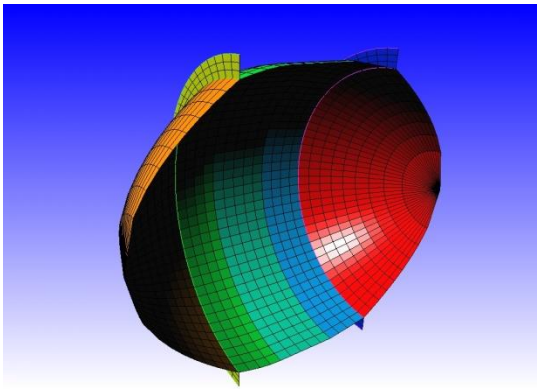
```
object section aft tank barrel
  curve1 sc
  length $aft_tank
  nodes_axial $aft_tank * $meshdens
  components_axial 1
  mark element LOX
object section aft tank aft curve
  curve2 aftdome
  c2_xscale 1.
  c2_yscale 1.
  length $aft_support
  taper power 1.0
  mark element LOX
  components_axial 1
  nodes_axial $aft_support * $meshdens
object section aft tank aft bulk
  curve1 body
  curve2 aftdome
  c1_xscale $fusescale
  c1_yscale $fusescale
  c2_xscale 1.
  c2_yscale 1.
  length 0.0
  components_axial 1
  nodes_axial 50 * $meshdens
  mark element bulk
  mark element fuselage
null
define aftaftbulk @transz
object frame aft aft ring frame
  count 2
```



```
  mark element fuselage
```



```
object dome aft tank aft dome
  curve1 aftdome
  length $aft_dome - $aft_support
  c1_xscale 1.
  c1_yscale 1.
  nodes_axial $aft_dome - $aft_support *
$meshdens
  components_axial 1
  taper para
  mark element LOX
```



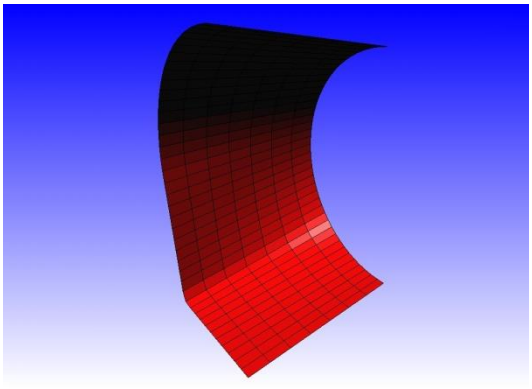
The position of the aft tank is computed from five previously saved lengths. The definition should all be on one line in the actual input file, not wrapped as it is in this manual.

```
define aft_tank_start $noseend + $fwd_tank + $fwd_tank_skirt +
$fuse_center_bay + $aft_tank_skirt
move
  transz $aft_tank_start
define aft_tank_end $aft_tank_start + @transz
write vrml orb-lox.wrl
store aft_tank
```

The next object created is a notional thrust structure. It makes extensive use of stiffeners created with `frame` and `dframe` objects. The first piece created accomplishes the transition from the half-loaf-of-bread "body" user-defined curve to a semi-circle.

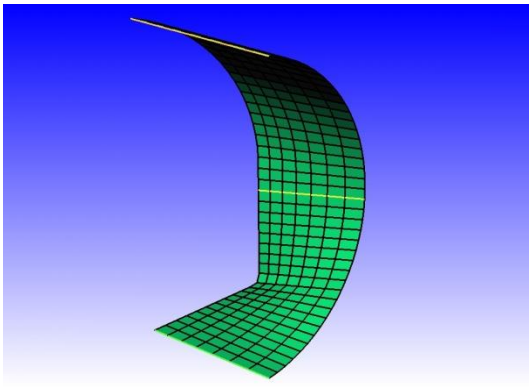
```
# ================= Thrust structure ====================
object section thrust cone
  curve1 body
  curve2 sc
  c1_xscale $fusescale
  c1_yscale $fusescale
  c2_xscale $thrustconescale
  c2_yscale $thrustconescale
```

```
  length $thrustconelength
  components_axial 1
  components_circ 1
  nodes_circ $circnodes
  nodes_axial $aft_skirt + 10. * $meshdens
  mark element fuselage
```



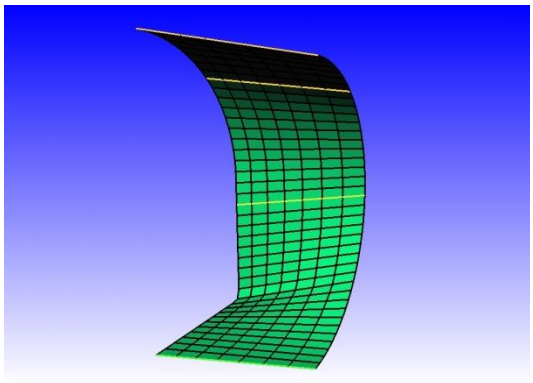
Five axial stiffeners are created. The first three (at 0, 50, and 100 percent of the circumference) are created as one object. Then, two individual axial stiffeners are added, one at the `$longeron_pos` position (18 percent) and one at 75 percent.

```
object frame thrust stiffeners
  count 3
  align axial
  mark element fuselage
```

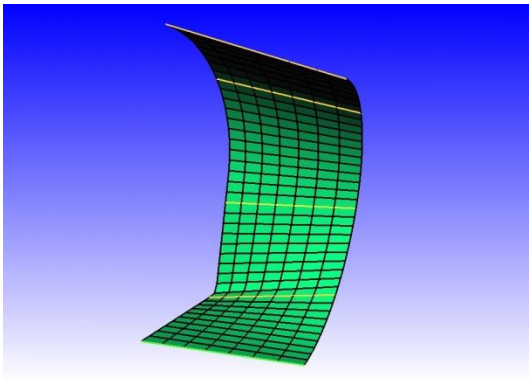


```
object frame thrust stiffeners
  count 1
  position $longeron_pos
  align axial
```

```
  mark element fuselage
```

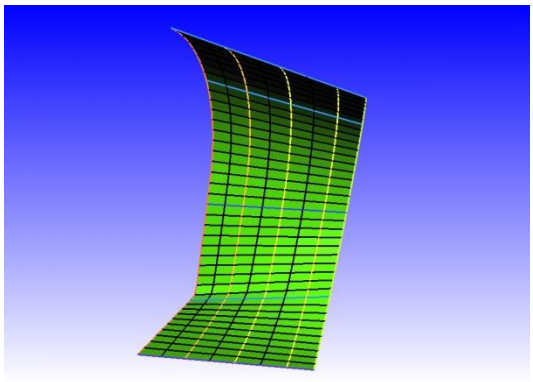


```
object frame thrust stiffeners
  count 1
  position 0.75
  align axial
  mark element fuselage
```



Five circumferential stiffeners are added:

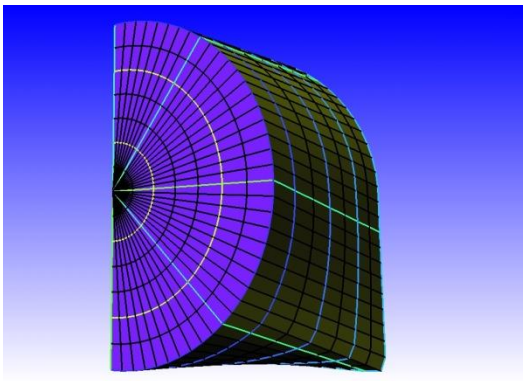
```
object frame thrust cone rings
  count 5
  align circ
  mark element fuselage
```



A circular flat plate is added with similar stiffeners:

```
object dome thrust plane
  taper bulk
  length 0.0
  components_axial 1
  nodes_axial $meshdens * 80.
  mark element fuselage
object dframe thrust rings
```

```
  align circ
  count 1
  position 0.2
  mark nodes engine ring
  mark element fuselage
object dframe engine rings
  align circ
  count 1
  position 0.7
  mark element fuselage
object dframe thrust diags
  align axial
  count 3
  mark element fuselage
object dframe thrust diags
  align axial
  position $longeron_pos
  count 1
  mark element fuselage
object dframe thrust diags
  align axial
  position 0.75
  count 1
  mark element fuselage
```



```
move
  transz $aft_tank_end
define thrust_end $aft_tank_end + $aft_skirt + 10.
write vrml orb-thrust.wrl
store thrust
```

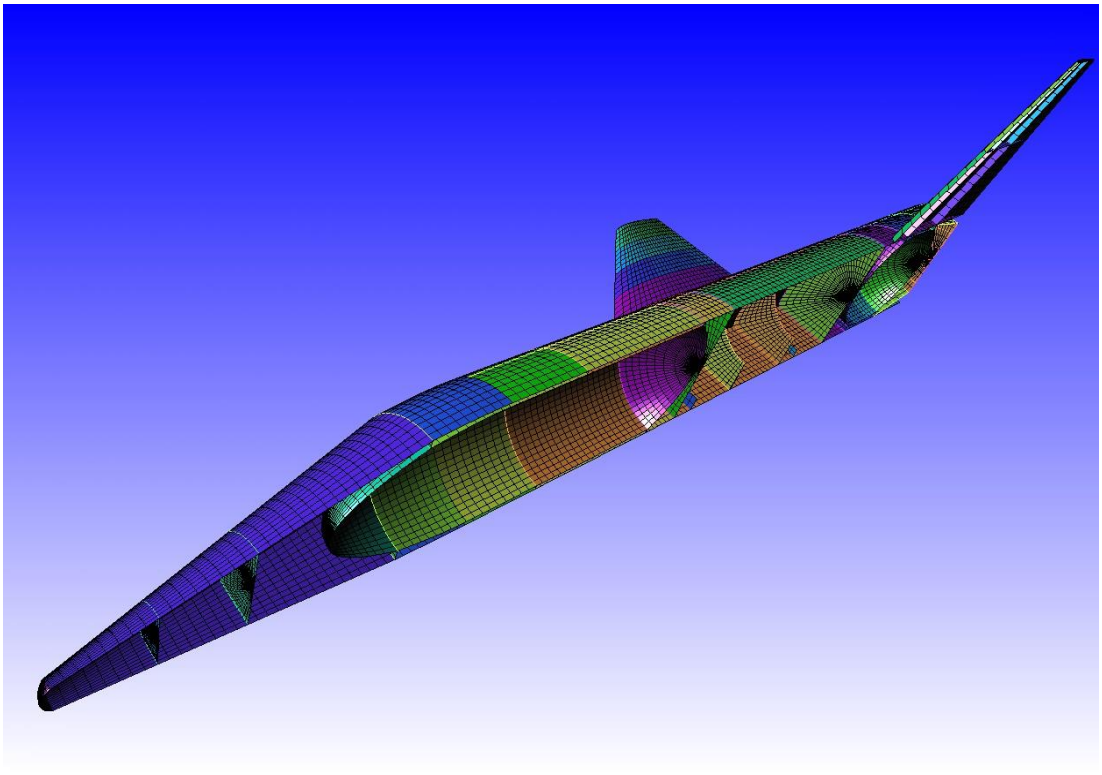
After positioning the thrust structure at the calculated location, it is saved to the clipboard.

All of the components of the vehicle have been created and stored. Next, they can be recalled in various combinations for use. The first combination is the full vehicle with all the components in the correct position.

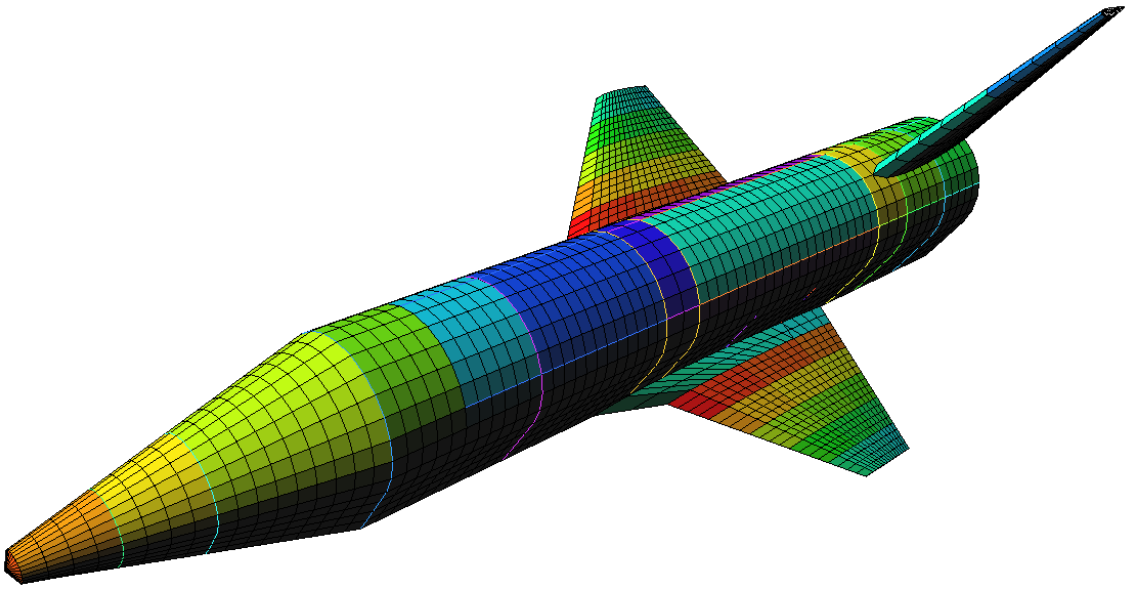
Each recall command performs a node equivalence operation that stitches the model together where nodes are coincident. This equivalence operation tends to be slow. Once they are recalled, the whole vehicle is rotated such that the x coordinate direction becomes the axial axis. Then, VRML and NASTRAN files of the full model are written.

Note that prior to actual analysis with the model, the wing and tail need to be stitched to the fuselage. This operation will be accomplished later in the input file.

```
# ==================== Assembly ======================
recall nose
recall fuselage
recall mainwing
recall tail
recall fwd_tank
recall aft_tank
recall thrust
# rotate so that x is aft
move
 roty 90
store vehicle
recall vehicle
# ================== Write models ====================
vrml rainbow
write vrml tsto2025half.wrl
write nastran tsto2025half.bdf
```



Now, the first variable defined in the file, `$fullvehicle` is checked. If it is true, a mirrored model is created and written out to a VRML file; if it is false, the mirror and VRML writing are skipped. Then, the final model (full or half) is stored as "stitchme."

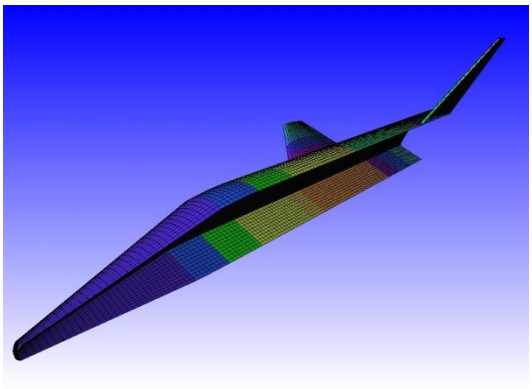


```
# =============== Mirrored model ========================
if $fullvehicle = 0
goto 100
mirror z
write vrml tsto2025mirrored.wrl
#
linelabel 100
store stitchme
recall stitchme
```

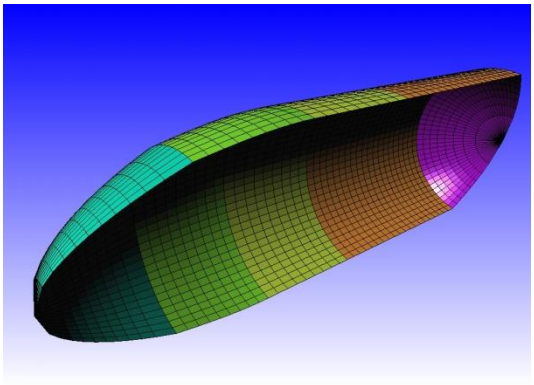
Next, the region mode is used to write out various partial versions of the model. These partial models retain the node, element, and property numbering of the full model. They are used for mapping of external aerodynamic loads (to the "OML" sub-model) and internal tank loads (to the "LH2" and "LOX" sub-models). Note the selection of elements based on the labels assigned with the "mark" command during model creation.

```
# ============ Models for mapping & analysis =============
region
  mkadd OML
  rwrite vrml tsto2025OML.wrl
```

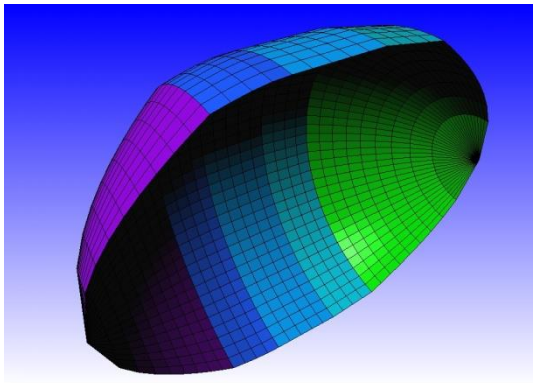
```
  rwrite nastran tsto2025OML.bdf
```



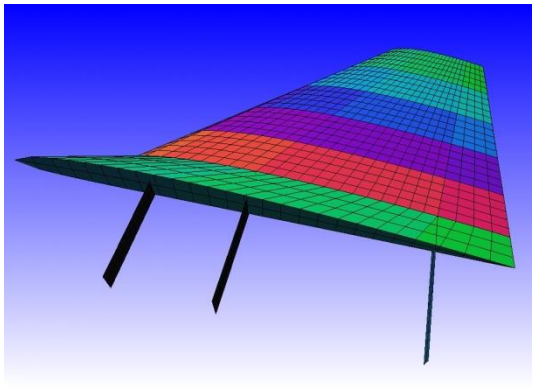
```
region
  mkadd LH2
  rwrite vrml tsto2025LH2.wrl
  rwrite nastran tsto2025LH2.bdf
```



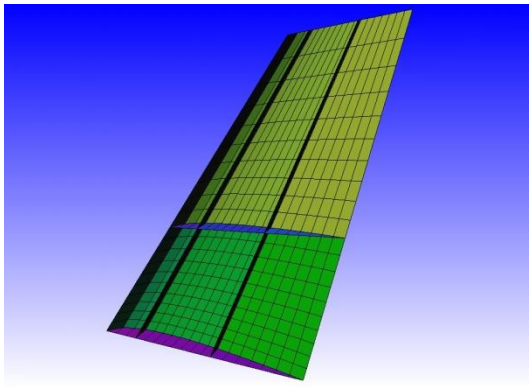
```
region
  mkadd LOX
  rwrite vrml tsto2025LOX.wrl
  rwrite nastran tsto2025LOX.bdf
```



```
region
  mkadd wing
  rwrite vrml tsto2025wing.wrl
  rwrite nastran tsto2025wing.bdf
```



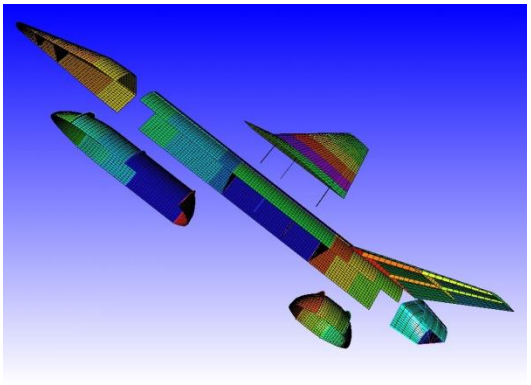
```
region
  mkadd tail
  rwrite vrml tsto2025tail.wrl
  rwrite nastran tsto2025tail.bdf
```



Next, an expanded version of the model is created for use in slides and presentations.

```
# ============== Expanded model for figures ===============
new
recall nose
move
  transz -100
recall fuselage
move
  transz 0
  transx -200
recall mainwing
move
  transx 200
  transy -100
```

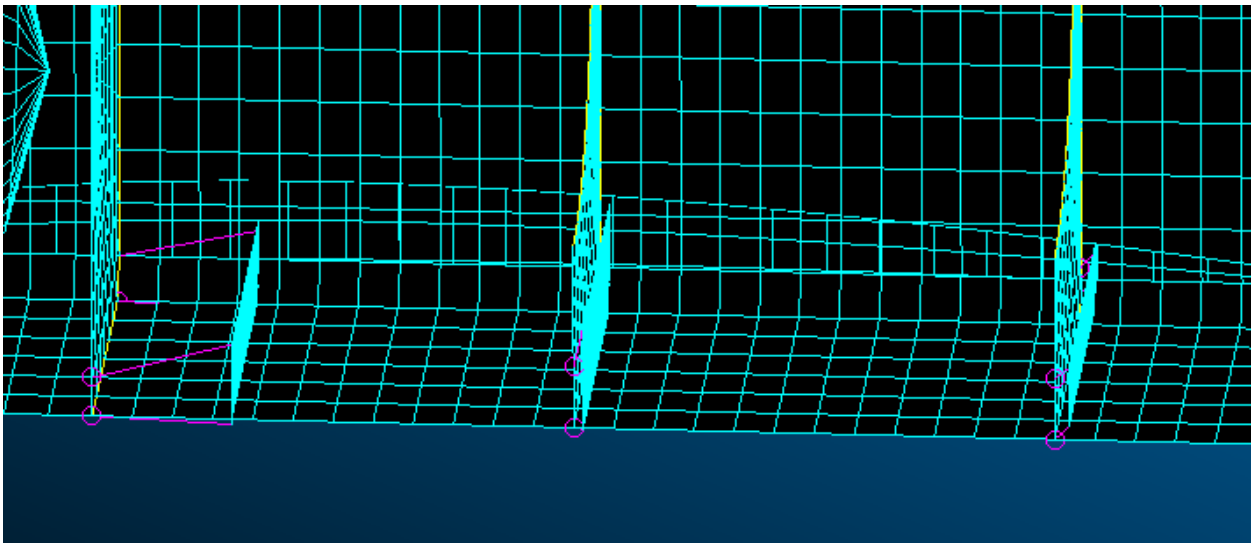
```
recall tail
move
  transy 100
  transx 200
recall fwd_tank
recall aft_tank
move
  transx -200
  transz -200
recall thrust
move
roty 90
write vrml tsto2025-exp.wrl
```



As previously discussed, one step that is required prior to using the model in a finite element analysis is to stitch the wing and the tail to the fuselage. The next section of *Loft* input accomplishes this stitching. It does this by using region mode to locate the corners of the wing and tail spars and the corners of the bulkheads near those spars. Then, RBE objects are created to connect the appropriate corners. Note that it first recalls either the half or full model that was saved as "stitchme" following the possible mirroring operation.

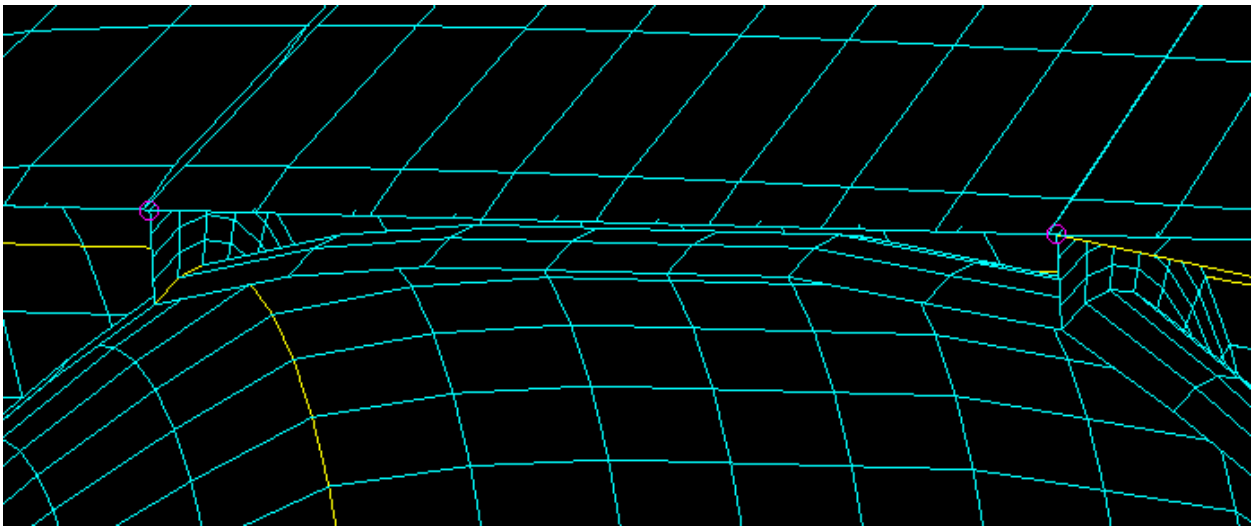
```
# =============== RBE connections for wing/tail ====================
linelabel 1000
new
recall stitchme
define spar1pos $spar1 * $mainchord / 100. + $wingoffset + $mainoffset
define spar2pos $spar2 * $mainchord / 100. + $wingoffset + $mainoffset
define spar3pos $spar3 * $mainchord / 100. + $wingoffset + $mainoffset
list mesh
# find corner nodes within fuselage for all 3 spars
region
   mkadd mainwing ribs spars CARRYTHR NODES
   irem xgt $spar1pos + 5.
   irem zlt $fusescale * -1. + 5.
   irem zgt $fusescale - 5.
   corner mainwing fwd corners
```

```
region
   mkadd mainwing ribs spars CARRYTHR NODES
   irem xlt $spar2pos - 5.
   irem xgt $spar2pos + 5.
   irem zlt $fusescale * -1. + 5.
   irem zgt $fusescale - 5.
   corner mainwing mid corners
region
   mkadd mainwing ribs spars CARRYTHR NODES
   irem xlt $spar3pos - 5.
   irem zlt $fusescale * -1. + 5.
   irem zgt $fusescale - 5.
   corner mainwing aft corners
null
# find corner nodes near spars on fuselage bulkheads
region
   ppadd payload bay fwd bulkhead
   irem yge -80.0
   corner Front bulkhead corners
region
   ppadd payload bay fwd support
   irem yge -80.0
   corner Mid bulkhead corners
region
   ppadd payload bay aft support
   irem yge -80.0
   corner Aft bulkhead corners
# connect spars to bulkheads with rbes
object rbe forward wing rbes
   group1 mainwing fwd corners
   group2 Front bulkhead corners
object rbe mid wing rbes
   group1 mainwing mid corners
   group2 Mid bulkhead corners
object rbe aft wing rbes
   group1 mainwing aft corners
   group2 Aft bulkhead corners
```



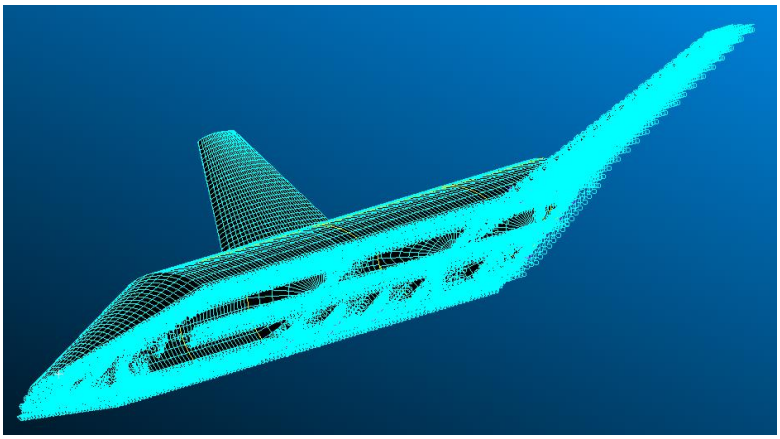
RBEs shown in purple

```
# tail attachment RBEs
#
# find nodes to connect
region
   mkadd aft tank fwd bulk ALL NODES
   irem yle $fusescale - 2.
   irem ygt $fusescale
   corner tank fwd bulkhead corners
region
   mkadd aft tank aft bulk ALL NODES
   irem yle $fusescale - 2.
   irem ygt $fusescale
   corner tank aft bulkhead corners
region
   mkadd tail stiffeners ROOT SPAR NODES
   irem xle $aftfwdbulk + $aftaftbulk / 2.
   corner tail aft spar nodes
region
   mkadd tail stiffeners ROOT SPAR NODES
   irem xge $aftfwdbulk + $aftaftbulk / 2.
   corner tail fwd spar nodes
object rbe forward tail rbes
   group2 tank fwd bulkhead corners
   group1 tail fwd spar nodes
object rbe aft tail rbes
   group2 tank aft bulkhead corners
   group1 tail aft spar nodes
list mesh
```



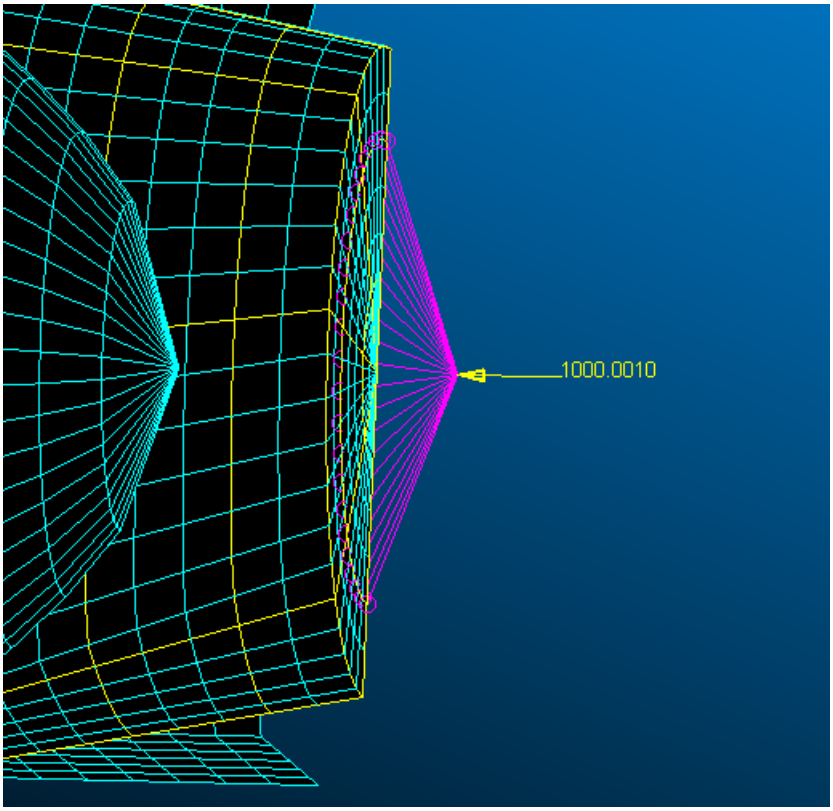
Next, if we are building a half model, create symmetry boundary conditions along the center plane. If we are building a full model, skip this step.

```
define bcset 0
if $fullvehicle = 1
goto 200
# centerline symmetry BCs - created for half model
define bcset 999
region
   iadd zge -0.01
   irem zge 0.01
   mark nodes centerline
object bc symmetry bc
   group1 centerline
   setid $bcset
   dof 345
linelabel 200
```



A RBE spider is then created with a 1000 pound thrust load applied to the thrust structure.

```
# force on engine thrust structure spider version
define loadset 1000
region
   ppadd thrust rings
   irem sphere $thrust_end 0. 0. 2. # del center alignment node
   mark nodes thrust ring
object node thrust spider
   x $thrust_end + 20.
   y 0.0
   z 0.0
   mark node fuselagenode
object rbe thrust
   group1 thrust spider ALL NODES
   group2 thrust ring
   approach spider
   dof 1
   mark rbe fuselagerbe
object force total thrust
   group1 thrust spider ALL NODES
   value1 -1000.0
   setid $loadset
   mark rbe fuselagerbe
```



Next, the fully stitched models are written out.

```
# write final stitched models
nastran sol 101
nastran subcase 500
nastran thick 1.0
nastran spc $bcset
nastran load $loadset
list mesh
write vrml tsto2025stitched.wrl
write nastran tsto2025stitched.bdf
```

The final section of the input file creates a surrogate model of just the fuselage including all of the bulkheads and the thrust structure. The tanks are removed and replaced with point masses that are connected to their support bulkheads with RBE spider connections, and a point mass representing the payload is connected to the payload bay support bulkheads. The process is similar to the wing stitching process used earlier. Here, the innermost nodes on each bulkhead are grouped for the forward tank, payload, and aft tank. Then new nodes are created at the center point of the three zones, a point mass is attached to each node, and RBEs are created to connect each point mass to its supports.

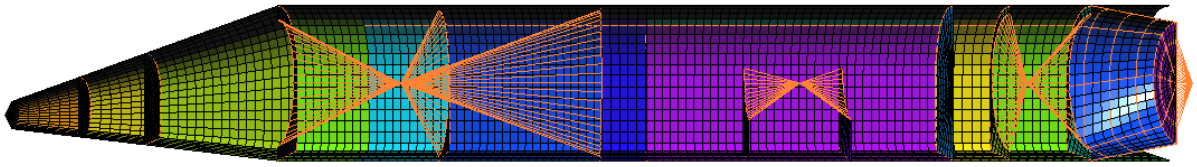
```
# Group all of the tank and payload bulkhead edge support nodes
linelabel 300
null
region
   mkadd fwd fwd ring frame ALL NODES
   mkadd fwd mid ring frame ALL NODES
   mkadd fwd aft ring frame ALL NODES
# remove the nodes on the fuselage side of the bulkhead
   ikeep xcyl 0. 0. 0. $tankscale + 1.
# remove beam alignment nodes
   irem xcyl 0. 0. 0. $tankscale / 2.
   mark node fwd tank support nodes
region
   mkadd aft fwd ring frame ALL NODES
   mkadd aft aft ring frame ALL NODES
   ikeep xcyl 0. 0. 0. $tankscale + 1.
   irem xcyl 0. 0. 0. $tankscale / 2.
   mark node aft tank support nodes
region
   mkadd fwd plb support frame ALL NODES
   mkadd aft plb support frame ALL NODES
   ikeep xcyl 0. 0. 0. $plb_scale + 1.
   irem xcyl 0. 0. 0. $plb_scale / 2.
   mark node plb support nodes
# create point masses at centers of each tank and payload bay
```

```
# and spider them to the corresponding support nodes
define fwdtankmass 1.0
define afttankmass 1.0
define payloadmass 1.0
object node fwd_fuel
   x $fwd_tank / 2. + $fwd_tank_start
   y 0.0
   z 0.0
   mark node fuselagemass
object node aft_fuel
   x $aft_tank / 2.  + $aft_tank_start
   y 0.0
   z 0.0
   mark node fuselagemass
object node payload
   x $plb_half + $plb_start + $noseend
   y 0.0
   z 0.0
   mark node fuselagemass
object mass fwd_fuel_mass
   value $fwdtankmass
   group1 fwd_fuel all nodes
   mark rbe fuselagerbe
object mass aft_fuel_mass
   value $afttankmass
   group1 aft_fuel all nodes
   mark rbe fuselagerbe
object mass payload_mass
   value $payloadmass
   group1 payload all nodes
   mark rbe fuselagerbe
object rbe fwd_fuel_rbes
   group1 fwd_fuel all nodes
   group2 fwd tank support nodes
   approach spider
   mark rbe fuselagerbe
object rbe aft_fuel_rbes
   group1 aft_fuel all nodes
   group2 aft tank support nodes
   approach spider
   mark rbe fuselagerbe
object rbe payload_rbes
   group1 payload all nodes
   group2 plb support nodes
   approach spider
```

```
   mark rbe fuselagerbe
null
list mesh
list groups
list rbe
```

The region operation makes use of the labeling that was applied for the various fuselage components as well as for the new nodes, point masses, and thrust force.

```
region
   mkadd fuselage
   mkadd fuselagenode
   mkadd fuselagerbe
   mkadd fuselagemass
   rwrite vrml tsto2025-fuselage.wrl
   rwrite nastran tsto2025-fuselage.bdf
end
```



Now that the main, partial, and surrogate models have all been generated and saved to various output files, spend a little time looking at some of those output models in a VRML viewer or in a NASTRAN pre-processor. You can also experiment with changing some of the vehicle dimensions or model settings (such as the `$fullvehicle` or `$meshdens` variables) and looking at the changes in the output models.