NASA/TM–20230001772

Loft: An Automated Mesh Generator for Stiffened-Shell Aerospace Vehicles

Lloyd B. Eldred
Langley Research Center, Hampton, Virginia

July 2023

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–20230001772

Loft: An 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

July 2023

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

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 mesh generation code that is designed for aerospace vehicle structures. From user input, Loft generates meshes for wings, noses, tanks, fuselage sections, thrust structures, and so on. As a mesh is generated, each element is assigned properties to mark the part of the vehicle with which it is associated. This property assignment is an extremely powerful feature that enables detailed analysis tasks, such as load application and structural sizing.

This memorandum is presented in two parts. The first part is an overview of the code and its applications. The modeling approach that was used to create the finite element meshes is described. Several applications of the code are demonstrated, including a Next Generation Launch Technology (NGLT) wing-sizing study, a lunar lander stage study, a launch vehicle shroud shape study, and a two-stage-to-orbit (TSTO) orbiter. Part two of the memorandum is the program user manual. The manual includes in-depth tutorials and a complete command reference.

Introduction

The ability to rapidly create, modify, and update a structural finite element model is a substantial asset in conceptual analysis. A wide variety of shapes, concepts, and layouts may be considered during the early trade study phases of a project. The large commercial finite element model creation programs are not well suited for this kind of operation. Such commercial codes can be used to quickly create a mesh of questionable quality for analysis using the code's automeshing capabilities. Or significant analyst effort can be expended to manually generate and set up a well-designed-for-analysis mesh. For the stiffened-shell class of vehicles, *Loft* can produce a well-designed mesh that is parametrically generated and suitable for conceptual trade studies for significantly less effort than required for a well-designed mesh with the commercial code. As an illustration, compare two-stage-to-orbit (TSTO) orbiter meshes in Figure 1 and Figure 2. Figure 1 was produced by *Loft*. Figure 2 was produced by using the automeshing capability of Patran on a CAD model of the outer mold line (OML). In particular, note mesh details at the wing leading edges. Further, the colors in Figure 1 illustrate the different sizing analysis regions that are automatically created using *Loft*. This partitioning of the mesh would need to be performed manually on the Patran model.

A large commercial meshing program is certainly capable of generating similar meshes to those produced by *Loft*, but at significantly more effort in positioning cutting planes, mesh seed positioning, property assignments, etc. And that commercial code can then be used to add a lot of small detail that is impossible in *Loft.* (A more efficient approach might be to add those details to the mesh that started in *Loft*). But, for rapid generation of high fidelity meshes for conceptual level design, *Loft* has clear advantages.

Figure 1. TSTO orbiter model created with *Loft*.

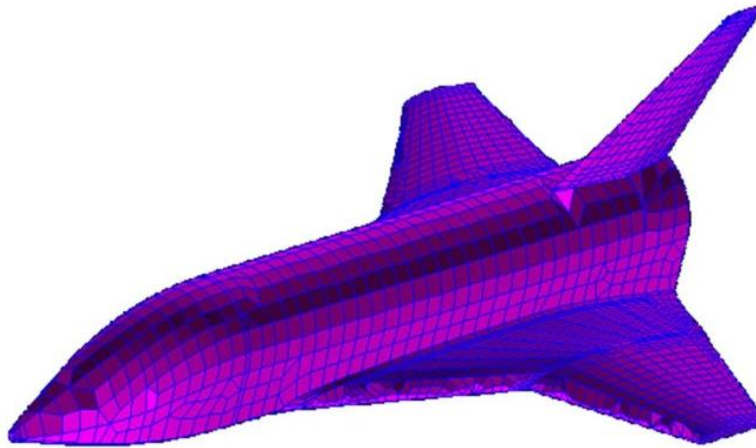


Figure 2. TSTO orbiter model created with Patran automeshing.

An initial application of the *Loft* code was to produce a TSTO upper stage model that was based on a NASA Intercenter Systems Analysis Team (ISAT) reference configuration. This model, which is illustrated in an expanded view in Figure 3, can be fully defined in a 100-line ascii-text *Loft* input file and the input file can be created in a few hours. A similar model that was created manually with a commercial code required substantial efforts on the part of three engineers over a period of one year. The commercial code based model did include significant additional detail, such as fillets; however, this level of detail is of little interest at the conceptual study stage.

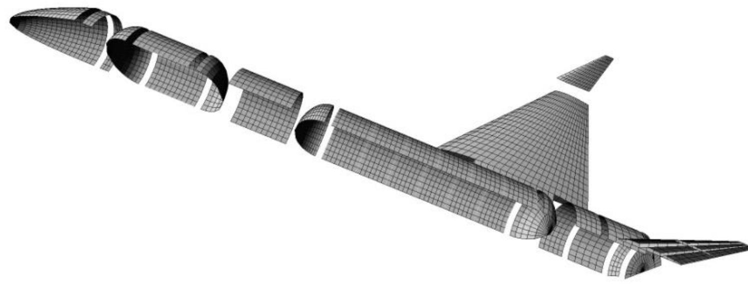


Figure 3. ISAT TSTO upper-stage created with *Loft*.

The model shown in Figure 3 includes tanks, thrust structure, wing, winglet, and tail. The wings use NACA four-digit airfoil cross sections and include ribs and spars. Ring frames are used around one tank, and longitudinal stiffeners are created along the other.

A powerful feature of *Loft* is its method for assigning properties to elements during model creation. Users specify the name of each engineering component. This name is then assigned to the corresponding elements' physical property fields. The user may optionally subdivide the component by specifying the number of material property definitions to be used across the object. These user-labeled definitions streamline the analysis and sizing process significantly. Contrast the effort that is associated with an analysis code that reports that element 58 has a negative margin of safety with that of a code that reports that "FWD LOX DOME" has the same failed result. This labeling significantly reduces the bookkeeping that is required to set up, post-process, and evaluate the results of a structural analysis.

Modeling Approach

The basic geometric entity in *Loft* is the "curve." This can be a two-dimensional (2-D) shape of any kind[*]. *Loft* contains a library of standard curve shapes, as well as three different ways in which the user can specify a nonstandard cross section. At its core, *Loft* linearly interpolates a three-dimensional (3-D) section between two arbitrary curves[†]. Commercial codes call this operation "lofting," thus the choice of program name. *Loft* can also taper a cross section down to a single point to create a dome, nose cone, or bulkhead. Figure 4 illustrates the 3-D shape that results from connecting a semicircle on the right to a half-diamond at center and then to an "M" shaped (user-defined) cross section on the left. The white lines on the figure are conformal ring frames that follow the interpolated shape along the left portion of the model.

---

[*] The term "curve" refers to a planar path requiring two coordinates (e.g. x,y) to describe. A mathematician would view such an entity as having only one dimension, length, and no thickness. Indeed, the actual lofting functionality of the program uses this one-dimensional view of the curve (see tutorial projects 3 and 5). Further, for most applications, curves within *Loft* should not be self intersecting other than possibly having coincident end points when a closed shape is desired.

[†] Similarly, the term "section" refers to a surface requiring three coordinates (e.g. x,y,z) to describe. A mathematician would consider this surface to have two dimensions, length and width, and no thickness.

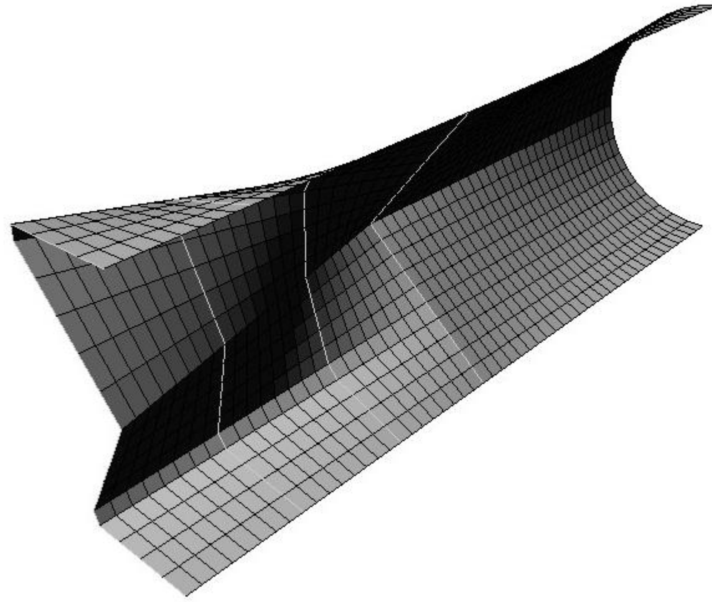


Figure 4. Lofting of three 2-D curves into a 3-D object.

Wings are created by using a similar approach. The user specifies span, chord, taper, sweep, and any desired 4- or 5-digit NACA airfoil shape for the wing root and wing tip. The code creates the corresponding trapezoidal wing section, complete with ribs, spars, and (as desired) carry through. Partial wings may be created to model ailerons. Figure 5 illustrates a wing and an expanded view of the same wing. The figure was created in *Loft* exactly as shown by requesting and offsetting different portions of the full wing mesh. Four ribs and two spars are shown.

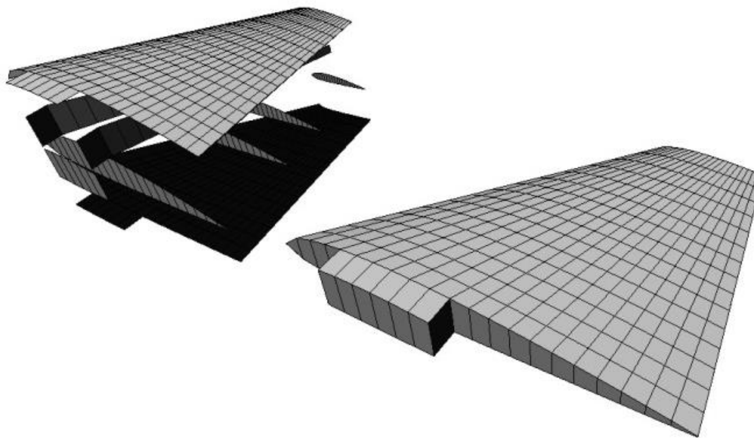


Figure 5. Expanded and normal view of a *Loft*-created wing.

User Interface

Loft uses an ascii-text input file as its user interface. *Loft* outputs a variety of standard mesh data files including NASTRAN bulk data [1], I-DEAS universal [2], ABAQUS input [3], Tecplot[4], Virtual Reality Modeling Language (VRML) 2.0 [5], and Stereo Lithography (STL) files for 3-D printing. All of the

figures in this memorandum were created by using a third-party VRML viewing program. *Loft* is written in portable C and has been compiled and used on a variety of computing platforms.

The *Loft* user creates a text input file with the text editor of their choice (e.g., notepad, vi, or emacs). Each engineering component, such as a nose, dome, barrel, intertank, and so on, is called an “object” in *Loft*. The user defines the first object by selecting an initial cross-sectional shape (curve) and its 2-D scaling. The user then specifies a second shape for the other end of the object, as well as the length, and the desired number of nodes in the circumferential and axial directions.

Each of these options is called a “parameter” in *Loft*. All parameters have a default value. Thus, the user need only supply values if the default value is not the desired value. When the user begins work on a second object, the default sizing and shape are set to those of the previous object to smoothly connect the two components. The default new object position is immediately aft of the previous object. Thus, if a user is creating an aircraft fuselage with a constant cross-sectional shape and dimension, those values only need be specified once; the input values then become the default values for all later objects. This treatment of default settings encourages the user to start at one end of the vehicle and move sequentially to the other end. Furthermore, it substantially simplifies the user’s task of defining a model and enables the 18-component, 4500-element model that is shown in Figure 3 to be completely defined in a 100-line input file.

In addition, this continuous updating of default values makes *Loft* a parametric modeling tool. The user can change the dimensions of the fuselage in one location and those changes propagate through the rest of the model. If the user changes the length of an object, later objects shift appropriately and retain their relative positions.

Mesh Manipulation

Loft also contains a powerful collection of mesh manipulation capabilities. These include translation, rotation, warping, inversion of element normal vectors, rotation of element material alignment vectors, and cloning. Figure 6 shows a shuttle-like stack that was created from the TSTO upper-stage half-model that is shown in Figure 3. That model was cloned and reversed, and the normal vectors of the mirror half were flipped. A single booster model was created and similarly cloned to form a second booster. A single

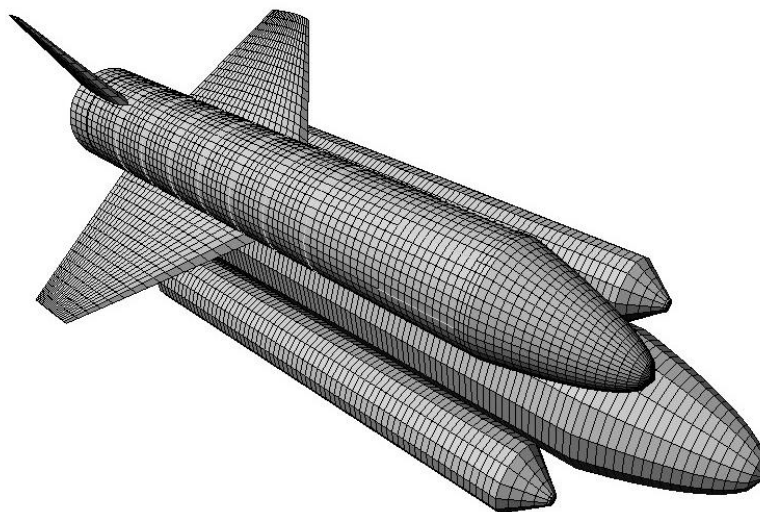


Figure 6: A shuttle-like configuration created with *Loft*'s cloning tools.

external tank model was then created. Finally, each vehicle component was appropriately positioned.

Loft can manipulate a mesh at a much finer level. Elements can be specified by object name, by property ID, by the arbitrary user “marks” that can be assigned during object creation, or by a specified volume. These selected elements can be queried, modified, or deleted. This capability allows damage to be modeled, partial models to be saved (e.g., only those elements labeled as part of the outer mold line (OML)), and so on. Figure 7 shows shroud doors that were created by changing the properties within a specified rectangular region of a mesh. The door frames were created with the same process. The ability to save partial models based on this mesh labeling is discussed in more detail with the TSTO orbiter example later in the document.

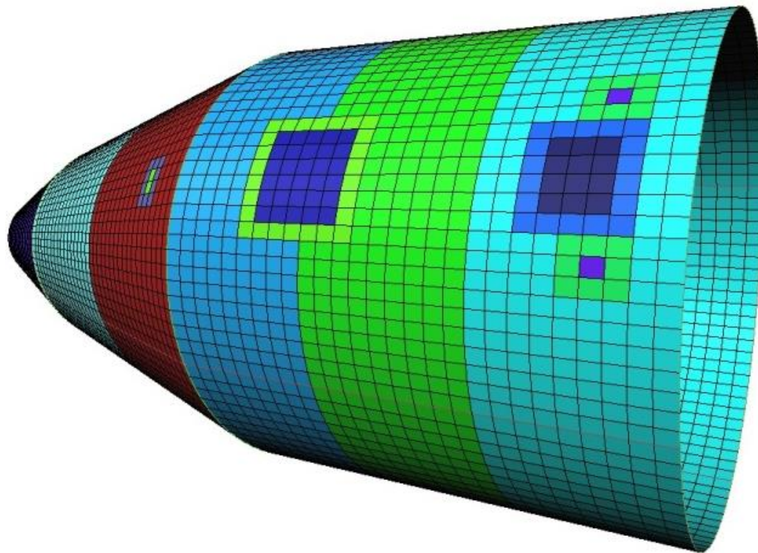


Figure 7. Shroud doors created by changing properties in a rectangular volume.

Limitations

Loft is intended as a tool for the conceptual design stage. Thus, some important limitations should be kept in mind and taken into account when the time comes to convert to a more time-consuming and more general mesh-creation tool.

Loft’s beam creation options are limited. Stiffening beams such as ring frames, longerons, and rib and spar caps are easily created, but free-standing beams or trusses must be created manually, by specifying coordinates of each end, or with another tool and merged with a panel mesh created in *Loft*. This merging can be accomplished either in that tool or the data can be read into *Loft* for merging. The support struts connecting the spherical tanks on the lander concept in Figure 8 were created manually but made substantial use of *Loft*’s math and variable support to automatically relocate appropriately when other vehicle dimensions changed.

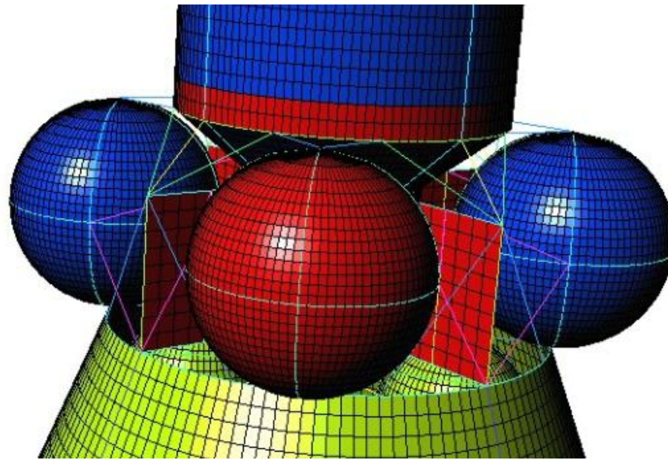


Figure 8. Tank struts created in *Loft*.

Another limitation has implications even at the conceptual level. While *Loft* does merge finite element nodes that are coincident, it does not attempt to merge or stitch dissimilarly meshed objects. A long fuselage model will stitch correctly as long as the circumferential node counts do not change. However, the wing, tail, and winglet of the booster in Figure 3 require manual stitching to the adjacent components before any analysis can be performed. This process can be simplified by positioning of ring frames at the desired attachment stations, but the final connection must be made manually. Stitching is discussed further in the lunar lander stage and the TSTO orbiter examples in the applications portion of this document.

Applications

Loft has been applied to a wide variety of aerospace analyses. Several of these applications will be discussed to demonstrate the code's capabilities.

NGLT Wing Sizing

Loft was used to determine the optimum rib and spar count for a Next Generation Launch Technology (NGLT) vehicle wing. A simple Visual Basic front-end tool was created that allowed the user to vary the basic wing geometry settings. Then, the user could push a button to: (1) call *Loft* to generate a mesh for

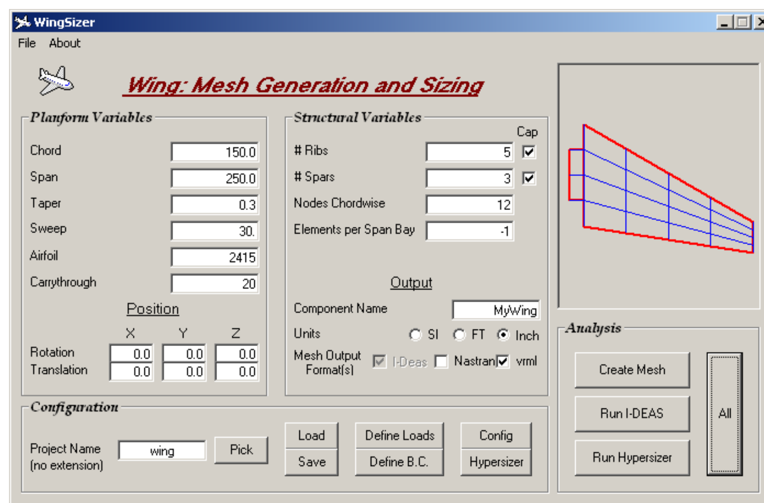


Figure 9. Visual Basic front end for wing sizing tool.

the specified wing, (2) call the finite element code I-DEAS to apply a specified pressure load and solve the finite element analysis (FEA) system, and (3) call HyperSizer [6] to compute the required weight of the wing, report back the weight, and report if any negative margins of safety were computed. Figure 9 shows the Visual Basic interface for the wing sizing tool.

This approach allowed a broad survey of the design space to be completed, including a variety of structural materials, in just a few days. For this particular work, the wing planform was fixed and the rib and spar counts were varied to determine the lowest weight configuration. Figure 10 illustrates a portion of the computed wing weight results.

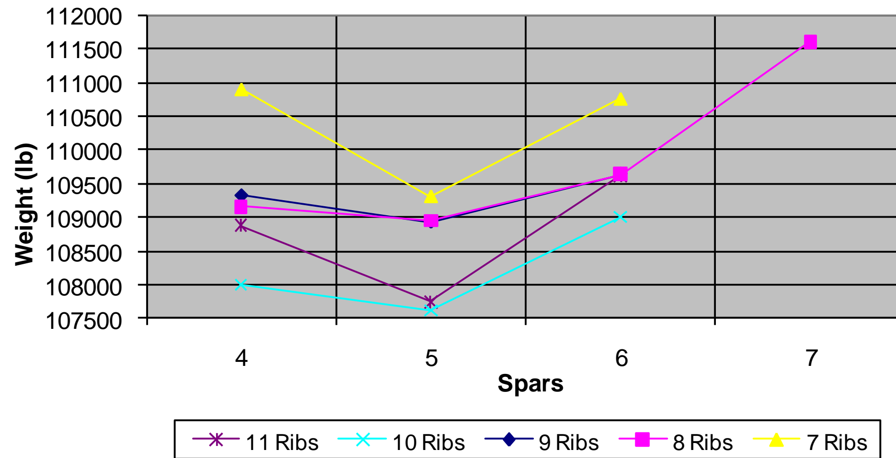


Figure 10. Variation of wing weight with rib and spar count.

Lunar Lander Stage

In the preliminary stages of NASA's Constellation program, a variety of lunar lander concepts were studied. The "DASH Lander" design consists of three stages: an ascent stage, a decent stage, and a retro stage. The retro stage is responsible for the lunar orbit insertion (LOI) burn and for a substantial portion of the lander's decent to the surface before being discarded to crash downrange of the actual landing site. Both the ascent and decent stages have substantial structural truss components and are not well suited to being modeled in *Loft*. However, the concept for the retro stage is similar to that of the Apollo service module shown in Figure 11 [7].

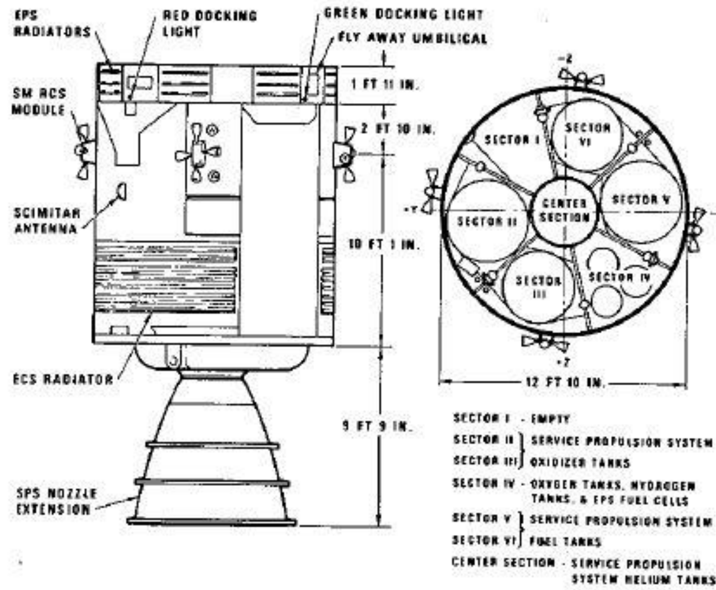


Figure 11. Apollo service module.[7]

Both the CAD and finite element models of the full lander stack are illustrated in Figure 12. On the right of the illustration, the external skin of the retro module has been removed from the sides and top, to show the internal detail. *Loft* was used to create the tanks, the external skin including the lander adaptor at the top, the cross module bulkheads, and all of the stiffening and attachment beams that lie along the skin and tanks. A few additional beams were manually added to actually connect the prepositioned load-bearing frames on the skin and bulkheads to those on the tanks.

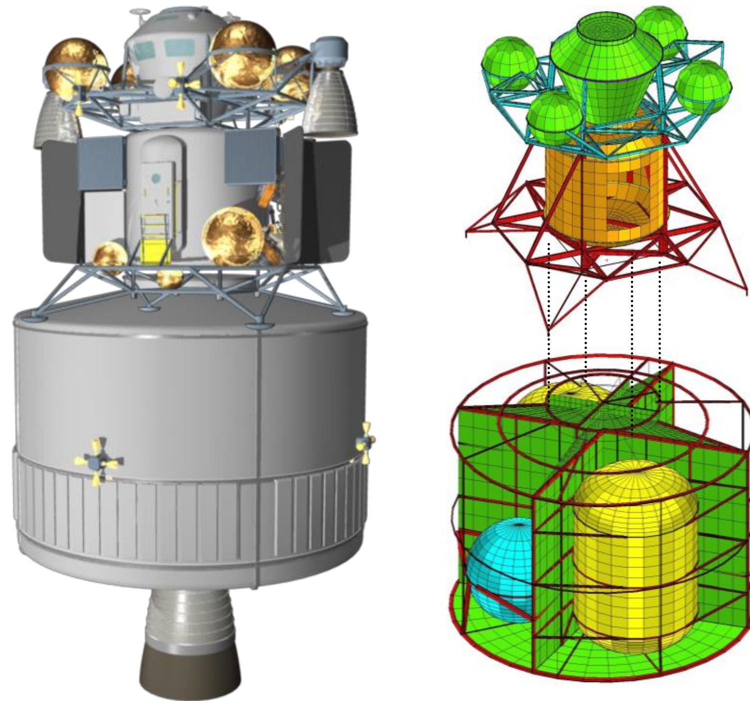


Figure 12. DASH Lander CAD and FEA models with FEA outer skin removed.

Following construction of the three component models (i.e., the ascent, decent, and retro modules), design loads were applied in NASTRAN, and the components were sized in HyperSizer. The beams on the right side of Figure 12 are shown at the actual sizes that were computed by the structural sizing analysis.

Ares V Shroud

Loft was used to create all of the finite element models that were used by the Ares V Shroud pre-phase A design team. Over the life of the project, this constituted approximately 20 distinct models. Of particular interest here are the 12 models that were developed in support of a shape optimization study for the shroud. These shapes are illustrated in Figure 13 and show conic, biconic, hemisphere, ogive, power-law, and blunted Haack shapes.

Each of the shroud concepts was modeled in *Loft*, and then analyzed and sized. Other team members performed aerodynamic, thermal-protection, and trajectory analyses to determine the changes in the delivered payload mass for each concept.

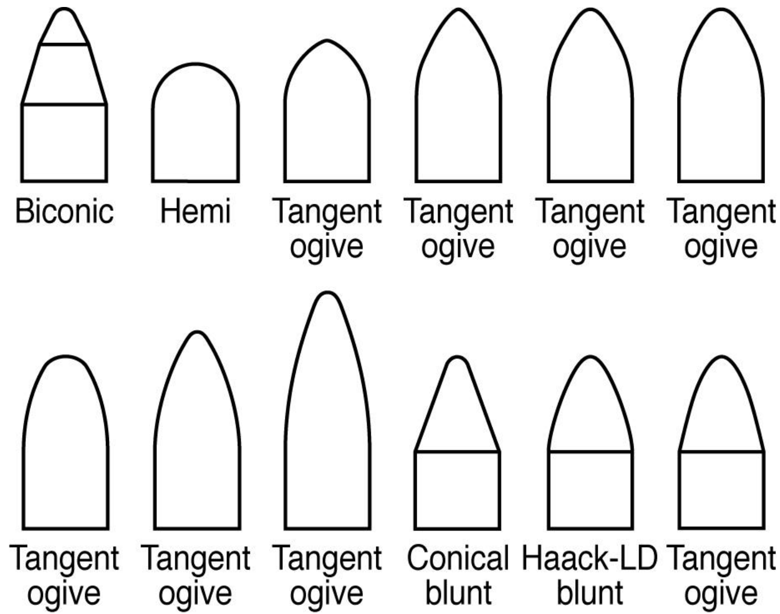


Figure 13. Ares V shroud shapes considered.

One of the biconic-shape analysis models is shown in Figure 14. The model includes separation joints, large access doors, and small fuel and purge doors. The color changes indicate the different sizing design regions of the shroud. These regions were defined completely within *Loft*. Prior to the analysis, boundary conditions were applied to the base of the structure, aerodynamic loads were mapped onto the finite element mesh, and the combined and scaled load cases were defined in the finite element analysis deck.

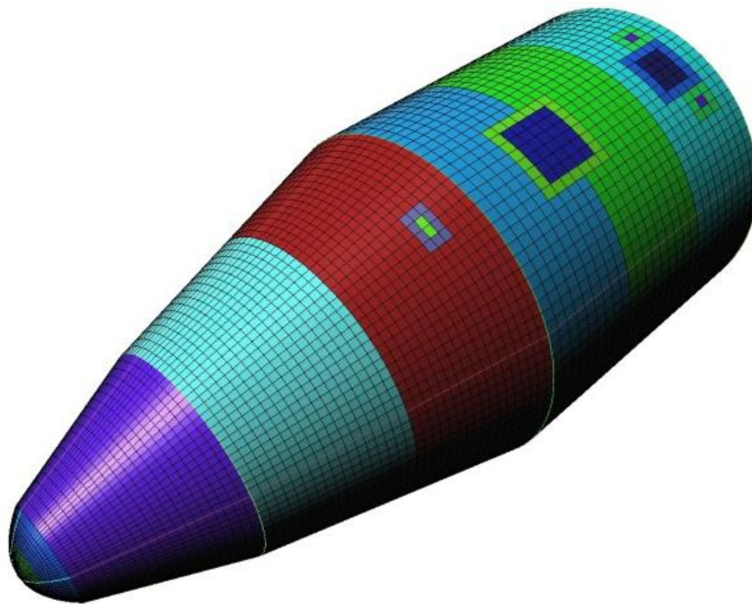


Figure 14. Bi-conic shroud model created entirely in *Loft*.

The *Loft* input file to create the four petal, bi-conic shroud in Figure 14 is 134 lines of ascii text. This count includes substantial comments for clarity. The following listings show the first 16 lines of the *Loft*

input file for this model. They are provided to illustrate the process that is used to define a model. More comprehensive and in-depth tutorials are provided in part two of this memorandum.

The first line of the partial input file is a comment. It explains that the next 4-line block of input defines a new curve named “qc” (for quarter circle). The first line of the block defines the type of user-defined curve (compound) and specifies the “qc” name. The second line identifies the built-in “circle” curve as the basis of the new shape. The last two lines of the block defines the parameters “sstart” and “sstop” which specify that the new curve is defined as the section of the “circle” curve from one-eighth to three-eighths of its circumference.

```
# define "qc" curve as quarter circle
curve compound qc
     child circle
     sstart 0.125
     sstop 0.375
```

The next block of the input file then uses this “qc” curve to construct the dark blue spherical cap by creating a dome object named “Nose Cap.” The next three parameter lines specify dimensions for the object in the x, y, and z (length) directions. The “taper” parameter specifies a parabolic curvature and “zdist” controls the spacing of nodes along the length of the dome. The last four parameters define the node and component (structural sizing region) counts in the axial and circumferential directions.

```
object dome Nose Cap
     curve1 qc
     c1_xscale 50.688
     c1_yscale 50.688
     length -29.266
     taper para
     zdist 0.6
     nodes_circ 27
     nodes_axial 16
     components_circ 1
     components_axial 2
```

The remainder of the input file (not shown here) defines the rest of the quarter circumference petal, creates three clone petals (for a total of four), marks the doors, and saves the completed model.

TSTO Orbiter

As part of a two-stage-to-orbit (TSTO) design study, a finite element model of the orbiter stage was constructed by using *Loft*. Because the fuselage cross section is not a shape that is contained in *Loft*’s curve library, a user-defined compound curve was specified. This compound curve combined a circular top, an angled flat side, a round bottom corner, and a flat bottom as shown in Figure 15. Figure 16 shows the finite element half-model of the vehicle. The last 34 pages of part 2 of this memorandum discuss the full orbiter input file in fine detail.

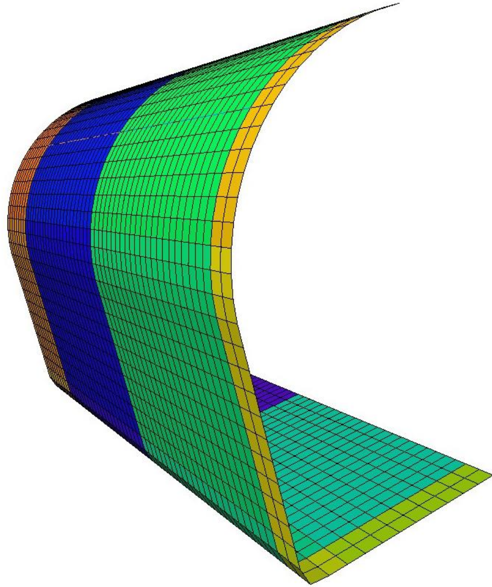


Figure 15. User-defined compound curve used for fuselage cross section.

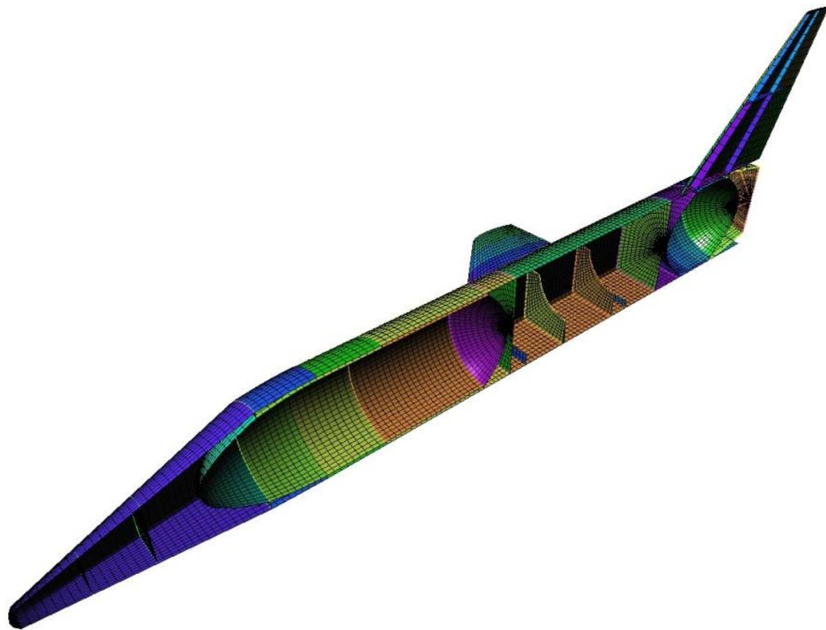


Figure 16. TSTO orbiter FEA model.

Figure 17 shows an expanded view of the model to illustrate wing and tank detail. After the manual stitching was accomplished, simple loads and boundary conditions were applied to the model. A finite element solution was performed to check for any mechanism behavior that would indicate insufficient stitching.

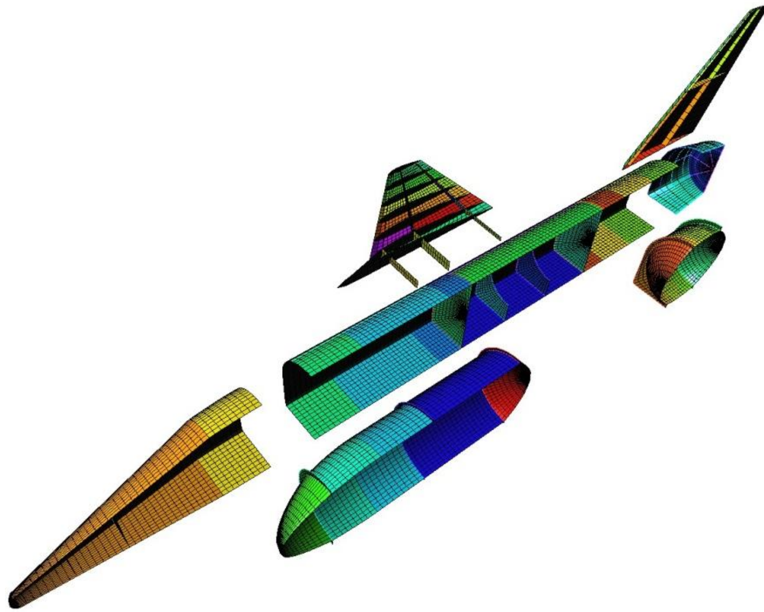


Figure 17. Expanded view of TSTO orbiter FEA model.

The input file for the orbiter contains commands to mark the components that are on the vehicle outer mold line with the label "OML." Similar marks are applied to the two tanks. These labels can be used to output a partial model, with all of the node and element indices intact. These partial models make the mapping of external aerodynamic loads or internal pressure loads to the appropriate portions of the vehicle easier and faster. The mapped data sets can then be applied directly to the full model. Figure 18 shows OML-only and tank-only models that were created from the full vehicle input file. Note that the OML model contains only the skins of the wings.

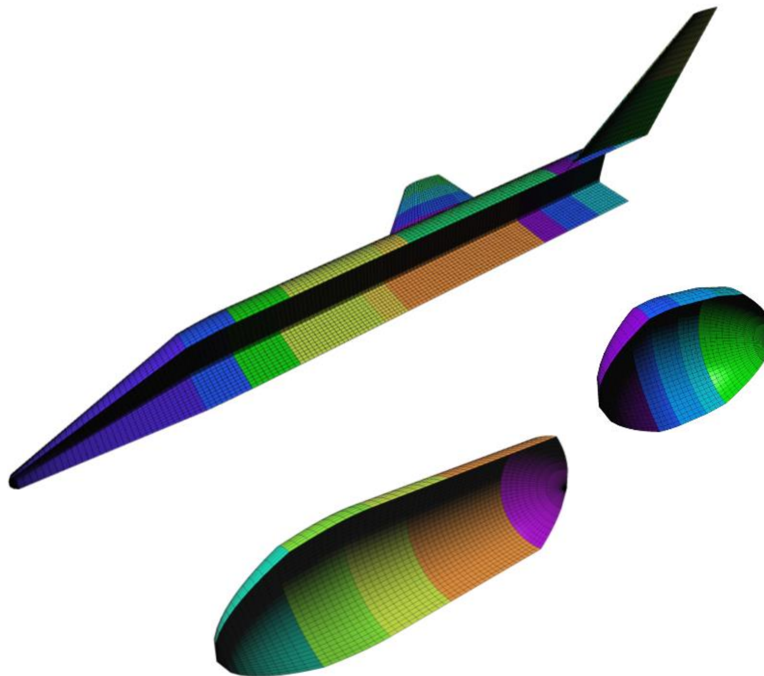


Figure 18. Three partial vehicle models created by labeling the full model.

Cerro et al. [8] describe the use of *Loft* as part of a complete conceptual vehicle sizing process.

Eldred et al. [9] describe the incorporation of *Loft* into a design of experiments driven multidisciplinary system to perform conceptual design of supersonic aircraft with complex wing and fuselage shapes as illustrated in Figure 19. The wings studied include a potentially large number of spanwise variations of chord lengths, airfoil shape, twist, and sweep angles. The fuselage configurations allowed for arbitrary

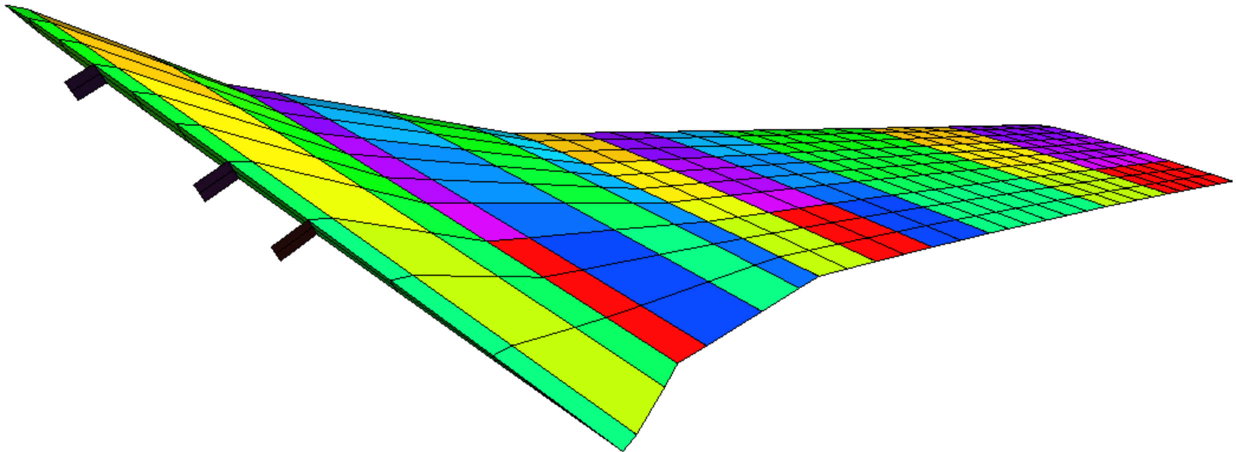


Figure 19. Complex supersonic wing model.

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.

User Manual

An extensive manual for users has been created for the *Loft* program and is included as part two of this document.

Chapter 1 of the user manual describes the basic terminology and the user interface for the program. Chapter 2 contains a variety of tutorials, beginning with a very basic commercial aircraft model and progressing to more advanced subjects, such as user-defined curves and the region mode. Chapter 3 describes the region mode in significant detail. A programmer's reference is included in Chapter 4. Chapter 5 describes two small, related, utility programs. Chapter 6 is a quick reference for all of the commands, parameters, curves in the library, and taper types that are used for domes and noses. Finally, two complete input files are provided with discussion and illustrations for each section of the files.

Summary

Loft is a very powerful automated mesh generator that is designed to allow the rapid production of detailed conceptual finite element models that are suitable for analysis and sizing. Its focus on stiffened-shell aerospace vehicles allows it to produce cleaner meshes than auto-meshing models from commercial codes. Suitable models for analysis can be produced much more quickly with *Loft* than with a commercial code, since the latter requires creation of the geometry and then manual definition of the mesh. The inher-

ent parametric nature of *Loft* makes it ideal for rapidly updating models for trade studies or for design refinement.

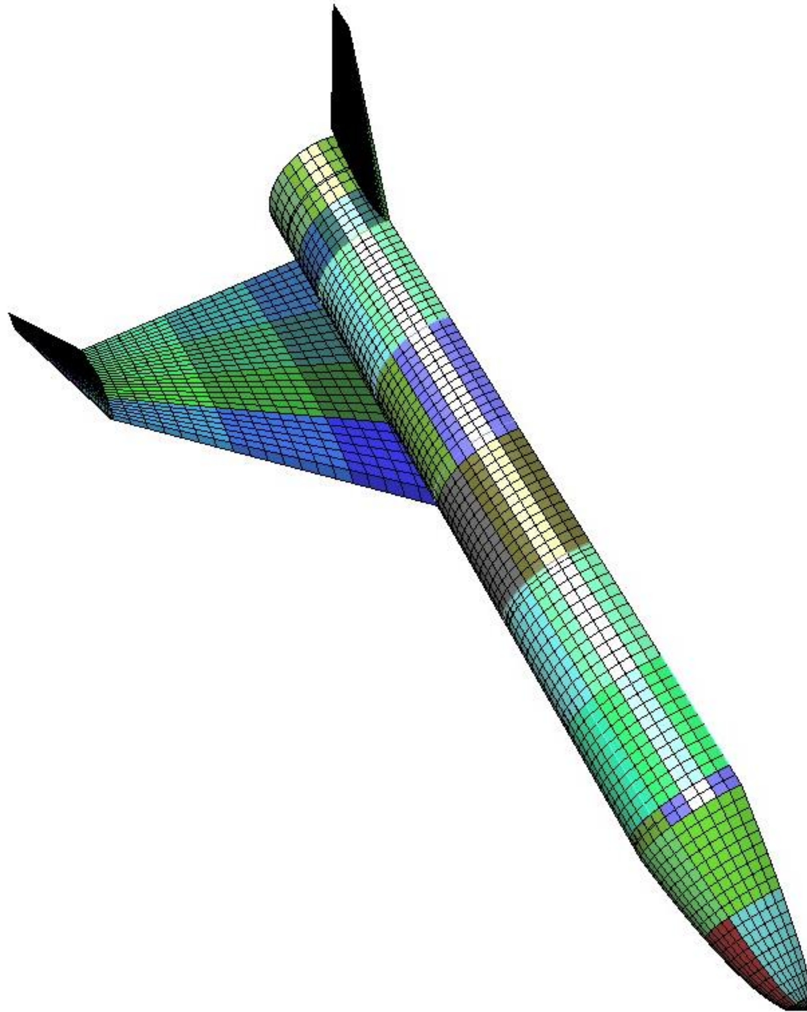
References

1. MSC Nastran, https://hexagon.com/products/product-groups/computer-aided-engineering-software/msc-nastran, accessed March 1, 2023.
2. Siemens I-DEAS NX, https://www.plm.automation.siemens.com/global/en/resource/i-deas-software-to-nx/96560, accessed March 1, 2023.
3. Simulia Abaqus FEA, https://www.3ds.com/products-services/simulia/products/abaqus/, accessed March 1, 2023.
4. Tecplot Data File Types, https://www.tecplot.com/2016/09/16/tecplot-data-file-types-dat-plt-szplt/, accessed March 1, 2023.
5. The Virtual Reality Modeling Language Specification, Version 2.0 ISO/IEC WD 13772, http://graphcomp.com/info/specs/sgi/vrml/spec/, accessed March 1, 2023.
6. Collier Research Corporation HyperSizer/HyperX, https://collieraerospace.com/, accessed March 1, 2023.
7. “NASA Apollo Command Module New Reference,” North American Aviation, 1968, p. 53
8. 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, 16-18th May, 2005
9. Eldred, Lloyd B., Padula, Sharon L. and Li, Wu; “Enabling Rapid and Robust Structural Analysis During Conceptual Design,” NASA/TM–2015-218687

Part 2

Loft: An Automated Mesh Generator For Stiffened-Shell Aerospace Vehicles

Program Manual



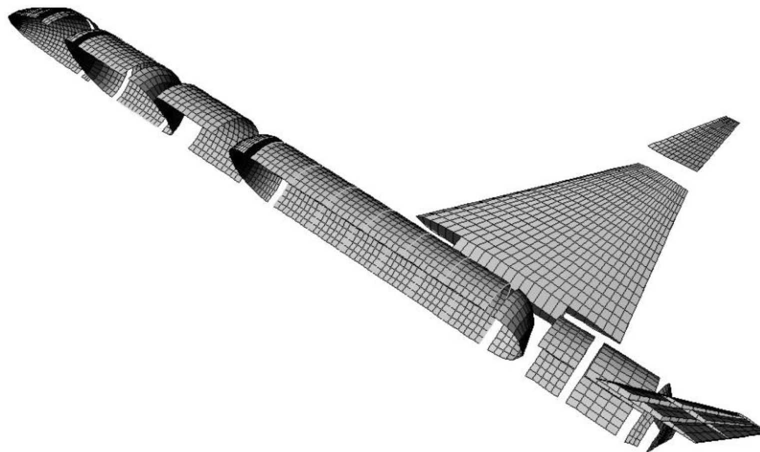
Chapter 1: Introduction

Loft is an automated 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 its meshes in NASTRAN bulk data deck, Seimens' I-DEAS Universal File format, Abaqus input file format, VRML 2.0 (Virtual Reality Modeling Language), Tecplot, and STL (STereo Lithography) files. 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 six parts. The first part is an introduction and overview of the program and how it works. The second section is a practical tutorial on constructing a variety of vehicles and components. The third part of the manual discusses the powerful region concept in detail. The fourth section of the manual is a technical/programmer's reference describing how the code is written and how to add to it. Part five documents various external utility programs that have been written for *Loft*. The final part is a reference guide giving details on all commands and objects.



Mesh Construction

Loft uses very basic finite elements: 4-node quadrilaterals, 3-node triangles, 2-node bars, 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 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 18 “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.

Formatting in this manual

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 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 a number of 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.

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 is 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 and solid rocket boosters, Shuttle with just the external tank, 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 a number of 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.

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). Wings are meshed with quadrilateral panel elements except at the leading edge of each rib where a triangular element is 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

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 is 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 when running. These groups mark nodes that are on curve end-points, 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."

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 meta-variables to be set (such as unit type). 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 parameters, add lines after the command with the parameter names followed by the new values. 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` command.

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.

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.

```
# 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 Windows another option is to create a text file with the desired command. It is recommended that a greater than symbol and then the name of a file is also added to capture the output from *Loft*. Your new file 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. 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. See tutorial project 7 on Variables and Math for more details.
@ - The at-sign symbol is used to recall a system variable. A list of variables is available in Chapter 6's "System Variable List" charts. See tutorial project 7 on Variables and Math for more details.
% - The percent sign is used to call a math function such as sine or squareroot. See tutorial project 7 for details and Chapter 6'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. See tutorial project 7.

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 are 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 all of the 2-D curves are defined in the X-Y plane, with Z as the extrusion direction. If, as in the example vehicle 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 parameter name and value(s) 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 parameters `transx`, `transy`, and `transy` override the default position setting and assign an absolute position to the item. The 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, the relative translation parameters are 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. All of 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.

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 result 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 exactly 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 its internal curve mnemonics, then the list of user-defined curves.

Interpolated curves are built from user-specified x,y coordinates pairs. At the moment, 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 the chapter 6, and tutorials on using all types of user-defined curves are in the tutorial project 3 found in chapter 2.

Chapter 2: Tutorials

Introduction

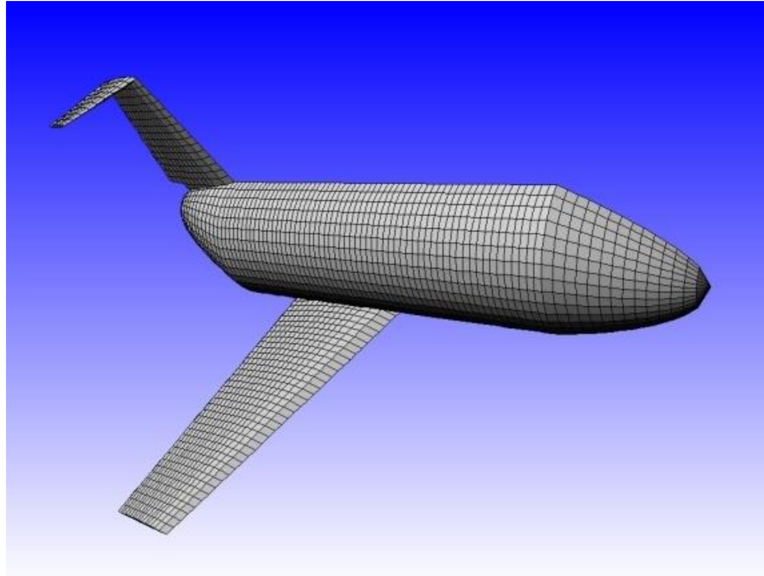
Loft is an easy-to-use program that takes very simple finite elements and builds detailed finite element meshes. A user controls *Loft* by creating a text input deck with their favorite editor such as notepad in Windows and vi or emacs in Unix/Linux.

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
Project 2: Converting Project 1 Mesh to a full vehicle
Project 3: Creating and using User-defined Curves
    Part A: Interpolated Curves
    Part B: User-defined Compound Curves
    Part C: User-defined Lofted curves
Project 4: A Tapered Four-Lobe Tank
Project 5: Controlling Circumferential Node Distribution
Project 6: Introduction to Regions
Project 7: Variables and Math
Project 8: Bodies of Revolution, Toroids, and Helixes

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 number of 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 deck 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 6 of this manual) 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 chart in Chapter 6 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.

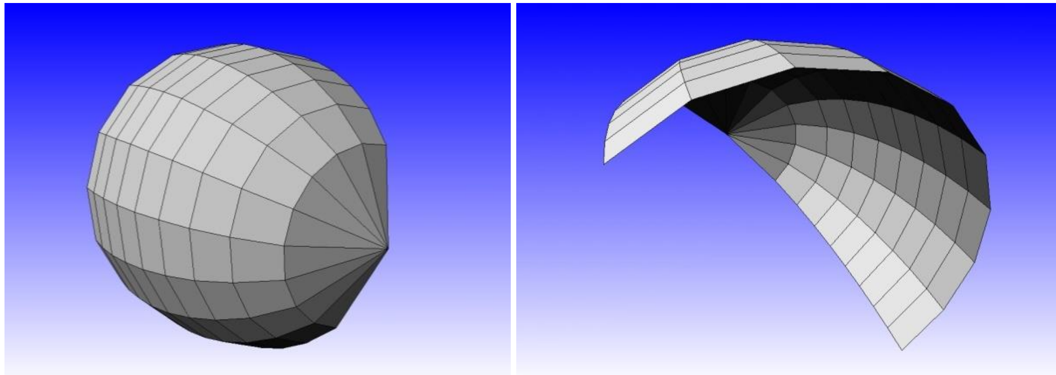
Now, let's see the result. To do that, an output command is added to the file:

```
# 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 `write` command reference in Chapter 6 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 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:



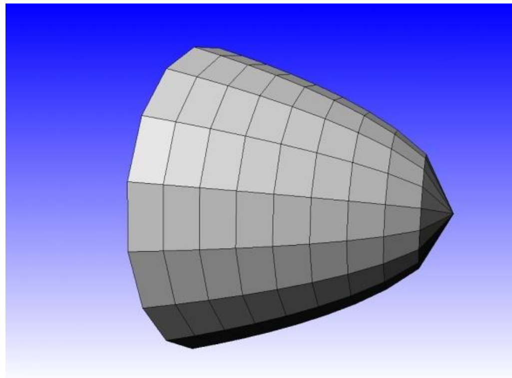
Obviously, the model could use some improvements. Open the input file in the editor again.

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 fol-

low that command directly; once another command is encountered (i.e., a `write` command) the previous command is finished.

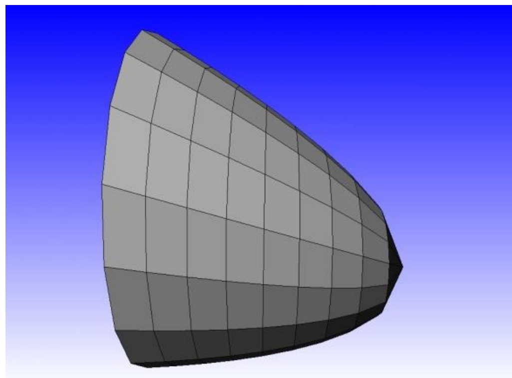
The first thing to change is the curvature of the nose. Referring to the "dome taper library" figure in Chapter 6, there are illustrations of differently shaped dome objects and the mnemonics necessary to use them. Change from the default spherical tapered dome to a parabolic tapered one.

```
taper para
```



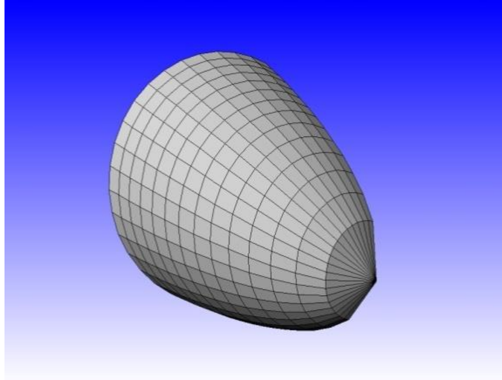
Now, drop the nose tip down a little so the pilots can see out.

```
zdroop 4.0
```



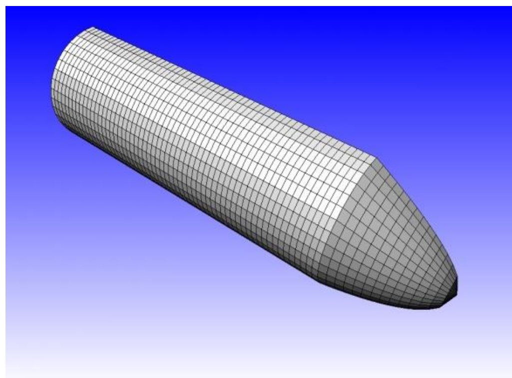
And make the mesh a little denser.

```
nodes_circ 21
nodes_axial 15
```



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

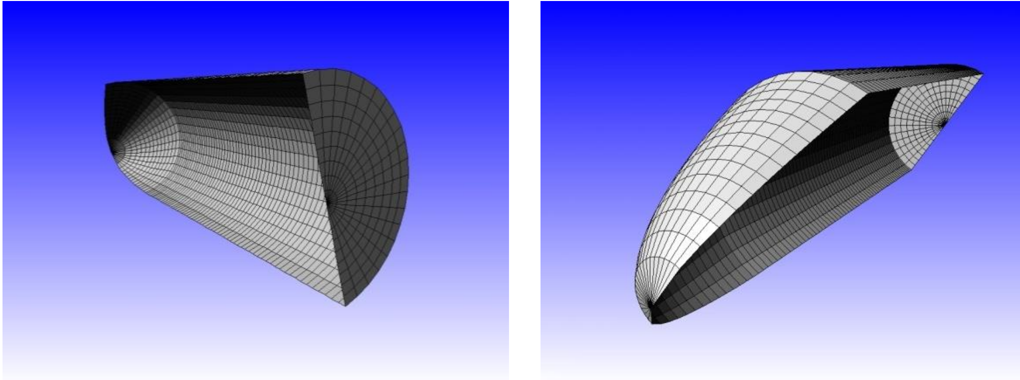
```
# Fuselage
object section Fuselage
length 50
nodes_axial 60
```



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.

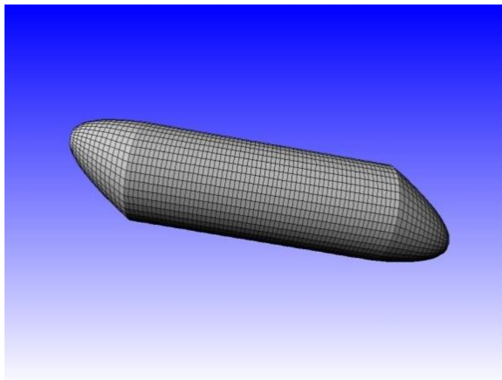
```
# Bulkhead
object dome Bulkhead
taper bulk
nodes_axial 10
```



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
```

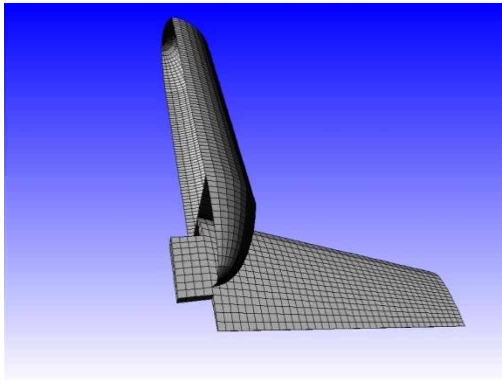


Next, move onto the wing.

```
# 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 the `wing` object reference in Chapter 6). 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 carrythrough. 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:

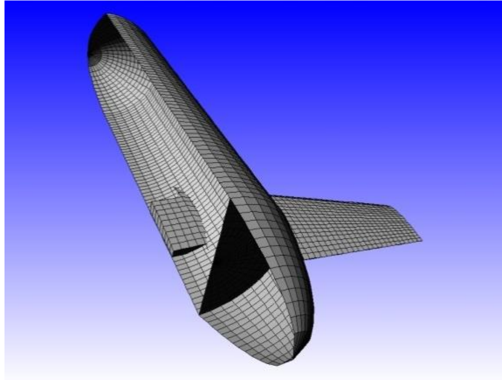


The wing shape is correct, but it's in the wrong place. Why is that? First, dome objects' lengths do not alter the default starting point of the next object. And the origin of a wing object is at its leading edge root. So, the leading edge root point of the wing is sitting at the rear center point of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

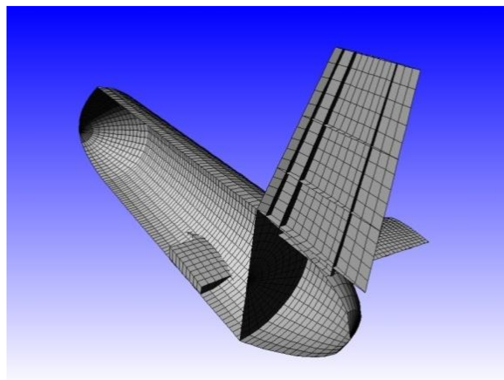
The x translation moves the carrythough to the centerline. The y translation moves the wing down to the bottom of the fuselage, and the z translation moves the wing forward.



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.

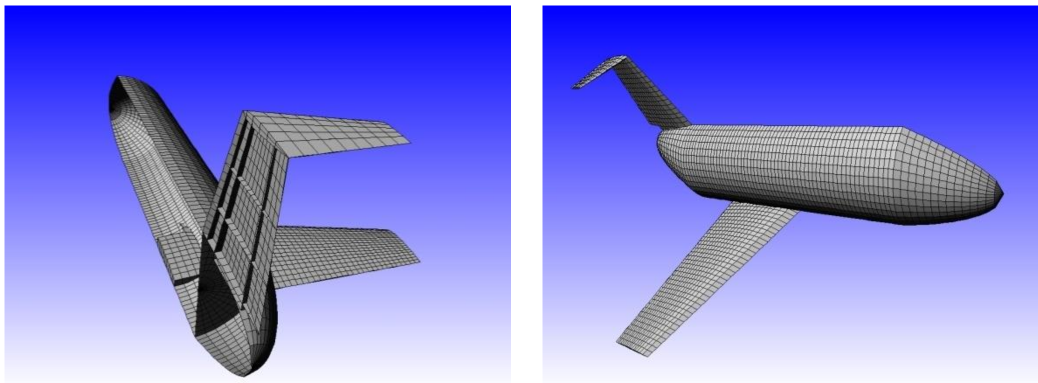


With the `halfwing` option, it's possible to see the internal spars and ribs on the tail, which are in the same position as on the main wing (since no change was specified).

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 documentation in Chapter 6 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.



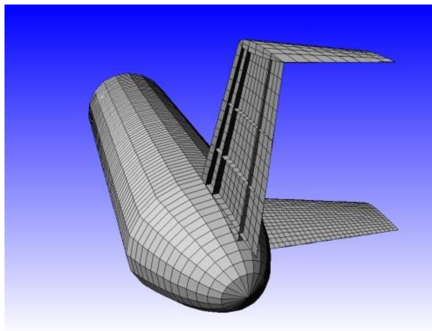
Note that the various wing objects are not actually connected (in a finite element sense) to the fuselage or each other at this stage. Before actually using this model to perform an analysis, some work should be done with the mesh density on the horizontal tail (to make it match that on the vertical 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 two different ways to accomplish this task. Both 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 first option is better if only a full model is desired. The second is 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:

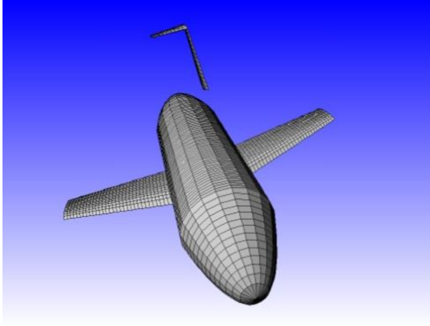


The new full circle curve1 parameter becomes the default for the rest of the fuselage objects by only changing the one line at the beginning of the file. You may also want to double the circumferential node density so that the spacing is the same as before: "`nodes_circ 41.`" Now, fix the wings.

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 6) 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
```

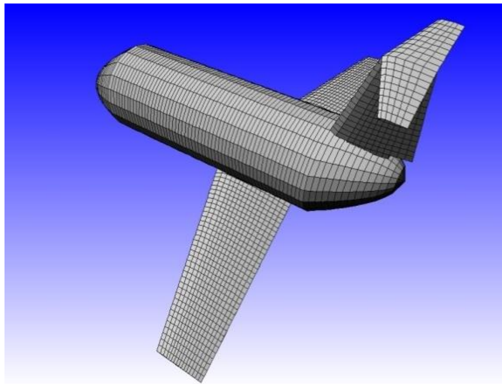
With all of the edits, the final input deck, with some (optional) added indentation for reading clarity, is:

```
# 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, manual stitching of the wing surfaces to each other and the fuselage would be necessary prior to any finite element analysis.

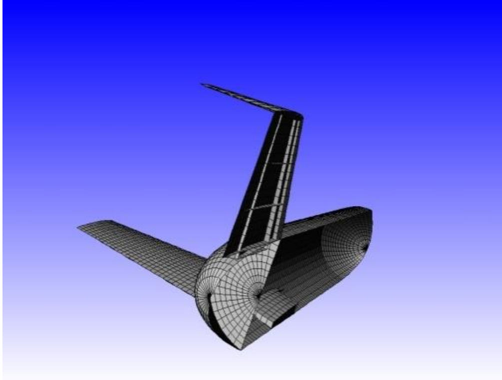


Approach 2: Clone the half model into a full model

This part of the tutorial will create a very similar mesh another way. Start by copying 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
```

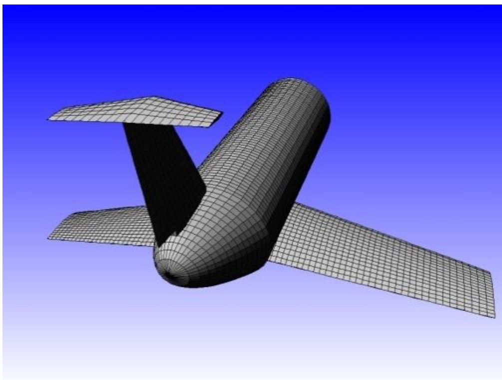
These commands start by moving the half model to the internal clipboard and naming it "SB." The store command clears and resets the active workspace. So, the next command recalls it back into active memory. The next three lines perform two stack level move operations. The `scalex -1.0` parameter changes 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:



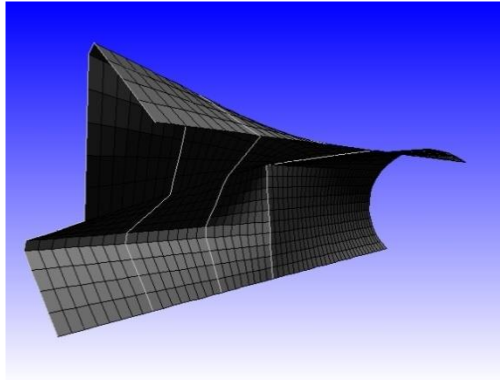
The meshes produced by these two approaches are in many ways identical. The nodes and the elements are in the same places (the cloned approach may have extra nodes and elements in the vertical tail due to being created as two halfwings). The real differences are subtle. If I-DEAS universal files were created by adding lines like

```
write unv project2a.unv
```

to each file and these universal files were imported into I-DEAS, 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.

Project 3: Creating and Using User-defined Curves

Part A: Interpolated Curves

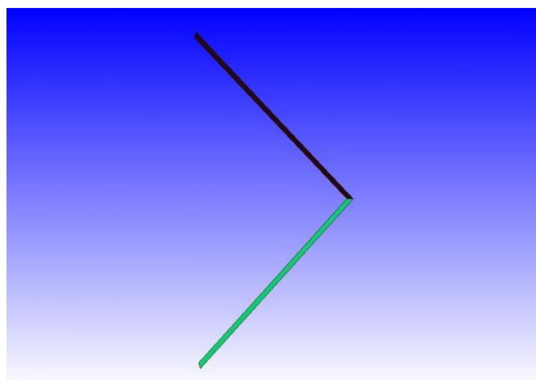


Loft's curve and curve family library covers the basic shapes used for many aerospace vehicle components. But, the library can't contain everything. This project explains how to use the interpolated curve definition capability to create user-defined shapes.

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:



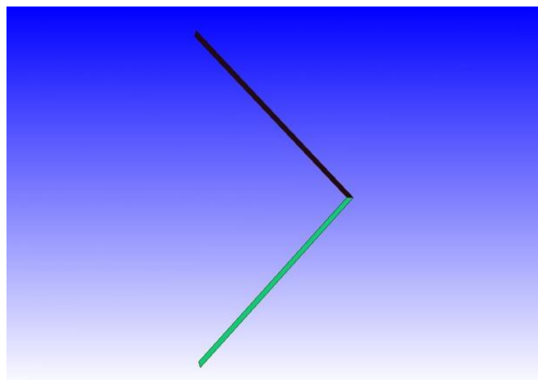
To define this shape to fit in a unit circle, start at the top. The coordinates are x=0, y=1. The midpoint of the shape is at x=1, y=0, and the bottom point is at x=0, y=-1. The command and parameters to specify these coordinates as a *Loft* interpolated curve named "sd" are:

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

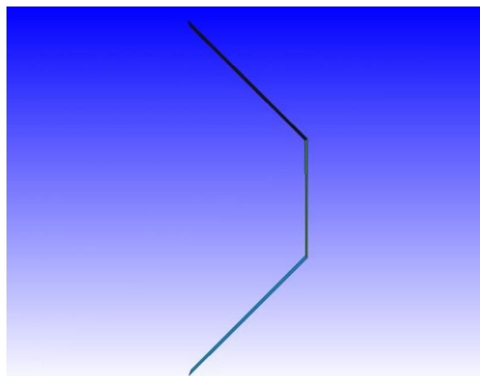
Once defined, the "sd" mnemonic can be used in any subsequent objects as if it were a curve in the library.

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:

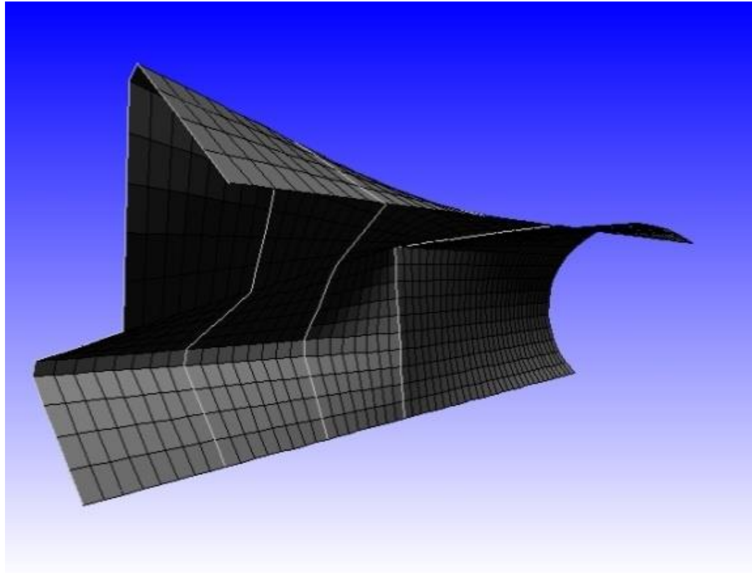


By the way, *Loft* will show this same corner-rounding behavior when using library curves and the other types of user-defined curves. The user may need to experiment with the number of nodes specified if hitting the corners exactly is important. See project 5 for some additional ways to address this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

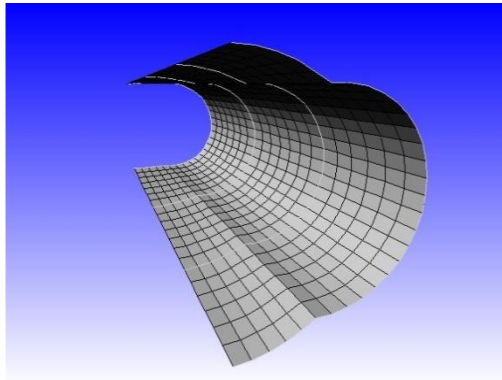
To finish this project, define a second interpolated curve (the M-shaped left side of the original figure) and then use both curves:

```
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.



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 curve, can be used.

This power comes at a price. *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 reblend 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 s value 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
```

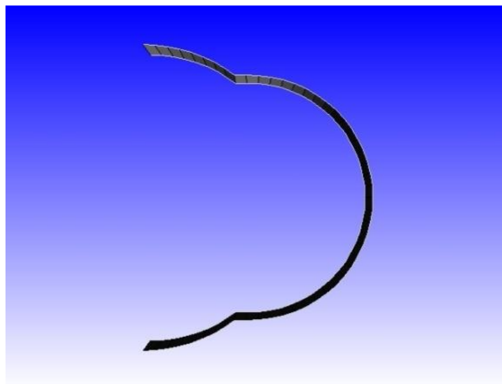
The last arc is part of the first curve, so that block is copied here:

```
child sc
x 0.0
```

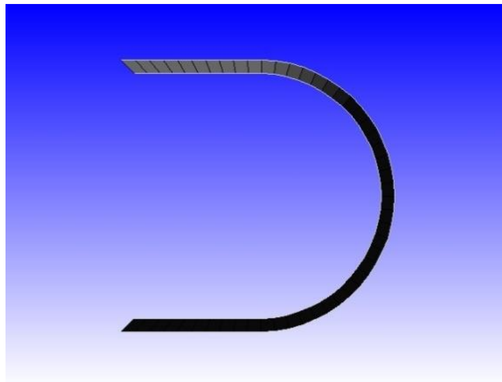
```
y 0.0
radius 5.0
```

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 for this tutorial project 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:

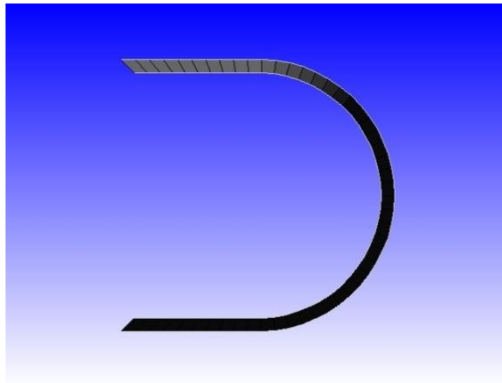
```
child cir
x 3.0
y 0.0
radius 3.0
sstop 0.5
```

(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
```

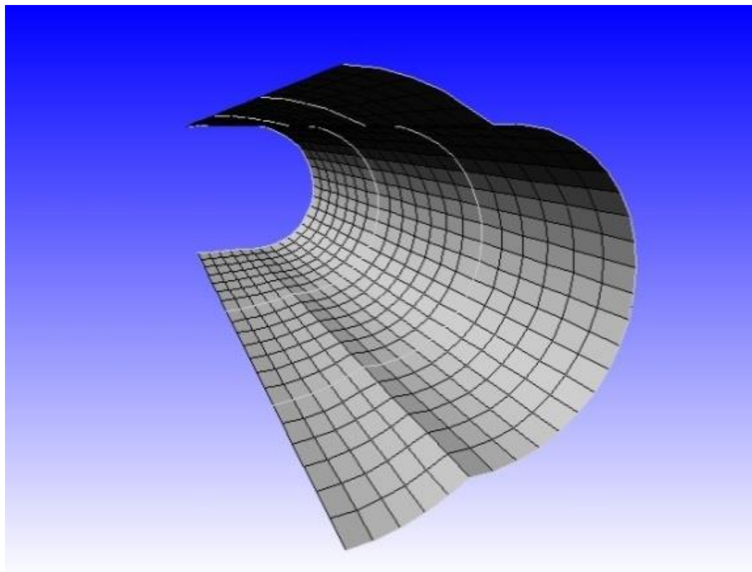
To generate a sample representation of the new compound curve just add:

```
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 Barrel
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
```

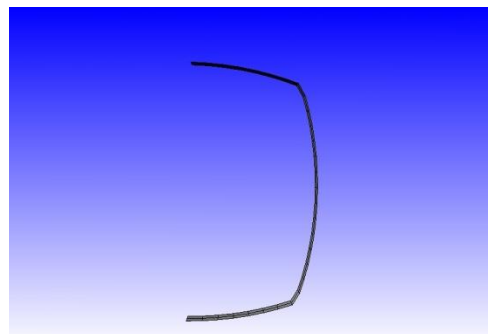


Part C: User-defined Lofted curves

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.

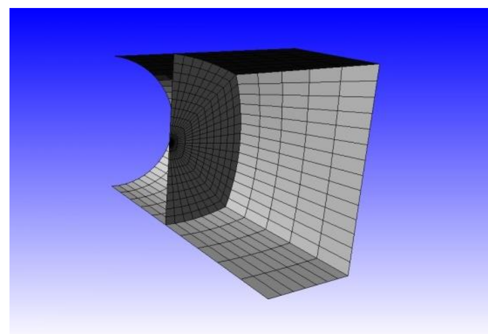
To create a lofted curve, the user specifies the curves that are to be blended to form the new cross section. As with the compound curve, any type of curve including user-defined curves can be used as the end shapes. The user then specifies the fraction along the transition from `curve1` to `curve2` 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
```

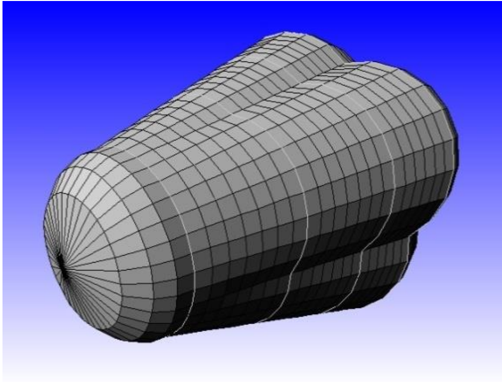
Care should be taken if node-stitching is desired to make sure that the bulkhead is positioned at a spot on the section object with nodes. In the above example, an odd number of nodes was used axially to ensure that a node line existed at the 50% axial station on the section. The lofted curve was defined 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
```

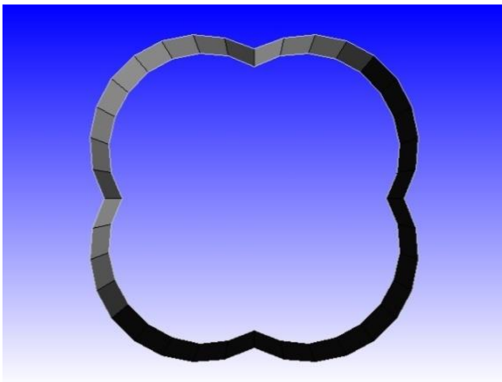
A very similar approach can be used to create a bulkhead that supports an internal structure such as a tank. The bulkhead would be constructed using a zero-length section object with one end curve defined as a lofted curve extracted from the desired position along the fuselage section and the other end as a lofted curve extracted from the tank object.

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:

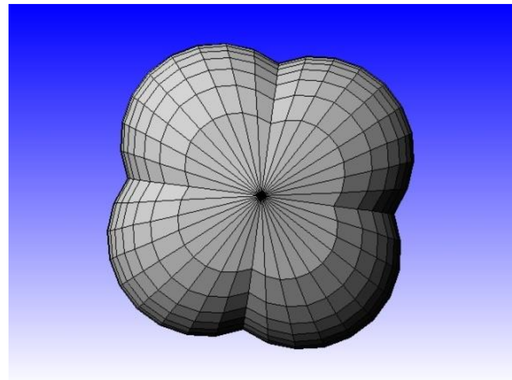
```
child cir
x 1.0
y -1.0
radius 2.0
child cir
x -1.0
y -1.0
radius 2.0
child cir
x -1.0
y 1.0
radius 2.0
```

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
# Set units and save
units feet
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 in Chapter 6). 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 the tutorial project 4.

This project discusses a number of 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.

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 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 in order 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
```

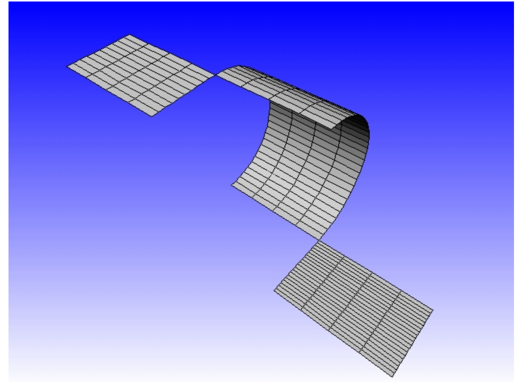
The global "s" start and stop points indicate what portions of the parent curve are contributed by each child. We can use those values to extract just those contributions into new compound curves:

```
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
```

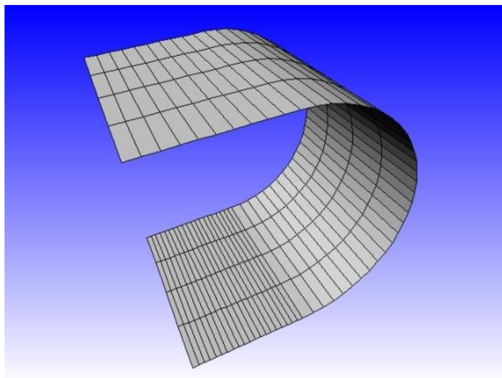
(Remember that the "roundbox" curve definition needs to be copied into this new input file – user-defined curves are not added to *Loft*'s internal library permanently.)

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:



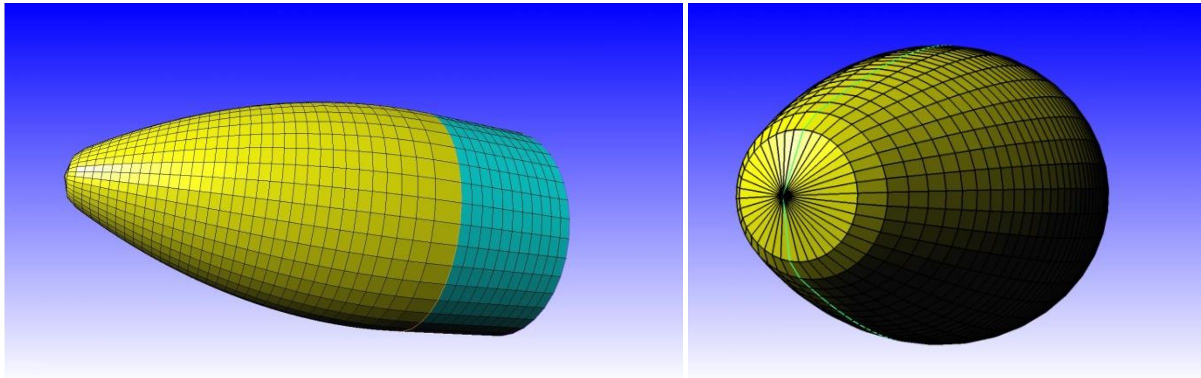
This sub-curve technique gives the user a lot of additional control on mesh density and locating important nodes, but it is a lot more effort than the other approaches. The main drawback in this approach is the difficulty in obtaining compatibility with meshes generated without sub-curves. Generally, 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 or modify portions of the current stack. This tutorial illustrates a small portion of these capabilities.

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.

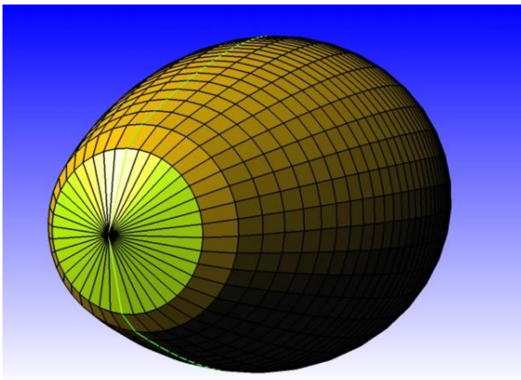


```
object dome Nose
curve1 cir
c1_xscale 1.0
c1_yscale 1.0
length -550.000
nodes_circ 41
nodes_axial 35
components_circ 1
components_axial 1
taper ogive
param1 55.
param2 983.230
param3 198.0
zdist 0.73
transz 618.0
object dframe Sep Joints
count 3
align axial
#
object section Barrel
length 200.0
c1_xscale 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 command 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 0.0 200. 30. 30.
pprem Nose Sep
setpp Nose Cap
```



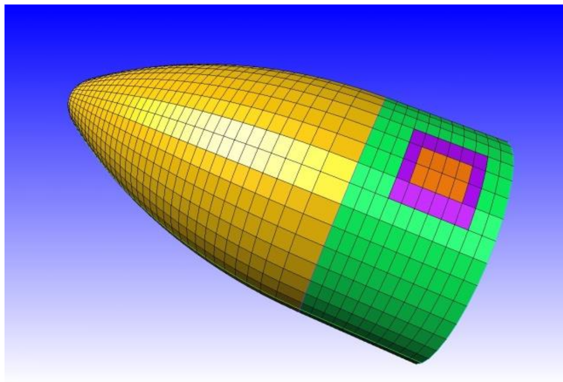
There are two parts of defining this region. The **inclusive** add parameter `iadd` adds all elements that have any nodes within the specified cylindrical area. In this case, the beam elements that represent the separation joint should not be updated. So, the remove by physical property name parameter `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 operations 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 commands to perform the two active parameters.

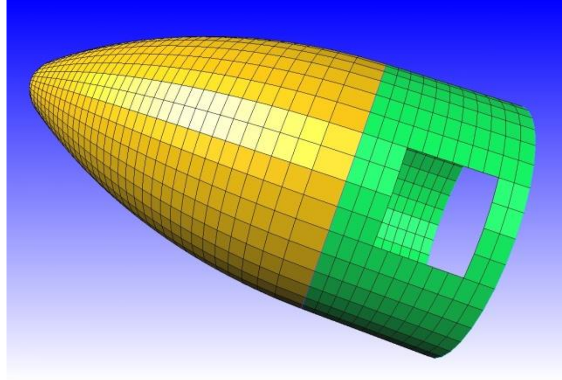
```
# Cut out a door with frame border
region
iadd box 732. 0. 198. 85. 72. 120
setpp Large Door Frame
region
eadd box 732. 0. 198. 85. 72. 120
setpp Large Door
```

Note that the two add operations 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
format vrml
filenew project6a.wrl
rwrite
```



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 commands `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 has to be updated. The basic math support in the *Loft* input file reader adds even more flexibility.

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 can be used to perform 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 and any desired number of characters) can be used for the variable name. In order 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 system variables in Chapter 6 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 in Chapter 6 for a full list of supported functions.

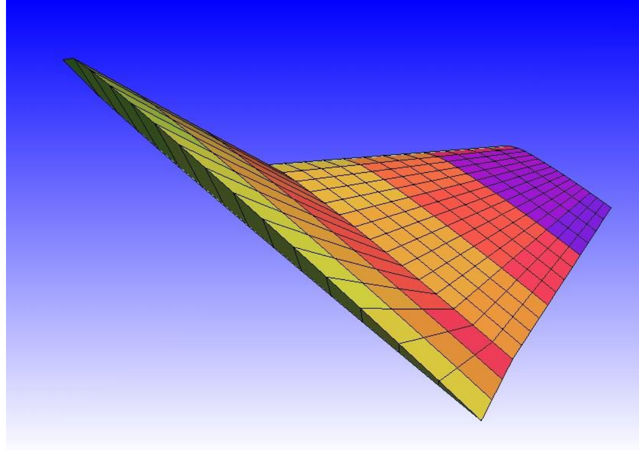
Math functions are called by using the percent symbol and the function mnemonic. They have to be placed at the end of a line after any variable or arithmetic. Multiple functions can be used on a single line. Each function will be applied to the preceding number in the order read.

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
```

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.



In this example, 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 `move`, with no parameter lines before the var1 definition would result in both variables have the same, expected, value of 10.

Project 8: Bodies of Revolution, Toroids, and Helixes

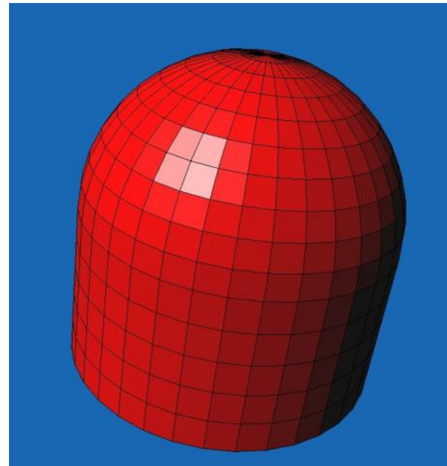
Any curve type can be used to create a body of revolution in *Loft*. Five parameters in the section object type can be used to create bodies of revolution, toroids, and helixes. These parameters are `radius`, `c1_rotation`, `c2_rotation`, `c1_yoffset`, and `c2_yoffset`.

There are 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 (the examples in this project include this move.)

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:

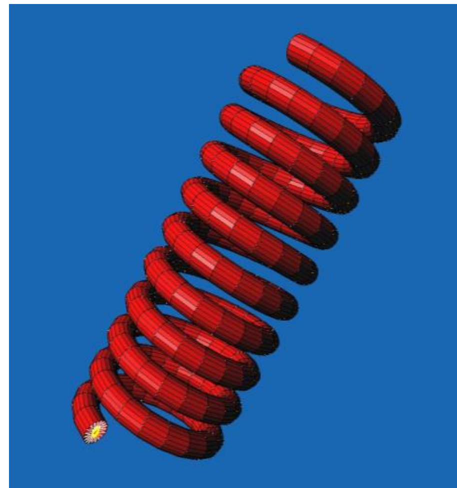
```
# Body of revolution
object section bor
   curve1 sbb
   curve2 sbb
   nodes_axial 36
   nodes_circ 21
   length 0
   radius 0
   c1_rotation 0
   c2_rotation 360
# save
write vrml bor.wrl
end
```



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.

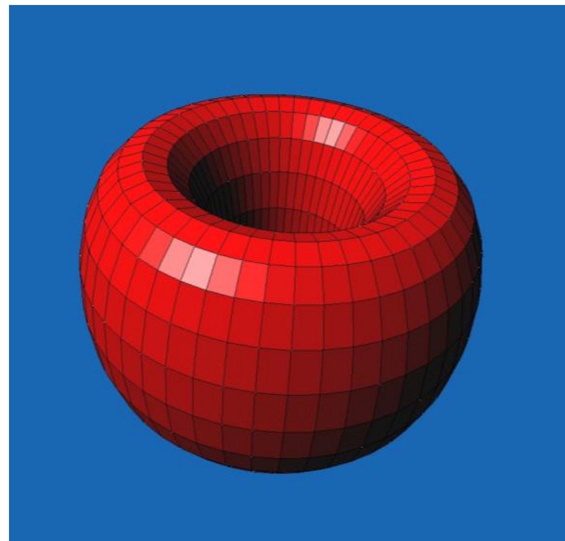
```
# Helix
object section Spring
   curve1 cir
   curve2 cir
   nodes_axial 360
   nodes_circ 10
   length 0
```

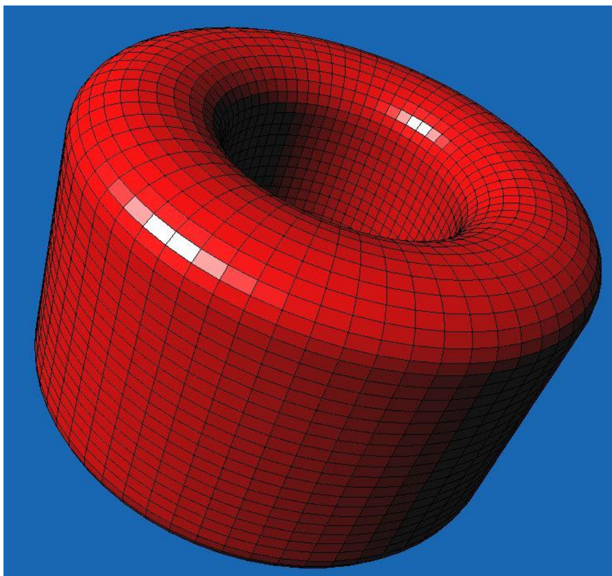


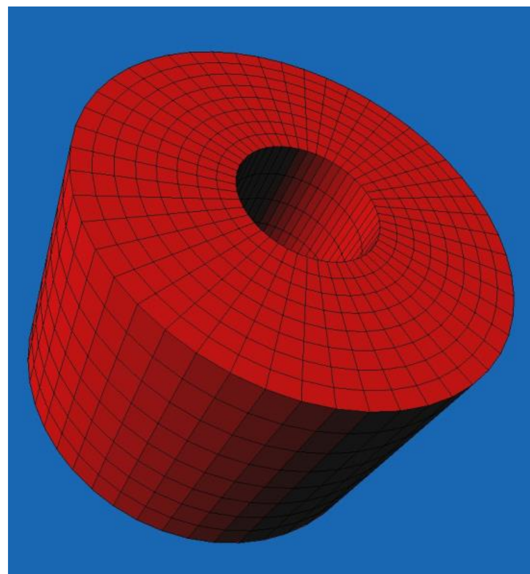
```
      radius 2
      c1_rotation 0
      c2_rotation 3600
      c2_yoffset 30
   move
      transx -2
   # save
   write vrml spring.wrl
   end
```

Any *Loft* curve type can be used including user-defined curves. Curve 1 and Curve 2 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 images below use three different cross section curves (`cir`, `fillet`, and `squ`), with the same 360-degree rotation, radius (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
```







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.

First, recall the distinction between *Loft* commands and *Loft* parameters. There are a very short list of *Loft* commands (`object, move, define, write, read, region`, etc.). Parameters modify how the active command is executed by changing a default value such as a dimension. For the `region` command all of the parameters are acted on when specified and are therefore more active than the term “parameter” implies, so the term “operation” is used instead.

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 material or physical property 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 operations that can be performed on a region. Passive operations are actions such as queries that do not change the mesh data. Active operations modify the mesh data in the region by changing properties, deleting nodes or elements, etc. **Only one active operation 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` commands 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 operation to complete the current region operation.

Region Operations

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. All region commands reset the list of selected nodes and elements to be empty.

Definition Operations

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

The volumetric selection operations 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 Operations

`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.

Usage: **`iadd`** *<type>* *<center of volume>* *<dimensions of volume>*
Type = "all", "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: `iadd sphere 10. 20. 25. 5.`

`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.** 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 removed. The type "all" will remove all nodes (and thus all elements) in the current stack. No dimensions are required for the "all" type.

Usage: **`irem`** *<type>* *<center of volume>* *<dimensions of volume>*
Type = "all," "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: `irem sphere 10. 20. 25. 5.`

`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. 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. The type "all" will add all nodes (and thus all elements) in the current stack. No dimensions are required for the "all" type.

Usage: **eadd** <*type*> <*center of volume*> <*dimensions of volume*>
Type = "all," "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: `eadd sphere 10. 20. 25. 5.`

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. **This operation does not delete anything from the mesh, it just removes the specified items from the region selection list**. 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. The type "all" will remove all nodes (and thus all elements) in the current stack. No dimensions are required for the "all" type.

Usage: **erem** <*type*> <*center of volume*> <*dimensions of volume*>
Example: `erem sphere 10. 20. 25. 5.`
Type = "all," "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: `erem sphere 10. 20. 25. 5.`

Property Selection Operations

mpadd – Add elements to the selected list based on their material property name. The material property name is used by *Loft* during object creation 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.

Usage: **mpadd** <*material property name*>
Example: `mpadd lox tank`

mprem – Remove elements from the selected list based on their material property name. The material property name is used by *Loft* during object creation 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 to restore those nodes to the active region list.

Usage: **mprem** <*material property name*>
Example: `mprem lox tank`

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.

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 to restore those nodes to the active region list.

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 several that contain its object name and different collections of nodes and elements. 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.

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 several that contain its object name and different collections of nodes and elements. 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 to restore those nodes to the active region list.

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

Passive Operations

Passive operations can be used to list information about all of 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 – Change all items in the selection list to not-selected and all not-selected items to selected.

Usage: **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**

fileout – Specify an output file to send `query` and `rwrite` outputs to. By default, this output is printed to the screen. Since the user may wish to save multiple results to the same file, all output is appended to the end of a (possibly) pre-existing file.

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

filenew – 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. 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. "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`

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`

rwrite – Write the selected items as if they were a complete mesh. Uses the values set by the format and `fileout` or `filenew` commands.

Usage: **rwrite**
Example: `rwrite`

Active Operations

Active operations attempt to change the portion of the stack's mesh that is contained in selected region in some way. This can be a property change, the addition of a mark, 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`

`mark` – Add a mark to all selected elements. Note the difference in syntax versus the object level mark command where one specifies "element" or "node" as well as a name.

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

`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 nose cap`

`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`**

`rotate` – Reorder element nodes to rotate element orientation

Usage: **`rotate`**

Chapter 4: Programmer's Guide and Reference

Introduction

This portion of the *Loft* 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 and is 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 some program operations are described, the C file and/or subroutine that performs the function may 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 chapter 6. (`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
tecplot.c
stl-ascii.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 list of marks (a marklist structure). 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_panels.`

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 mesher 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 mesher routine if the object is of your new type:

```
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 mesher 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 a 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 several 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` 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 el-

ements 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.

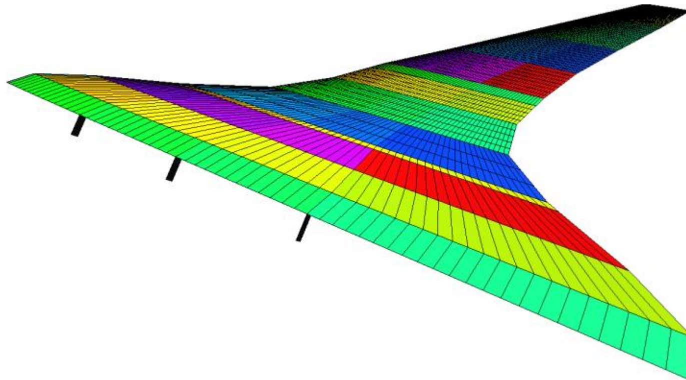
Be careful when selecting your curve's mnemonic to avoid collisions with other curves. For instance, if you want to use the mnemonic "ssquiggle," you need to add your check to `curvefunctionptr` before the check for the semi-square curve, since that check compares the first two characters of the curve 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 5: 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 an utilty 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 the *Loft* 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. "upper-skinelems.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.

`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.

`wfuse` - Half of average width of vehicle fuselage along wing. This value is used to create non-skinned carrythrough.

`n` - Supplied number of wing sections.

n lines of **`Xle,Yle,Zle,Xte,Yte,Zte,Tmax`** where

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

`Xte` = x location of section trailing edge

`Yle,Yte` = 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. (for example, an input of `nspars 2` will give the exact same result as `sparpos 33.33` and `sparpos 66.666`.) 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*, if ribs are desired at 0 and 100% span; they will need to be specified in this file (using an `nribs` value of 2, the default, will create just the 0 and 100% ribs.).

Parameter List (can be specified in any order):

`nribs` = Number of evenly spaced ribs to create (default 2)

`nspars` = Number of evenly spaced spars to create (default 0)

`ribpos` = Percent span (0-100) location to create a rib (no default)

`sparpos` = Percent chord (0-100) location to create a spar (no default)

mesh = Finite element mesh density per unit length (higher values produce a denser mesh) for all three mesh directions. The three specific parameters meshthick, meshspan, and meshchord are reset to this value when the general mesh parameter is used.

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)

rotx = Specifies a desired rotation about the x (axial) axis (dihedral) of completed wing (default 0.0)

roty = Specifies a desired rotation about the y (spanwise) axis (angle of attack) of completed wing. (default 0.0)

rotz = Specifies a desired rotation about the z (vertical) axis of the completed wing. (default 0.0)

weight = Specifies total 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. (default 0.0)

mergetol = Specifies tolerance for *Loft*'s node equivalence operation. Any nodes that are at the specified value or closer will be merged together. (default 0.02)

minthick = Integer value specifying minimum percent thickness for wing sections. (default 1)

naca = 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. (default “00XX”)

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).

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.

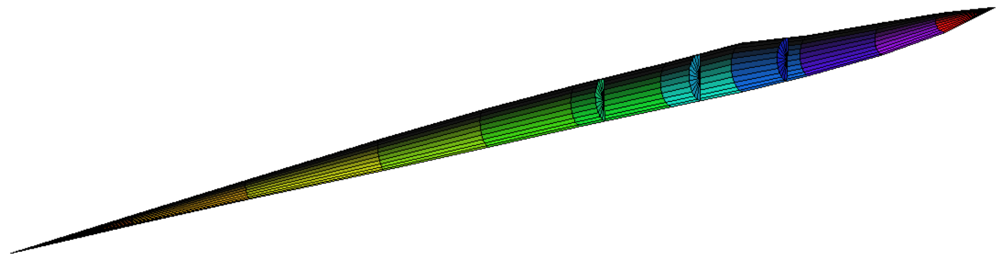
Example rootnodes.txt file:

```
$ 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 roots.

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.

> **`x0, y0, z0`** = Coordinates of fuselage reference location (nose).

n = Supplied number of fuselage sections.

n lines of *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 of the fuselage at that station

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 = Finite element mesh density per unit length (higher values produce a denser mesh) in the axial direction. (default = 3.0)

meshcirc = Finite element mesh density per unit length (higher values produce a denser mesh) in the circumferential direction. (default = 3.0)

bulkhead - 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. (default none)

Example: `bulkhead mainwing 28.75`

rotx = Desired rotation about the x (axial) axis of completed fuselage (roll). (default 0.0)

roty = Desired rotation about the y (spanwise) axis of completed fuselage (pitch). (default 0.0)

rotz = Desired rotation about the z (vertical) axis of the completed fuselage (yaw). (default 0.0)

curve = Specifies a *Loft* curve name to be used for the fuselage. The rhorz and rvert values read from hrm2fusecoords.out will scale this shape to the specified dimensions. (default sc)

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 6: Command & Object Reference

Alphabetical Command List

`Curve` – Define a user curve

Usage: **`curve`** <*type*> <*mnemonic*>
type = “interpolated,” “compound,” “lofted”
mnemonic = user 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 program (optional)

Usage: **`End`**
Example: `end`

`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`

`List` – Output various lists to the screen

This command is intended for debugging purposes. The options “groups” and “marks” are synonymous.

Usage: **`list`** <*type*>
Default: none
type = “ccurves,” “icurves,” “lcurves,” “stacks,” “variables,” “groups,” “marks,” “mprops” (material properties), “pprops” (physical properties), “ribs,” “spars,” “mesh” (gives various data counts), or “all.”
Example: `list stacks`

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 or element) and the group name. A marked group can contain either nodes or elements, but not both. Current labels can be listed using the `list groups` or `list marks` command.

Usage: **mark** <*type*> <*name*>
Default: none
Example: `mark element OML`

MergeTol – Tolerance 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`

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 is less than or equal to 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 of "`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 starting 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

Usage: **`nastran`** *<parameter>* *<value>*
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.
Examples: `nastran grid 8, nastran cylx`

`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 `write` command to save the previous one.) All defaults are reset to their initial values.

Usage: **`new`**
Example: `new`

`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 effects the output of the `query` parameter in region mode.
Value = amount to offset the indices. Note: *Loft* internal indices start from 0. NASTRAN, for instance, does not support elements or nodes numbered 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. (under development)

Usage: **quality**
Example: `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" (I-DEAS' universal (.unv) not currently supported.)
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 operations that are described in chapter 3. These abilities partially overlap the `list` and `subtract` commands.

Usage: **region**
Example: `region`

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

Usage: **`reset`**
Example: `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 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" or "feet", "inch", "cm" or "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
Example: `vrml reverse`

Default: primary

`Write` – Write current stack 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 file to be saved to
Example: `write vrml rocket.wrl`

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.

Common Parameters

All object types except the individual beam object 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).

`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)

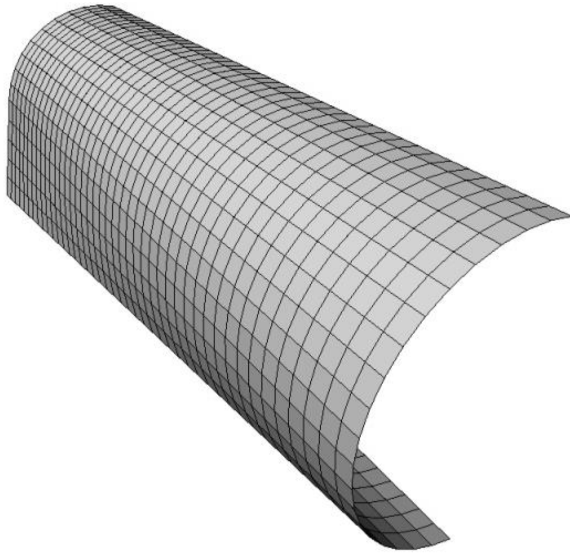
`gwarppx`, `gwarppy`, `gwarppz`, `gwarpnx`, `gwarpny`, `gwarpnz` – distort the part of the object in the region specified by the last two letters (p means positive, n means negative, and x, y, and z are the coordinate axes) by the specified three values. Scaling of the original coordinates varies linearly 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 or element) and the group name. A marked group can contain either nodes or elements, but not both.

Example: `mark element OML`

Section



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).

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

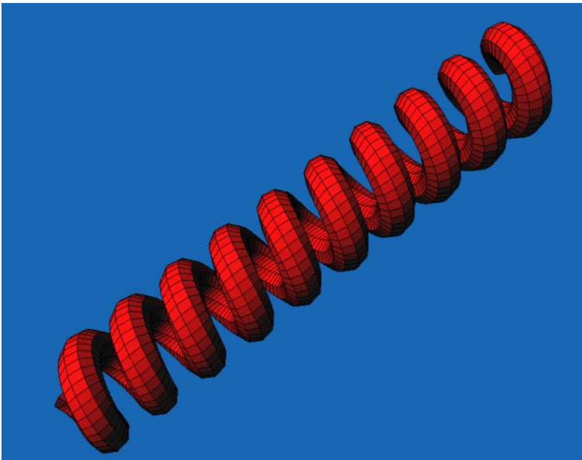
`c2_yscale` – factor to scale y dimensions of curve 2 by
default = 1, or last y scale

`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 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 c2_s

c2_s – scheme 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 circumfrential 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

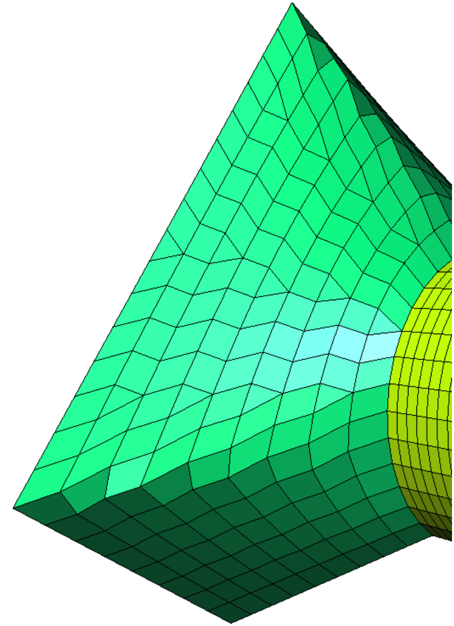
`axial_cpos` – position of one axial component edge in percent. Values can be the word "reset" to remove the current list of positions, or between 0 and 100 to set the percentage where elements created after that location will be in a new component. Multiple positions can be set. Use of this parameter 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` – This setting controls how quickly curve 1 transitions to curve 2. 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 chapter 6. 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

TSection



A *TSection* 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.

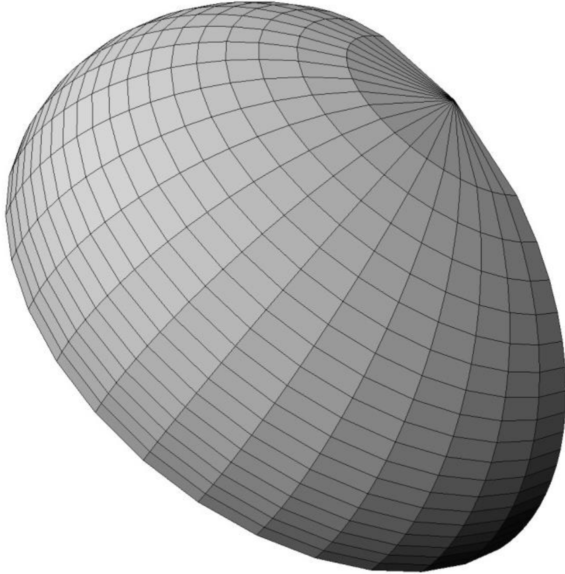
No TFrame object has been created to allow ring frames and longerons to attach to a TSection. A conventional frame object may be used. It should stitch well along edges of the section but will generally not attach properly across the middle of a Tsection. If such a mid-frame is desired, use multiple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

`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 *dome* 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 circumferntial 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

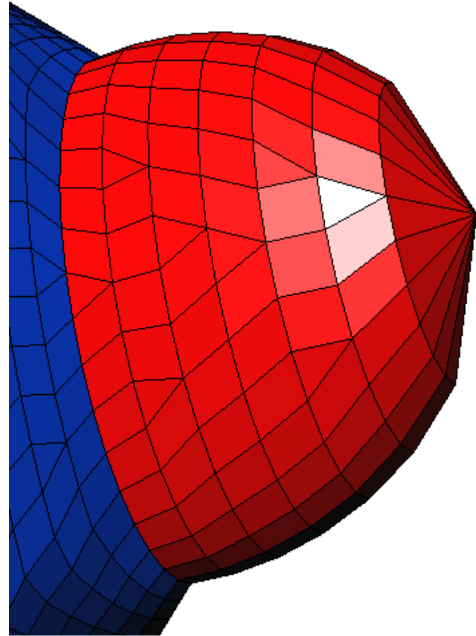
`droop` – mnemonic for droop schedule (see droop library)
default = line

`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: $z_i = \text{length} * (i/\text{nodes_axial})^{\text{zdist}}$
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).

TDome



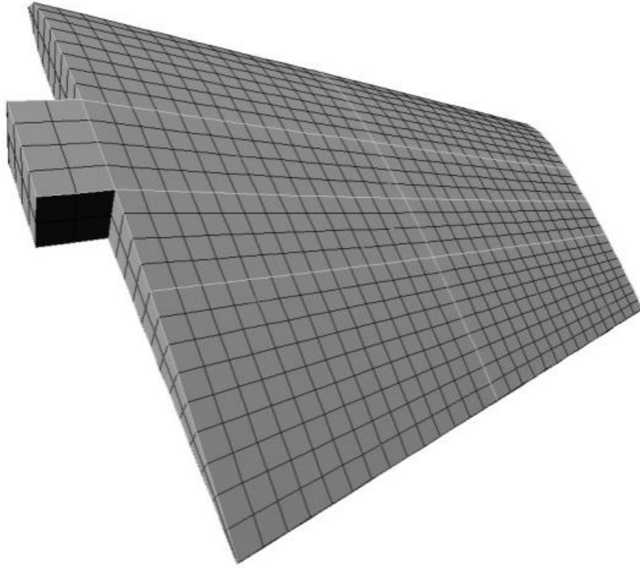
A ***TDome*** is an under-development variation on the dome object type. This object allows the user to specify a different value of `nodes_circ` at each end of the section. This results in a number of triangular elements being created to gradually change from one node count to the other.

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:

`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 lifting surface (e.g., a right wing, a tail, a winglet, etc.) 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 and tip, 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 both the specific parameters `rootnaca` and `tipnaca`. 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 "`rootnaca 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.

Parameter List (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` – Airfoil NACA designation (contains camber and thickness data) for wing root. Currently only 4 and 5 digit airfoils are supported, but more series may be added in the future.
Default: 2410

`ripnaca` – Airfoil designation for wing tip.
Default: 2410

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

`tipaoa` – Tip twist angle in degrees. Wing half-chord at the tip is the rotation axis and 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 (y-direction).
Default: 0

`tipvert` – Vertical offset of wing tip. Positive is up (y-direction). Can be used to produce wing dihedral.
Default: 0

`wingbox` – Carrythrough length. May be zero. At least 2 spars must be specified if a carrythrough 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 carrythrough 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 carrythrough 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. 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 carrythrough. 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

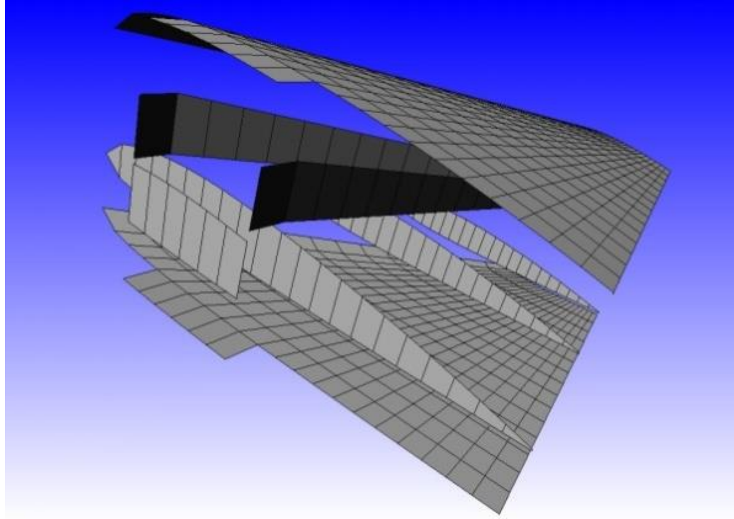
`stop` – Percentage of chord length to stop generating the object. Any spars that are specified at higher positions than this value are ignored. The `start` and `stop` parameters are used to generate partial wing objects (e.g., control surfaces).
Default: 100

`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 specific parameters `rootnaca` and `tipnaca` are reset to this value.

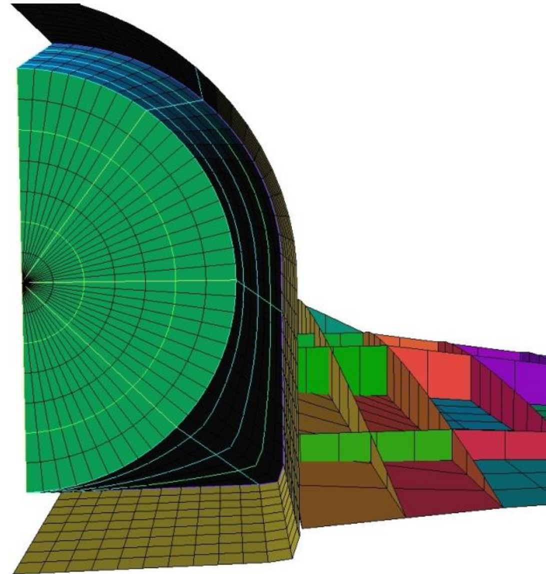
`nribs` – Number of wing ribs, including root and tip. Must be greater than or equal to 2. When used, the current `ribpos` parameter settings are erased and the specified number of new evenly spaced ribs are placed in the `ribpos` list.

`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 *section* object – taking their shape and dimensions from that section. To attach stiffeners to a *dome* object, use the *DFrame* object described below. 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.

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.

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. Values are “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 one.
Default: 0

`type` – Type of 1-D object to generate. Values are "`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

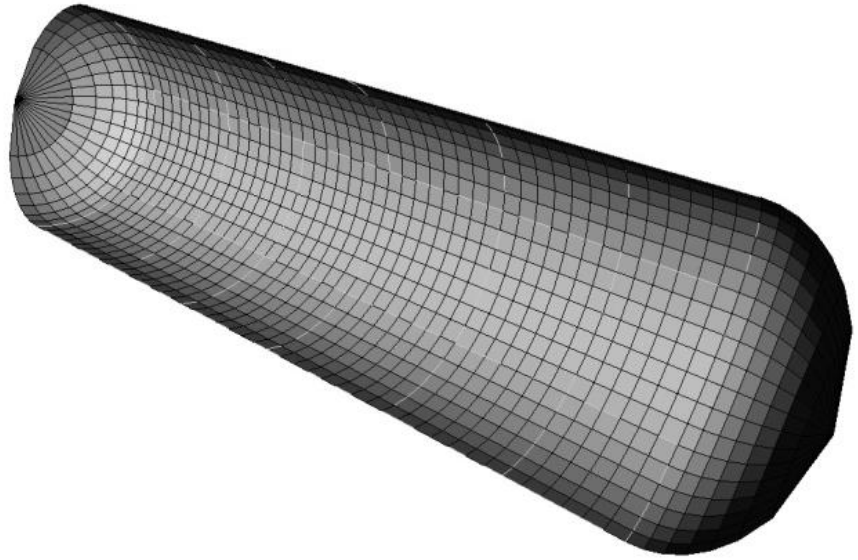
`type` – Kind of 1-D object to generate. Values are "`beam`," "`rod`," or "`bar`" (`rod` and `bar` are the same).
Default: beam

`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.

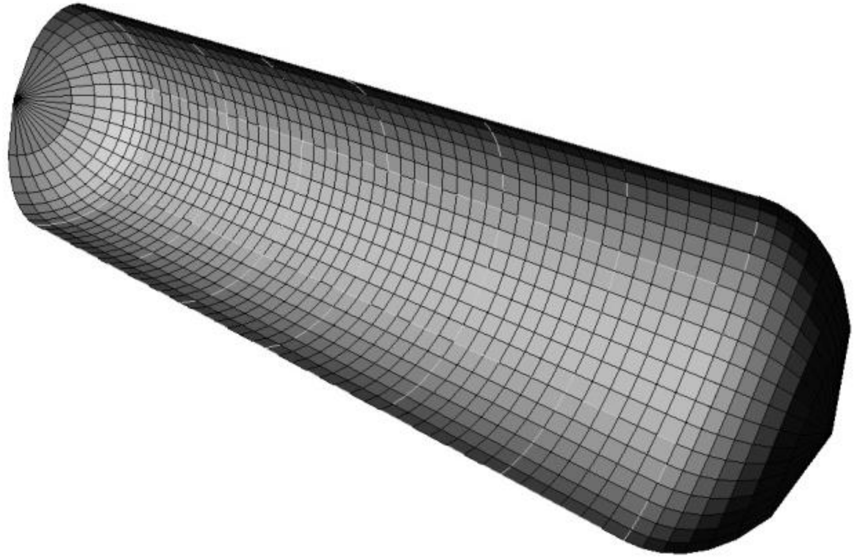
The tank local origin point is the centerpoint of curve 1 (the center of the front of the barrel section). Use of a tank object does update the global default creation point to the center of curve 2.

Additional Parameter List

See list for the section object type above for a base list of parameters.

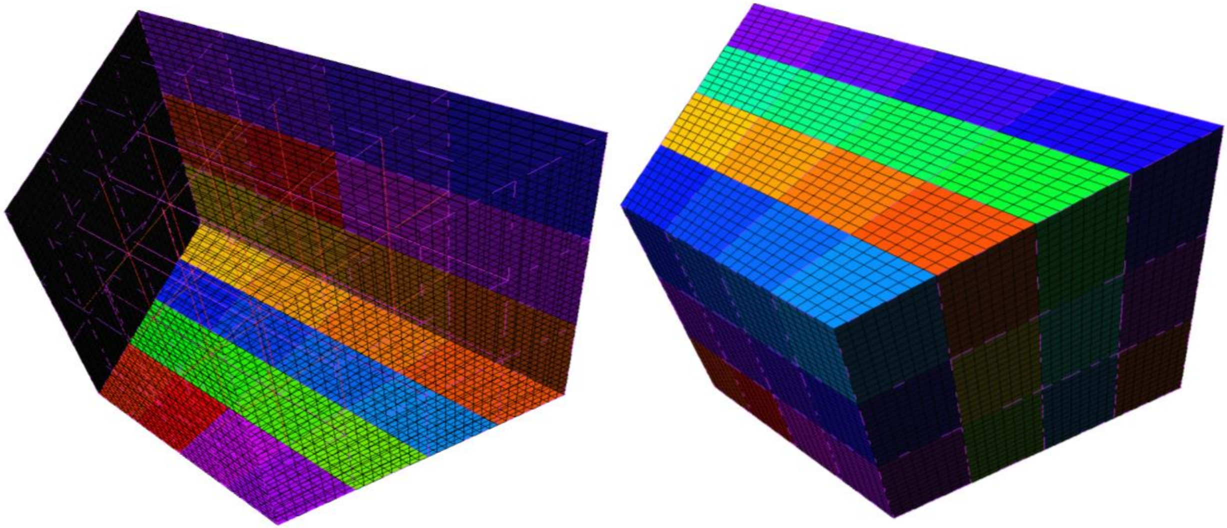
`domelength` – Length of the elliptical domes
Default: 0.707 * Average of corresponding section end's `scale_x`,`scale_y`

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

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 clockwise 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/sc curves, then the user will need to specify the portions of each curve that is to be used. See project 3B 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 sc), 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

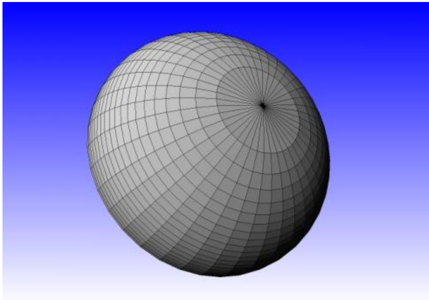
Filleted box – "fillet" – Square with rounded corners (**Note:** the distribution of s along the filleted box is not exactly by circumference.)
Parameter meaning – radius of fillet, between 0 and 1
Default = 0.25

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.)

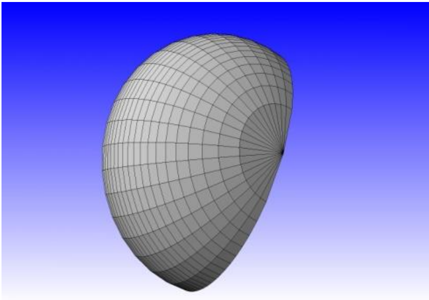
Parameter meaning – radius of fillet, between 0 and 1

Default = 0.25

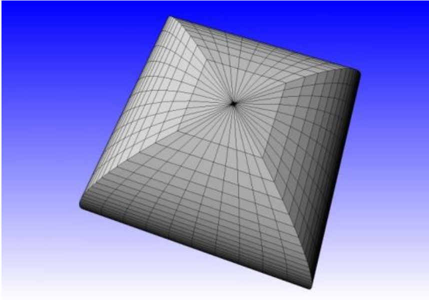
Library Curves illustrated with Dome Objects



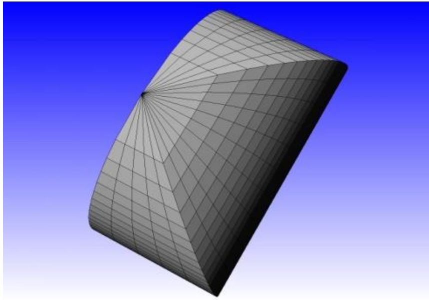
Circle



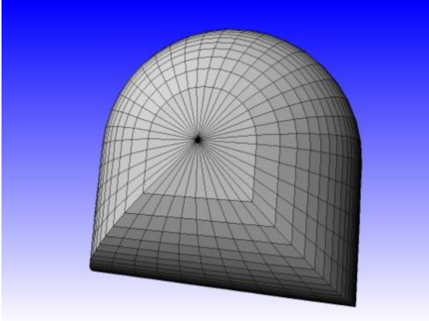
Semi-Circle



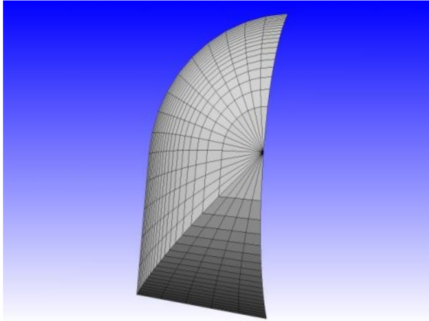
Square



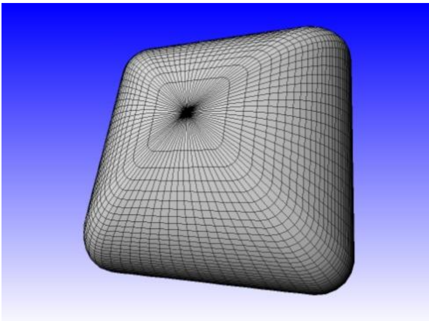
Semi-Square



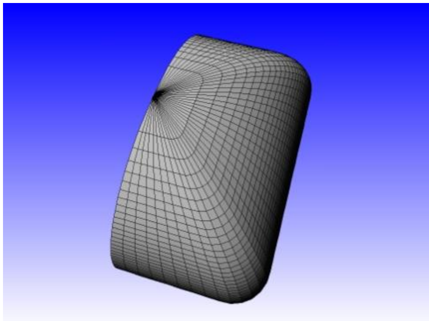
Breadbox



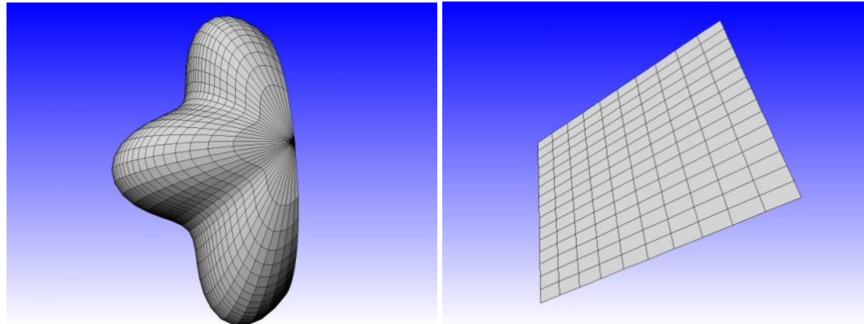
Semi-breadbox



Fillet



Semi-Fillet



Cosine Wiggle Line or Horizontal Line

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)

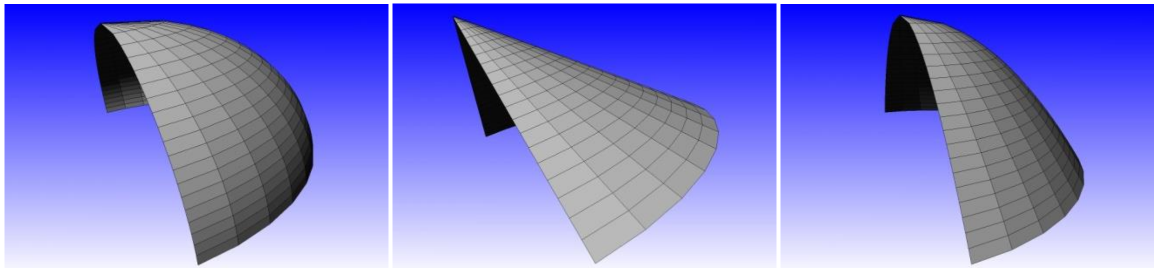
Parabolic – “para” – Power law nose shape
param1 = exponent of taper schedule. Default = 0.5 for a true parabola

Elliptical – “elli” – Elliptical taper for tank domes

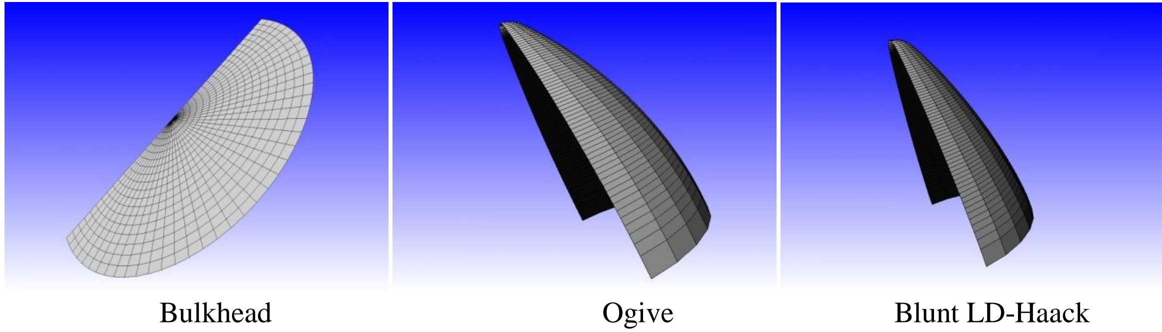
Ogive – “ogive” – Tangent ogive nose with spherical nose cap
param1 = nose cap radius. Default = 1.0
param2 = radius of main section curve. Default = 0.0
param3 = radius of nose base. Default = 1.0

Haack – “haack” – LD-Haack nose shape with optional spherical blunt cap (controlled by param3)
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

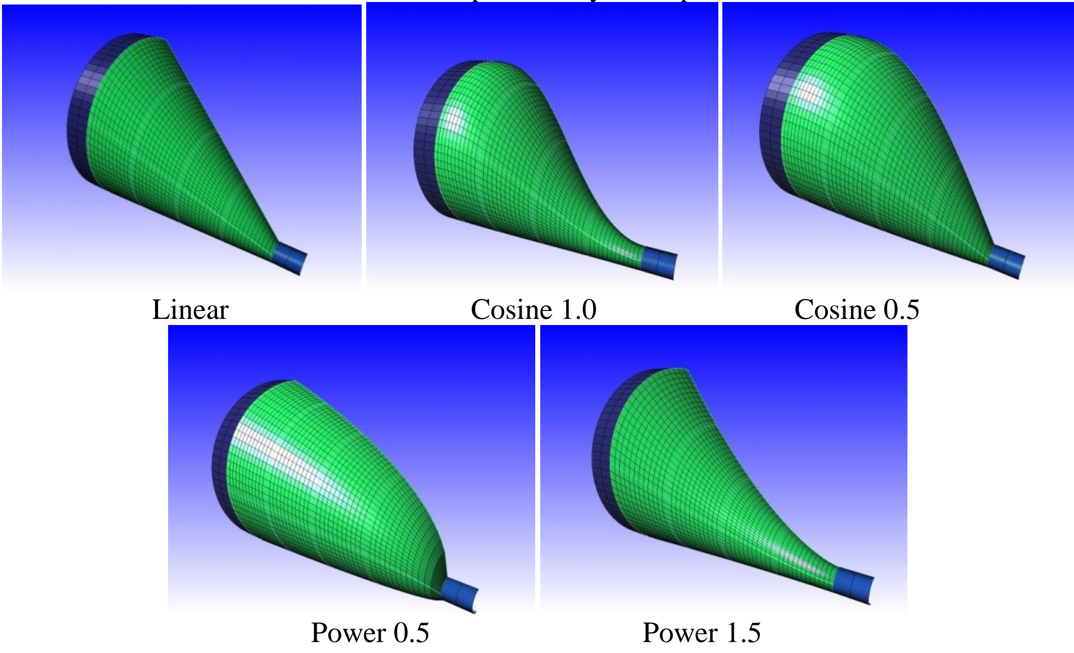
This is a list of the currently coded section taper schedules and the meaning of the `value` options. The pictures show a section object that interpolates between one semi-circle and a larger, offset semi-circle. Circumferential and axial frames are added.

Linear – “line” - Linear taper

Power – “power” – Power curve taper
value = exponent of taper schedule. Default = 1.0 which is linear

Cosine – “cosine” – Cosine schedule, offers tangency posibilities
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

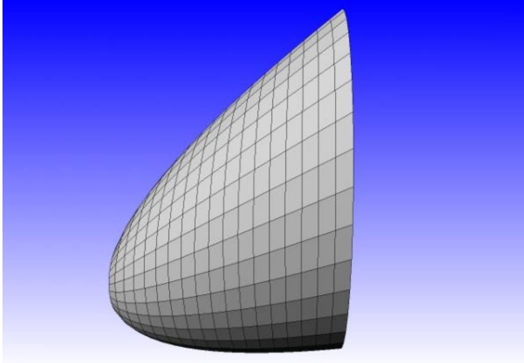
Droop Library

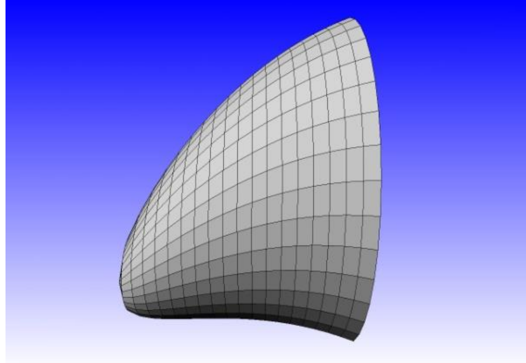
This is a list of the currently coded dome droop schedules.

Linear – “line” – Nose centerline descends linearly

Parabolic – “para” – Nose centerline descent smoothly increases

Droop Library Examples





Linear Parabolic

System Variable List

This is a chart listing 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

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 the 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 the 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 tranz – z position of next wing	@wing.transz

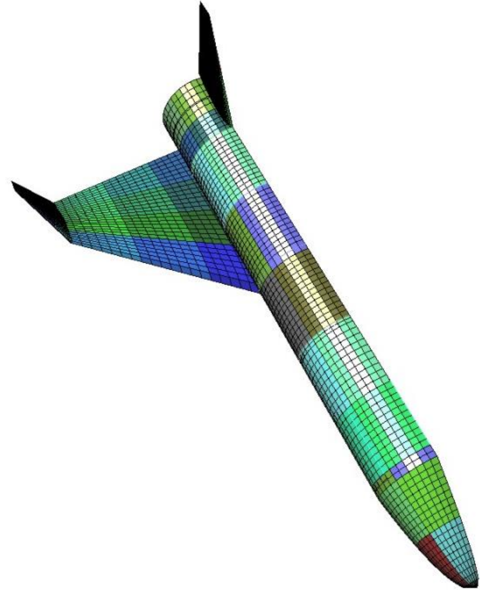
Math Function List

For these functions, all angles are in radians.

Function	Invoked by
Sine	%sin
Cosine	%cos
Tangent	%tan
Arcsine	%asin
Arccosine	%acos
Arctangent	%atan
Exp	%exp
Square root	%sqrt
Cube root	%cbrt
Absolute value	%abs
Integer or truncation	%int

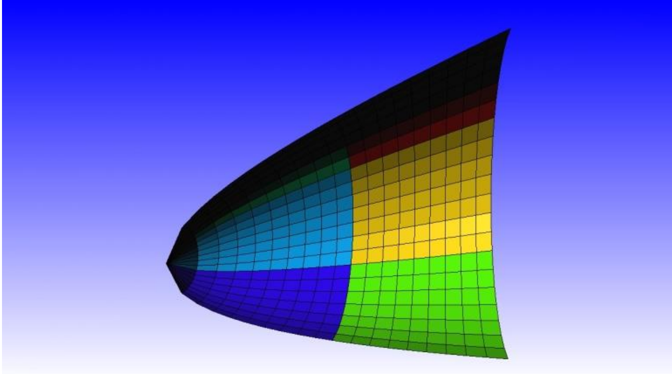
Chapter 7: Example Input Decks

Loft Example Input Deck #1

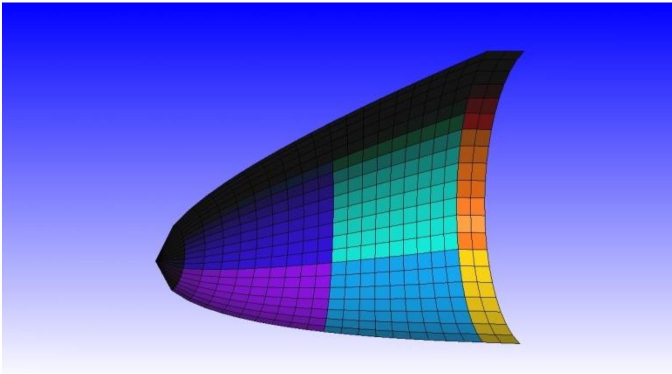


The first full example *Loft* input deck builds a simple conceptual level finite element model of a Two State To Orbit (TSTO) booster (BST) vehicle. A lot of the design details of the vehicle, such as stiffeners, are very notional and the wing carrythrough passes through the aft tank. Also, the wing objects are not stitched to the fuselage. It contains approximately 100 lines of basic *Loft* commands and parameters. It does not make use of user-defined curves, the region mode, or perform any store/recall operations.

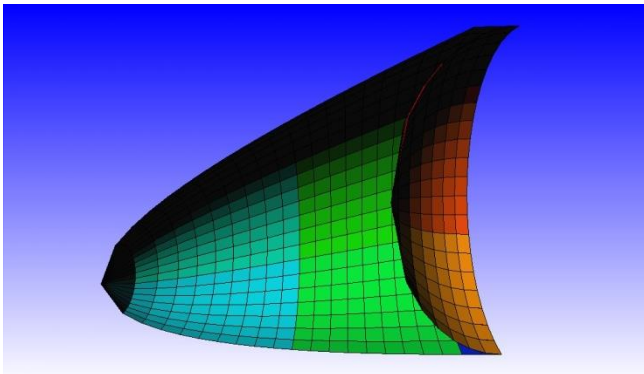
```
# Testing full vehicle based vaguely on
# ISAT Reference vehicle Mach 3.4 TSTO Vehicle
# Booster
# Our nose
object dome BST Nose
curve1 sc
c1_xscale 15.589
c1_yscale 15.589
length -36
taper para
nodes_circ 21
nodes_axial 20
droop line
zdroop 8
components_axial 2
```



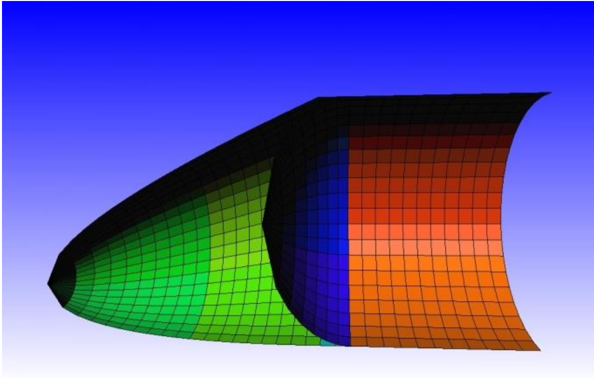
```
# Short fuselage extension to get nose
# not to impinge on forward tank
object section BST Nose Barrel
length 3.885
nodes_axial 3
components_axial 1
```



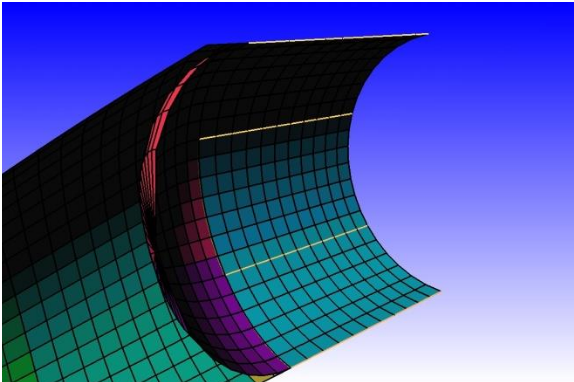
```
# Forward LOX (liquid oxygen) Tank
object dome BST LOX FW Dome
length -11.02
taper elli
nodes_axial 8
components_axial 1
```



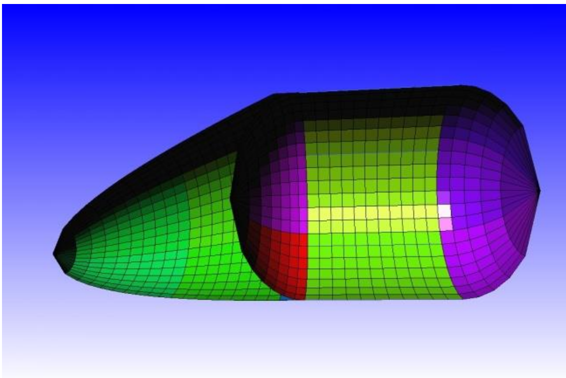
```
object section BST LOX Barrel
length 23.205
nodes_axial 12
components_axial 1
```



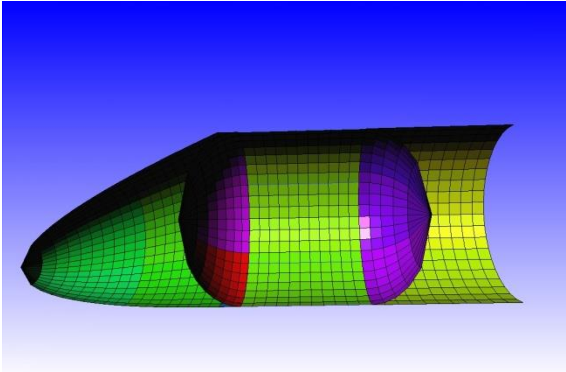
```
object frame BST LOX Frame
align axial
```



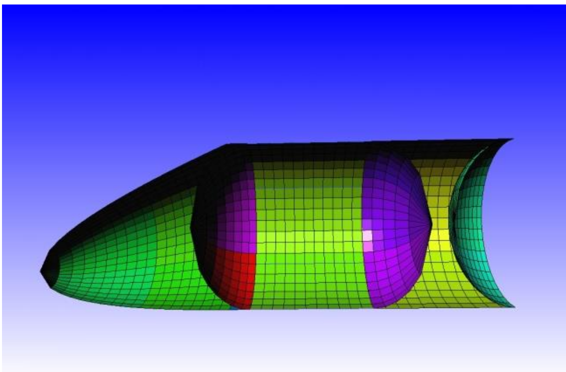
```
object dome BST LOX AFT Dome
length 11.02
taper elli
nodes_axial 6
components_axial 1
```



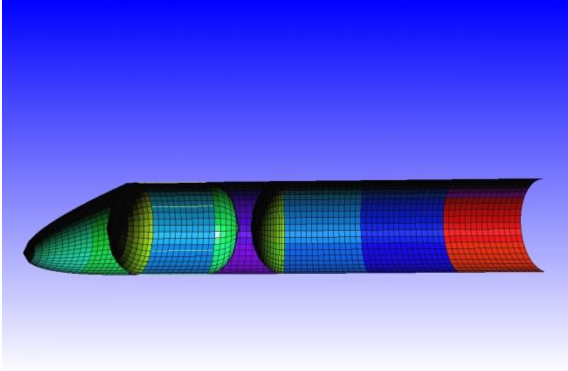
```
# ITA (Intertank adaptor)
object section BST ITA
length 26.04
nodes_axial 12
components_axial 1
```



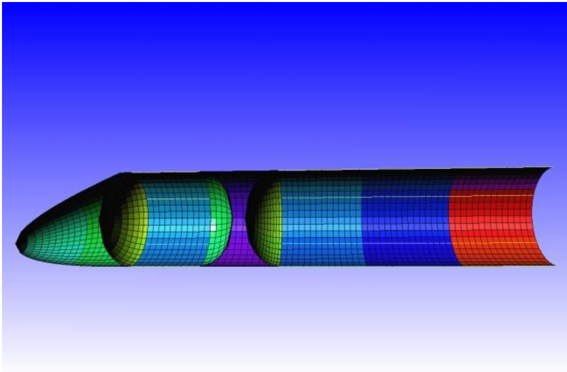
```
# LH2 (liquid hydrogen) Tank
object dome BST FW Dome
length -11.02
taper elli
nodes_axial 12
components_axial 1
```



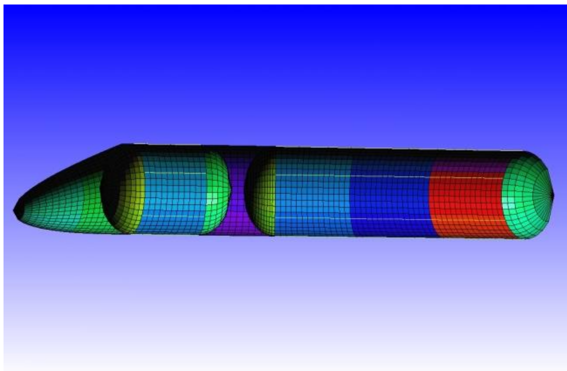
```
object section BST LH2 Barrel
length 87.35
nodes_axial 44
components_axial 3
```



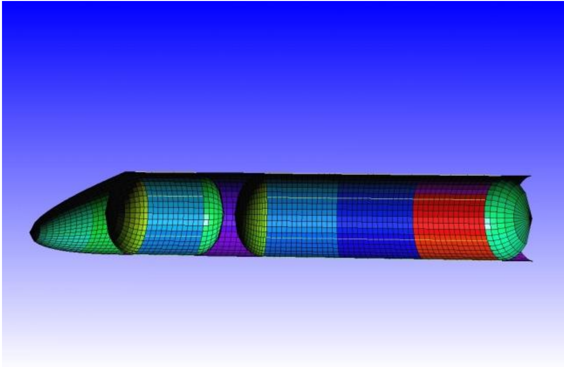
```
object frame BST LH2 frame
```



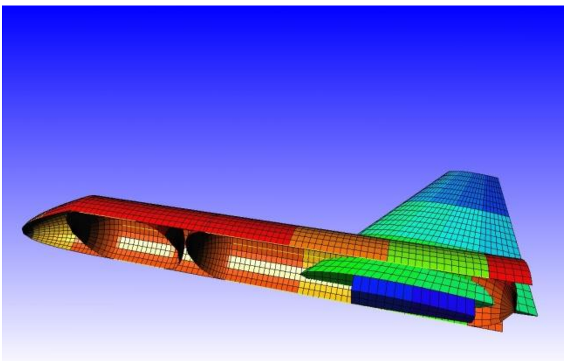
```
object dome BST LH2 AFT Dome
length 11.02
taper elli
nodes_axial 6
components_axial 1
```



```
# Tank shroud
object section BST Tank Shroud
length 11.02
nodes_axial 6
components_axial 1
```

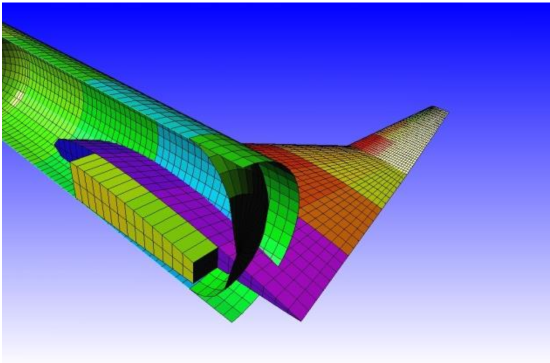


```
# Wing
object wing Main Wing
chord 80
span 60
taper 0.25
sweep 40
wingbox 6
transx 6
relz -70
rely -12
nribs 4
nspars 3
meshchord .4
meshspan .4
meshthick .4
naca 2412
```

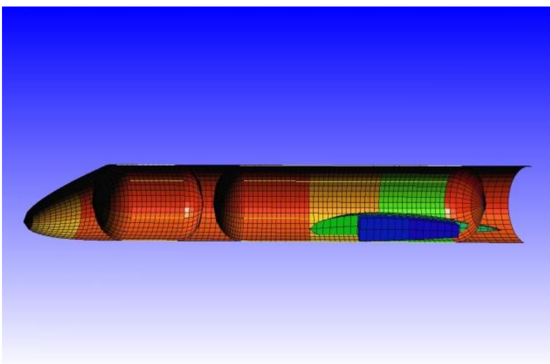


```
# Tip fin
object wing Winglet
chord 20
span 20
wingbox 0
transx 66
relz 50.35
rotz 50
```

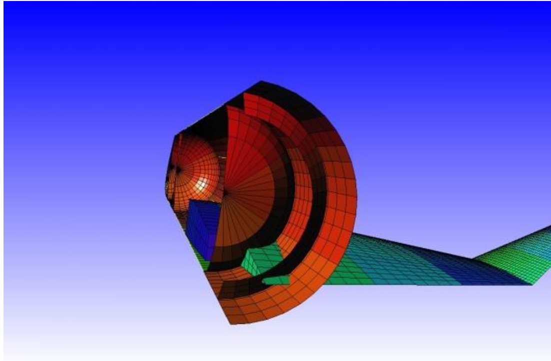
```
meshchord 1.6
meshspan 1.6
meshthick 1.6
```



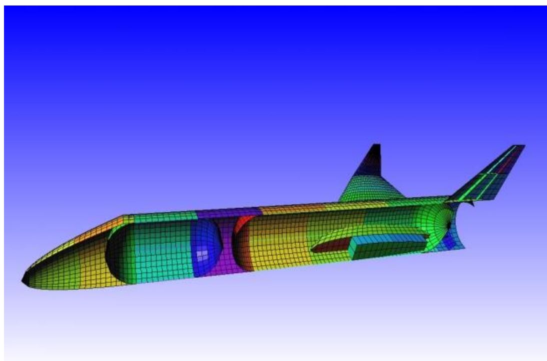
```
# TS (Thrust structure) shroud
object section BST TS Shroud
length 16.5
nodes_axial 6
components_axial 1
```



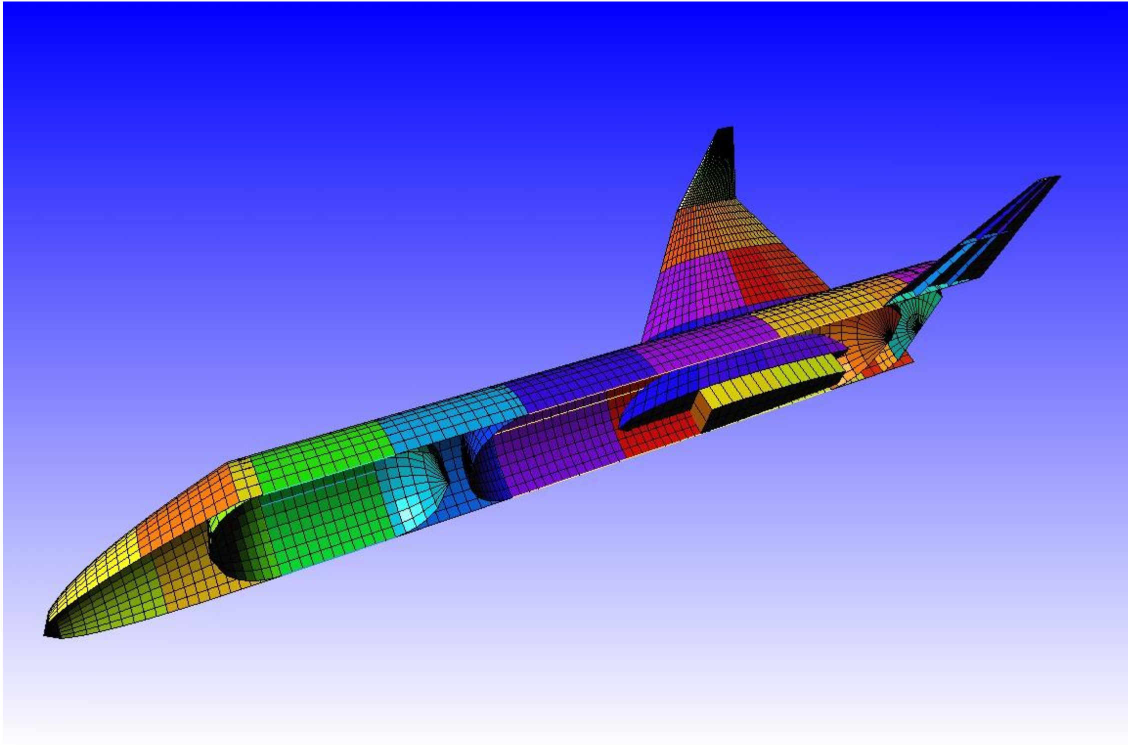
```
# Put a chopped off cone inside the shroud
# to represent the thrust structure
# note the relz parameter's use
object section BST Thrust Structure
length 3
c2_xscale 12
c2_yscale 12
relz -10.5
nodes_axial 4
components_axial 1
```



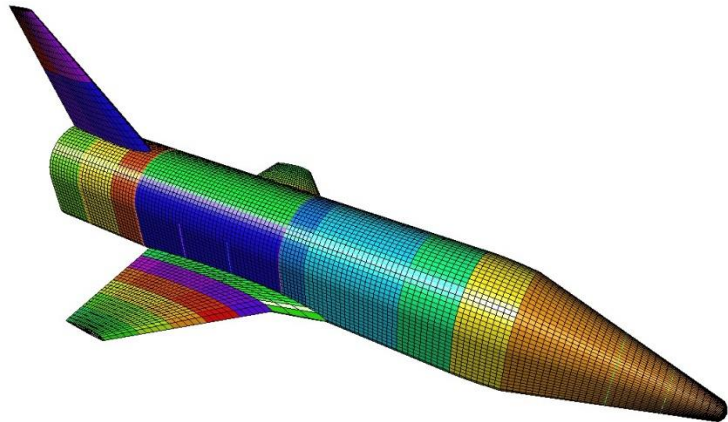
```
# Vertical tail on line of symmetry
object wing Tail
naca 0612
nribs 3
nspars 2
halfwing bottom
chord 30
span 30
transy 15.589
rotz 90
relz -20
mesh .4
```



```
# bulkhead to close off thrust structure
object dome BST Thrust Bulkhead
taper bulk
components_axial 1
# save
write vrml full-color.wrl
end
```



Loft Example Input Deck #2



The second full example *Loft* input deck 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 deck have been addressed in this model with carefully positioned stiffeners and wings. The deck contains approximately 800 lines of *Loft* commands and parameters.

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.

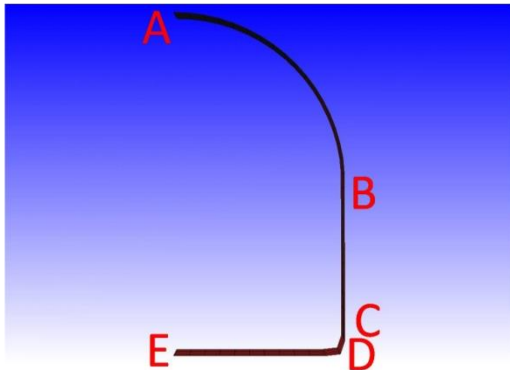
```
# Loft input deck to generate
# LaRC TSTO-2009-2A Orbiter
# aft LOX packaging
#
# Units are in inches
#
```

The first command defines the 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 scale factor used for the fuselage is defined here for the same reason. The third variable defines the position of the forward-most bulkhead on the forward tank as measured from the constant cross section portion of the fuselage. This dimension is needed in order to produce the user-defined lofted curves that define the bulkhead.

```
define circnodes 41
define fusescale 102.
```

```
define bulletbulk 100. # dist fwd of fslg to place bulkhead
```

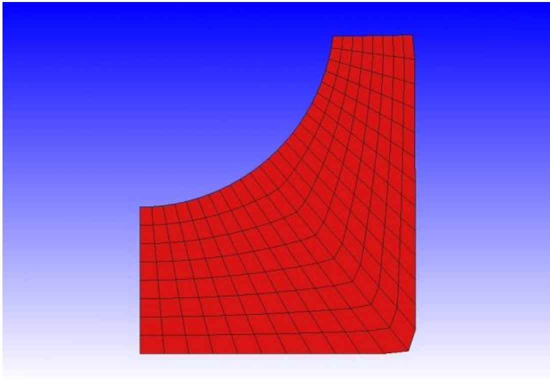
The first major section of the input deck defines all of the user-defined curves needed to construct the vehicle.The first such curve is the half-slice-of-bread cross sectional shape of the fuselage. The final shape is made of two circular portions: one at the top and one at the bottom outside corner, 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.




```
# Define child curves of unit half cross section
# (cross sectional shape fits in -1 to 1 square space)
# point defintion:
#     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
```

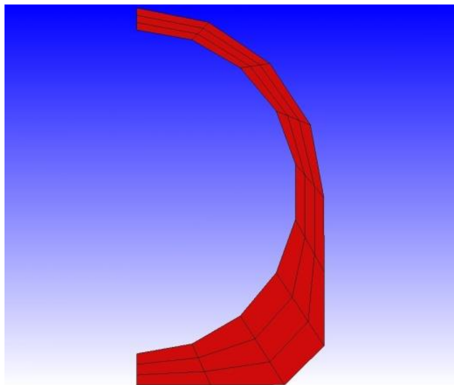
```
    x 0.941176
    y -0.94117647
    child mylineDE
```

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.



```
# Payload Bay Support bulkhead curves
# plb1 = semi-circle bay shape
# plb2 = sidewall & floor shape
curve compound plb1
  child sc
  sstart 0.54
  radius 72. / 102.
  x 0.0
  y 24.0 / 102.
curve compound plb2
  child body
  sstart 0.4
```

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
curve lofted forebullet
  curve1 sc
  curve2 body
  c1_xscale 18.0
  c1_yscale 18.0
  c1_yoffset -42.0
  c2_xscale $fusescale
  c2_yscale $fusescale
  c2_yoffset 0.0
  taper cosine .5
  station 450.54 - $bulletbulk / 450.54
curve lofted fore25
  curve1 sc
  curve2 forebullet
  c1_xscale 18.0
  c1_yscale 18.0
  c1_yoffset -42.0
  c2_xscale 1
  c2_yscale 1
  c2_yoffset 0.0
  station 0.25
curve lofted fore50
  curve1 sc
  curve2 forebullet
  c1_xscale 18.0
  c1_yscale 18.0
  c1_yoffset -42.0
  c2_xscale 1
  c2_yscale 1
  c2_yoffset 0.0
  station 0.5
curve lofted dome50
  curve1 sc
```

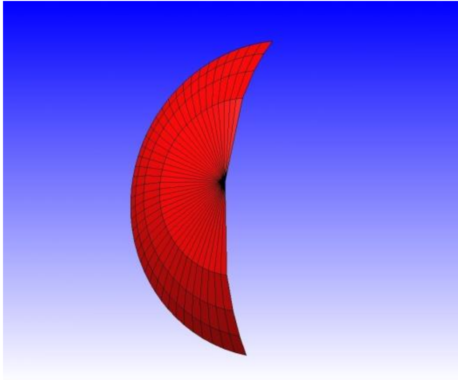
```
  c1_xscale 96.
  c1_yscale 96.
  taper elli
  station 0.50
list ccurves
list lcurves
```

Following the completion of the curve definition section, the `list` debugging command is used to confirm the creation of all of the desired curves. 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
```

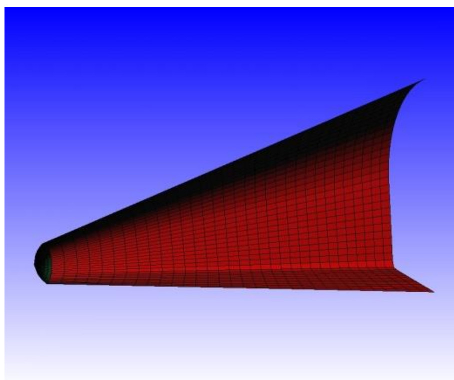
The input deck then starts defining the vehicle, starting at the nose. Note the use of the previously defined "circnodes" variable. Also notice that all external components are given the "OML" mark.

```
#
# Build vehicle
#
# ====================== Nose =======================
define caplength -9.
object dome nosecap
  curve1 sc
  c1_xscale 18.0
  c1_yscale 18.0
  c1_yoffset -42.0
  length $caplength
  nodes_circ $circnodes
  nodes_axial 5
  taper para
  components_axial 1
  components_circ 1
  transz $caplength
  mark element OML
```



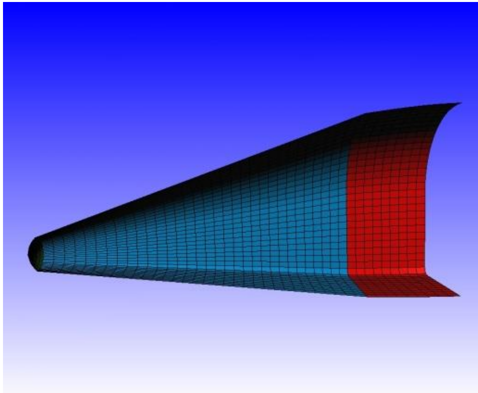
The nose length dimension supplied by the vehicle designer was 441.54 inches from the tip of the nose to the start of the constant cross section portion of the fuselage. The length of the section is computed parametricly from the length of the components on either end. Thus, the nose or the supported-length of the forward tank barrel could change, and this component would be updated to maintain the desired total length. The `nodes_axial` variable is chosen to be a multiple of four (32) plus one so that nodes are positioned at 25 and 50 percent of the component. The nose-gear bulkheads will be placed at these positions and will stitch to the fuselage correctly.

```
object section forebody
  curve2 forebullet
  c2_xscale 1.0
  c2_yscale 1.0
  c2_yoffset 0.0
  length 441.54 - $caplength - $bulletbulk
  nodes_axial 33
  components_axial 1
  mark element OML
```



```
object section forebody2
  curve2 body
  c2_xscale $fusescale
  c2_yscale $fusescale
```

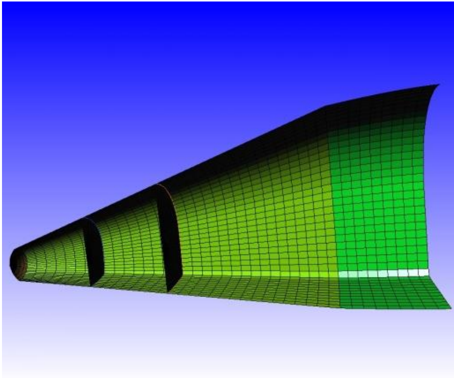
```
    c2_yoffset 0.0
    length $bulletbulk
    taper cosine .5
    nodes_axial 10
    components_axial 1
    mark element OML
```



The `move` command below has no parameters after it. Thus, it does not actually move anything. But, it does force *Loft* to generate the "forebody2" 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. The "offset" variable is used to position the two nose-gear bulkheads that immediately follow. 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.

```
move
define noseend @transz
define offset 441.54 - $caplength - $bulletbulk / 4
#
object dome Nose Gear Front Bulk
  curve1 fore25
  c1_xscale 1.0
  c1_yscale 1.0
  zdroop 30.0
  transz $offset + $caplength
  length -0.0001
  nodes_axial 8
  zdist 0.6
  components_axial 1
object dframe nose fwd ring frame
  count 1
object dome Nose Gear Rear Bulkhead
  curve1 fore50
  zdroop 20.0
```

```
  transz 2 * $offset + $caplength
  length -0.0001
  nodes_axial 8
  zdist 0.7
  components_axial 1
object dframe nose aft ring frame
  count 1
```



Finally, the completed nose is written to a VRML output file and 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.

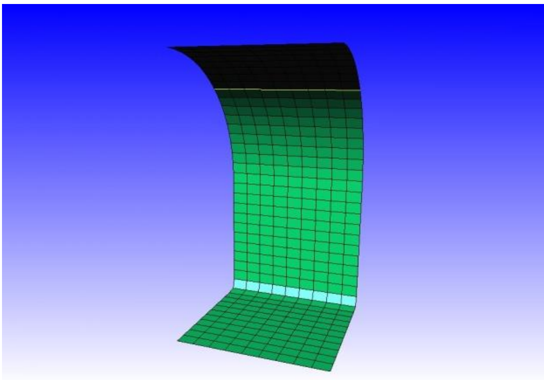
```
write vrml orb-nose.wrl
store nose
```

The global variable section below defines the length and position of all of the main fuselage components including tanks. These are collected in one place to make model updating easier. All of the later objects reference these dimensions. The “fuse_center_bay” variable definition line is wrapped onto two lines in this document and should actually be on one long line. As with the nose’s “forebody” object length, this variable is used to maintain the desired overall length of 1013 inches when tank and skirt dimensions are updated.

```
# ================= Global Variables =====================
# Tank barrel lengths
define fwd_tank 325.
define aft_tank 43.
# Skirts over domes
define fwd_tank_skirt 62  # used only at aft of front tank
define aft_tank_skirt 76  # used at front & aft of aft tank
define aft_skirt 103.
define longeron_pos 0.18
define  fuse_center_bay  1013.  -  $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
```

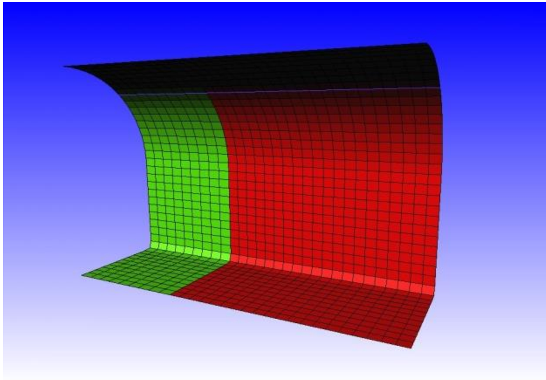
The constant cross-section portion of the fuselage is defined in seven sections. These cuts were made to force the creation of nodes at axial stations that will later have bulkheads. Each fuselage portion also has a longeron created at 18 percent around the curve. The longeron runs the length of the 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 10
  nodes_circ $circnodes
  components_axial 1
  components_circ 1
  mark element OML
object frame longeron1
  count 1
  align axial
  position 0.18
```

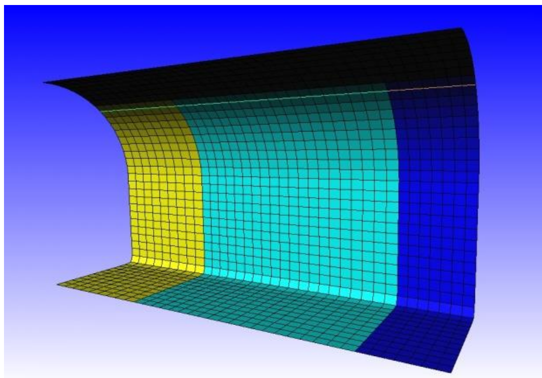


```
object section fuselage1.5
  curve1 body
  curve2 body
  c1_xscale $fusescale
  c1_yscale $fusescale
  c2_xscale $fusescale
  c2_yscale $fusescale
  length $mid_bulk
```

```
  nodes_axial 21
  nodes_circ $circnodes
  components_axial 1
  components_circ 1
  mark element OML
object frame longeron1
  count 1
  align axial
  position 0.18
```



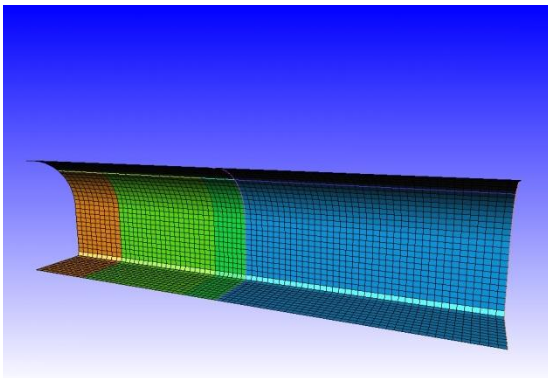
```
# Along fwd tank aft dome
object section fuselage2
  length $fwd_tank_skirt
  nodes_axial 8
  nodes_circ $circnodes
  components_axial 1
  mark element OML
object frame longeron2
  count 1
  align axial
  position 0.18
```



```
define plb_start @transz
```

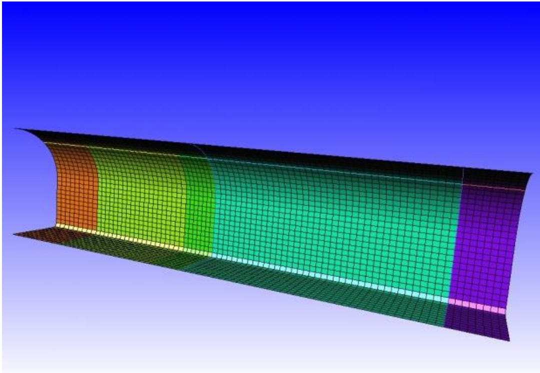
In this case, no dummy `move` command is necessary to force `@transz` to have the desired value; the longeron object definition caused the generation of the “fuselage2” object and the updating of the `@transz` system variable. Note that the selection of `nodes_axial` as a multiple of three plus one allows the later exact positioning and stitching of the payload support bulkheads at 1/3 and 2/3 of the payload bay length.

```
# Payload Bay fuselage
object section fuselage_center_bay
  length $fuse_center_bay
  nodes_axial 40
  nodes_circ $circnodes
  components_axial 1
  mark element OML
object frame longeron3
  count 1
  align axial
  position 0.18
object frame forward pl ring
  count 1
  align circ
  position 0.0
object frame aft pl ring
  count 1
  align circ
  position 1.0
```

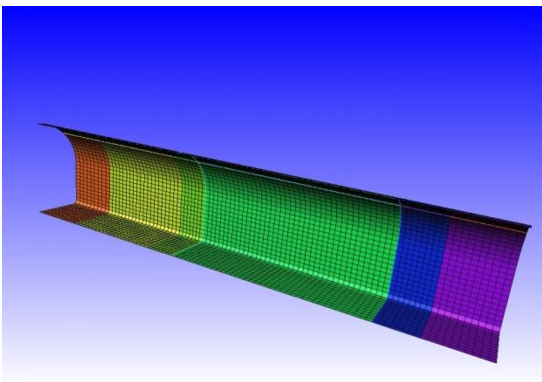


```
# Fuselage along Aft tank fwd skirt
object section fuselage4
  length $aft_tank_skirt
  nodes_axial 9
  nodes_circ $circnodes
  components_axial 1
  mark element OML
object frame longeron4
  count 1
```

```
  align axial
  position 0.18
```

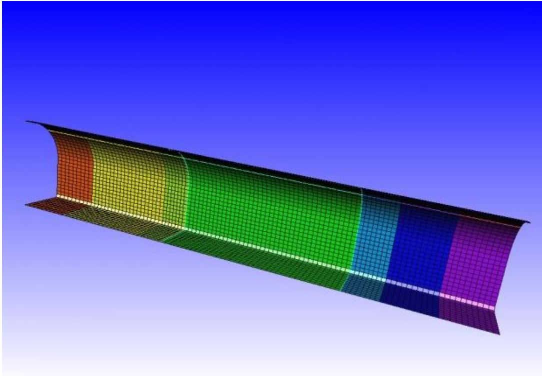


```
# Fuselage along Aft tank barrel
object section fuselage5
  length $aft_tank + 64
  nodes_axial 11
  nodes_circ $circnodes
  components_axial 1
  mark element OML
object frame longeron5
  count 1
  align axial
  position $longeron_pos
```



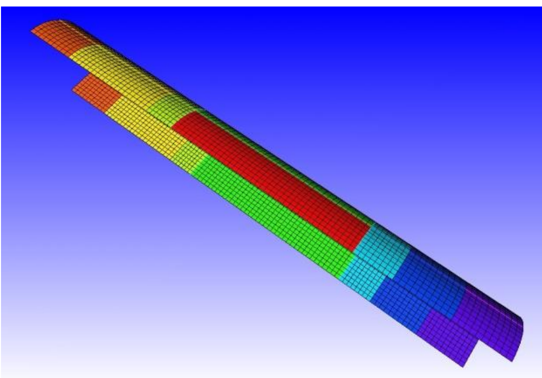
```
# Fuselage along Aft tank aft skirt
object section fuselage6
  length $aft_skirt
  nodes_axial 11
  nodes_circ $circnodes
  components_axial 1
  mark element OML
object frame longeron6
  count 1
```

```
  align axial
  position 0.18
define fuseend @transz + $noseend
```



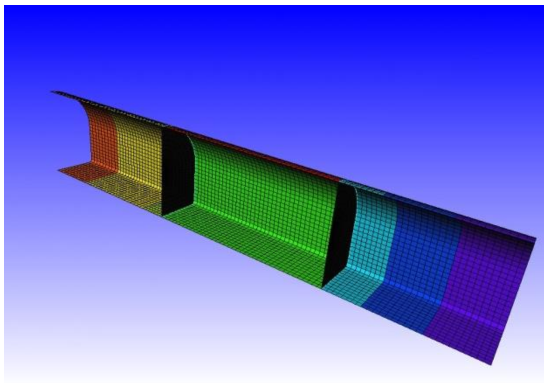
The next step is to add some detail to the payload bay. First, some dimensions are computed based on the previously defined variables. Then the region command is used to modify the physical property assignment of elements along the upper section of fuselage object "fuselage3." These updated elements represent the payload bay doors.

```
# =================== Payload bay =======================
define plb_length $fuse_center_bay
define plb_half $plb_length / 2
define plb_third $plb_length / 3
define plb_center $plb_start + $plb_half
region
  iadd box 0. 102. $plb_center 130. 130. $plb_length
  pprem fuselage2
  pprem fuselage4
  setpp payload doors
```



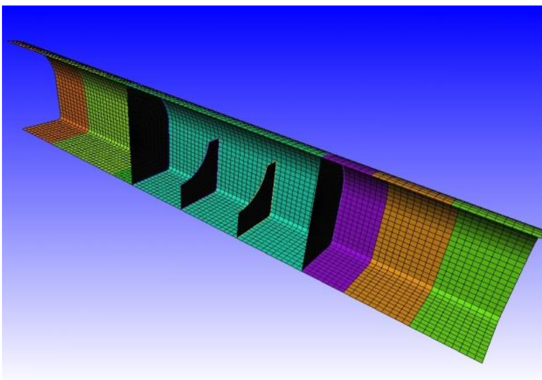
Then full bulkheads are added at the front and rear of the payload bay and partial, support, bulkheads are added at the 1/3 and 2/3 positions in the bay.

```
object dome payload bay fwd bulkhead
  curve1 body
  c1_xscale $fusescale
  c1_yscale $fusescale
  taper bulk
  transz $fwd_tank + $fwd_tank_skirt
  transy 0.0
  transx 0.0
  nodes_circ $circnodes
  components_axial 1
object dome payload bay aft bulkhead
  curve1 body
  taper bulk
  relz $plb_length
  transy 0.0
  transx 0.0
  components_axial 1
```



```
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 9
  nodes_circ $circnodes * 0.6 + 1
object frame fwd plb support frame
  count 1
  align axial
  position 0.0
object frame fwd plb support frame
  count 2
```

```
    align circ
  object section payload bay aft support
    curve1 plb1
    curve2 plb2
    length 0.0
    relz $plb_third
    components_axial 1
    components_circ 1
    nodes_axial 9
    nodes_circ $circnodes * 0.6 + 1
  object frame aft plb support frame
    count 1
    align axial
    position 0.0
  object frame aft plb support frame
    count 2
    align circ
```



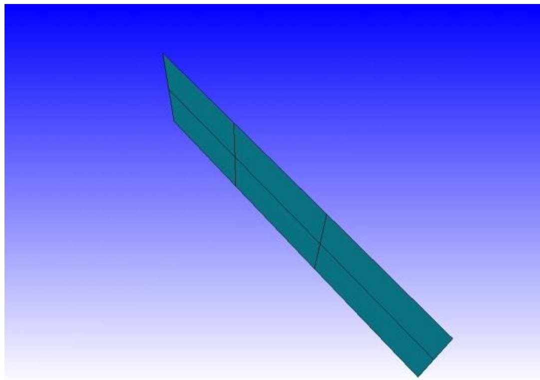
Finally, the completed fuselage component is moved so that it is immediately aft of the nose using the previously created "noseend" variable. A VRML output file of the component is created. Then the full stack is moved onto *Loft's* internal clipboard and a new stack is started.

```
  move
    transz $noseend
  write vrml orb-fuselage.wrl
  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. The first line defining the variable "tan75" uses an externally calculated value based on the 75.179 degree leading edge sweep angle of the strake (the %tan operation could also have been used). Note

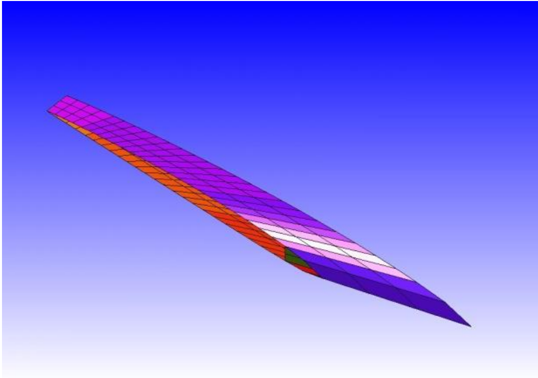
the extensive use of the `gen_XX` flags and the use of the `mark` command to mark only the wing skin as "OML."

```
# ======================== Wing ==========================
define tan75 3.77924
define spar1 10.
define spar2 36.
define spar3 82.
# First generate the spar we want to keep
object wing strake spar
  chord 498.196
  span 31.
  taper 377.777 / @wing.chord
  sweep 75.179
  rootnaca 2407
  tipnaca 2408
  sparpos $spar1
  ribpos reset
  notip 1
  meshchord 0.06
  meshspan 0.125
  meshthick 0.1
  transz 712.65
  relx 103
  rely -95
  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 elements
object wing strake
  sparpos reset
```

```
    sparpos $spar1
    sparpos $spar2
    sparpos $spar3
    notip 1
    gen_spars off
    mark element OML
    mark element wing
```



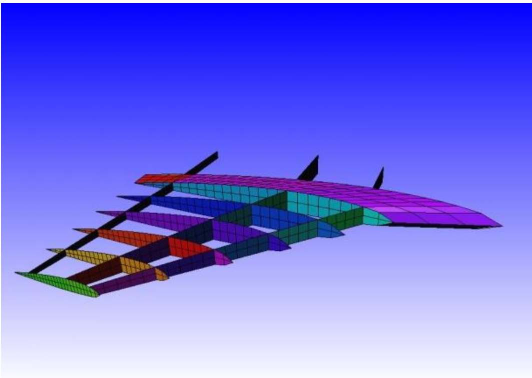
```
  define strakespan @wing.span
```

No dummy command is required to capture the system variable update here because the default dimension variables are updated immediately on specification.

The main wing is also specified as two objects. The reason for this is to apply the “OML” mark to only the wing skin. Note the extensive use of system variables based on the strake dimensions. This allows the user to change a dimension in only one location and have the wing still stitch together properly.

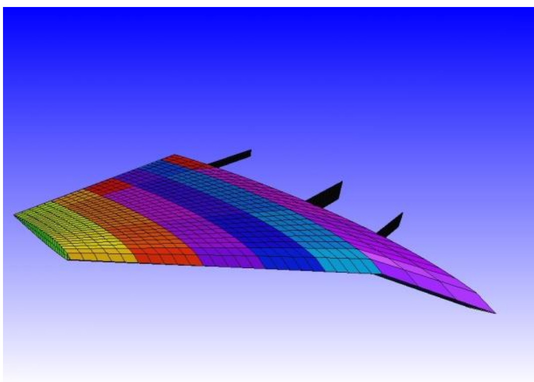
```
  object wing mainwing ribs spars
    chord @wing.chord * @wing.taper
    span 233.
    meshchord @wing.mesh_chord / @wing.taper
    taper 113.235 / @wing.chord
    sweep 45.854
    rootnaca 2408
    tipnaca 2313
    ribpos reset
    ribpos 20.
    ribpos 40.
    ribpos 60.
    ribpos 80.
    relx $strakespan
    relz $tan75 * $strakespan
    wingbox 103 + $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
```

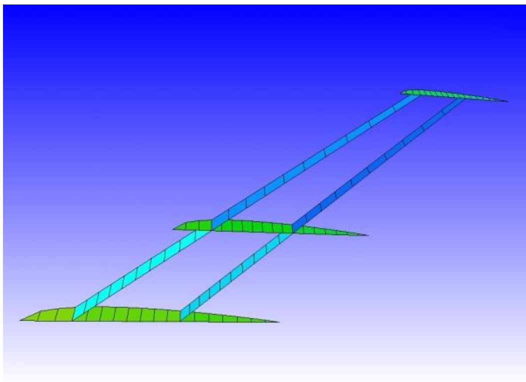


```
write vrml orb-wing.wrl
store mainwing
list stacks
```

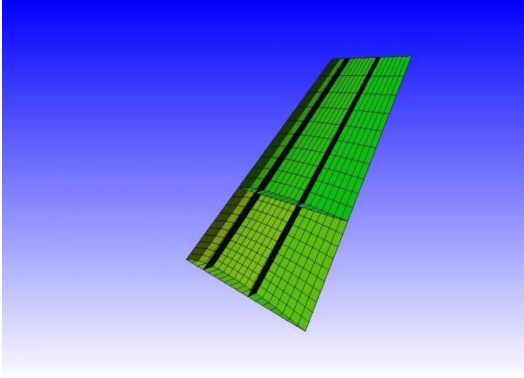
The `list stacks` debug command lists all of the stacks that have been stored on the internal clipboard. Next, the tail will be created as a new stack. As with the main wing components, it is created as two objects so that the skin can be marked as "OML."

```
# ===================== Tail ========================
object wing tail stiffeners
```

```
    chord 260.337
    span 281.5
    taper 77.955 / @wing.chord
    sweep 47.
    rootnaca 0613
    tipnaca 0618
    sparpos reset
    sparpos 19
    sparpos 60
    halfwing bottom
    ribpos reset
    ribpos 50
    wingbox 0.
    meshchord 0.08
    meshspan 0.08
    meshthick 0.02
    transz $fuseend - @wing.chord
    rely 102.
    transx 0
    rotz 90
    gen_up_skin off
    gen_low_skin off
    mark element tail
```



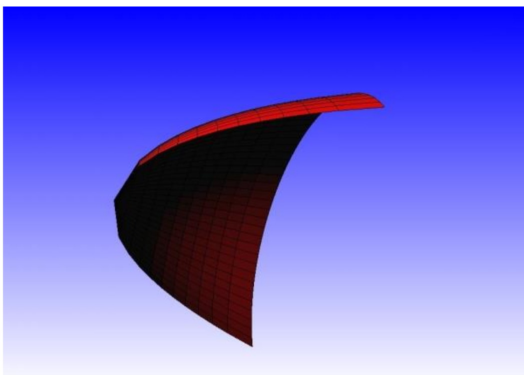
```
  object wing tail skin
    halfwing bottom
    gen_ribs off
    gen_spars off
    gen_up_skin on
    gen_low_skin on
    mark element OML
    mark element tail
```



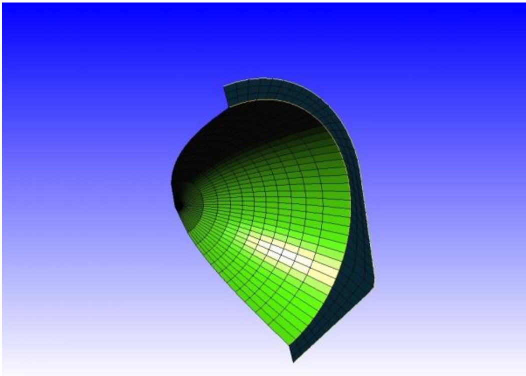
```
write vrml orb-tail.wrl
store tail
list stacks
```

After the tail object is written out as a VRML file and moved onto the internal clipboard, again the list of stored stacks is requested. 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. Note also that the tank walls are all given the mark "LH2." This mark will be used later to extract just these elements from the full model.

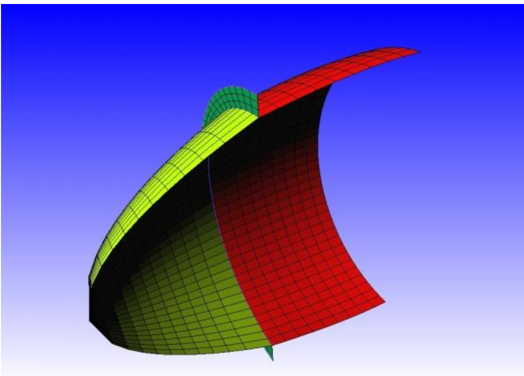
```
# ======================== Fwd Tank ======================
object dome fwd tank fwd dome
  curve1 dome50
  c1_xscale 1.
  c1_yscale 1.
  length -1 * $half_lh2_nose
  transx 0.0
  transy 0.0
  zdist 0.7
  transz $noseend - 100.
  nodes_axial 12
  nodes_circ $circnodes
  components_axial 1
  components_circ 1
  taper para
  mark element LH2
```



```
object section fwd tank fwd bulk
  curve2 forebullet
  length 0.0
  components_axial 1
  nodes_axial 4
  mark element bulk
object frame fwd fwd ring frame
  count 2
```

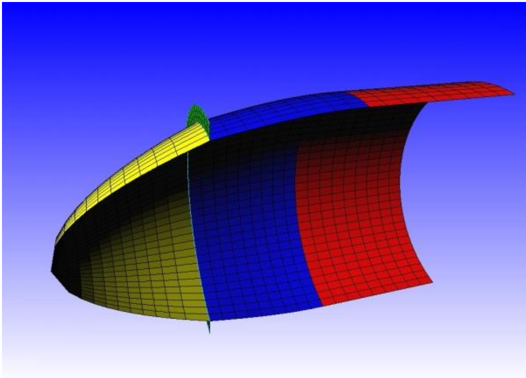


```
object section fwd tank dome2
  curve1 dome50
  curve2 sc
  length $half_lh2_nose
  c1_xscale 1.
  c1_yscale 1.
  c2_xscale 96.
  c2_yscale 96.
  nodes_axial 10
  components_axial 1
  taper cosine 0.5
  mark element LH2
```

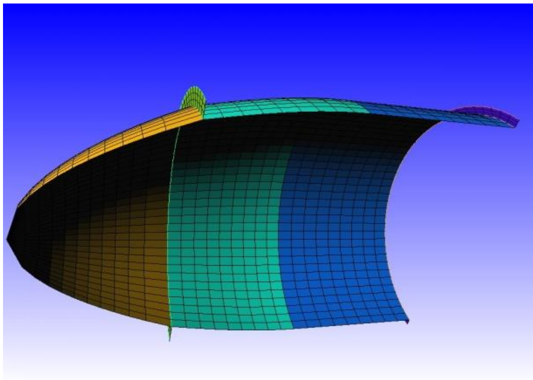


```
object section fwd tank barrel pt 1
  length $mid_bulk - $half_lh2_nose
  nodes_axial 10
  components_axial 1
```

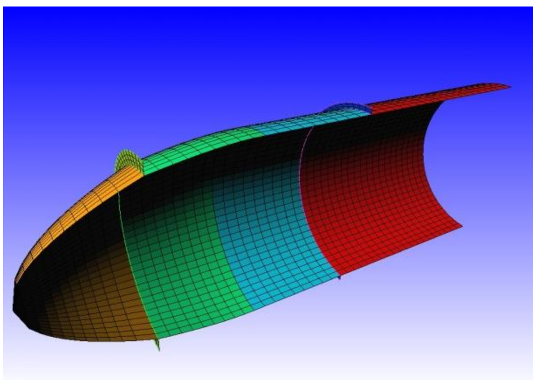
```
  mark element LH2
```



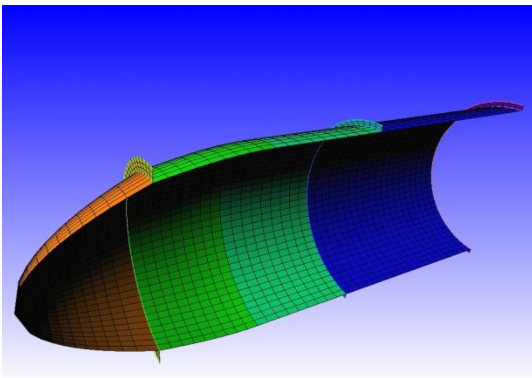
```
object section fwd tank mid bulk
  curve1 body
  curve2 sc
  c1_xscale $fusescale
  c1_yscale $fusescale
  length 0.0
  components_axial 1
  nodes_axial 4
  mark element bulk
object frame fwd mid ring frame
  count 2
```



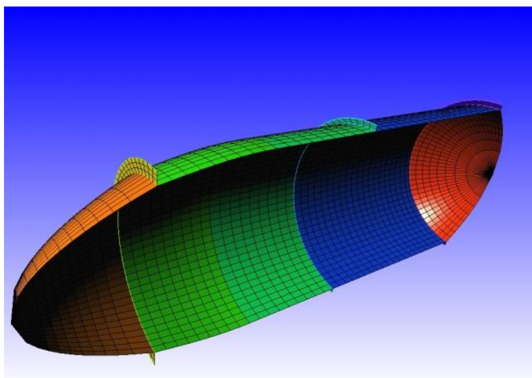
```
object section fwd tank barrel pt 2
  length $mid_bulk
  nodes_axial 21
  components_axial 1
  mark element LH2
```



```
object section fwd tank aft bulk
  curve1 body
  curve2 sc
  c1_xscale $fusescale
  c1_yscale $fusescale
  length 0.0
  components_axial 1
  nodes_axial 4
  mark element bulk
object frame fwd aft ring frame
  count 2
```



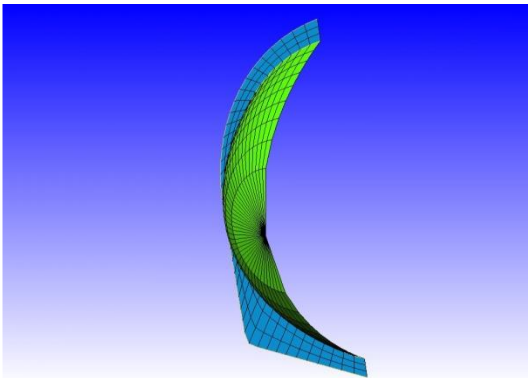
```
object dome fwd tank aft dome
  length 50
  nodes_axial 9
  components_axial 1
  mark element LH2
```



```
write vrml orb-lh2.wrl
store fwd_tank
```

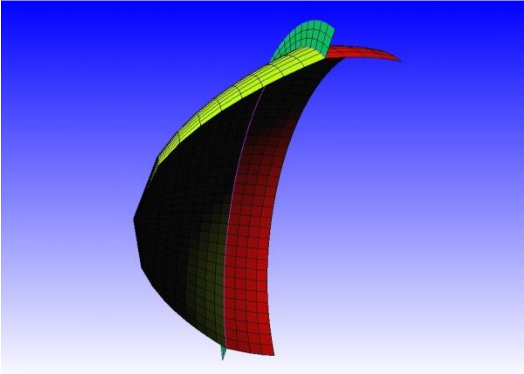
The aft tank is built in a similar process to the forward tank. It is shorter but still has mid-dome bulkheads like on the front of the forward tank. The lofted curve to connect to the dome is defined here rather than at the top of the input file; it could be moved to the top of the file if desired.

```
# ====================== Aft Tank =========================
define aft_dome 96
define aft_support $aft_dome / 3.
curve lofted aftdome
  curve1 sc
  station 1 / 3
  taper elli
  c1_xscale 96.
  c1_yscale 96.
object dome aft tank fwd dome
  curve1 aftdome
  length $aft_support - $aft_dome
  c1_xscale 1.
  c1_yscale 1.
  nodes_axial 10
  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 4
  mark element bulk
object frame fwd aft ring frame
  count 2
```

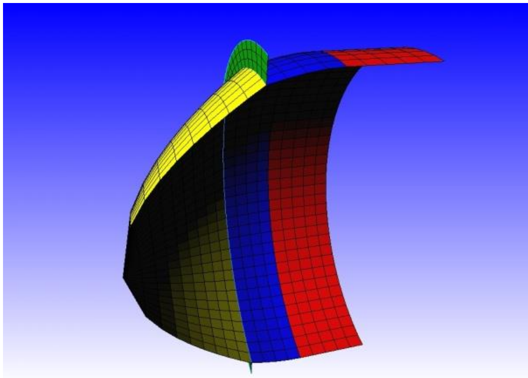


```
object section aft tank fwd curve
```

```
  curve2 sc
  c2_xscale 96.
  c2_yscale 96.
  length $aft_support
  taper cosine 0.5
  mark element LOX
  components_axial 1
  nodes_axial 5
```

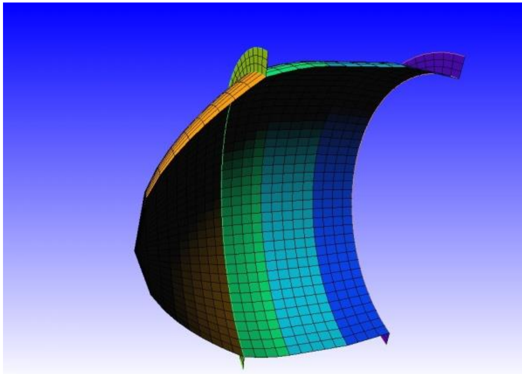


```
object section aft tank barrel
  curve1 sc
  length $aft_tank
  nodes_axial 6
  components_axial 1
  mark element LOX
```

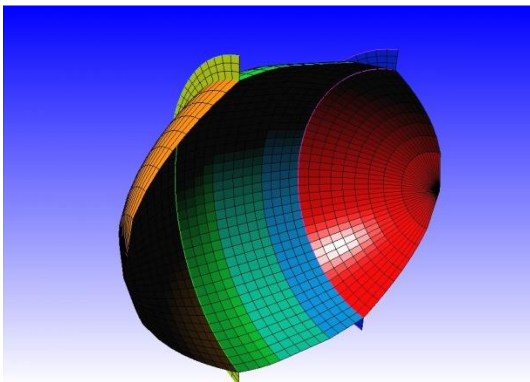


```
object section aft tank aft curve
  curve2 aftdome
  c2_xscale 1.
  c2_yscale 1.
  length $aft_support
  taper power 1.0
  mark element LOX
  components_axial 1
  nodes_axial 5
object section aft tank aft bulk
```

```
    curve1 body
    curve2 aftdome
    c1_xscale $fusescale
    c1_yscale $fusescale
    c2_xscale 1.
    c2_yscale 1.
    length 0.0
    components_axial 1
    nodes_axial 4
    mark element bulk
  object frame aft aft ring frame
    count 2
```



```
  object dome aft tank aft dome
    curve1 aftdome
    length $aft_dome - $aft_support
    c1_xscale 1.
    c1_yscale 1.
    nodes_axial 10
    components_axial 1
    taper para
    mark element LOX
```

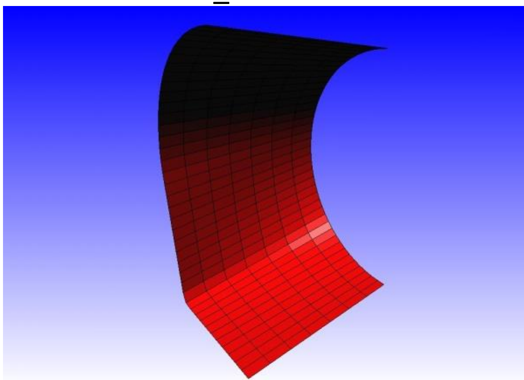


The position of the aft tank is computed from five previously saved lengths. The definition should all be on one line in the actual input file, not wrapped as it is in this manual.

```
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
```

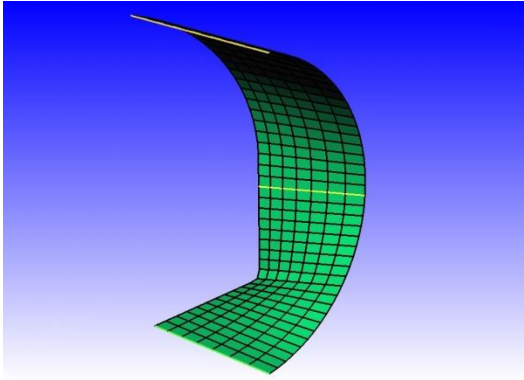
The next object created is a notional thrust structure. It makes extensive use of stiffeners created with Frame and DFrame objects. The first piece created accomplishes the transition from the half-loaf-of-bread "body" shape to a semi-circle.

```
# ================= Thrust structure =====================
object section thrust cone
     curve1 body
     curve2 sc
     c1_xscale $fusescale
     c1_yscale $fusescale
     c2_xscale 80.
     c2_yscale 80.
     length $aft_skirt + 10.
     components_axial 1
     components_circ 1
     nodes_circ $circnodes
     nodes_axial 8
```

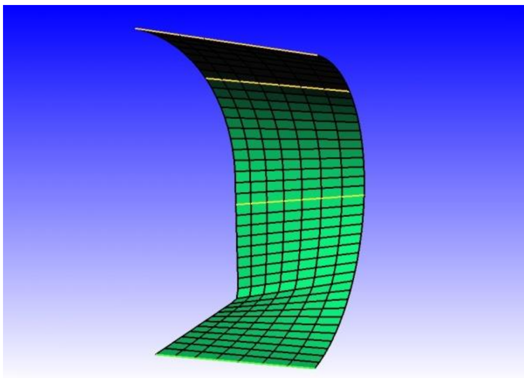


Five axial stiffeners are created. The first three (at 0, 50, and 100 percent of the circumference) are created as one object. Then, two individual axial stiffeners are added, one at the `$longeron_pos` position (18 percent) and one at 75 percent.

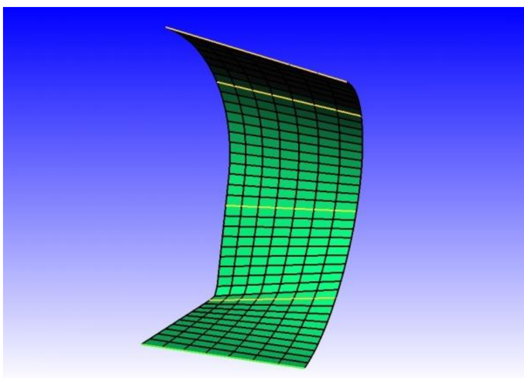
```
object frame thrust stiffeners
  count 3
  align axial
```



```
object frame thrust stiffeners
  count 1
  position $longeron_pos
  align axial
```

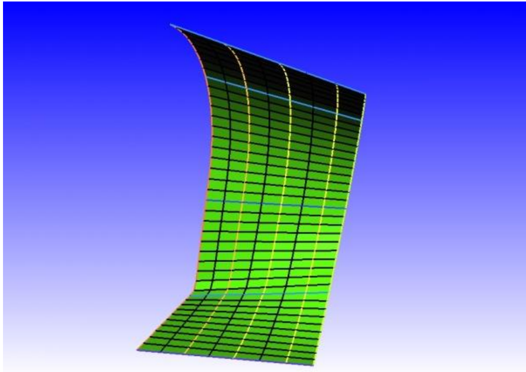


```
object frame thrust stiffeners
  count 1
  position 0.75
  align axial
```



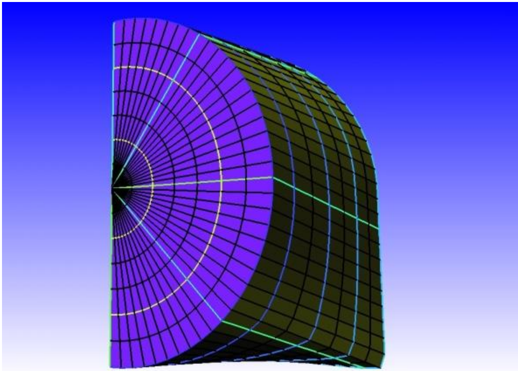
Five circumferential stiffeners are added:

```
object frame thrust cone rings
  count 5
  align circ
```



A circular flat plate is added with similar stiffeners:

```
object dome thrust plane
  taper bulk
  length 0.0
  components_axial 1
  nodes_axial 8
object dframe thrust rings
  align circ
  count 1
  position 0.2
object dframe thrust rings
  align circ
  count 1
  position 0.7
object dframe thrust diags
  align axial
  count 3
object dframe thrust diags
  align axial
  position $longeron_pos
  count 1
object dframe thrust diags
  align axial
  position 0.75
  count 1
```



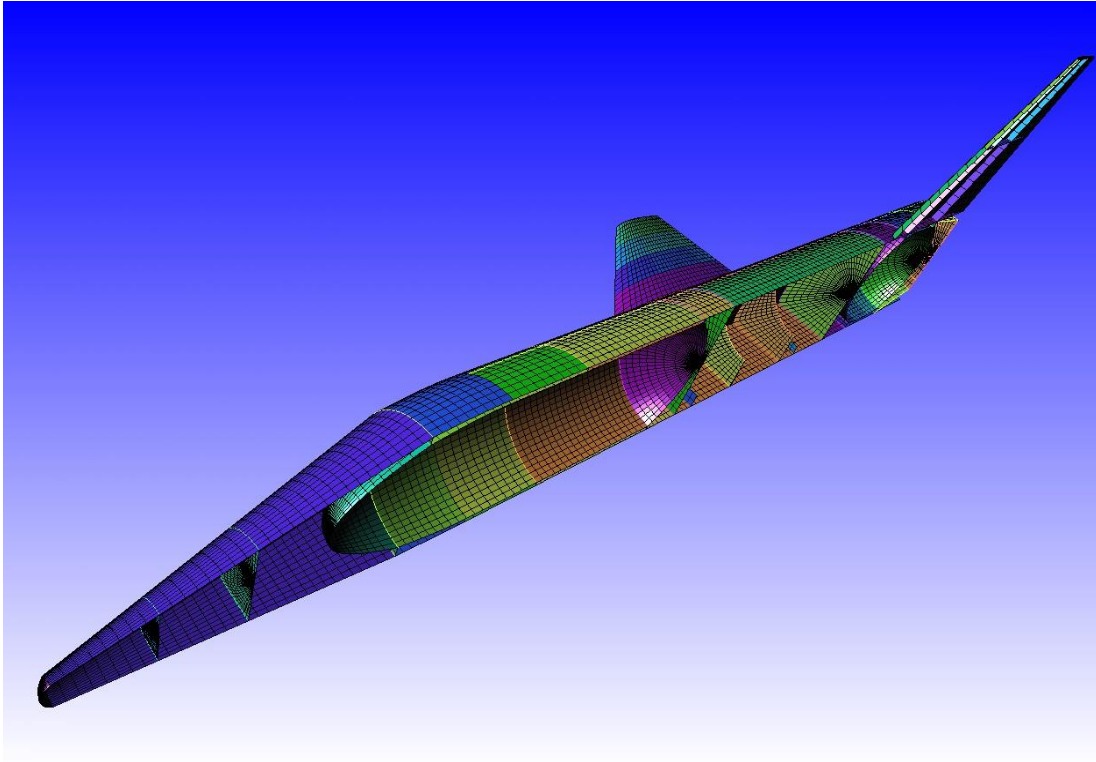
```
move
  transz $aft_tank_end
write vrml orb-thrust.wrl
store thrust
```

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.

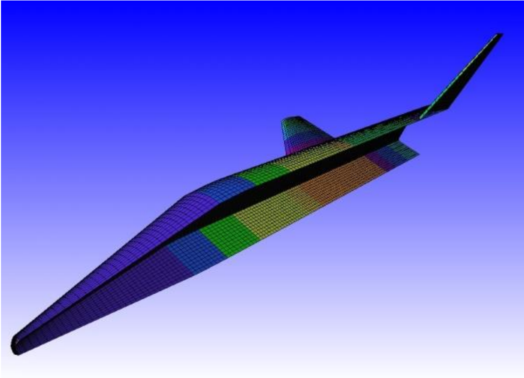
Note that prior to actual analysis with the model, the wing and tail need to be manually stitched to the fuselage. A short discussion of this stitching will be provided after the end of the input file discussion.

```
# ===================== 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
# =================== Write models ====================
vrml rainbow
write vrml tsto-2009-2B.wrl
write nastran tsto-2009-2b.bdf
```

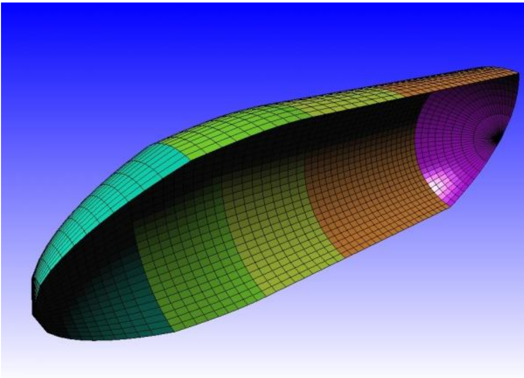


Next, the region mode is used to write out various partial versions of the model. These partial models retain the node, element, and property numbering of the full model. They are used for mapping of external aerodynamic loads (to the “OML” sub-model) and internal tank loads (to the “LH2” and “LOX” sub-models). Note the selection of elements based on the labels assigned with the `mark` command during model creation.

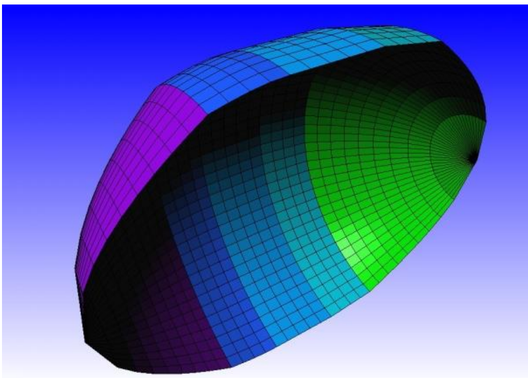
```
# ============ Models for mapping & analysis =============
region
  mkadd OML
  filenew tsto2009-2b-OML.wrl
  format vrml
  rwrite
  filenew tsto2009-2b-OML.bdf
  format nastran
  rwrite
```



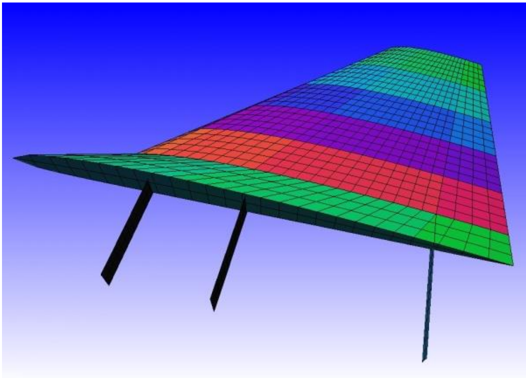
```
region
  mkadd LH2
  filenew tsto2009-2b-LH2.wrl
  format vrml
  rwrite
  filenew tsto2009-2b-LH2.bdf
  format nastran
  rwrite
```



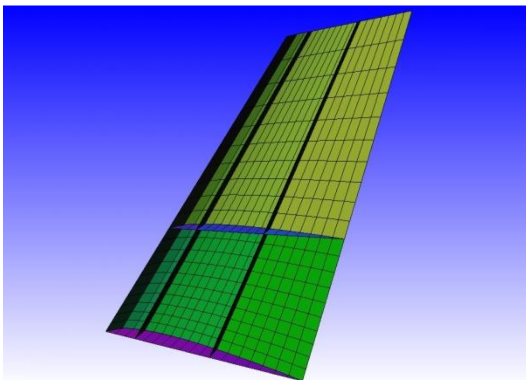
```
region
  mkadd LOX
  filenew tsto2009-2b-LOX.wrl
  format vrml
  rwrite
  filenew tsto2009-2b-LOX.bdf
  format nastran
  rwrite
```



```
region
  mkadd wing
  filenew tsto2009-2b-wing.wrl
  format vrml
  rwrite
  filenew tsto2009-2b-wing.bdf
  format nastran
  rwrite
```



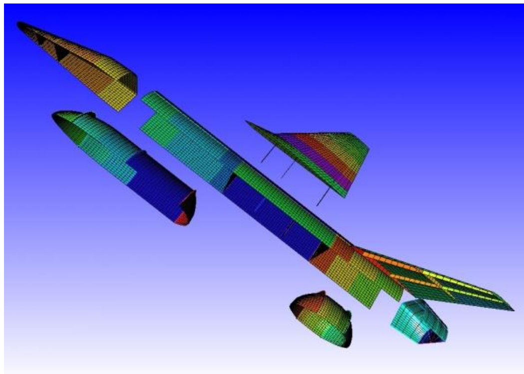
```
region
  mkadd tail
  filenew tsto2009-2b-tail.wrl
  format vrml
  rwrite
  filenew tsto2009-2b-tail.bdf
  format nastran
  rwrite
```



Finally, an expanded and a mirrored version of the model are created for use in slides and presentations.

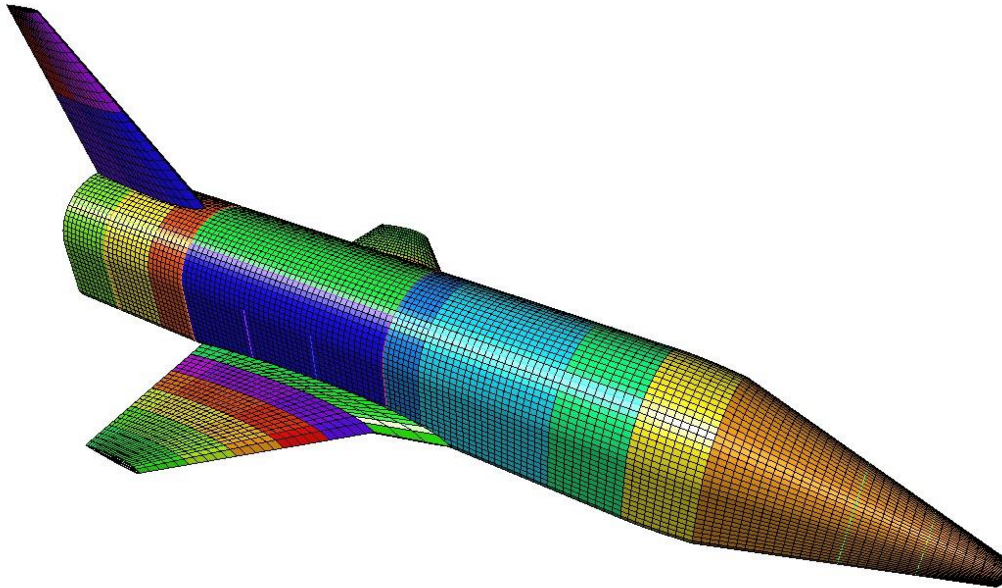
```
# ============== Expanded model for figures ===============
new
recall nose
move
  transz -100
```

```
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 tsto2-2009-2b-exp.wrl
write nastran tsto2-2009-2b-exp.bdf
```



```
# ================== Mirrored model =======================
new
recall nose
recall fuselage
recall mainwing
recall thrust
recall tail
store OML
recall OML
move
  scalex -1
  flip
```

```
recall OML
write vrml tsto2-2009-2b-mirrored.wrl
end
```



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. One way to do this is to load the three component models into a commercial modeling package such as Patran or FEMAP and identify the nodes that we wish to connect.

In this case, the nodes on the wing carry through spars at the wing root and at the centerline need to be connected by rigid elements to the nearest nodes on the fuselage structure, where there are prepositioned stiffeners. Similarly, the spar nodes at the tail root are connected to the aft tank bulkhead ring frames.

To test that adequate stitching has been added, start by applying symmetric boundary conditions on the centerline nodes of the vehicle. Since X is now the axial direction, Y is lateral, and Z is vertical, these constraints set the Y translation Ty=0, the X rotation Rx=0, and the Z rotation Rz=0. Then select an arbitrary node (such as the nose tip) to hold completely fixed. There is also a line of beam alignment nodes running down the center of the model that are not attached to any element. They can be manually constrained, or NASTRAN can fix them with the AUTOSPC option.

Finally, apply unit force loads to the wing and tail tips and run a static analysis. If the model runs without error and the deflected shapes look reasonable, then the stitching has been successful.