Automated Multidisciplinary Optimizations of Conceptual Rocket Fairings

Ronald S. Smart
Brigham Young University - Provo

Automated Multidisciplinary Optimizations of Conceptual Rocket Fairings

Ronald Scott Smart

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Mechanical Engineering

Brigham Young University

August 2011

ABSTRACT

Automated Multidisciplinary Optimizations
of Conceptual Rocket Fairings

Ronald Scott Smart
Department of Mechanical Engineering
Master of Science

The purpose of this research is to develop and architect a preliminary multidisciplinary design optimization (MDO) tool that creates multiple types of generalized rocket fairing models. These models are sized relative to input geometric models and are analyzed and optimized, taking into account the primary objectives, namely the structural, thermal, and aerodynamic aspects of standard rocket flights. A variety of standard nose cone shapes is used as optimization proof of concept examples, being sized and compared to determine optimal choices based on the input specifications, such as the rocket body geometry and the specified trajectory paths. Any input models can be optimized to their respective best nose cone style or optimized to each of the cone styles individually, depending on the desired constraints.

Two proof of concept example rocket model studies are included with varying sizes and speeds. Both have been optimized using the processes described to provide delineative instances into how results are improved and time saved. This is done by optimizing shape and thickness of the fairings while ascertaining if the remaining length downstream on the designated rocket model remains within specified stress and temperature ranges. The first optimized example exhibits a region of high stress downstream on the rocket body model that champions how these tools can be used to catch weaknesses and improve the overall integrity of a rocket design. The second example demonstrates how more established rocket designs can decrease their weight and drag through optimization of the fairing design.

Keywords: Ronald Scott Smart, fairing, nose cone, computer aided design (CAD), optimization, finite element analysis (FEA), multidisciplinary design optimization (MDO), computation fluid dynamics (CFD)

ACKNOWLEDGMENTS

This thesis is a culmination of the knowledge of those upon whose shoulders I stand. It is more importantly a culmination of the efforts of those who have lifted me up to those shoulders. There are more than can be recognized, but it is to them that I give my thanks, and will here list a key few. I am forever indebted to my parents for giving me the guidance that led me to this success. I am grateful for my wife who has encouraged me to strive forward and to keep my priorities in check. I am appreciative to all those at ATK, both for funding and for making this accomplishment possible. A special thanks to Dr. C. Greg Jensen who helped me to accomplish it. Mostly, I am thankful for the inspiration that comes from God through prayer because without this none of these efforts would have been realized.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

The competing forces in the creation of high performance products in industry, such as rockets, are cost and quality. These two forces have vector directions that can be complete opposites of each other which cause a strain between the demand for speed and agility of the rocket and the demand for short time to market and less manpower. Obtaining the best possible design for a set of performance parameters can be an expensive lifetime pursuit, but obtaining the quick and minimalist cost solution can literally blow up in your face. Minimizing the time and effort for creation of an exceptional design is the struggle in industry that can tear a company in two. This can occur if efforts are not placed on working to better align these two vector forces of cost and quality to create less strain on the company. Aligning these forces supplies a component of momentum to the products that will help the company keep a competitive edge on the market. Getting these forces aligned when creating a rocket is as important as understanding the physics that create and control the rocket's flight trajectory. The methods in this thesis improve the ideal product envelope by making the initial product domain more feasible and therefore more usable. Quality is achieved through quicker more detailed optimization of rocket fairings which can be implemented on many differing rocket designs.

1.1 Problem Statement

This thesis proposes, implements, demonstrates and tests a method that converges on optimal designs for conceptual rocket body models based on the integration of multidisciplinary analyses. Included analyses are the structural, thermal and aerodynamic aspects of any rocket model's trajectory.

Products and jobs that are not deemed as exceptionally difficult or taxing are often said to be "not rocket science" for a reason. The multiple levels of consideration that must be made before acceptable success is achieved are extensive. Handling multifaceted calculations that have extensive interrelations can be a daunting task that can require more than even seasoned engineers can calculate alone. Townsend states "As the application development became more complex with increasing levels of fidelity and numbers of disciplines, the need for applying software engineering practices became evident." (Townsend 2002) The increased levels of fidelity are inherent in all advances made in the field of aerodynamics, and the evident need of software engineering practices is central to this research. A great advantage in the methods of this thesis is in the integration and automation of the software tools which provide for faster and more detailed designs. Increased computing power has led to a shift from purely empirical based studies to designs primarily achieved based on analytical research and testing. These advancements allow time to search the vast space of available design possibilities before even creating a physical model. However, they come with a caution of possible loss in accuracy that is dependent on how efficiently the models truly encapsulate the problem. There are many different forms of analysis, and handling the aspects of all of them simultaneously can be challenging and is a prominent subject of current research. (Alexandrov 2002)

The methods in this thesis are designed to integrate the various related fields, as will be discussed in this chapter, and to provide insight for designers into how best to manage all aspects of the problem. One of the difficulties of this type of integration is getting the information from one software system to another such as from a CAD tool to a flow meshing tool or to a structural meshing tool and from there to their respective solvers (See Chapter 2 for more detail on CAD and meshing). Methods for transferring files can take many forms such as IGES and STEP, which carry geometric information from CAD systems, but are not always accurate or perfectly compatible with the various packages.

By optimizing and integrating the automated tools created for this thesis, multidisciplinary design optimization (MDO) can be performed. Defined, MDO is an engineering discipline that incorporates the use of a collection of tools that cover multiple fields of the design process. MDO is another powerful aspect of this thesis and is more complex to generate than a traditional optimization due to the excess of separate but interrelated disciplines. "Aerospace vehicles generally require input of design variables from a variety of traditional aerospace fields such as aerodynamic, structure, propulsion, performance, cost and trajectory. Traditional optimization methods cannot always be applied as they use variables from one field only. Multidisciplinary techniques are required for this class of design problems." (Roshanian 2006) In engineering and aerodynamic optimization, there is an ever increasing need for the tools to cover more than the aspects of any one engineering discipline. This thesis presents a multidisciplinary design optimization that overcomes the difficulties of dealing with the various interrelated fields.

1.2 Thesis Objective

The objective of this thesis is to show the development and architecture of a preliminary MDO tool that expedites the creation, evaluation and selection of optimal rocket fairing models. These models are to be analyzed and optimized taking into account the primary objectives, namely the structural, thermal, and aerodynamic aspects of standard flights for each rocket used in the implementation. A variety of standard nose cone shapes will be used as optimization proof of concept examples, being sized and compared to determine optimal choices based on the input specifications such as the rocket body geometry, and the specified trajectory path. The optimal convergence will be determined through the integrated analytical consideration of the conflicting objectives based on controllable user input parameters.

A display of a current simple MDO design process used commonly in industry is given in Figure 1-1. Often, as illustrated in this process, the work done by engineers is done for an individual task with no thought of repeatability. The input is manually inserted into a manually built CAD model that is then manually input into a manually created FEA model. This model is then checked for adequacy and it either passes or fails (for more on FEA see chapter 2). When the minimal requirements are finally met, the sufficient design is accepted and the now unnecessary process is scrapped, as indicated by the garbage can. Although there is no reuse, the engineers have learned how to make related models better the next time. The majority of lost time comes in rework and labor delays.

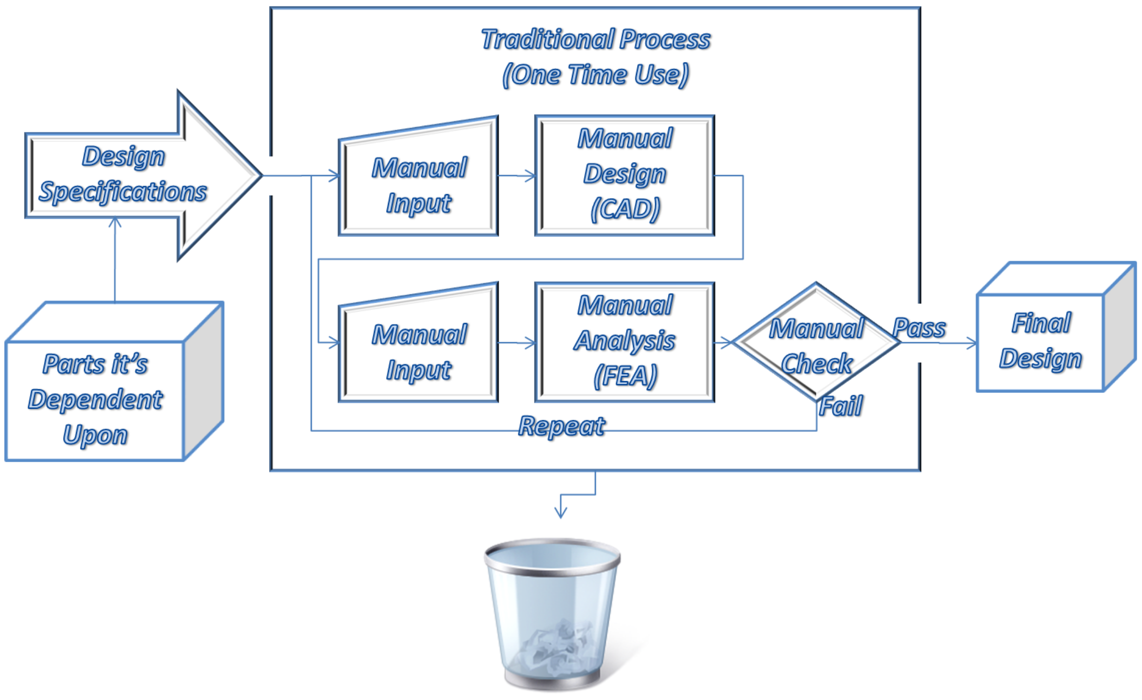


Figure 1-1: Traditional Simple MDO Design Process

1.3 Benefits

An improved process that is becoming more desired in industry is given in Figure 1-2. The benefit in repeatability of the work being done by engineers is being seen, and the reuse of computation models is growing. These reusable models are often called parametric models because of their ability to update based on the key parameters supplied. Instead of a garbage can, a recycle bin is displayed indicating much of the substance of this process can be reused for future iterations. This is a large step towards an automated MDO process.

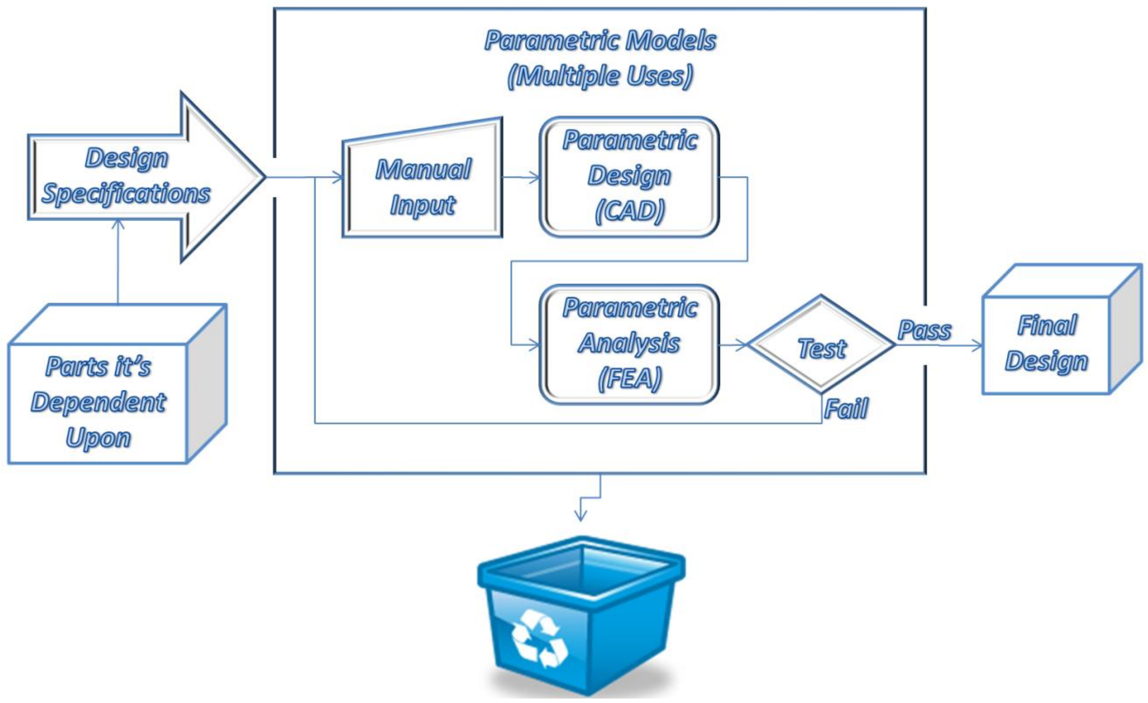


Figure 1-2: Parametric Design Process

The process given in Figure 1-3 builds upon the parametric process and creates some additions specific to the needs of a multidisciplinary analysis of a fairing. A much better communication between the various disciplines is required to allow for a user free, and also more seamless transition. The benefits of the parametric models are still encapsulated, and because more specifics are known about the process, additional improvements can be made that streamline the process, and remove unnecessary down time.

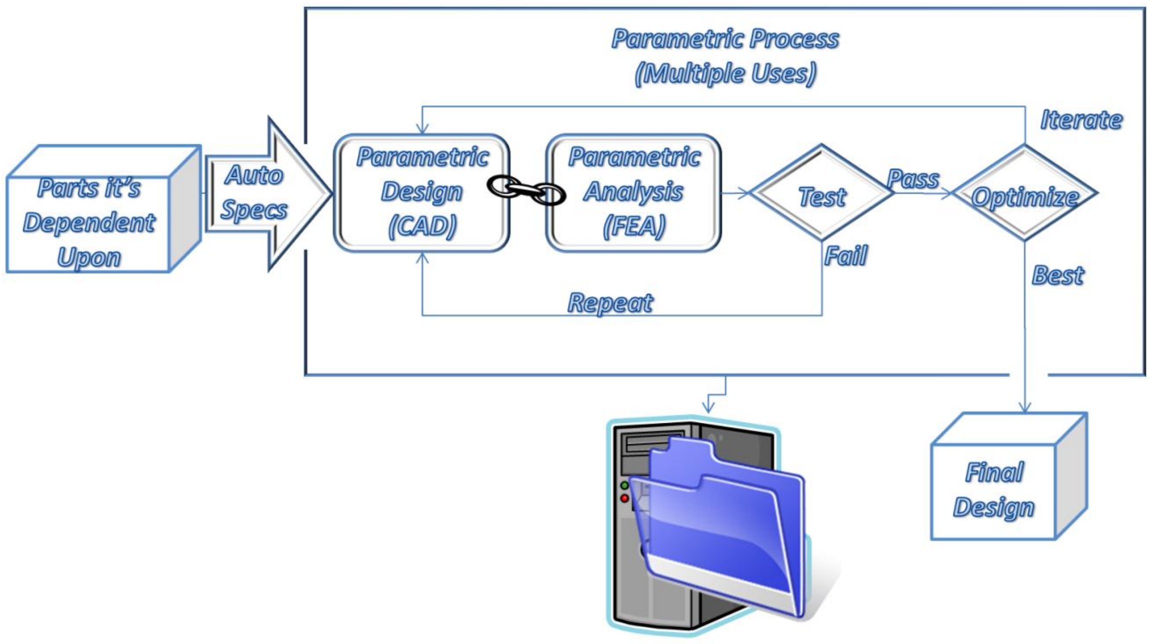


Figure 1-3: Proposed Specific Process

Many of the benefits of this proposed process come from the level of integration between the multiple models. A higher coordination between the CAD and analysis models allows for improved fluidity of the project and a greater allowance for automation. Because of these improvements the process can go a step further and be put into an optimization loop. This enables the process to find better results in a quicker fashion than the trial and error of a person making repeated attempts at improvement. The process is shown being saved on a computer for future use, because the same tool can be used again to optimize the next preliminary rocket model.

1.4 Thesis Outline

Chapter 2 covers related and significant challenges and foundational literature necessary to construct the basis for this research. The foundation of Computer Aided Design (CAD) and of Finite Element Analysis (FEA) is discussed as well as Multidisciplinary Design Optimization (MDO).

Chapter 3 presents the developed method which consists of nine parts; first, the method describes the automation process which outlines the plan for execution of the process. Following this is a description of how the geometry of the rocket will be referenced and described in the Rocket Body Geometry section. The purpose of the outer mold line of the rocket and how it is created is described in the OML creation section. The unique creation of the fairing geometry is described next, followed by the breakdown of the rocket model into discretized pieces is shown as well as how these pieces are managed. There is also a description of the structural geometric model creation, and input into how data is transferred between each program. Next, the Structural FEA is described and then the CFD calculations are given and described in detail.

Following this chapter, a more detailed description of the process and tools used will be given in Chapter 4. Chapter 5 presents a discussion of the results, and Chapter 6 gives conclusions and future recommendations.

2 Literature Review

This chapter contains research and foundational information regarding the background, developments, and issues related to the automated multidisciplinary optimization concepts that are put forth in this thesis. There are many complications with the aerodynamics of supersonic flight that must be overcome that may or may not have been represented adequately in specific cases before. “Many aspects of transitional and turbulent flows are not fully understood…. In the absence of detailed experimental or computational databases to better understand these physical phenomena, we are left with excessive design conservatism and unrefined conceptual designs.” (Martin 2001) This issue can be simply termed as a lack of knowledge, or better stated as a lack of the means of getting knowledge in a timely manner. If there is a way to overcome this issue, and to overcome excessive conservatism, it would open up new paths of possibility through breakdown of the barriers of ignorance.

There are many tools being created that surpass previous limitations and give increased accuracy to solutions which provide the foundation for the work of this thesis. The tools that will be described here are mostly computational in nature, but these were grounded upon the works of others who sought to describe the physical world through equations and laws. Three main categories of discussion competently encapsulate the areas of research and work that have been extended in this thesis: the subject of parametric CAD, the realm of computational analyses (such as FEA and CFD), and the methodologies of rocket body design.

A short description of Computer Aided Design (CAD) is provided here, with its methods for automation and developments upon which this thesis is based. This is followed by a similar foundation in the area of Finite Element Analysis (FEA) and Computational Fluid Dynamics (CFD). Then, some methodologies for integration and automation of rocket body design such as a Multidisciplinary Design Optimization (MDO) are presented here as foundation for use in this research. These sections also mention areas that are marginal or lack robustness and consistency with current design processes.

2.1 Parametric CAD

Computer Aided Design (CAD) systems first started as electronic 2D-drafting devices, the first of which was the Sketchpad System (Sutherland 1963). From these early beginnings, CAD packages have advanced from 2D models, to wire frame models, then to surfaces and finally into the current advanced 3D modeling packages of today. Large steps forward have been taken since the 1960's wherein computers have greatly improved, and the new area of Computer Aided Design (CAD) began to become a reality for more and more designers. For more information regarding the basics of CAD and its formulation, see - (Zeid 2005).

Many CAD tools provide for vast possibilities of improved development and research. Today's research focus in CAD has been in parameterization of models to allow for reuse of designs and for quick adjustments and improvements to be made to current designs. (Taylor 1992) (Dye 2007) (King 2006) Such advancements provide for design improvement possibilities and lead to orchestrated changes being made more quickly and reliably. The benefits of CAD models are essential to the work of this thesis and make such quick changes and iterations possible.

There are some difficulties associated with competent CAD designs. It can take more time to create a generic CAD tool than to make a specific model once. If the design of the product changes dimensionally, parametrics can easily compensate. However, if the design changes topologically, the tool can become outdated before it has justified the time and cost of its creation. Making the tool applicable to extensive changes is a primary goal of this thesis.

2.2 Finite Element Analysis (FEA)

Another advance in computing has been in the area of finite element analysis (FEA). FEA is a technique used to determine the static and dynamic structural and aerodynamic analyses of a part. This is done by discretizing the geometry of a CAD model into small finite elements (each of which is influenced by its neighbors) and determining the strengths and weaknesses of each element and then from this, interpreting the integrity of the whole. Before such advancements, calculations were limited to known physical calculations based on standard geometries. Determining validity of the actual complex design was based ultimately on the performance of what was physically produced after the cost of creation had already been incurred.

The branch of analysis related to aerodynamics that will be used is Computational Fluid Dynamics (CFD). CFD, as its name indicates refers to the calculation of the governing equations of fluid motion on a computer using numerical methods.

CFD is being validated for increasingly challenging configurations. However, the setup and solution time for CFD is an impediment to extensive use early in the design process, where many configuration variations need to be considered. As a result, simpler methods are used during conceptual or preliminary design, even if they neglect important aspects of the flow physics." (Gatzke 1998) Overcoming the conceptual limitations of some preliminary design methods is another contribution of this thesis.

As with CAD, the development has grown from simple 2D into 3D models, many of which still must be approximated due to size and computation time, even using advanced super computers. The vast amount of possibilities makes it impossible to test them all. Many rocket models, as have been seen during industry experience are inherently axisymmetric in design. Consequently, this is how the models are designed for this thesis. The reason for using this type of model is that it keeps modeling and computations simpler and still fits a large quantity of standard rockets.

CFD has been used for variety of simulations and analyses of flow over a rocket. Some key results given by various studies include the drag coefficient of the rocket and pressure and velocity fields in the surrounding air. Many individual cases have been explored which are very similar to those presented in this thesis. They are often painstakingly meshed and then analyzed to determine performance of a design that cannot be easily adjusted. This thesis presents methods to fix this issue through increased adjustability of the models and integration of the tools created.

2.3 Optimization Using Automation and Integration (MDO)

MDO uses optimization techniques to solve design problems involving a number of disciplines. It is also known as multidisciplinary optimization and multidisciplinary system design optimization (MSDO).

"CAD models from conceptual design often follow the "over-the-wall" approach for downstream analyses such as FEA and CFD. The over-the-wall approach consists of four domain experts namely: the designer, airsolid designer, mesh expert, and CFD expert. When the designer has proposed a design it is transferred to the airsolid designer to create the fluid domain, hereafter referred to as an airsolid, then to the mesh expert to create the mesh, and finally the CFD expert to run the CFD analysis. These experts frequently use CAD neutral formats as the model moves from one domain to the next and even duplicate efforts. This over-the-wall approach creates opportunity for error or design escapes that cost a company large amounts of time and money. Since the CAD-to-CFD process is time consuming, CFD has played a limited role in conceptual design, especially where complex models are involved." (King 2006) This transitional issue is part of what MDO can be used to overcome through its implementation.

Another issue that MDO solves is that of optimizing on a specific aspect of the design can lead to disregard for other aspects of the design. As an example, an optimization based purely on CFD results could lead to a design that is not structurally sound. MDO incorporates the influence of all relevant disciplines considered together. An optimum found using this technique is superior to the design found by optimizing based on the individual disciplines one at a time, since it can utilize the interactions between the disciplines. This is currently the standard optimization practice used in the aerospace world.

Disregarding one or more important disciplines during a design study can exploit the inaccuracies in the representation and lead to a false optimal design at an infeasible region. The cost of MDO is that including all disciplines simultaneously significantly increases the complexity of the problem. "A major task in MDO remains the integration of analysis codes, since each discipline's tools require a different description of the design. This may not be onerous for simple systems completely described by a simple set of design variables. For aerodynamic and structural applications however, the design variables determine the form of complicated geometries. As the design variables change during optimization, so too does the geometry. The new geometry is then used to build meshes and extract other properties, such as mass and inertia for evaluation of the design." (Crawford 2004) Creating sound links between CAD, FEA, and CFD is an important aspect of MDO because integration of these tools allows for much quicker and more fluid analytical optimization. (Lee 1999)

In the past, development of structural and aerodynamic vehicles has been performed by separate groups representing the various disciplines; and each group's expertise was of course focused on their particular objective. These conflicting objectives often led to discord among the different groups and also a much longer time for the disjoint objectives to converge upon a feasible, let alone optimal, design. The variety of software packages involved, each focusing on its own area of expertise and supporting only its emphasized results, only serves to increase the problem. "The ceaseless increase in computational hardware capability has only exacerbated this problem, as engineers and researchers struggle with the use and development of computer programs that have become increasingly more complex in order to fully utilize the available computational power." (Alonso 2004)

Various fields have now begun to try and incorporate MDO, including automobile design, naval architecture, electronics, and computer creation. A large number of these applications have appeared in the field of aerospace engineering, mostly involving aircraft, but also in the beginnings of spacecraft design. There is currently no tool that covers the optimization of rocket fairings based on the input of varying CAD models.

In the past, basic 2D models or physical scaled models have been used to give preliminary estimates before the creation of a more finalized product. "Often an aircraft is represented by a simple model during the conceptual and preliminary designs. Because simple models are neither accurate nor complete, optimization of these models could lead to an impractical design." (Samareh 1996) This is of vital concern in this project because the models are to be preliminary but not to be at the expense of accurate results. Therefore, improved detail and assurance of the validity of the models is essential and will be an important part of this research. A balance will be sought to encapsulate only the level of detail required so as to maintain the minimum required computational time.

To recap the process and the acronyms used to be sure they are understood, the objective will be restated here. This thesis will make use of selected rocket CAD models to optimize a fairing design for each using an MDO process. An MDO process generates Parametric CFD models and FEA structural models, which are integrated and analyzed based on the parameters of the MDO, to determine what fairing model is best for the rocket mission being considered.

3 Methods

The methods described in this chapter are representative of specific processes that are unique, but are related, to other fields. Although they have been applied to the specific example of this thesis, many of the methods are applicable in other areas related to apexes or cowls of vehicles, or to the other related geometries of the methods. Most attempts at relating this research to outside applications will be made in the conclusions and future work chapter of this thesis. However, the tools used in the methods will be generalized to demonstrate the independence of the methods from any specific tool. The methods defined in this chapter are: The Automation Process, Rocket Body Geometry, OML Creation, Fairing Geometry Creation, Slice Properties Implementation, Structural Geometry Generation, Structural FEA, CFD Calculations, and Optimization. The detailed implementation of these methods will be discussed in detail in chapter 4.

3.1 Automation Process

In Figure 3-1 the predicted process is shown which was used as the outline for this thesis. This process outlines what the inputs are and what steps must happen to get an optimal rocket fairing model that has been analyzed for flow, stress and temperature. Each block description listed has many execution choices that will greatly influence the speed, accuracy and quality of the result of the process. In the top left, there is an arrow that shows where the process begins.

The additional arrows show the course of the design and analysis of the fairing from initial gathering of the input rocket dimensions, to the addition multiple nose cone designs, to the analyses of the cones made. The flow, structural, and thermal analyses lead to a variety of choices from which to pick the most exceptional design. This served as a foundation for this thesis. Additions have been made to these beginnings that improve the results such as the optimization, and integration of the models.

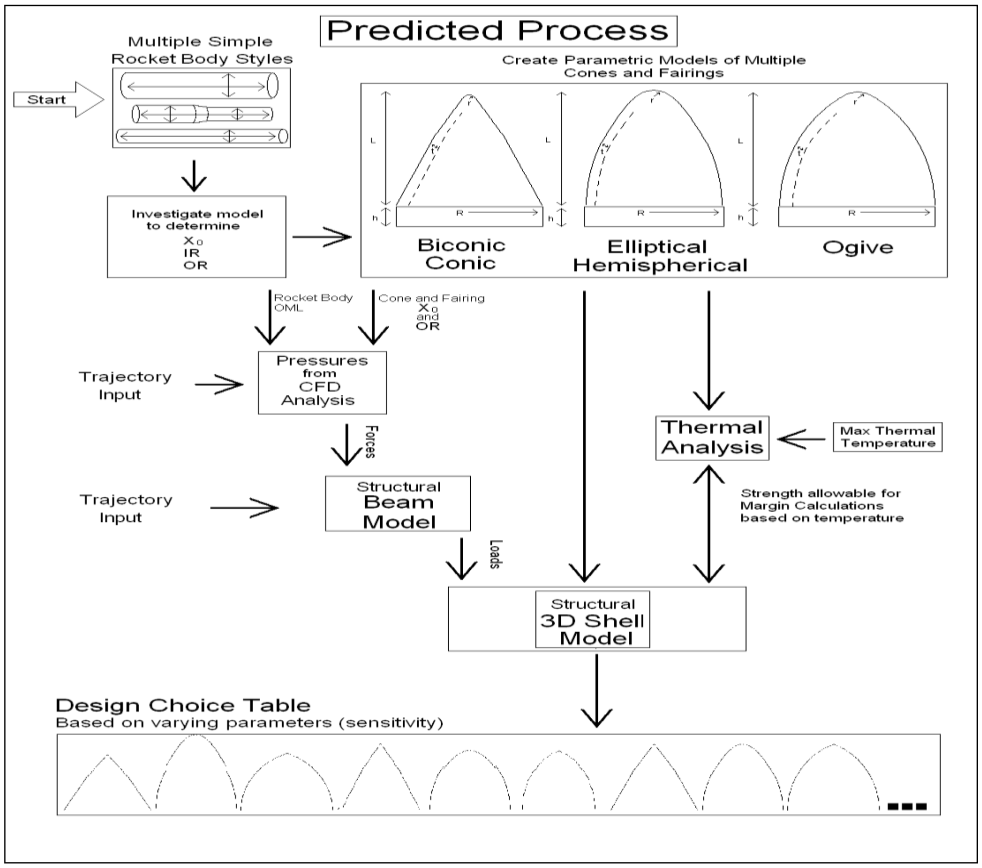

Predicted Process
Multiple Simple Rocket Body Styles
Start
Create Parametric Models of Multiple Cones and Fairings
Investigate model to determine X0 IR OR
Biconic Conic
Elliptical Hemispherical
Ogive
Rocket Body OML
Cone and Fairing X0 and OR
Trajectory Input
Pressures from CFD Analysis
Forces
Thermal Analysis
Max Thermal Temperature
Strength allowable for Margin Calculations based on temperature
Trajectory Input
Structural Beam Model
Loads
Structural 3D Shell Model
Design Choice Table
Based on varying parameters (sensitivity)

Figure 3-1: Overall Process Design

3.2 Rocket Body Geometry

A CAD rocket body model is an input to this process which is the entire model excluding the cone itself that will be generated. It requires solid models of the geometry with material properties assigned to all of the various parts to obtain meaningful results. The interior parts play no role in the CFD, but are included if available in the calculations for the FEA model. The rocket body model is used by a few steps at the beginning of the process. All relevant rocket information is gathered and kept in a data structure so that further access to the model itself is not required during later process steps and computational speed can be maintained.

For future descriptions of the parts, a representation created by Scott et al will be used. (Scott 2009) For more detail regarding this, see this reference. "*A* will represent an assembly-type part (a parent to at least one other part) and *C* will represent a component type part (one having no children). The superscript of *A* or *C* will represent the hierarchical level of the part and the subscript will represent its position relative to its sibling parts, e.g. $A^2{}_1$ is the first child of its parent and is a second level assembly, and $C^3{}_2$ is the second child of its parent and is a third level component. No subscript will be used when referring to the collection of all parts on a certain hierarchical level."(Scott 2009) The various levels are displayed here in Figure 3-2.

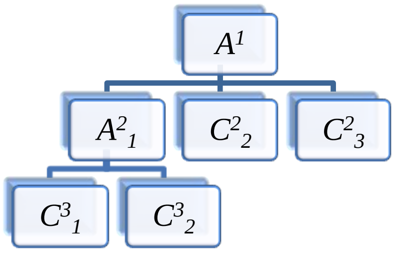


Figure 3-2: Assembly Part Notation as Arranged by Scott et al

“The notation from set theory for boundaries (*bS*) will be modified by a subscript 2 or 3 to distinguish between two-dimensional boundaries b_2 (sketch geometry) and three-dimensional boundaries b_3 (faces and edges of the solid). b_2C^2 represents the two dimensional boundary of a component. b_3C^2 represents the three dimensional boundary of a component. I_{ij} is a unit of the control structure which represents the interface between components i and j. The clearance between components is denoted by ε and the red chain link symbol denotes the sketch constraints between I_{ij} and b_2C^2. A^1 is the top level assembly that contains the control structure.” (Scott 2009) These key terms are displayed here in Figure 3-3 which show a cross section view of aset of parts that are revolved about a central axis. Similar revolutions are presented throughout this thesis by merely showing the cross sectional cut of such parts and indicated by the oranges lines in Figure 3-3.

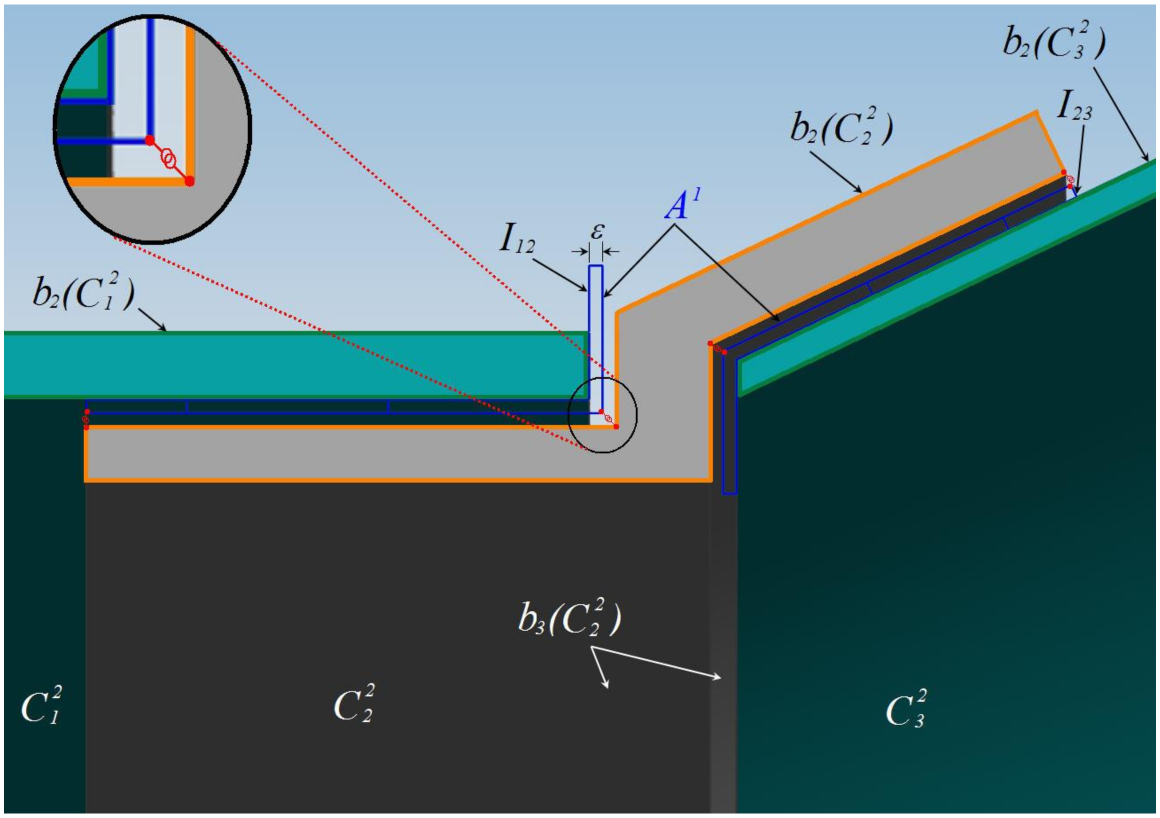


Figure 3-3: Key Terms Used by Scott et al

3.3 OML Creation

For a CFD analysis, an exterior representation of the model is required. For the FEA models, inner and outer radii are needed for calculations. In models such as this, it can be very difficult to find a good way to unite the parts of the model into good representatives of the exterior. The exterior is also termed the Outer Mold Line (OML) and the interior is termed the Inner Mold Line (IML). Below, in the figures from Figure 3-4 to Figure 3-11, the developed process for accomplishing this task is shown.

Figure 3-4 shows a simple example of an initial part cross section layout. The part is displayed as a cross sectional slice in the axial direction which would be revolved about the center axis represented as a dotted line. This example is a simple group of tubes that increase in diameter from left to right with a kind of cap on the top. Combining all the part models into one single part can be done using a few steps that begin with the exterior boundaries of the parts. The lines that make up each boundary will be termed L_k and are defined in Equation 3-1. The use of the $\sum$ symbol in this equation and others that follow is not to be interpreted as the summation of all numbers into one value. The $\sum$ symbol instead represents a collection of all the points or lines that fit the criteria as indicated by the surrounding subscripts. There are a varying total (termed w) of lines in each boundary of a component.

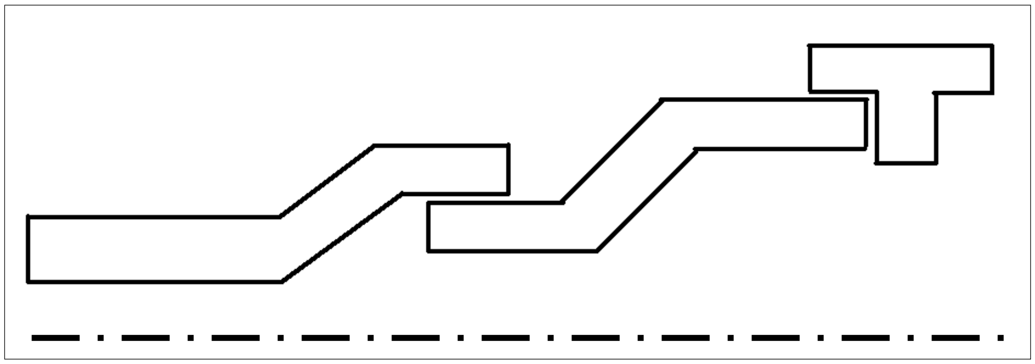


Figure 3-4: Initial Assembly Axial Slice

$$L_{ext\_sum} = \sum_{k=1}^{w} \sum_{i=1}^{n} \sum_{j=1}^{m} L_k b_2 C_i^j \qquad (3\text{-}1)$$

To start the parts are interrogated for all of their edges. In Figure 3-5, the points on each of these corners are labeled in red. Some of these points are in the main areas of interest for uniting the model into a common body.

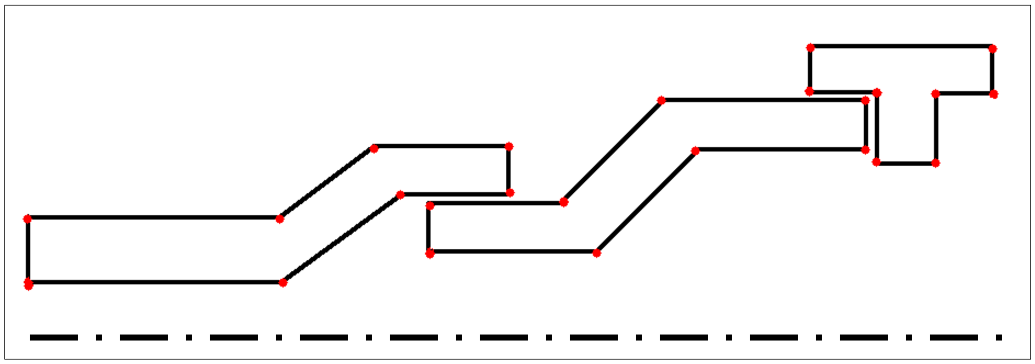


Figure 3-5: Points from Boundary Line Intersections

These points can be defined as the intersections of each exterior line with all of the other exterior lines (also known as the complement *C*) from each part as mathematically described in

Equation 3-2 where Pt_q is the current point. Then in Equation 3-3, Pt_{edges} is shown as the summation of all line intersections.

$$Pt_q = L_q \cap L_q^{\ C} \tag{3-2}$$

$$Pt_{edges} = \sum_{q=1}^{H} Pt_q \tag{3-3}$$

These edge points are used to find intersections with the lines on all the other parts of the model. This is done by investigating vertically relative to each edge for intersections in all of the part models. Vertical lines through each Pt_q titled Lv_q in Equation 3-4 are used to intersect any other lines that they cross in all of the assembly. The resulting additional points that were not already included are labeled here in green in Figure 3-6.

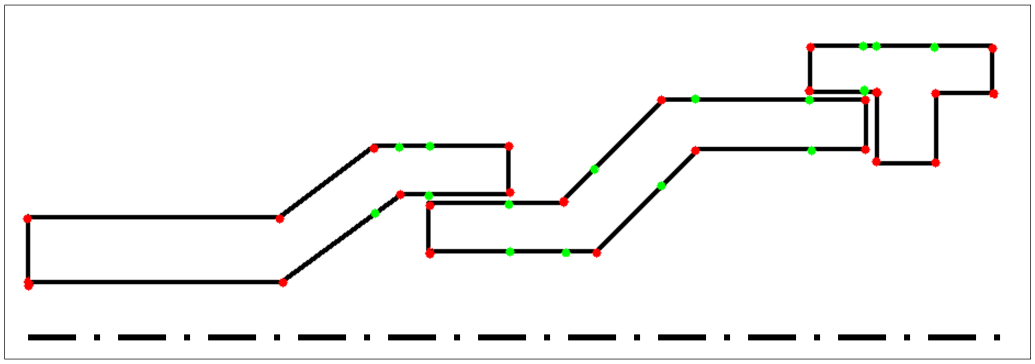


Figure 3-6: Primary Points Based on All Edge Intersections of Parts in Y

Together with the points in red, they make up what are termed the Primary Points (*Pt_{prim}*) as shown in Equation 3-4.

$$Pt_{prim} = \sum_{q=1}^{H} \sum_{k=1}^{w} L_k \cap Lv_q \tag{3-4}$$

Sometimes, there are more than just two points that fall in a vertical line because multiple parts overlap at the location. Picking which two points are the extreme max and min at every location gives you the inner and outer radii of that location. So the vertical maximum and minimum points at each x location of the Primary Points are determined by comparison. These extremes all lie on the exterior or interior of the desired resultant shape. This is mathematically defined here in Equation 3-5 and is represented visually as the circled points in Figure 3-7.

$$Pt_{ext} = \sum_{z=1}^{D} \max\left(\sum_{q=1}^{H} Pt_{zprim} \cap L_q \right) + \sum_{z=1}^{D} \min\left(\sum_{q=1}^{H} Pt_{zprim} \cap L_q \right) \tag{3-5}$$

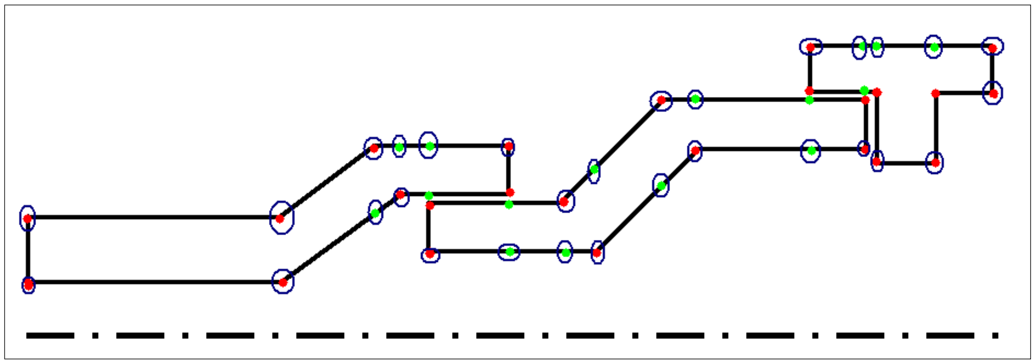


Figure 3-7: Minimum and Maximum of Primary Points in Y on Other Edges and Parts

When these max and min primary points are attached correctly, they make a representation of the model that is close, but still has some areas that are poorly defined as seen in Figure 3-8. This is because some corners that may have looked like they were purely inside the model were really just a surface that was vertically aligned with another outside surface and was incorrectly discarded.

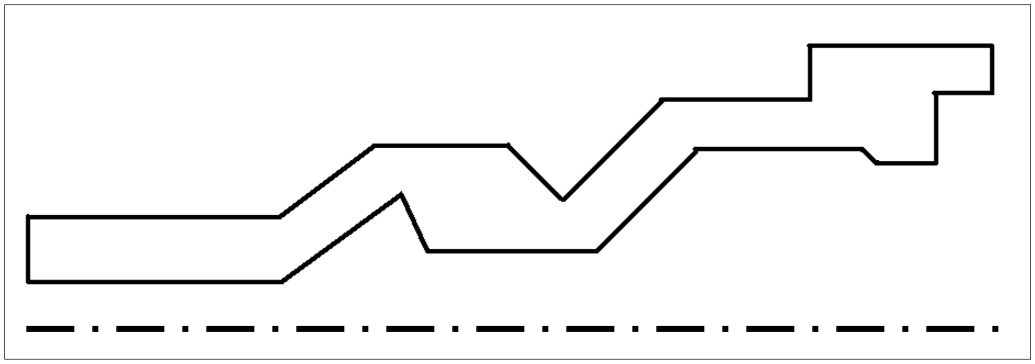


Figure 3-8: Solid Made from Current Max and Min Primary Points

To correct the model and improve these surfaces, additional points are placed on the model shifted slightly to the right and left of each Primary Point and these are added to the set of Primary Points. This shift distance can approach the neighbor points infinitesimally so that it is well within the tolerances of the part, but still unique from the Primary Point. The additional points are shown here in Figure 3-9 as purple vertical hash marks. Picture these purple marks approaching the neighbor points until they are not visibly distinguishable (yet are still unique). These values shifted just left and right in the x-directions are presented as *q*+ and *q*- terms in Equation 3-6.

$$Pt_{advprim} = \sum_{qadv=1}^{Hadv} \sum_{k=1}^{w} L_k \cap Lv_q + L_k \cap Lv_{q+} + L_k \cap Lv_{q-} \tag{3-6}$$

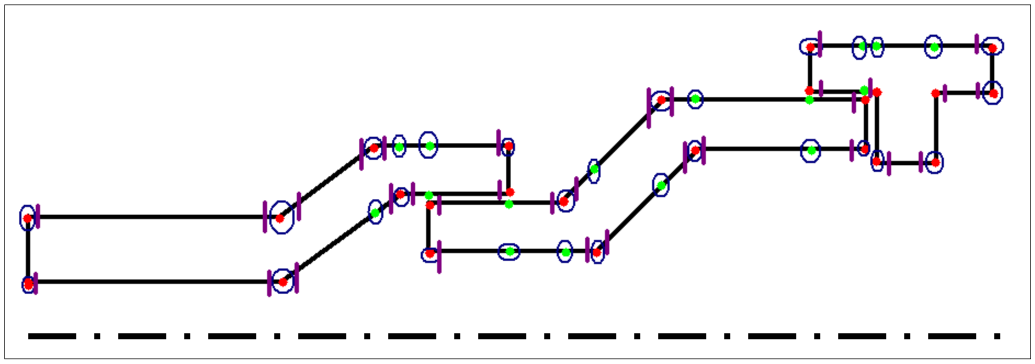


Figure 3-9: Additional Primary Points from Shifting Left and Right of Current Points

These new points shown as purple vertical hash marks can then also be intersected with all the other part edges in the model which add the yellow hash marks as seen here in Figure 3-10. These points are termed the Key Points in this document.

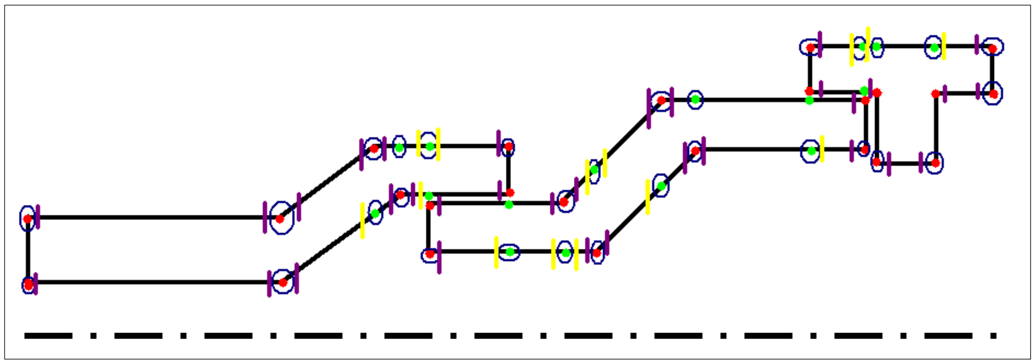


Figure 3-10: Key Points Based on New Set Placed on Other Edges and Parts in Y

$$Pt_{advext} = \sum_{zadv=1}^{Dadv} \max\left(\sum_{qadv=1}^{Hadv} Pt_{zadvprim} \cap L_{qadv} \right) + \sum_{zadv=1}^{Dadv} \min\left(\sum_{qadv=1}^{Hadv} Pt_{zadvprim} \cap L_{qadv} \right) \qquad (3\text{-}7)$$

By taking these max and min exterior key points as was done before, the new solid shown in Figure 3-11 is created by sorting through these points and connecting them in the correct order. Note that for each max point there is always a corresponding min point so it can be easily determined which ones are part of the OML and which are part of the IML surfaces.

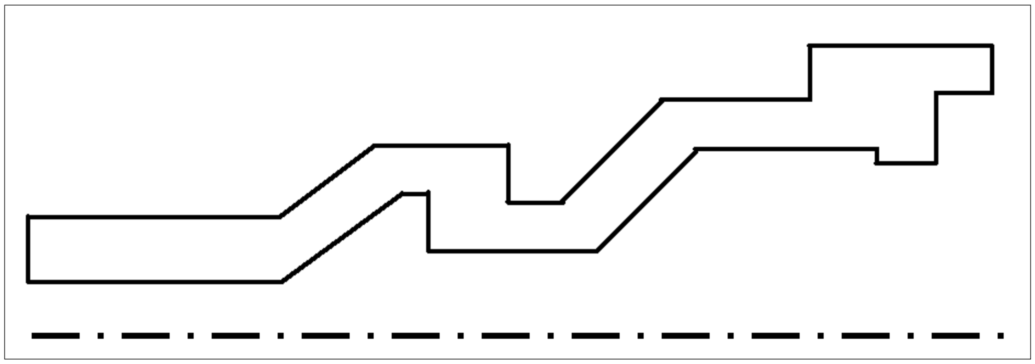


Figure 3-11: Solid Based on New Set of Max and Min Key Points

Other applications are achievable with this tool that are outside the scope of this project, but that could be future work for other areas of research. The possibilities of this tool apply not only to other areas directly related to rockets, but to any axisymmetric models in existence where it is desired to create a common solid, exterior, or interior. Examples could range from models as small as a pen, a dart, or a flashlight, to much larger bodies such as a fuselage, pipes or shafts in vehicles.

3.4 Fairing Geometry Creation

There are a variety of equations that are used in generating Fairings. The equations calculate the y value relative to the x location with the left facing tip of the nose cone being the zero location.

From Equation 3-8 to Equation 3-16, the cone equations that can be applied to this process are discussed. They each have some general common variables such as the x location which varies as it traverses the cone with the initial value of zero being the tip and increasing until it reaches the base. Then there is also the Length which is the distance of the cone from tip to base (L). The radius of the nose cone (r) is defined as the value at the base.

A distance is also set on the front which allows for a radius blend (tip_rad). This distance is not the value of the radius at the tip, but how much room you would like to give to a radius blend that will hold tangent to the cone and round perfectly at the top. Ensuring that this edge is tangent is very important since a harsh edge can negatively impact travel in supersonic flight. "Flow at a compression corner, such as the junction between a cylinder and a conical frustum, causes separation of the boundary layer. Shockwaves form at the flow separation and reattachment points during supersonic flight." (Yang 2008)

Figure 3-12 shows the new cone radius in orange and the original curve in blue with a tip_rad = 7. The tip ends up a little shorter than the initial equation would have. The new radius holds tangent to the curve connection and also perpendicular at the bottom so that the tip is rounded and not pointed. This clipping of the nose tip from an infinitely sharp edge to a finite value allows it to be manufactured and is vital to the analysis. "The amount of heat transfer is inversely proportional to a square root of nose radius, and so the nose bluntness has a considerable effect on the surface heat transfer, even though it has slight effect on the surface pressure distribution." (Lee 2001) So though it may not be critical to the pressure distribution and might be disregarded by some, the thermal aspect is strongly influenced by its shape and so is included in this research.

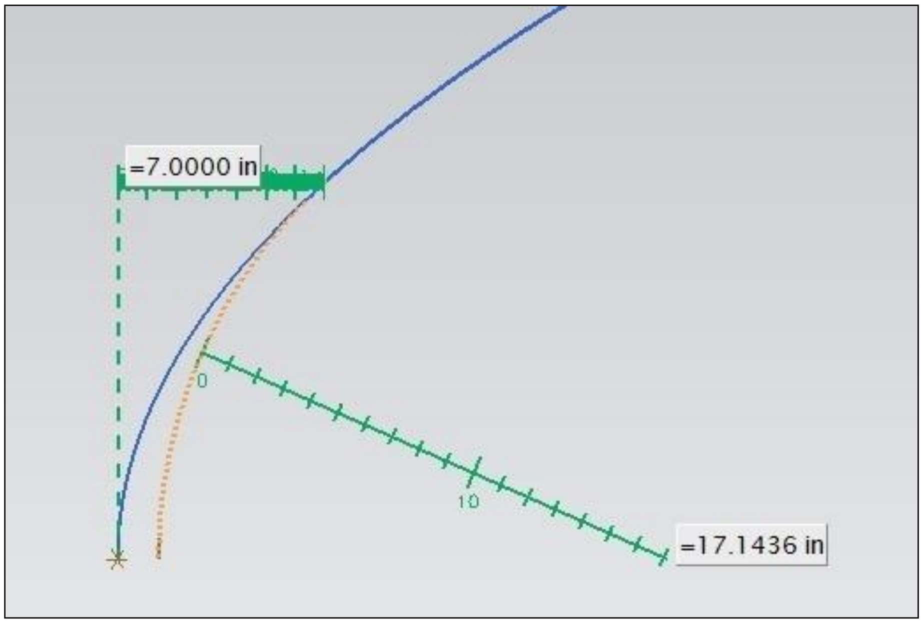


Figure 3-12: Description of tip_rad Dimension Relative to True Radius

All of the following curve equations are defined relative to their y-axis value (labeled as $Y_{thistype}$) which varies as you proceed in x. In all of the figures from Figure 3-13 to Figure 3-19, the common values are set with L=100, r=50, and tip_rad=7. All other values unique to the equation type are described in their captions. In both Equation 3-8 and Equation 3-9, the calculation of an ogive curve is shown. This is a common style of cone configuration. It is written relative to the y-axis value (labeled as Y_{OGIVE}) which varies as you proceed along the x-axis. Figure 3-13 shows an example of the Ogive curve.

$$Y_{OGIVE} = \sqrt{\rho^2 - (x - L)^2} + (R - \rho) \tag{3-8}$$

$$\rho = \frac{R^2 + L^2}{2R} \tag{3-9}$$

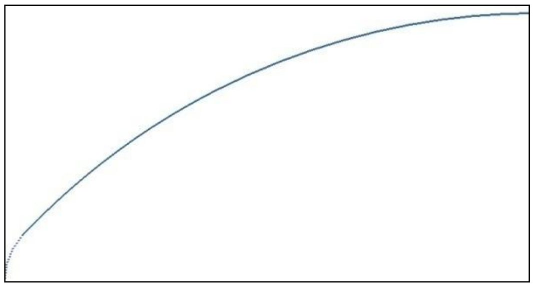


Figure 3-13: Ogive Curve Example

Another equation used to control the shape of a nose cone is an elliptic curve (labeled as $Y_{Elliptic}$). It is defined in Equation 3-10. An example is shown in Figure 3-14.

$$Y_{Elliptic} = r \times \sqrt{1 - \frac{(L - \mathrm{x})^2}{L^2}} \tag{3-10}$$

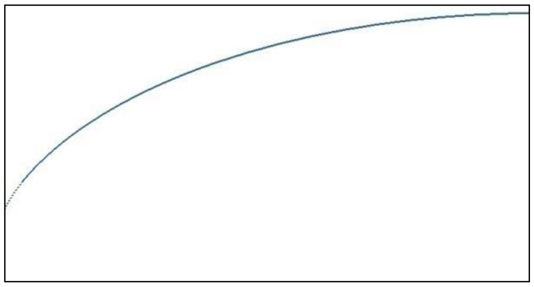


Figure 3-14: Elliptic Curve with L(100), r(50), and tip_rad=7

A standard cone equation is described here in Equation 3-11 which is based merely on a line defined by the radius (r) and the length (L) relative to the y-axis (labeled as Y_{Conic}).

$$Y_{Conic} = \frac{x \times r}{L} \tag{3-11}$$

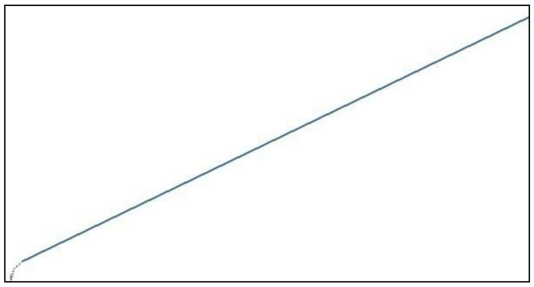


Figure 3-15: Cone Line Equation Example

An alteration to the basic cone equation is described here in Equation 3-12. The bi-conic cone has two parts, and so it has another radius (r_1) and length (L_1) control. The equation changes for X≥L_1. Figure 3-16 shows an example bi-conic cone curve.

$$Y_{Biconic} = if\ (x \le L_1)\ then\left(\frac{x \times r_1}{L_1}\right) else\left(r_1 + \frac{(x - L_1) \times (r - r_1)}{L}\right) \tag{3-12}$$

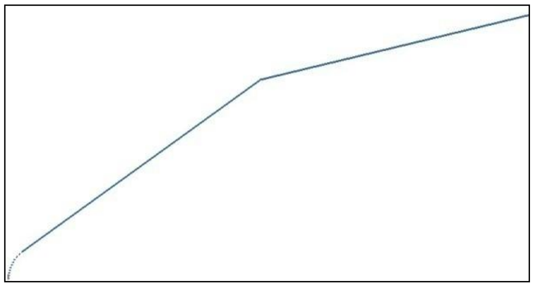


Figure 3-16: Bi-conic Curve Example with r1=3*r/4 and L1=L/2

The parabolic curve equation is shown here in Equation 3-13. This equation has a constant value *K* that can vary from 0 to 1. This controls how slim the parabola is, varying from a true parabola (*K*=1) to a cone (*K*=0). The added spectrum of choices provided by the *K* value gives a variety of possibilities that are not necessarily considered true parabolas, but provide for more flexibility in design. An example of just such a parabolic blend, with *K*=0.5 is shown in Figure 3-17.

$$Y_{Parabolic} = \frac{r \times \left(\frac{2x}{length} \right) - K \times \left(\frac{x}{length} \right)^2}{2 - K} \tag{3-13}$$

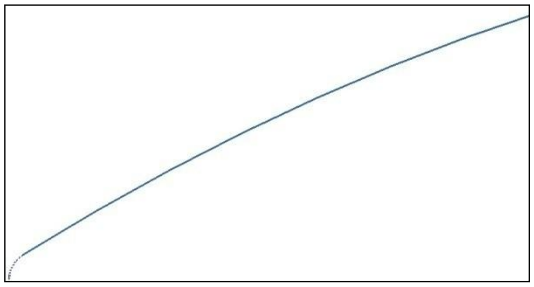


Figure 3-17: Parabolic Curve Example with *K*=.5

In a power curve equation, as given in Equation 3-14, the variable *n* can vary from 0 to 1.

$$Y_{Power} = r \times \left(\frac{x}{length} \right)^n \tag{3-14}$$

As *n* increases from 0 to 1, the cone transitions from a cylinder to a cone in shape. At n=0.5, the resultant is a parabola. An example is shown here with n=0.5 in Figure 3-18.

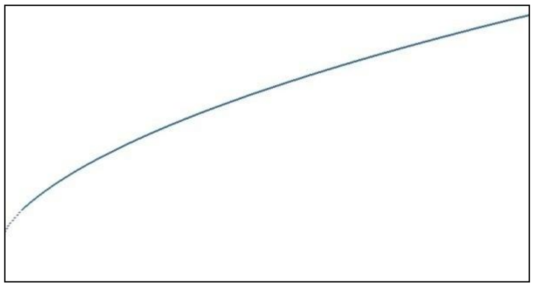


Figure 3-18: Power Curve Example with n=.5

The HAACK equation shown in Equation 3-15 is the only one that is not based on geometric shapes, but is derived to minimize the drag. Like the parabolic equations, the HAACK series nose cones are not quite tangent to the body at their base, but they are close to tangent. It is so close that it is hard to see with the human eye that they are not parallel where they meet. The value θ is broken out into a separate equation to simplify the equations and is specifically defined in Equation 3-16.

$$Y_{HAACK} = \frac{r \times \sqrt{\theta - \frac{1}{2}\sin(2\times\theta) + \mathrm{C}\times\sin(\theta)^3}}{\sqrt{\pi}} \tag{3-15}$$

$$\theta = \frac{\pi}{180°} \times \arccos\left(1 - \frac{2\times x}{L}\right) \tag{3-16}$$

There is also a C parameter that can vary from 0 to 1. When C=0, the cone is called an LD-HAACK which means it's the minimum drag for the given length and diameter, and when C=1/3, it is called LV-HAACK which indicates minimum drag for a given length and volume. The LV-HAACK is also often referred to as the Von Karman, or the Von Karman Ogive, and is commonly used for rocket fairings.

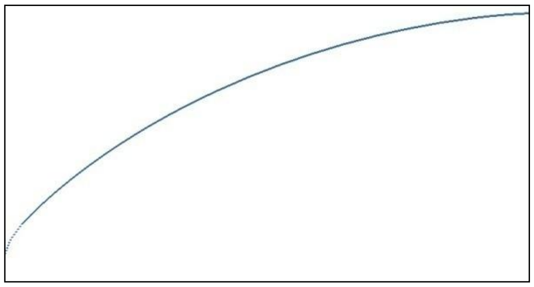


Figure 3-19: HAACK Curve Example with C=1/3

All of these equations need to be updated parametrically and be interchangeable while still maintaining the same common control parameters. These cones are attached to the structural and to the CFD fairing models which are described in more detail later.

3.5 Slice Properties Implementation

The information regarding the implementation of the slice properties tool can be organized into three categories. This list describes how the sections are categorized:

1. Slice Properties Calculations
2. Slice Properties Usage
3. Slice Properties Storage

3.5.1 Slice Properties Calculations

A simple description of a slice is all the material inside a region R bounded on both sides in the x dimension. It can be mathematically described by Equation 3-17 where *Sw* is the resultant Slice and is the width of the increment *Rw*.

$$S_w = \sum_i^n R_w \cap b_2 C_i^2 \qquad (3\text{-}17)$$

The region R_w is given a fixed length in x, starting at the most extreme left position in x and then by shifting one increment to the right with each iteration. The set of all desired information termed I_{slice} that needs to be obtained from each slice is achieved using a function termed F_{set}() and is shown in Equation 3-18.

$$I_{slice} = F_{set}\left(\sum_{w=0}^{w_{\max}} S_w\right) \tag{3-18}$$

Each slice can theoretically be created in any CAD package using the intersection of the region and all the parts that fall within that block or within two bounding planes. The concept is much easier than the implementations, as can be seen in chapter 4. One difficulty is that not all parts that are created in an assembly are defined using the same coordinate system. As an example, say the slice is 1 inch from the assembly coordinates of x=3 to x=4. If the model was completely modeled relative to this same coordinate system, you could just create a block in each part from x=3 to x=4 and intersect that with the parts. If however one of the parts was modeled from x=0 to x=1 and then was positioned in the assembly at x=3 to x=4, creating an intersection block at x=3 to x=4 inside that part would completely miss the intended part and misrepresent the region of x=3 to x=4. This problem intensifies when the parts have different orientations from the assembly.

This problem is solved by creating the block part in the main assembly, at the assembly coordinates desired, and then using transformation matrices to copy the block into the various parts at the right position, which are now referenced relative to the local coordinate systems. Next is to determine which portion of the parts intersects with the block.

When you have the determined slice of all parts in the assembly, the next step is to separate exterior parts from interior parts. This can be done using methods similar to the OML tool described earlier, replacing the concept of lines with solids. The exterior pieces need their density, thickness, and outer radius determined. These pieces are then directly represented in a beam model.

The interior parts are represented as lumped masses so more properties are calculated like the inertial moments in the x, y and z directions (Ixx, Iyy, Izz), and the center of gravity (CG). These can be calculated individually for each of the parts in the slice. Then the mass properties are summed to determine the lumped mass properties of the entire slice.   The reason and usage of the lumped masses is explained in the following sections of this chapter.

Some difficulties with the models are the doors and other access panels that dot the exterior of the rocket body. A simple version of the model, with no doors, is adequate at this stage. The challenge lies in determining the appropriate skin thickness and outer radius for these regions. Around the doors tends to have a thicker region of support that is not indicative of the rest of the revolved solid. So avoiding this region for the representative slice can be helpful. This thickness usually occurs inward and so does not affect the OML Creation result of the outer mold line. The process in section 3.3 describing this OML Creation tool is made to bridge the gaps of such holes to create a seamless surface. This is done by selecting a typical region far from the door hole as the representative thickness and outer radius. More specific discussion of this is detailed in the implementation section.

3.5.2 Slice Properties Usage

Beam representations of the slices will be linked together and lumped masses are placed at their centers. This gives a simple but adequate structural representation of the rocket body.

These beam models can be used to perform many tests to determine the structural integrity of the model.

3.5.3 Slice Properties Storage

It is better to wait until more information is known about these models (such as the forces applied) so that all the information can be concurrently imported to the structural model, but this cannot happen until flow analysis is done. The flow analysis requires some of the same input geometric data so these details had to be calculated first and subsequently saved for later. This leads to the divide between the initial slicing and the structural model creation. Because of timing issues related to when the slice and structure model are created, the process requires a method for information storage and retrieval.

Other analyses can also be done with this information so it is helpful to have a readable format that is user friendly and organized well. It is also beneficial to have a format that is automatically readable by a computer program without need of parsing the information again that was previously available. Because these formats have alternative uses, it was found to be more beneficial to make both copies.

3.6 Structural Geometry Creation

Analyses of such models can occur on multiple levels of fidelity. The models can be 2D beams, 3D shells, or 3D solid models. By moving to a higher fidelity level, you move to a realm which can hold higher accuracy but also higher complexity and runtime. So, when a sufficiently accurate result can be obtained by a lower fidelity model, this is the ideal choice because the results are computed in less time which allows the process to be optimizable. Because the goal of this thesis is to create preliminary fairings that are more simplistic and axisymmetric in shape, the 3D shell models are a good level of fidelity for the nose cone. The rest of the rocket is a more secondary study, which has previously received and will yet receive detailed review in a rocket development process so it is sufficient to use a faster 2D beam representation of this portion.

Beam models have multiple types, each with different properties and features. Many programs such as Abaqus FEA, ANSYS, LS-DYNA, and forms of Nastran have similar elements. Only general element details will be described. For specific element examples, see chapter 4 under the beam section. One specific type of beam element has been identified as a useful option for this project. Also, a form of a 1D mass element has been found with desirable attributes. The chosen type of beam element has two radial cross sections associated with it. It is defined on each end by an inner radius and outer radius or by a radius and thickness. The beam is computed to be a taper from the one section to the other and is ideal for this analysis. This element also has a density input and a length. From these, the volume and mass can be computed and used by the solver. The 1D mass element type has no cross sectional area but has mass and inertial properties. These masses are used to exhibit the attributes of all parts interior to the model. They are attached to the 2D beam models which are representative of the exterior parts of the model. It should be noted that though the lumped masses are attached in line with the beam model, their center of gravity is not necessarily co-axial with the rocket. The Slicer Dicer tool gathered this information earlier and stored it for use in this model.

There are many options here for different levels of analysis, but for this thesis, a combination of these different mesh qualities has been implemented. The initial plan for the structural analyses was to make a 2D beam model of the fairing attached to the rocket body. This was to be created as a sufficient analytical result for the structural integrity of the rocket body and as an initial estimate of the nose cone which would then be refined. The refinement was to be a separate, higher detail 3D sheet model that focused on the fairing alone. The forces calculated in the initial beam model were for determining the loads that the rocket body itself (being propelled up) is placing on the 3D sheet fairing model. These loads could then be input to a more detailed sheet model of just the fairing itself. Upon further research, it was determined that these two processes could be done better simultaneously. Instead of creating a beam model of the fairing, the detailed sheet mesh can be attached directly to the rocket body beam model. This reduces the time and the difficulty of transferring data from one model to another. Also, there was no need to make an inferior fairing beam model that would contain less detail before proceeding to the higher fidelity model. The calculation of the forces that would be placed into the higher fidelity model would also be less accurate when compared to the direct connection of the models.

The beam element information was gathered earlier in the OML Slicer Dicer tool and is now applied here. Each of the beam elements has its own Ixx, Iyy, Izz, CG, Moment, Inner Radius, Outer Radius and Material type.

3.7 Structural FEA

As mentioned before, the analytical rocket models have different levels of fidelity. The first part is the beam model of the entire rocket and the second is a structural shell mesh of the fairing itself. Both models have pressures, forces, and temperatures applied to them based on the results of the CFD analyses previously performed. From this information, the data can be analyzed using structural analysis programs such as Abaqus FEA, ANSYS, LS-DYNA, and

Nastran. From these analyses, the stresses can be compared to the yield strength or to some safety factor to determine if it meets the desired criteria.

3.8 CFD Calculations

Much like the structural model, there are multiple levels of fidelity that can be obtained ranging from an empirical based code, or a panel code up to a 2D or 3D mesh. The panel codes which calculate based on small regions called panels and the empirical based codes use previously gathered empirical data to calculate and predict results. The 2D meshes are representative of the influenced air flowing around the model, with one side representing the wall of the rocket and the opposite side often representing the end of the region of air influenced by the motion of the rocket. Remaining sides are often inlets and outlets for the air to flow through. Similarly for the 3D models, the mesh represents the air around the rocket. The most interior wall of elements is the rocket wall, and the most exterior is the farthest away air represented. There is often an inlet and outlet condition on each end.

Navier-Stokes equations (as given in Equation 3-19) characterize any situation of fluid flow in the x-direction, and a corresponding equation for the y-direction. The ρ is density, p is pressure, V is the velocity vector, μ is viscosity, and u is velocity in the x-direction. The partial derivative of ρu with respect to time has been removed due to the steady state condition. The inviscid assumption, which is justified hereafter, further simplifies this equation by removing the divergence term on the right-hand side.

$$div(\rho uV) = -\frac{\partial p}{\partial x} + div(\mu grad u) \tag{3-19}$$

The mass continuity equation is used in the solution. The partial derivative of density with respect to time, which normally appears as the first term, was also zero due to the steady state condition, leaving Equation 3-20.

$$div(\rho u) = 0 \tag{3-20}$$

The conservation of energy equation is also used and is listed here in Equation 3-21. This is the last governing equation, where *k* is conductivity, *T* is temperature, S_i is the momentum source, and *Φ* is the dissipation function.

$$div(\rho iu) = -pdivu + div(kgradT) + \Phi + S_i \tag{3-21}$$

3.8.1 Pressure and Drag Coefficients

The coefficients of pressure and drag are used for the flow calculations in this thesis. The coefficient of drag (C_D) is listed here in terms of the Drag Force (F_D) as is given in Equation 3-22.

$$F_D = \frac{1}{2}\rho v^2 C_d A \tag{3-22}$$

The coefficient of pressure (C_P) is also shown here in Equation 3-23.

$$C_P = \frac{P - P_\infty}{\frac{1}{2} P_\infty V_\infty^2} \tag{3-23}$$

3.9 Optimization

Others have tried many different techniques to arrive at an optimal solution. The vast combinations make it impossible to exhaustively search the feasible space. Different algorithms, such as gradient based algorithms, genetic algorithms, and response surface methodology have been employed to more quickly, yet adequately, traverse the space. These cases have usually been created encompassing only one or two aspects of the problem because integrating the various software packages can be tedious and hard to automate. This limited integration of tools often leads to improper exploitation of inaccuracies. (Unal 1996) (Kim 2000) (Lee 2006)

Similar applications have been performed on engine aircraft. Tappeta et al (Tappeta 1999) developed an MDO optimization for engine components involving similar tools. "The basic disciplines used for simulating the multidisciplinary environment are fluids, modal, structural and thermal analysis models." (Tappeta 1999) These same areas are as applicable to rocket fairings as to other aerodynamic fields.

The multi-objective optimization can be described as shown in Figure 3-20. The μ_i values are the objectives to minimize. The μ_1 term represents the objective to minimize the weight of the cone, and the μ_2 term is the objective to minimize the stress.

A generalized reduced gradient method was used here that alternates the variables so that on each iteration only one is dependent and all the others are independent. The goals are subject to the constraints listed, and are in terms of the structural integrity (SI), and the stresses in the beams (σ_{bi}).

$$\min_{x} \left\{ \mu_1(x), \mu_2(x), \mu_3(x), \mu_4(x) \right\}$$

Where μ_1= Weight

μ_2= Stress

Subject to:

$$\sigma_{bi} \leq \frac{\sigma_{bCLi}}{SF_1} \qquad \forall_i \in \{1,2,\ldots n\}$$

$$\sigma_{C\max} \leq \frac{\sigma_{allow}}{SF_2} \qquad C \in \{1,2,3,4,5,6,7\}$$

$$T_{bi} \leq \frac{T_{bMax}}{SF_3} \qquad \forall_i \in \{1,2,\ldots n\}$$

$$T_C \leq \frac{T_{Max}}{SF_3} \qquad C \in \{1,2,3,4,5,6,7\}$$

$$W_C \leq W_{Max} \qquad C \in \{1,2,3,4,5,6,7\}$$

Where b_i = Beam Slice

Where C = Cone Type

Figure 3-20: Multi-Objective Optimization Problem

4 Implementation

The implementation described in this chapter is an instance of the type of process that can be applied to other unique products. The implementation example here is of rocket nose cones. Although they are applied to a specific example in this chapter, the methods are applicable in other areas related to apexes or cowls. This section shows how an input rocket is taken and, using various codes and techniques, is optimized based on the input flight parameters.

4.1 Process Flow Descriptions

In Figure 4-1 the process has been broken down into its primary steps. Each of the pieces lists its function and the software and coding style used to create it. The starting location is the input part model. This model can be very large and complex and the time it takes on the initial steps is adversely affected by the size of the input model. Fortunately, the original part model is not involved in the optimization loop. Hence, once the required information is extracted, the model can be set aside and future iterations can be made independent of the model. The first three steps, shown in the purple box, perform the data gathering and extraction preparation functions required as input to the optimization process. The remaining process steps in Figure 4-1 are run repeatedly according to the controls imposed on the iSIGHT-FD optimization run. This portion of the method is run on a computer indicated by the blue box. The iterations are all stored in separate folders and can be kept as a history of the process. The steps that are shown in the green box are computationally intense and should be run on a supercomputer or a batch cluster. In this way, the purple, blue, and green computer systems can even be different operating systems. Only the purple steps require user input and the green and blue computers must share a common drive. The blue computer requires a visual display for the NX6 batch files to run.

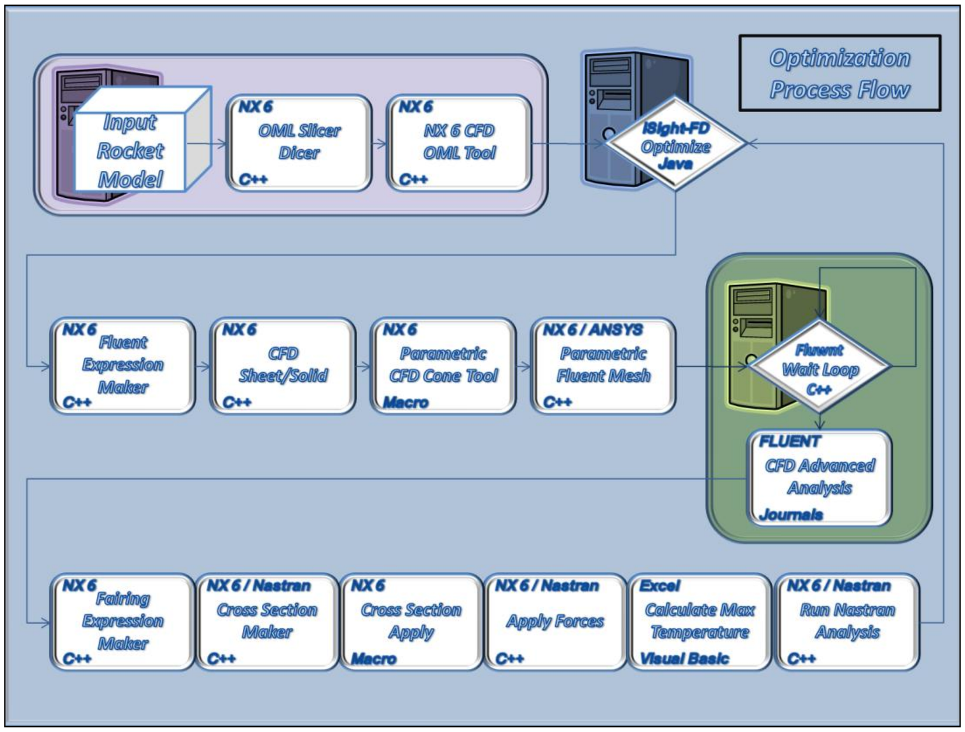

Optimization Process Flow
Input Rocket Model
NX 6 OML Slicer Dicer C++
NX 6 NX 6 CFD OML Tool C++
iSight-FD Optimize Java
NX 6 Fluent Expression Maker C++
NX 6 CFD Sheet/Solid C++
NX 6 Parametric CFD Cone Tool Macro
NX 6 / ANSYS Parametric Fluent Mesh C++
Fluent Wait Loop C++
FLUENT CFD Advanced Analysis Journals
NX 6 Fairing Expression Maker C++
NX 6 / Nastran Cross Section Maker C++
NX 6 Cross Section Apply Macro
NX 6 / Nastran Apply Forces C++
Excel Calculate Max Temperature Visual Basic
NX 6 / Nastran Run Nastran Analysis C++

Figure 4-1: Overall Process Design

Non-proprietary rocket examples will be used throughout the following chapters to help show how the process progresses. The first rocket (Figure 4-2) has fictitious components, but the body regions are representative of what is needed for the process.

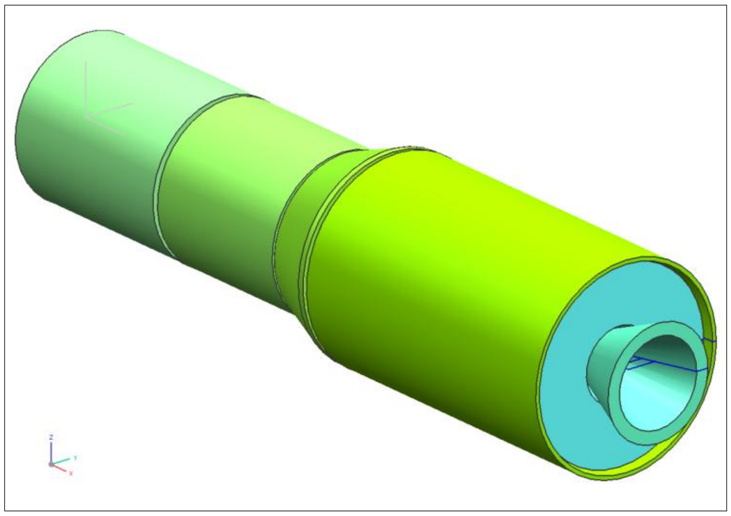


Figure 4-2: Simple Example NX6 Rocket Body Model

Figure 4-3 shows a cross sectional cutaway of the example rocket. Some of the components are not axisymmetric and some are not centered on the x-axis. These parts can be more difficult to represent in a simplified model, and is one of the purposes of slicing the model and creating the lumped masses that are representative of these interior parts. These lumped masses can be given multiple levels of fidelity based on how much detail is desired.

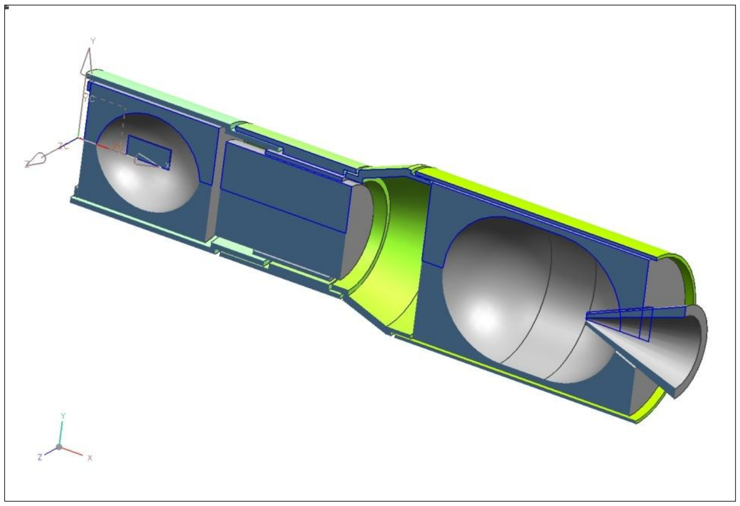
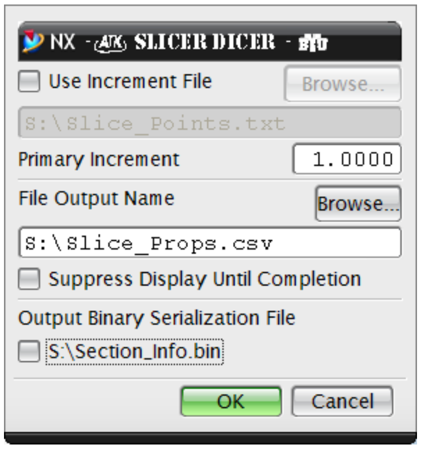


Figure 4-3: Cross Section View of Simple Example Model

4.1.1 OML Slicer Dicer Overview

The Slicer Dicer tool takes in a rocket body and "slices" the body into small, one inch segments (or any size chosen by the programmer). The sizes of the slices can be one set increment or a file of key points and lengths can be input to specify the divisions. The program then takes these slices and determines the properties of each, namely the inertial properties (Ixx, Iyy, Izz), the center of gravity (CG), and moments of the slice. These results are stored in a serialized array. Serialization involves storing C++ data, such as classes and structs, in a way that allows it to be used more seamlessly later in other programs. This is because it can be used without ever writing the data to a text file which in turn would need to be parsed, having the data reassigned to their original structures for a downstream program. This serialized data is used by the Section Maker tool. The data is also exported to a comma delimited text file that is helpful in other advanced rocket applications not required by this project. The rest of this section describes how the OML Slicer Dicer process is performed.

Figure 4-4 is the Graphical User Interface (GUI) associated with the OML Slicer Dicer application. The choices allow the user to perform various tasks based on their needs. If a greater level of customization is wanted, an increment file could be selected which would manage more details than the *Primary Increment* input allows for alone. The *File Output Name* is written out to the desired location with details into the operation and results. The *Suppress Display Until Completion* option makes the program run faster but progress cannot be seen on the display as it happens. For the optimization process, the *Output Binary Serialization File* choice is selected so that the data is stored for later use.


NX - ATK SLICER DICER -

Use Increment File

S:\Slice_Points.txt

Primary Increment 1.0000

File Output Name

S:\Slice_Props.csv

Suppress Display Until Completion

Output Binary Serialization File

S:\Section_Info.bin

Figure 4-4: Slicer Dicer GUI

4.1.2 OML Slicer Dicer Steps

The code for this portion was created using C++ for use in NX6. This is the most complex portion of all the code for this project. The data management of the operations and control of boundaries is more complex than the initial description details.

To start, the rocket model to test is opened and the code begins by cycling through all the parts in the assembly to determine the left and right extremes in the model. Planes have been placed extremely far away in the parent assembly (-15000 and +15000 units away). The complexity is due to the fact that each part may have been modeled in a different coordinate system. Checking the distance from the planes using the *UF_MODL_ask_minimum_dist* function determines how far each datum plane is apart relative to its own system. So even if in an assembly, the parts appear to be next to each other, if they are not brought in relative to the assembly, the distance will not give the desired result. This can be done by wave linking. Waving in or wave linking describes the action of taking the locations of one object in an assembly and placing it in another part model at the same relative distance as it was in the assembly. When the planes are wave linked in relative to the individual parts, the

UF_MODL_ask_minimum_dist function returns the correct distance values between all the components and can be used for determining the extreme ends. All the points are stored and the minimum of the minimum values and the maximum of the maximum values are determined to be the extreme ends of the model.

Once the basic rocket dimensions are determined, the *Use Increment File* check box in Figure 4-4 can be selected to allow for higher detailed control. If it is chosen, the input file is selected, read and key points with related increments are used as the file creates the slices. An angle control is also applicable, which allows the user to avoid complex regions on the model that would not be representative of the general thickness of the region, such as a door or a rivet hole. The angle control is really defined by two numbers, which are vector based in the YZ plane, and is described in more detail in the Appendix. Otherwise, the *Primary Increment* input value, seen in Figure 4-4 is used and a purely planar (XY plane) is used for the OR and IR calculations. The weight, inertial properties, moments and CG are all calculated off of the true dimensional quantities and are not rounded like the OR and IR dimensions. If the check box for suppression of the display is set, the *UF_DISP_set_display* function is used to suppress the updates of the picture during the operations until completion of the execution. It is easier to see how the program progresses through the part without it being suppressed but it is slower due to the screen updates that occur.

A revolved disc is created at the master level assembly that represents the volume of the region to be sliced. As a note, this disc could be changed for projects outside the scope of this project to be smaller so as to just catch the interior of the model or be turned into a donut just to catch a ring of results. The revolved disc can be seen in Figure 4-5 as a thin purple disc. This example segment is one inch thick.

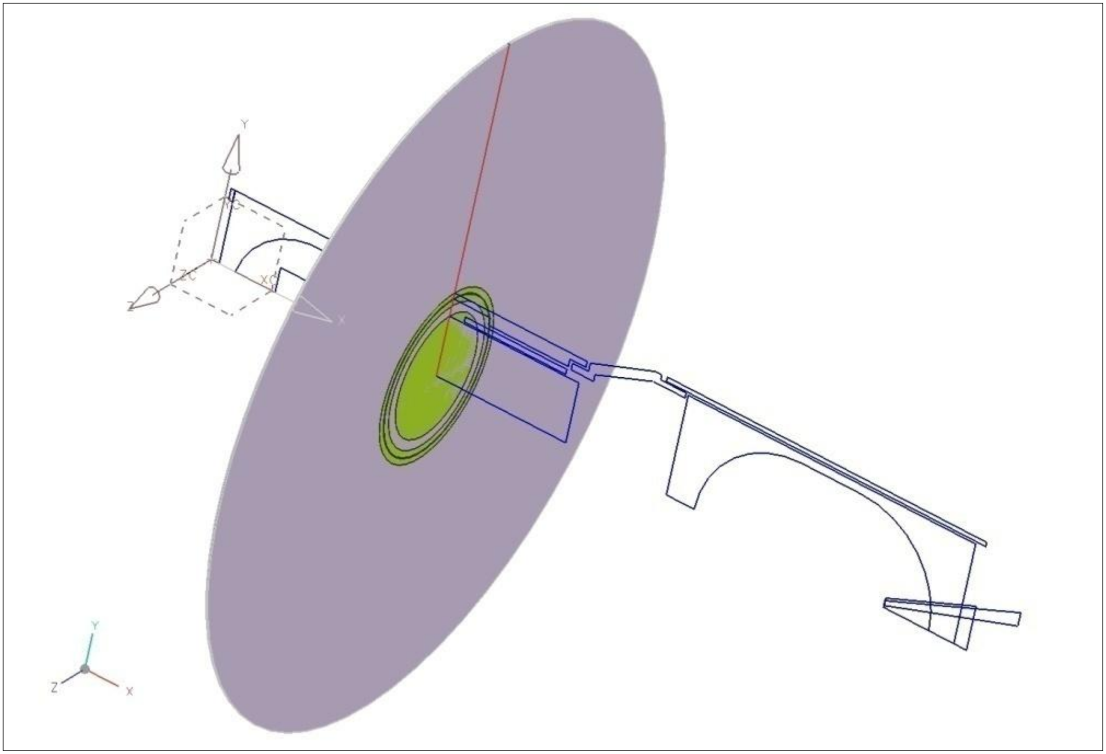


Figure 4-5: One Thin Purple Slice Region of the Simple Example Model

The function that actually creates the slices is then called recursively until every part has been cycled through. The parts are cycling first based on component and then in each component by solid body, because multiple solid bodies can be in a component. Each time it comes to a new body, the disc created in the master assembly is wave linked in to the current component and then used to intersect the current solid body. This ensures that the disc is at the proper slice location relative to the assembly coordinate system and not relative to the individual parts. In Figure 4-6 below, an example slice is shown corresponding to the disc slice shown in Figure 4-5.

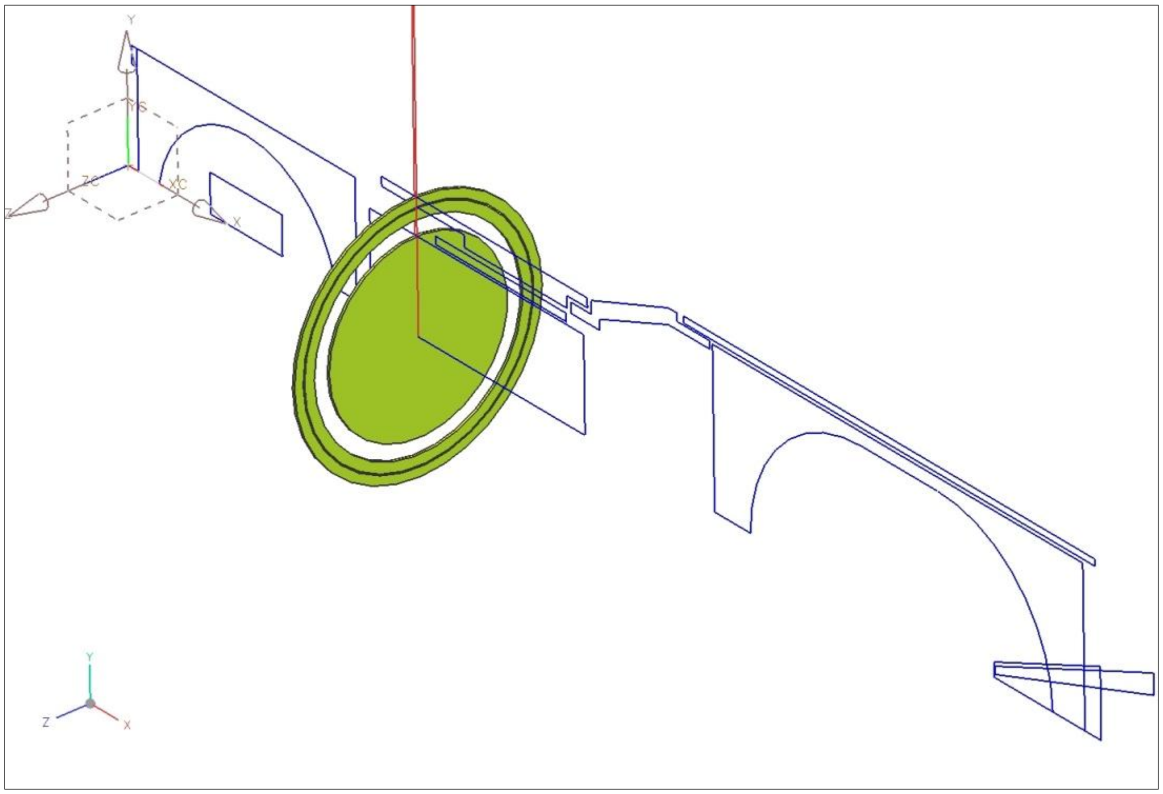
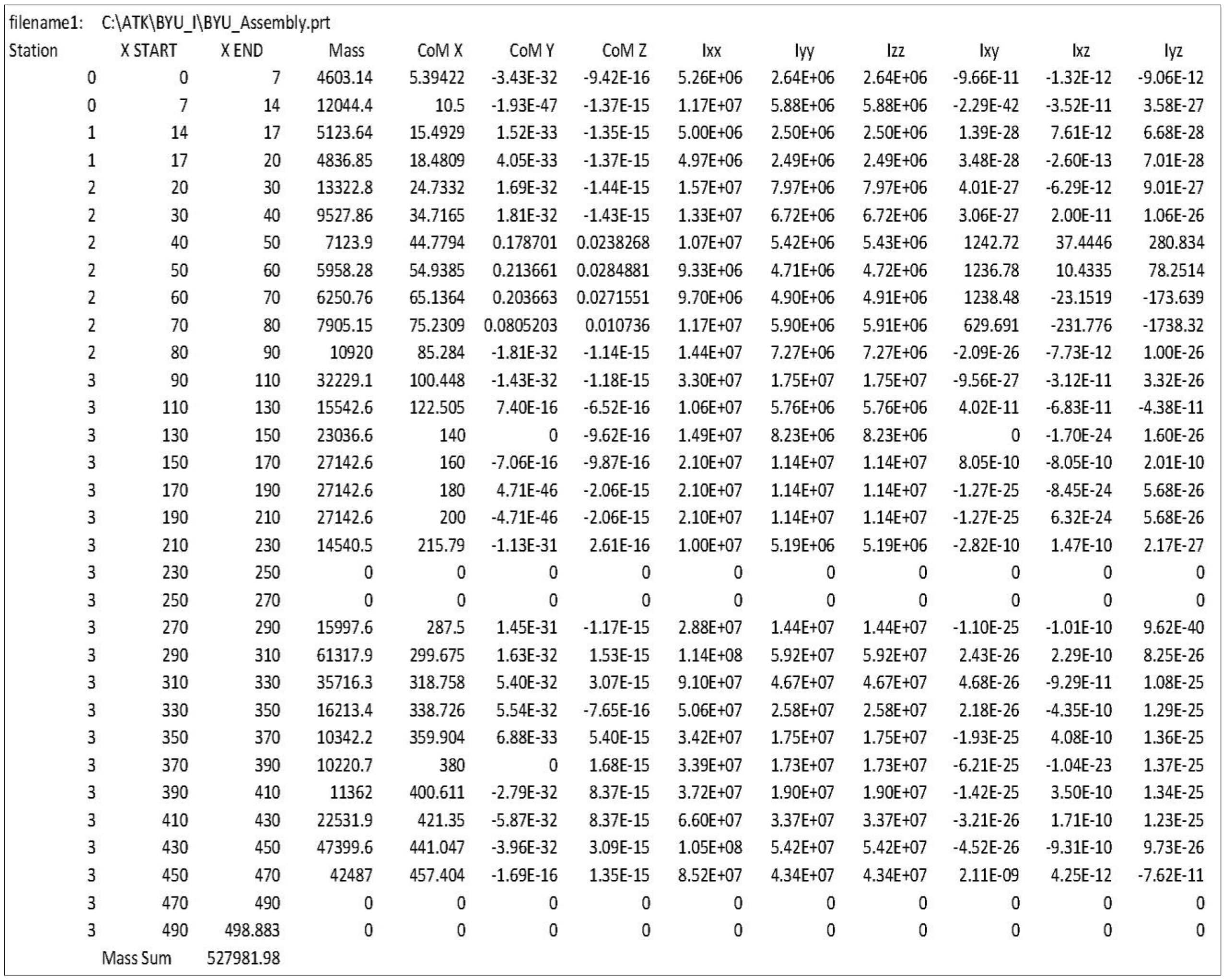


Figure 4-6: One Resultant Slice of Simple Example Model

Now that the slices have been created, it is important to determine which pieces in a given slice constitute a slice of the exterior case. Users will want the mass properties together and not separated into interior and exterior items for some applications of this tool. This is especially true for the text output option which writes mass properties to a Comma Separated Values (CSV) file, like the one shown here in Figure 4-7. In cases like this, where all the items are desired as a whole, you do not want a serialized file for future use; so unchecking the serialization check box allows you to skip the *Serialize_Section_Info* option.



filename1:	C:\ATK\BYU_I\BYU_Assembly.prt											
Station	X START	X END	Mass	CoM X	CoM Y	CoM Z	Ixx	Iyy	Izz	Ixy	Ixz	Iyz
0	0	7	4603.14	5.39422	-3.43E-32	-9.42E-16	5.26E+06	2.64E+06	2.64E+06	-9.66E-11	-1.32E-12	-9.06E-12
0	7	14	12044.4	10.5	-1.93E-47	-1.37E-15	1.17E+07	5.88E+06	5.88E+06	-2.29E-42	-3.52E-11	3.58E-27
1	14	17	5123.64	15.4929	1.52E-33	-1.35E-15	5.00E+06	2.50E+06	2.50E+06	1.39E-28	7.61E-12	6.68E-28
1	17	20	4836.85	18.4809	4.05E-33	-1.37E-15	4.97E+06	2.49E+06	2.49E+06	3.48E-28	-2.60E-13	7.01E-28
2	20	30	13322.8	24.7332	1.69E-32	-1.44E-15	1.57E+07	7.97E+06	7.97E+06	4.01E-27	-6.29E-12	9.01E-27
2	30	40	9527.86	34.7165	1.81E-32	-1.43E-15	1.33E+07	6.72E+06	6.72E+06	3.06E-27	2.00E-11	1.06E-26
2	40	50	7123.9	44.7794	0.178701	0.0238268	1.07E+07	5.42E+06	5.43E+06	1242.72	37.4446	280.834
2	50	60	5958.28	54.9385	0.213661	0.0284881	9.33E+06	4.71E+06	4.72E+06	1236.78	10.4335	78.2514
2	60	70	6250.76	65.1364	0.203663	0.0271551	9.70E+06	4.90E+06	4.91E+06	1238.48	-23.1519	-173.639
2	70	80	7905.15	75.2309	0.0805203	0.010736	1.17E+07	5.90E+06	5.91E+06	629.691	-231.776	-1738.32
2	80	90	10920	85.284	-1.81E-32	-1.14E-15	1.44E+07	7.27E+06	7.27E+06	-2.09E-26	-7.73E-12	1.00E-26
3	90	110	32229.1	100.448	-1.43E-32	-1.18E-15	3.30E+07	1.75E+07	1.75E+07	-9.56E-27	-3.12E-11	3.32E-26
3	110	130	15542.6	122.505	7.40E-16	-6.52E-16	1.06E+07	5.76E+06	5.76E+06	4.02E-11	-6.83E-11	-4.38E-11
3	130	150	23036.6	140	0	-9.62E-16	1.49E+07	8.23E+06	8.23E+06	0	-1.70E-24	1.60E-26
3	150	170	27142.6	160	-7.06E-16	-9.87E-16	2.10E+07	1.14E+07	1.14E+07	8.05E-10	-8.05E-10	2.01E-10
3	170	190	27142.6	180	4.71E-46	-2.06E-15	2.10E+07	1.14E+07	1.14E+07	-1.27E-25	-8.45E-24	5.68E-26
3	190	210	27142.6	200	-4.71E-46	-2.06E-15	2.10E+07	1.14E+07	1.14E+07	-1.27E-25	6.32E-24	5.68E-26
3	210	230	14540.5	215.79	-1.13E-31	2.61E-16	1.00E+07	5.19E+06	5.19E+06	-2.82E-10	1.47E-10	2.17E-27
3	230	250	0	0	0	0	0	0	0	0	0	0
3	250	270	0	0	0	0	0	0	0	0	0	0
3	270	290	15997.6	287.5	1.45E-31	-1.17E-15	2.88E+07	1.44E+07	1.44E+07	-1.10E-25	-1.01E-10	9.62E-40
3	290	310	61317.9	299.675	1.63E-32	1.53E-15	1.14E+08	5.92E+07	5.92E+07	2.43E-26	2.29E-10	8.25E-26
3	310	330	35716.3	318.758	5.40E-32	3.07E-15	9.10E+07	4.67E+07	4.67E+07	4.68E-26	-9.29E-11	1.08E-25
3	330	350	16213.4	338.726	5.54E-32	-7.65E-16	5.06E+07	2.58E+07	2.58E+07	2.18E-26	-4.35E-10	1.29E-25
3	350	370	10342.2	359.904	6.88E-33	5.40E-15	3.42E+07	1.75E+07	1.75E+07	-1.93E-25	4.08E-10	1.36E-25
3	370	390	10220.7	380	0	1.68E-15	3.39E+07	1.73E+07	1.73E+07	-6.21E-25	-1.04E-23	1.37E-25
3	390	410	11362	400.611	-2.79E-32	8.37E-15	3.72E+07	1.90E+07	1.90E+07	-1.42E-25	3.50E-10	1.34E-25
3	410	430	22531.9	421.35	-5.87E-32	8.37E-15	6.60E+07	3.37E+07	3.37E+07	-3.21E-26	1.71E-10	1.23E-25
3	430	450	47399.6	441.047	-3.96E-32	3.09E-15	1.05E+08	5.42E+07	5.42E+07	-4.52E-26	-9.31E-10	9.73E-26
3	450	470	42487	457.404	-1.69E-16	1.35E-15	8.52E+07	4.34E+07	4.34E+07	2.11E-09	4.25E-12	-7.62E-11
3	470	490	0	0	0	0	0	0	0	0	0	0
3	490	498.883	0	0	0	0	0	0	0	0	0	0
	Mass Sum	527981.98										

Figure 4-7: Slice_Props.csv File Based on Interior Parts of Example

4.1.3 OML Slicer Dicer Element Properties

In each slice, there are two different elements that have properties to be defined. The first type is a beam element. These beam elements have an exterior radius and thickness at both ends. These ends define the tapered slope of the element from the one side to the other (much like a cone with the tip cut off).

By using two equal length elements per slice the center has a node location for attachment. A lumped mass is placed at this location which defines the interior weight, CG, and inertial properties. It can be defined as being offset from the node it is attached to in any direction. It was determined that it is easier to divide each slice region into two beam elements with an averaged cross section at its center. Then when the lumped mass is attached at this center point, there is no need to adjust for the offset of attaching at one of the ends. The slice data is stored in block sets, which represent these two beams with a lumped mass at their center. This is much more understandable to someone using the model later for other analyses because the lumped masses are not placed at locations which are offset from their true weight centers. See section 6.2.1 for more on these properties.

4.1.4 OML Slicer Dicer Storage

In Figure 4-8 is listed the C++ class of information that is stored and used throughout the optimization process. Each element from a slice is placed into this structure. The function that is used for recovering the information is also listed. It provides the means by which the information can be broken down to a binary level and reconstituted for use later on.

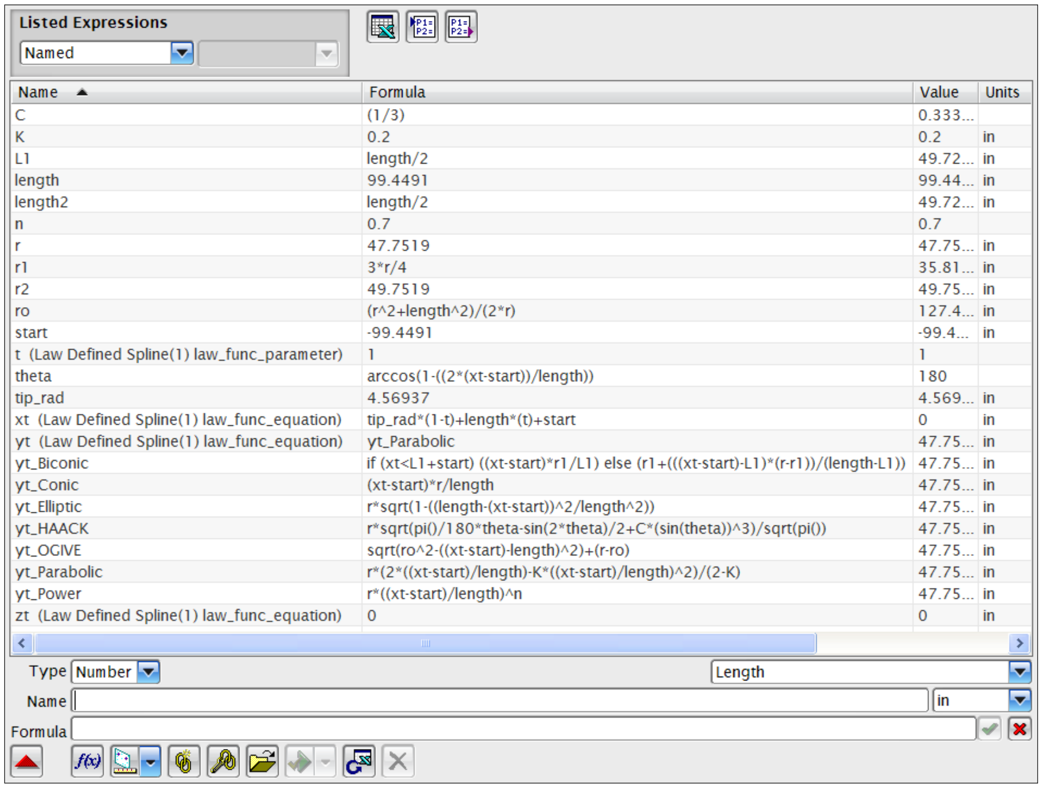
This process of storing the information at a binary level is known as serialization. It provides a way of both storing and accessing the information without needing to deal with extensive issues of text formatting and data management. This method is easier to manage than say an example of storing a text list of all the information, which could be very large in size. This text file would then require a program to parse through it over and over again at multiple stages of the optimization to regain the desired information.

```
#ifndef C_SECT_INFO_H
#define C_SECT_INFO_H
class C_Sect_Info
{
//private:
public:
        double  Mass;   double  CGx;    double  CGy;    double  CGz;
        double  Ixx;    double  Iyy;    double  Izz;
        double  Ixy;    double  Ixz;    double  Iyz;
        double          Offset_X;       double  Offset_Y;       double          Offset_Z;
        double  Density;                double  Pr_Axe[9];      double  Pr_Mom[3];
        double  Start_IR;       double  Start_OR;
        double  End_IR;         double  End_OR;
        double  Start_X;                double  End_X;
        int     Mat_Num;                int     Node_Num;
//public:
    C_Sect_Info(double ma, double cgx, double cgy, double cgz, double ixx,double
iyy,double izz,double ixy, double ixz, double iyz,double off_x, double off_y, double
off_z, double density, double start_ir, double start_or, double end_ir, double end_or,
double start_x, double end_x,int mat_num, int node_num )// : mass(ma)
        {
                Mass    =       ma;
                CGx     =       cgx;
                CGy     =       cgy;
                CGz     =       cgz;
                Ixx     =       ixx;
                Iyy     =       iyy;
                Izz     =       izz;
                Ixy     =       ixy;
                Ixz     =       ixz;
                Iyz     =       iyz;
                Offset_X        =       off_x;
                Offset_Y        =       off_y;
                Offset_Z        =       off_z;
                Density =       density;
                Start_IR        =       start_ir;
                Start_OR        =       start_or;
                End_IR  =       end_ir;
                End_OR  =       end_or;
                Start_X =       start_x;
                End_X   =       end_x;
                Mat_Num =       mat_num;
                Node_Num        =       node_num;

        }
    ~C_Sect_Info() { }
};
#endif // C_SECT_INFO_H

int Serialize_Section_Info()
{
        ofstream out("S:\\Section_Info.bin", ios::binary);
        for(unsigned int i=0;i<section_info.size();i++)
        {
                out.write((char*)&section_info.at(i), sizeof(section_info.at(i)));
        }
    cout<<"*Closing File Stream...n"<<endl;
    out.close();
        cout<<"----------End of Serialize Function"<<endl;
        //system("pause");
        return 0;
}
```

Figure 4-8: Class C_Sect_Info Used for Storing Properties for Other Programs

4.1.5 Cone Expression Maker

This C++ code is run in NX6 and is the basis for a parametric part file. It includes all the equations for nose cones, as described in Chapter 3. By including all the choices for the different cone types, the model can seamlessly transition from one type to another. The current cone type is indicated by the choice yt and is currently set to yt_Parabolic. It also sets the initial values for the radius and length of the nose cone. Any other values related to specific cone types are also set at a default, such as the bi-conic, where the secondary length is set to half the primary length. An example list of expressions for the simple part just covered is shown in Figure 4-9.



Name	Formula	Value	Units
C	(1/3)	0.333...	
K	0.2	0.2	in
L1	length/2	49.72...	in
length	99.4491	99.44...	in
length2	length/2	49.72...	in
n	0.7	0.7	
r	47.7519	47.75...	in
r1	3*r/4	35.81...	in
r2	49.7519	49.75...	in
ro	(r^2+length^2)/(2*r)	127.4...	in
start	-99.4491	-99.4...	in
t (Law Defined Spline(1) law_func_parameter)	1	1	
theta	arccos(1-((2*(xt-start))/length))	180	
tip_rad	4.56937	4.569...	in
xt (Law Defined Spline(1) law_func_equation)	tip_rad*(1-t)+length*(t)+start	0	in
yt (Law Defined Spline(1) law_func_equation)	yt_Parabolic	47.75...	in
yt_Biconic	if (xt<L1+start) ((xt-start)*r1/L1) else (r1+(((xt-start)-L1)*(r-r1))/(length-L1))	47.75...	in
yt_Conic	(xt-start)*r/length	47.75...	in
yt_Elliptic	r*sqrt(1-((length-(xt-start))^2/length^2))	47.75...	in
yt_HAACK	r*sqrt(pi()/180*theta-sin(2*theta)/2+C*(sin(theta))^3)/sqrt(pi())	47.75...	in
yt_OGIVE	sqrt(ro^2-((xt-start)-length)^2)+(r-ro)	47.75...	in
yt_Parabolic	r*(2*((xt-start)/length)-K*((xt-start)/length)^2)/(2-K)	47.75...	in
yt_Power	r*((xt-start)/length)^n	47.75...	in
zt (Law Defined Spline(1) law_func_equation)	0	0	in

Type: Number — Length
Name: — in
Formula:

Figure 4-9: NX6 Expressions Set by Cone Expression Maker Tool

4.1.6 Parametric Mid Cone Tool

This code is a text based macro run in NX6 that sets the necessary options for creation of a *law curve* as defined in NX6 documentation. The functionality in the C++ library of NX6 was not yet capable of performing a *law curve* operation programmatically so this workaround was instituted.

After this tool is run, a user can even interactively change the options in the expressions list of the part model to update the cone style. This includes parametric changes in lengths and also allows for complete changes in style. For example, a change from a power curve to an elliptic or a bi-conic style could be made without disrupting the surface sheet created or any other associated features like a FEA or CFD mesh.

4.1.7 OML Maker Tool

This tool is used to determine the outer mold line (outer radius along the length) of the rocket. The large number of parts that are present in a file make it difficult to determine which are related to the exterior and how to bridge the gaps between adjacent parts. The basic principles of how this is performed were shown in Chapter 3. Figure 4-10 is the OML Maker GUI.

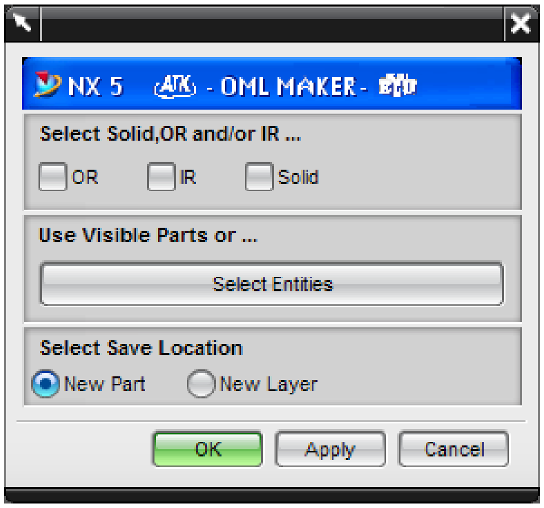


Figure 4-10: OML Maker GUI in NX5

There are three outputs available that might be used based on the user's needs. Any combination of the OR, IR and Solid check box options can be selected. The exterior parts can be selected using the *Select Entities* button. They are highlighted as they are selected and can be reselected to deselect. Also, desired parts can be chosen by making only those parts visible. Examples of both of these methods are shown in two cases below (Figure **4-11** through Figure 4-14).

In Figure **4-11**, the solids that are not wanted as part of the exterior have been hidden and only the desired exterior solids remain. Using this input and running the OML Maker Tool, where OR and IR are checked, results in the model shown in Figure 4-12. Only a cross section cut of the resultant sheets is displayed so that the internal lines can be seen. The Solid check box can also be selected to get a solid combination of these IR and OR sheets.

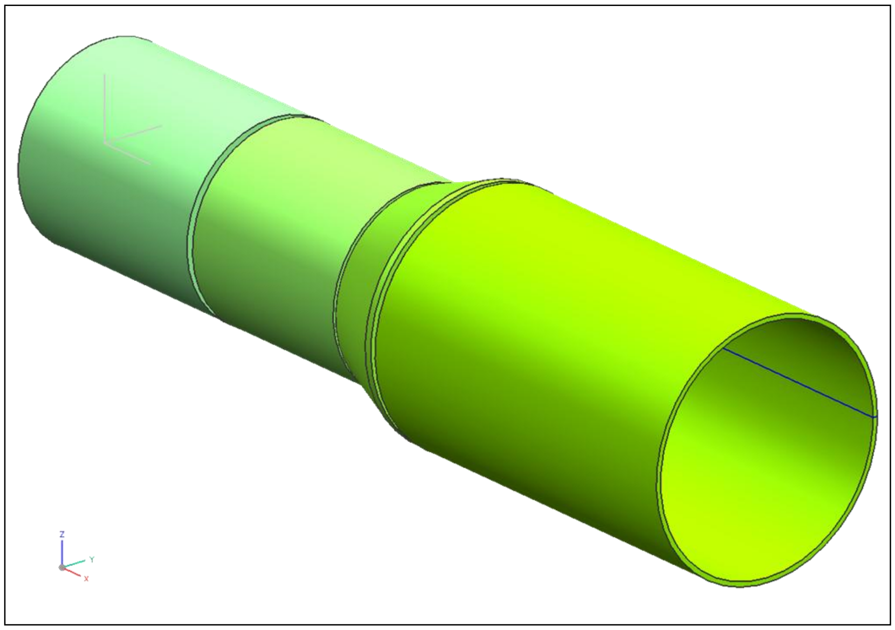


Figure 4-11: Desired Exterior Pieces Shown on Model

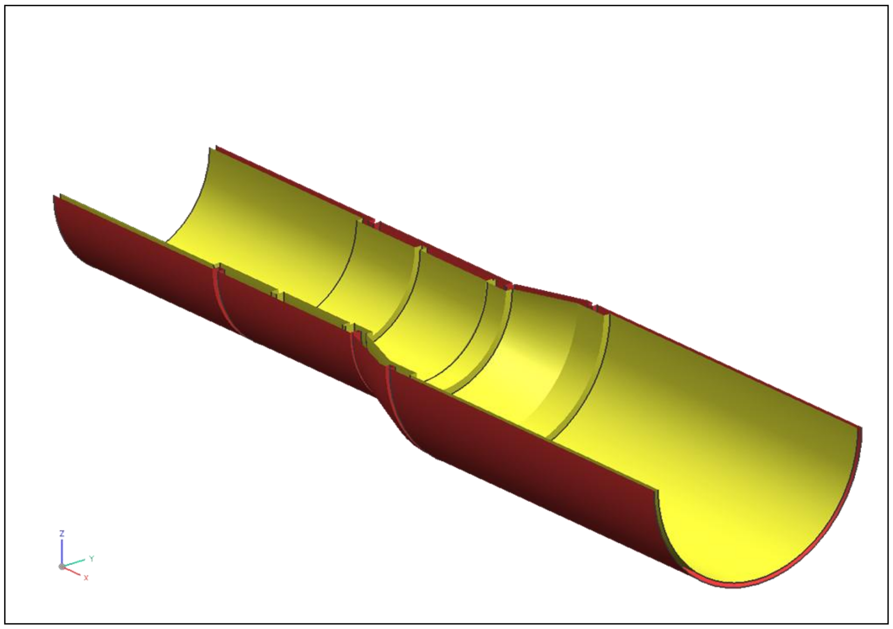


Figure 4-12: Resultant IR and OR Sheets of Model

The example in Figure 4-13 shows the solids selected using the *Select Entities* button and Figure 4-14 shows the resultant if only the OR check box is selected. The resultant sheets and solid are ideal for creating meshes for flow or for structural analyses, and can be placed into separate part files using the *New Part* radio button.

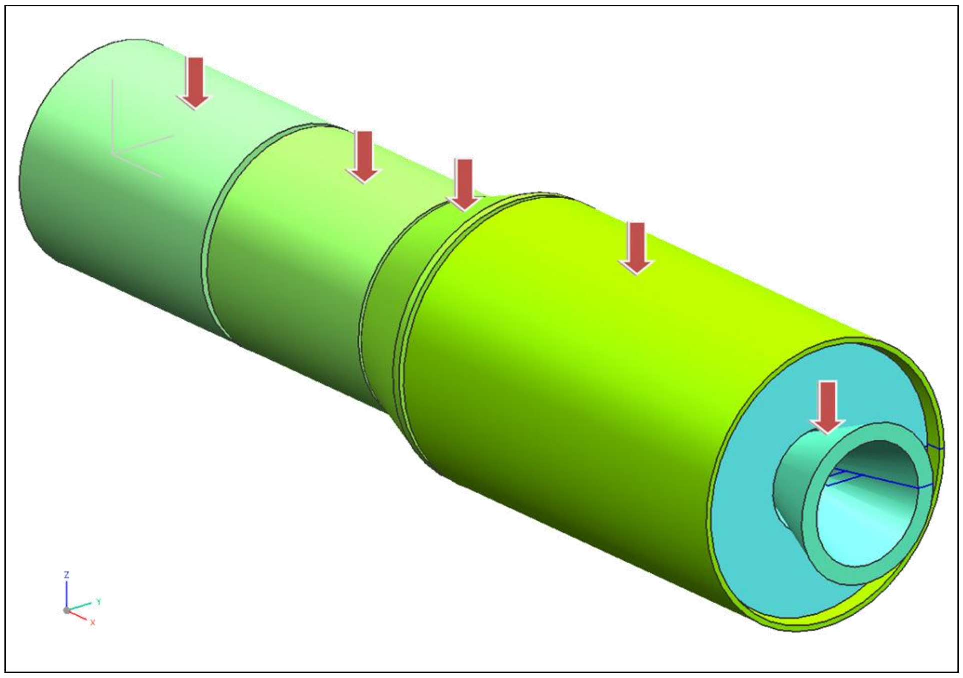


Figure 4-13: Selected Solids on Model

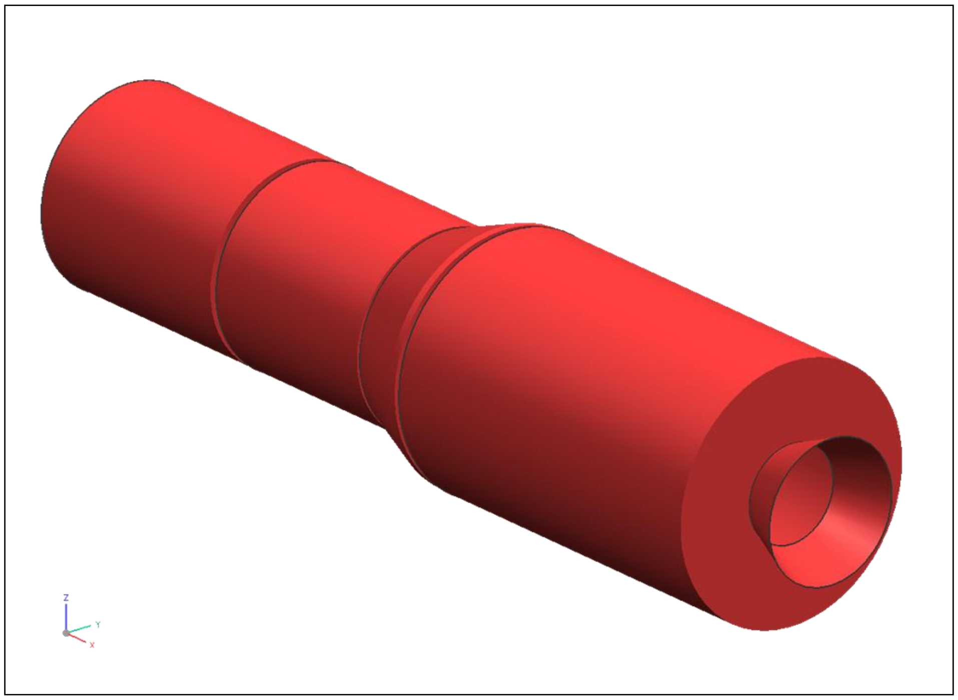


Figure 4-14: Resultant OR Sheet Based on Selections

4.1.8 CFD Parametric OML Tool

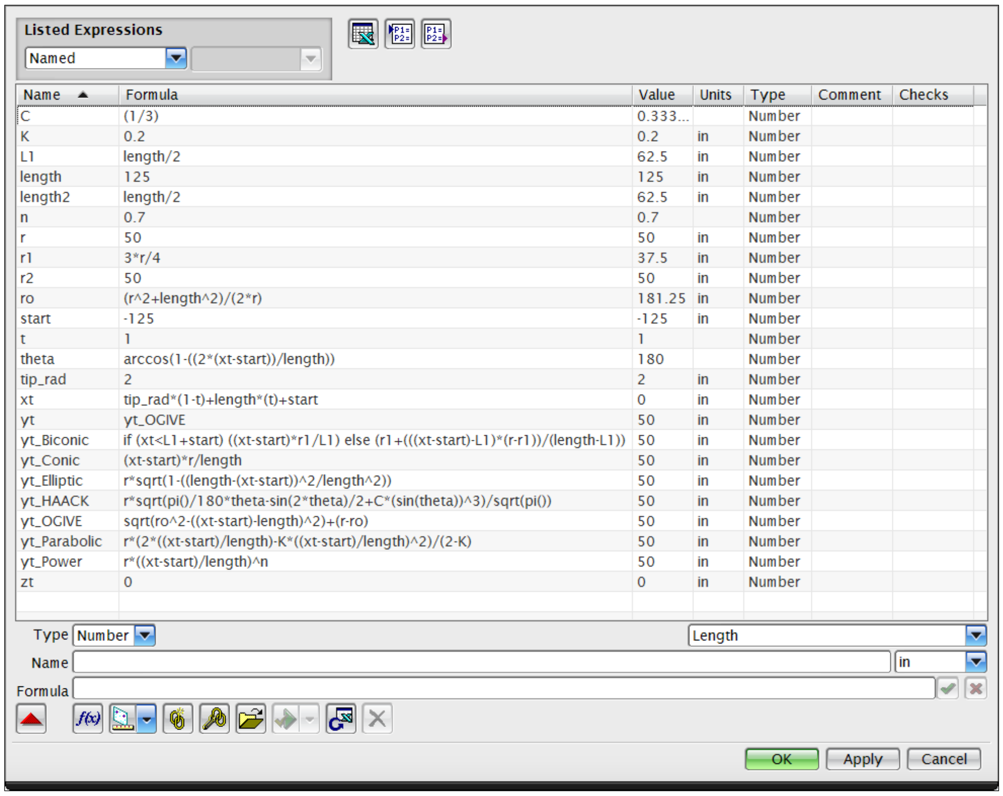
This tool is used to calculate the outer mold line of the rocket for creation of a parametric sheet for meshing. It is run in NX6 and is built using C++. This tool is the more focused version of the OML Maker tool used for setup of the CFD Fluent Mesh.

4.1.9 Missile DATCOM File Writer

The C++ code developed for this section was created in Visual Studio .NET 2005 and is run using an executable. It is made to do multiple runs in Missile DATCOM, which is a United States Air Force code based on experimental data. The data is tabularized into a database that allows for a realistic resultant output of pressures along the surfaces based on the inputs given. The inputs used for this project are: the mach number (MACH), the altitude (ALT), the alpha angle (ALPHA), the type of nose cone, which has 6 types, (TNOSE), the length of the nose (LNOSE), the diameter at the base of the nose (DNOSE), the length of the center body (LCENTR), the diameter of the back of the center (DCENTR), the length of the aft portion (LAFT), and the diameter of the back of the aft portion (DAFT). There are other commands which are used in the input decks for this program but they are constants and are not varied in this study. Some are used to create specific output files for use by following programs such as the PRESSURES command card, which writes all the pressures for the various runs selected to a file for later use. This option has proven to be very helpful for simple rocket designs, but Missile DATCOM is limited to a single stage shape rocket with no complex section combinations. This makes this tool far less capable than Fluent and more limiting than it is worth for most optimizations. Some effort could be made to make this option more capable, but this falls to a future work scenario and is described in Chapter 6.

4.1.10 Fluent Expression Maker

The code for this section is similar to the cone expression maker described above; the principle difference is the replacement of the cone thickness with an exterior shape (OML) equation. This new geometry is used as the basis of the mesh for calculating the effects of flow (CFD) over the rocket. This program was created using C++ and is run in NX6. A representative set of expressions, based on the simple example above, is listed here in Figure 4-15.



Name	Formula	Value	Units	Type	Comment	Checks
C	(1/3)	0.333...		Number		
K	0.2	0.2	in	Number		
L1	length/2	62.5	in	Number		
length	125	125	in	Number		
length2	length/2	62.5	in	Number		
n	0.7	0.7		Number		
r	50	50	in	Number		
r1	3*r/4	37.5	in	Number		
r2	50	50	in	Number		
ro	(r^2+length^2)/(2*r)	181.25	in	Number		
start	-125	-125	in	Number		
t	1	1		Number		
theta	arccos(1-((2*(xt-start))/length))	180		Number		
tip_rad	2	2	in	Number		
xt	tip_rad*(1-t)+length*(t)+start	0	in	Number		
yt	yt_OGIVE	50	in	Number		
yt_Biconic	if (xt<L1+start) ((xt-start)*r1/L1) else (r1+(((xt-start)-L1)*(r-r1))/(length-L1))	50	in	Number		
yt_Conic	(xt-start)*r/length	50	in	Number		
yt_Elliptic	r*sqrt(1-((length-(xt-start))^2/length^2))	50	in	Number		
yt_HAACK	r*sqrt(pi()/180*theta-sin(2*theta)/2+C*(sin(theta))^3)/sqrt(pi())	50	in	Number		
yt_OGIVE	sqrt(ro^2-((xt-start)-length)^2)+(r-ro)	50	in	Number		
yt_Parabolic	r*(2*((xt-start)/length)-K*((xt-start)/length)^2)/(2-K)	50	in	Number		
yt_Power	r*((xt-start)/length)^n	50	in	Number		
zt	0	0	in	Number		

Type: Number — Length
Name: — in
Formula:

Figure 4-15: NX6 Expressions set by Fluent Expression Maker Tool

4.1.11 CFD Sheet/Solid Maker

This file generates a sheet that is an axisymmetric representation of the effected air around the rocket body. This code was written in C++ for use in NX6. Stated simply, this program removes the revolved cross-sectional shape of the rocket from the sheet, leaving behind dimensional parameters that build an associative link to the rocket body geometry. Only half the model is needed for the 2D case when symmetry is applied. The cone is not yet trimmed from the sheet. This is again due to the difficulty of programmatically making law curves, see Figure 4-16. The air solid size can be controlled and increased as needed for the type of analyses being performed. Because a farfield condition is being used here, this size should be sufficient and will be tested and discussed in more detail later.

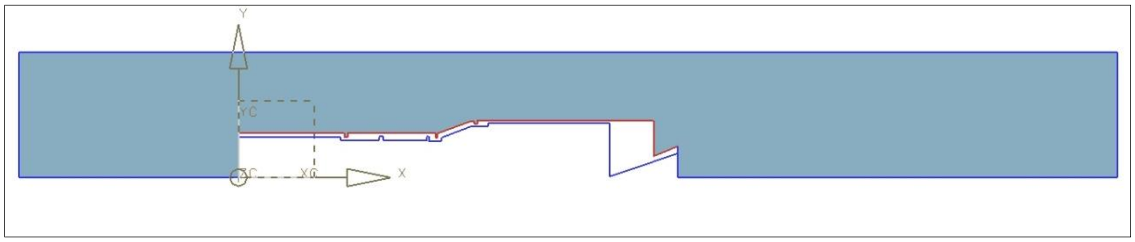


Figure 4-16: NX6 Sheet for Fluent Mesh

4.1.12 Parametric CFD Cone Tool

This tool takes the sheet made in the previous step and attaches a subtracted parametric cone that now makes the sheet easily updatable for the series of analyses. This is done in NX6 using the macro journaling option. This equation represents the exterior shape of the nose cone, see Figure 4-17.

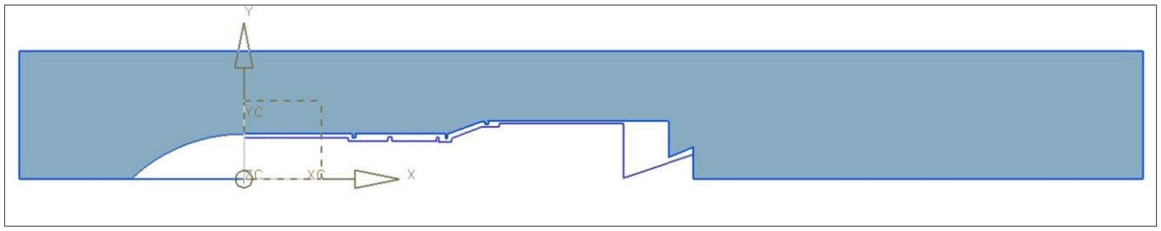


Figure 4-17:NX6 Sheet with Parametric Nose Cone Attached

An initial radius is also applied on the front of the cone for a desired length to make it so it can be manufactured and less prone to melting on the tip. This is another input to the NX6 equations that can be updated to modify the cone. This ensures that no aerodynamic anomalies occur as described by Khalid et al. “Very high temperatures are generated near the leading edges of hypersonic projectiles, such as at the tip of the nose or the upstream surface of the fins, which may cause gas dissociation effects in the flow and other structural deformations that result in aerodynamic implications. The intense temperatures near the leading edges can cause material melting and induce bluntness, which is often asymmetric.” (Khalid 2003)

A few additional surfaces are also added here to help with the mesh creation in Fluent. One that fills the total rocket cavity and another that follows below the center axis of the rocket for the entire length of the preparatory CFD Sheet Model. Both a small tip radius and the additional sheets are included as an example in Figure 4-18.

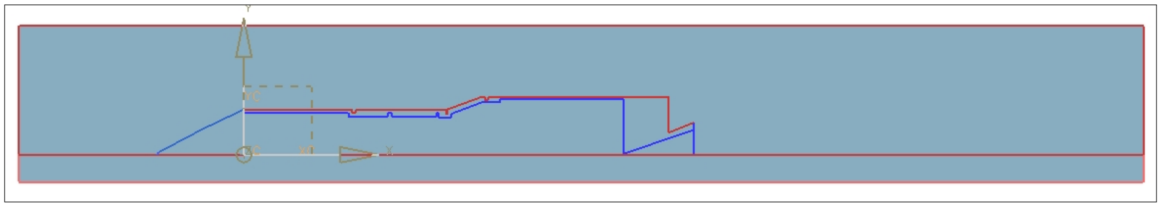


Figure 4-18: Example Sheet with Basic Nose Cone Ready for Fluent

4.1.13 Parametric Fluent Mesh

Because the OML solid made earlier has all the equations of the nose cone types built in, the sheet is fully parametric and can now be updated manually or programmatically. This sheet is then given a 2D triangular ANSYS mesh as shown in Figure 4-19. However, this is done in NX6 using the simulation tools. The triangle elements were found to be the most effective for the supersonic flights tested although quad elements can also be used. Better, more refined meshes were created by using a weighting system in NX6 to apply more elements closer to the surface of the rocket. However, this was not always uniform and was excluded to keep consistency between the models. Consequently, a higher element density was used instead to maintain the fidelity (see the results section under Verification and Validation). This causes the process to be slower and could be an area of future research and improvement.

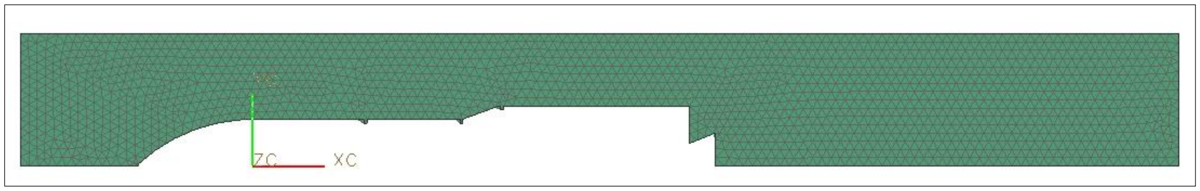


Figure 4-19: 2D Triangular Mesh of Simple Example

This 2D mesh surface can also be revolved to create a 3D representation of the rocket. One advantage of going to a 3D model is that it is easier to automate because the edges do not have to be defined and named. More importantly, moving to a 3D model allows the user to change the angle of attack, which alters the pressures on the rocket and therefore the structural integrity of the model. This is shown in Figure 4-20.

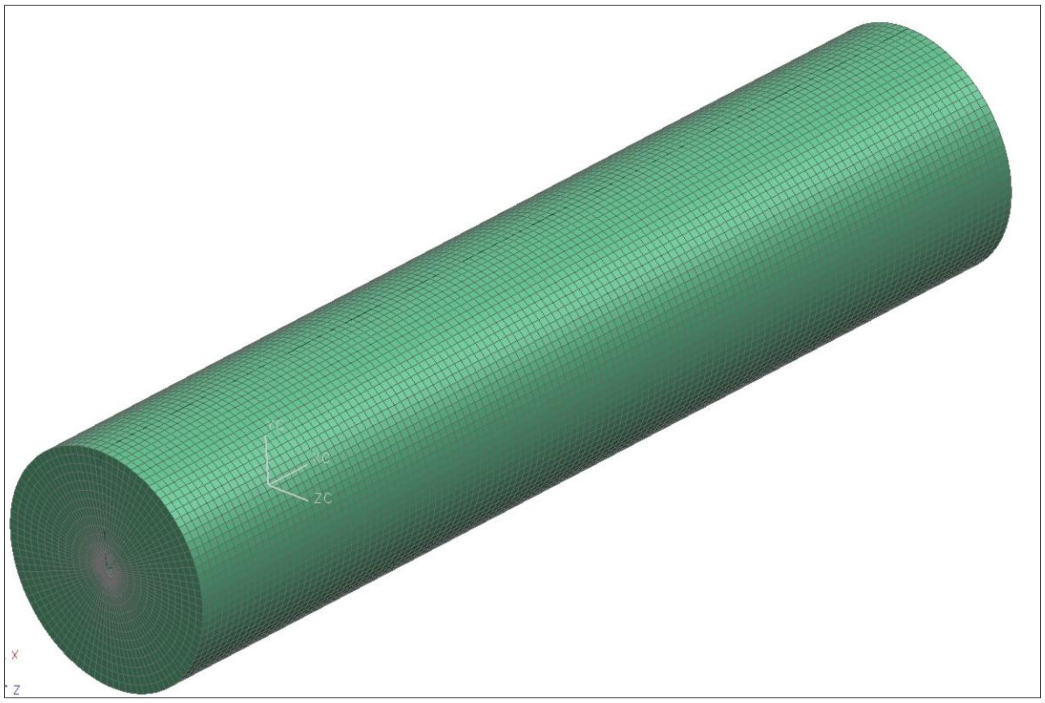


Figure 4-20: 3D Hexagonal Mesh of Simple Example

The interior of this model is hollow in the shape of the rocket, since this mesh is representative of the surrounding air. This mesh may be easier for the programmer to implement, but comes at a large cost in time for computations since the model is exponentially increased in element count, and depending on the model, can have a diminishing rate of return.

4.1.14 CFD Advanced Analysis

The selected options for Fluent can be seen in the following figures. Each analysis iterates tens of thousands of times until convergence of the residuals is made to within a defined tolerance. For the optimization, this must be automated. The automation of this section of code is done using Fluent journal scripts. The same steps written in the journals can be performed interactively using the Fluent GUIs. Each GUI choice that was selected to create a journal script will be displayed so that the interactive process can also be seen. The outputs, such as pressure and temperature, are sent to comma delimited text files (.csv extensions) for use later in the structural models. The descriptions here are for the 2D case but the same techniques can be applied to the 3D case for defining schemes and for setting up boundary conditions.

A *Second-Order Upwind* scheme was used; like any upwind scheme, it sets the value of any property at the boundary of a cell equal to the node directly upwind of it. This method is considered to be more accurate for high speeds (and high Péclet numbers) than the simpler *Central* scheme. Its order means that nodes are mathematically related not just to their neighboring nodes, but nodes that are two away in any direction.

The process for running Fluent manually is shown below. To show menu selections, the “>” symbol is used to denote the next level in the menu hierarchy. Variations will occur with version numbers and operating systems. This one is shown for windows Fluent 6.3.26, and is using the two-dimensional scheme for the axisymmetric rockets.

To start the program press Start > Programs > Fluent Inc > Fluent 6.3.26.

Select the *2ddp* option and then click *Run*. This tells the program to run using its double precision solver, which uses 64 bits instead of 32 bits, which creates higher precision but takes more memory. For more information and examples, see the Fluent infinite wedge example of Cornell University. (Bhaskaran 2002)

At this point we have a mesh from NX6 that has been exported as an ANSYS input mesh ".inp" file. This can be imported using File > Import > ANSYS > Input File and selecting the input file.

The boundaries have not yet been described correctly in NX6 for use in Fluent and now the trick is to create the boundaries correctly in Fluent without resorting to using another meshing program such as Gambit, TGrid or HyperMesh. In Figure 4-21, the desired mesh is shown in green with the unique boundary conditions required for solving shown in black, blue and yellow.

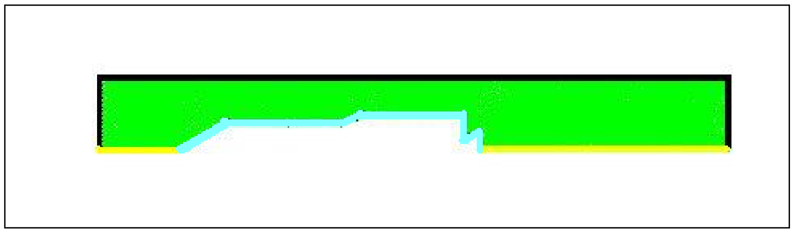


Figure 4-21: Desired Mesh with Required Boundary Conditions

Unfortunately, there is no way to simply define boundaries in Fluent. Usually a preprocessing tool such as Gambit, TGrid or HyperMesh would be used, but these are difficult programs to automate. There is, however, a way to change the type of boundaries in Fluent if they have previously been made. A default exterior boundary type exists for the black line which corresponds to the gas farthest from the model. This can serve perfectly as the *farfield* boundary which is needed in future sections. A default boundary type also exists between two meshes. By creating an extra mesh that is connected to the interior of the mesh (which is the exterior of the rocket) we create a boundary that can be described as a *wall* type, which is the blue in Figure 4-22. The *wall* type defines the rocket surface that air will flow over.

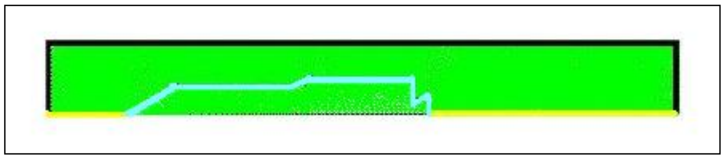


Figure 4-22: Additional Mesh for Wall Boundary Creation

Lastly, the axial boundary needs to be defined, which is displayed in yellow. The most accurate implementation discovered to handle this involves making a copy of the entire desired mesh about the center axis. This creates a boundary that connects in a straight line along the axial center. This method consistently defines the line correctly, both in front of and behind the rocket, with no misalignment or entanglement into the rocket interior mesh. These mesh additions are computationally added in the Parametric Fluent Mesh section above. This is also why some additional sheets were made Parametric CFD Cone Tool section. In Figure 4-23 the resultant mesh is shown.

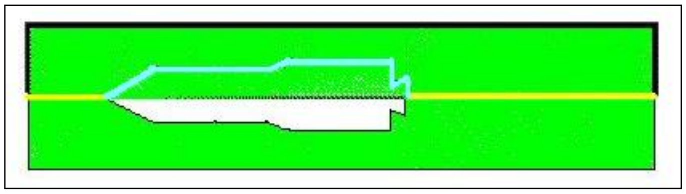
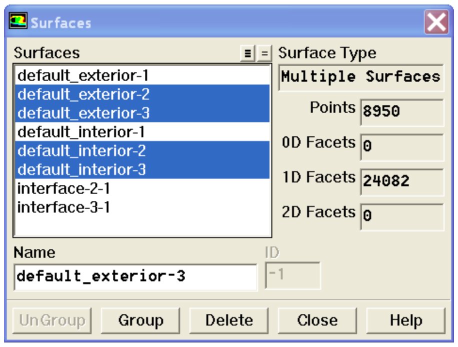


Figure 4-23: More Mesh Additions for Axial Boundary Creation

Now there are two extra meshes which are not desired for use in the Fluent run. This cost some time to create and some memory to store, but will luckily not be an addition to the optimization solver time because it can now be deleted. By going to Surface > Manage and selecting the extra meshes, the surfaces can be deleted as shown in Figure 4-24. In this same figure is shown the interface surfaces that are going to be used for the boundary conditions, which remain even after the deletion of one of the entities.



Surface Type: Multiple Surfaces
Points: 8950
0D Facets: 0
1D Facets: 24082
2D Facets: 0
Name: default_exterior-3
ID: -1

Figure 4-24: Surface Manager in Fluent

In Figure 4-25, the three boundary conditions are displayed. The black one is the default-exterior-1 and will be renamed “farfield” for the remainder of the thesis. The green is the default_exterior-2 and represented the interface between the mesh flow and the rocket mesh inside. It will be named “wall” for the remainder of the thesis. The orange (which actually shows up green in Fluent but has been changed for a clear division between the boundaries) represents the interface between the desired mesh and its mirrored copy about the axis and will be called “axis2” for the remainder of this thesis.

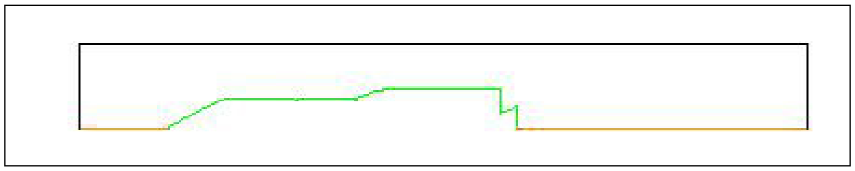


Figure 4-25: Boundaries Described in Fluent

There are two solver options using different equations: Pressure-Based and Density-Based. The documentation for Fluent discusses the time for selecting each of these options.

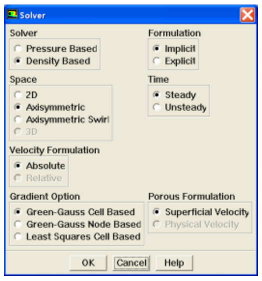
“Historically speaking, the pressure-based approach was developed for low-speed incompressible flows, while the density-based approach was mainly used for high-speed compressible flows. However, recently both methods have been extended and reformulated to solve and operate for a wide range of flow conditions beyond their traditional or original intent.

In both methods the velocity field is obtained from the momentum equations. In the density-based approach, the continuity equation is used to obtain the density field while the pressure field is determined from the equation of state.

On the other hand, in the pressure-based approach, the pressure field is extracted by solving a pressure or pressure correction equation which is obtained by manipulating continuity and momentum equations.” (Fluent Inc. 2006)

Both can do a decent job but the selection was made to go with the density-based solver based on the work of Bhaskaran et al. “Since we expect an oblique shock for our problem and the density-based solver is likely to resolve the shock better, let's pick this solver.” (Bhaskaran 2002)

Next, the journal selects *Density Based* in the Solver menu as shown in Figure 4-26.



Solver

Solver: Pressure Based [✓ Density Based]

Formulation: [✓ Implicit] Explicit

Space: 2D [✓ Axisymmetric] Axisymmetric Swirl 3D

Time: [✓ Steady] Unsteady

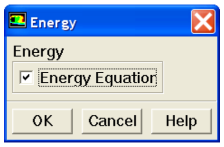
Velocity Formulation: [✓ Absolute] Relative

Gradient Option: [✓ Green-Gauss Cell Based] Green-Gauss Node Based Least Squares Cell Based

Porous Formulation: [✓ Superficial Velocity] Physical Velocity

Figure 4-26: Fluent Settings

In a compressible flow model, the continuity and momentum equations are coupled to the energy equations, and so the energy equation option is selected for this problem. Turning on the energy equation is done by going to Define > Models > Energy and checking the box next to *Energy Equation* and then by clicking *OK* as shown in Figure 4-27.



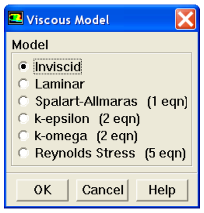
Energy

Energy: [✓ Energy Equation]

Figure 4-27: Energy Equation Option

If the only interest in the run was pressure results, the *Inviscid* option would be selected, which by ignoring the viscosity of the fluid, allows for a quicker and clearer cut result. However, if the temperature results are important, the Laminar option is selected to include the increased complexity of the air and allow for resultant heat to occur at the body of the model. Care must be taken in the selection based on what is expected. Khalid et al emphasized this point. "Even control surfaces at moderate hypersonic speeds close to Mach numbers of 7 experience heat-transfer effects that can cause fatal structural deformations, leading to total loss of control." (Khalid 2003) This is perhaps one of the greatest required trade-offs of this thesis.

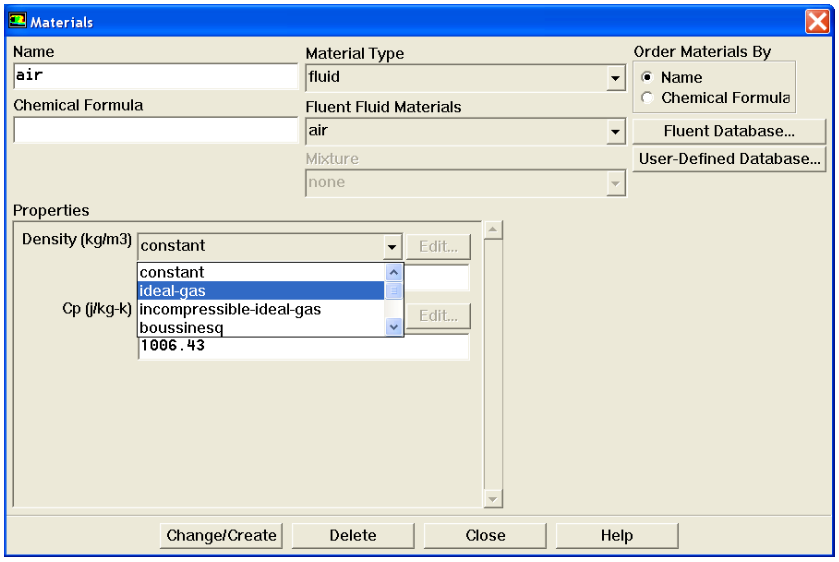
Through experimentation, the *Inviscid* option improved the accuracy of the flow results. However, loss of accuracy to the heat-transfer effects cannot be overlooked; in this case the *Laminar* option was selected.



Model: [✓ Inviscid] Laminar / Spalart-Allmaras (1 eqn) / k-epsilon (2 eqn) / k-omega (2 eqn) / Reynolds Stress (5 eqn)

Figure 4-28: Inviscid or Laminar Options

The next step is to define the materials used. This is done by clicking Define > Materials. The air the rocket is going through is chosen next under Fluent Fluid Materials. Then the Density option is set to *ideal-gas* as shown in Figure 4-29.



Name: air
Chemical Formula:
Material Type: fluid
Fluent Fluid Materials: air
Mixture: none
Order Materials By: [✓ Name] Chemical Formula

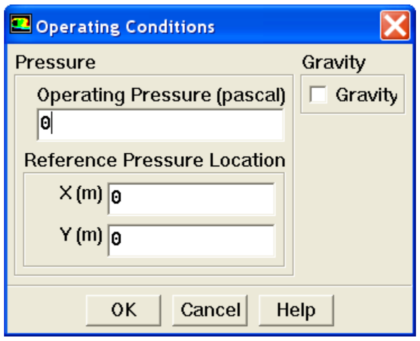
Properties
Density (kg/m3): constant
constant
[✓ ideal-gas]
incompressible-ideal-gas
boussinesq
Cp (j/kg-k): 1006.43

Figure 4-29: Material Type Options

This sets up the program to use the ideal gas law which is shown here in Equation 4-1. It relates the pressure of the gas to the density and Mass in a way that is independent from the amount of gas being considered. In the equation, the p is the pressure, the ρ is the density, and the R is the ideal gas constant. The M is the molar mass of the material, and the T is the Temperature.

$$p = \rho \frac{R}{M} T \tag{4-1}$$

The operating conditions are defined next by selecting Define > Operating Conditions. The Operating Pressure is set to zero as shown in Figure 4-30. This is used by Fluent to calculate the absolute pressure by adding the gauge pressure to it, as shown in Equation 4-2.



Operating Pressure (pascal) 0

X (m) 0

Y (m) 0

☐ Gravity

Figure 4-30: Operating Conditions

$$p_{abs} = p_{op} + p_{gauge} \tag{4-2}$$

The gauge pressure is used internally by Fluent to handle round-off errors and the absolute pressure is only calculated when required.

"Round-off errors occur when pressure changes *Δp* in the flow are much smaller than the pressure values *p*. One then gets small differences of large numbers. For our supersonic flow, we'll get significant variation in the absolute pressure so that pressure changes *Δp* are comparable to pressure levels *p*. So we can work in terms of absolute pressure without being hassled by pesky round-off errors." (Bhaskaran 2002) Setting the Operating Pressure to zero is the way to have Fluent work in terms of the absolute pressure and is the best option for the rocket simulations.

The next step is to define the boundary conditions. This is done using Define > Boundary Conditions and setting the "farfield" zone to be a pressure-far-field boundary type.

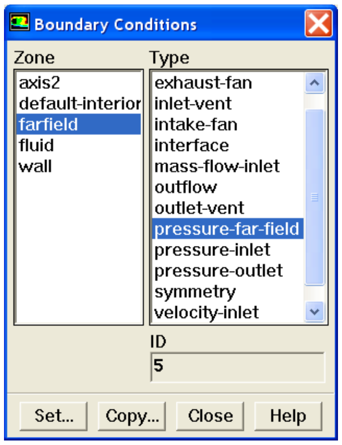
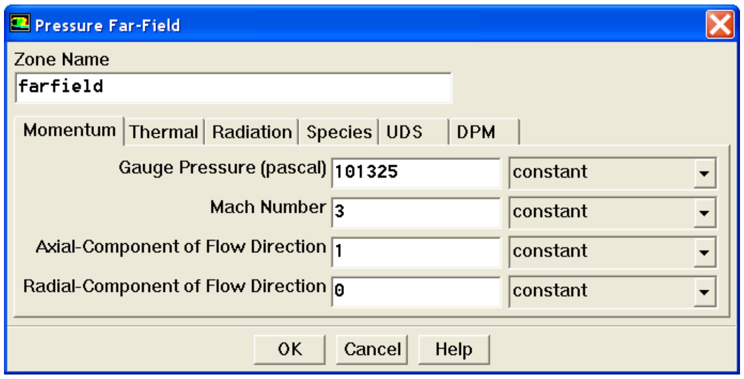


Figure 4-31: Boundary Conditions

The next step is to set the Gauge Pressure (as described in Equation 4-2) to be the desired pressure which is representative of the cruise altitude. In this case the pressure is at 101325 Pa, which is sea level. This is accepted by clicking the Set button. Also, the *Mach Number* has been set to 3. The rocket's drag is strongly influenced by these two values. As altitude changes, so does the air viscosity, speed of sound and air density. All of these are intertwined with the resultant Mach number value. Under *X-Component of Flow Direction*, the value of 1 has been inserted which states the boundary condition named "farfield" will have a flow in the x-direction.

For this example, the ambient temperature is initially set to 300K using the *Thermal* Tab.

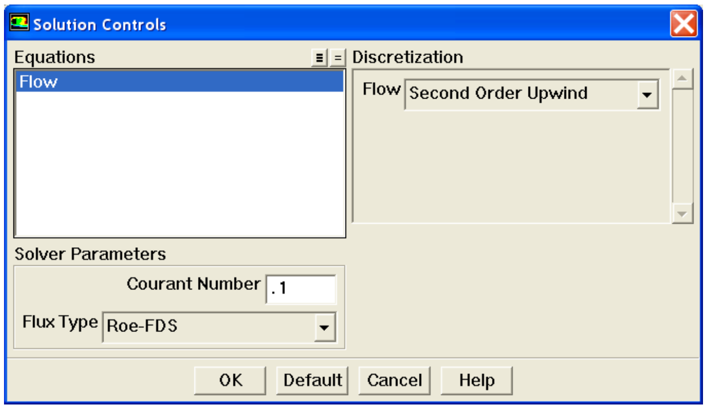


Zone Name: farfield

Gauge Pressure (pascal) 101325 constant
Mach Number 3 constant
Axial-Component of Flow Direction 1 constant
Radial-Component of Flow Direction 0 constant

Figure 4-32: Pressure Far-Field Settings

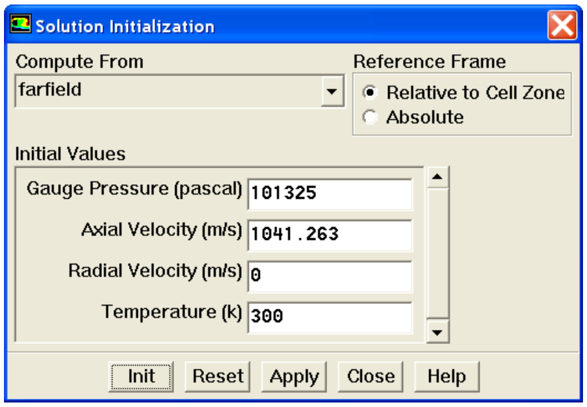
Similarly, the other boundaries must be set to the proper types. The boundary named "wall" is of course set to be a wall type and the one named "axis2" is set to be an axis type. In time it can be helpful to change the *Courant Number* shown in Figure 4-33 from a lower value to something higher as it becomes more stable, but this can lead to instability so for this project it has been left low (at 0.1).



Equations: Flow
Flow: Second Order Upwind
Courant Number .1
Flux Type Roe-FDS

Figure 4-33: Solution Controls

A Second Order Upwind scheme is set as shown in Figure 4-34 by clicking Solve > Initialize > Initialize. Initialization defines or sets initial values for every node in the computational domain. This scheme assumes that if flow is going from left to right that the right side of each element can be approximated as the value of the left side. When the elements are small enough, this technique allows you to traverse the domain more quickly and with enough iteration, it still achieves a good accuracy. The solution is given some initial values, which are best computed from the "farfield" boundary condition that has been set earlier.


Solution Initialization

Compute From: farfield

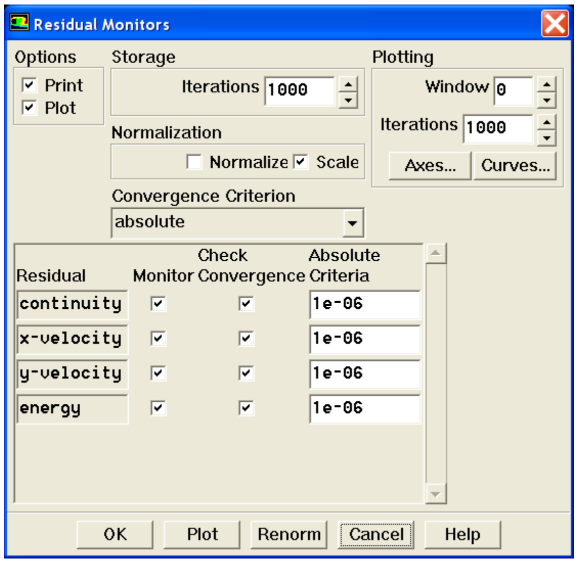
Reference Frame: [● Relative to Cell Zone] Absolute

Initial Values

Gauge Pressure (pascal)	101325
Axial Velocity (m/s)	1041.263
Radial Velocity (m/s)	0
Temperature (k)	300

Figure 4-34: Solution Initialization

The convergence criteria are set by clicking Solve > Monitors > Residuals, as shown in Figure 4-35. The criteria to check for convergence are set to a very small value, namely 1e-06. This means that the solution will not stop until the residuals go down to that range when iterating. For the interactive version, the check boxes for *print* and *plot* can be set so that it can be seen how far the results have come. The iteration is then run by clicking Solve > Iterate, then selecting the number of iterations, then by clicking OK. All of these steps have been automated and are stored in the journal file for use by the optimization to create solutions.



Options: ✓ Print, ✓ Plot

Storage: Iterations 1000

Normalization: ☐ Normalize ✓ Scale

Convergence Criterion: absolute

Plotting: Window 0, Iterations 1000

Residual	Monitor	Check Convergence	Absolute Criteria
continuity	✓	✓	1e-06
x-velocity	✓	✓	1e-06
y-velocity	✓	✓	1e-06
energy	✓	✓	1e-06

Figure 4-35: Residual Monitors

4.1.15 Section Maker

This section of the code begins the structural model creation. The OML Slicer Dicer Tool and Parametric Mid Cone Tool both have stored serialized data sets of information that can now be used to create the structural model. The code for retrieving the stored data is listed here in Figure 4-36. This allows the unknown quantities of data to be reconstituted into a useable format without the need for parsing through a text file to determine all the information again. As can be seen in the code, if the file is successfully found, the process then iterates looking for elements of the C_Sect_Info class as described in Figure 4-8. It is printing the mass portion of the structure as verification of retrieval of each of the C_Sect_Info elements.

```
if (exists("C:\\Section_Info.bin"))
{
        std::ifstream in("C:\\Section_Info.bin", ios::binary);
        C_Sect_Info temp_mass_r(1.0);
        while (!in.eof())
        {
                int blahsss = sizeof(temp_mass_r);
                in.read((char*)&temp_mass_r, blahsss);
                the_masses.push_back(temp_mass_r);
                cout<<"Mass t-Object has value "<< temp_mass_r.Mass
<<endl;
        }
        the_masses.pop_back();
}
else
{
        cout<<"error: no mass file found"<<endl;
        return 0;
}
```

Figure 4-36: Code for Retrieving the Section Info Data

Initially, the plan was to make a 2D beam model (in Nastran) like the one in Figure 4-37 to recreate the fairing and the rocket body. This was to be created as a sufficient analytical result for the structural integrity of the rocket and as an initial estimate of the nose cone, which would then be refined in another study. The refinement was to be a separate, higher detail 3D sheet model that focused on the Fairing alone as in Figure 4-38. The forces found in the initial beam model were to be used as a means of determining the loads that the rocket body itself (being propelled up) is placing on the 3D sheet fairing model.

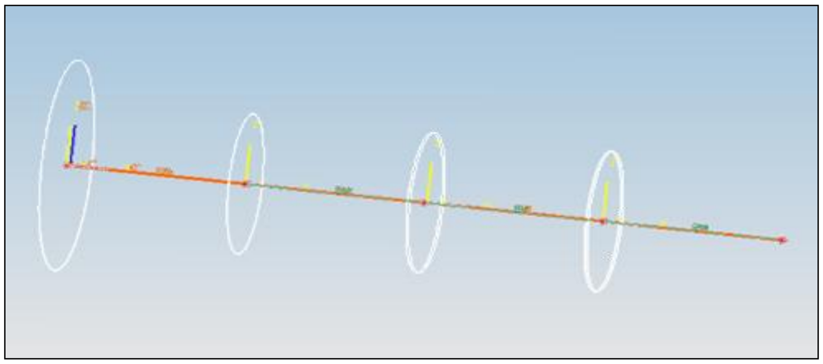


Figure 4-37: Beam Model Example

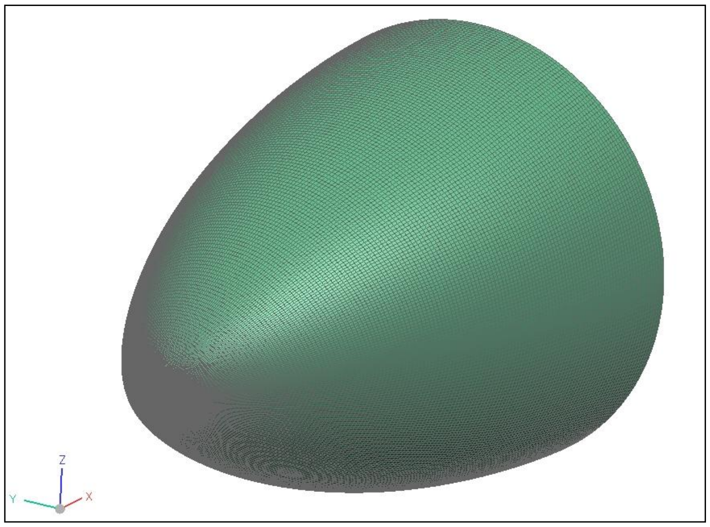


Figure 4-38: Example Rocket Fairing Structural Mesh (HAACK)

These loads could then be input to a more detailed sheet model of just the fairing itself. Upon further research, it was determined that these two processes could be done better together and instead of creating a beam model of the fairing, the detailed sheet mesh was attached directly to the rocket body beam model as shown in Figure 4-39.

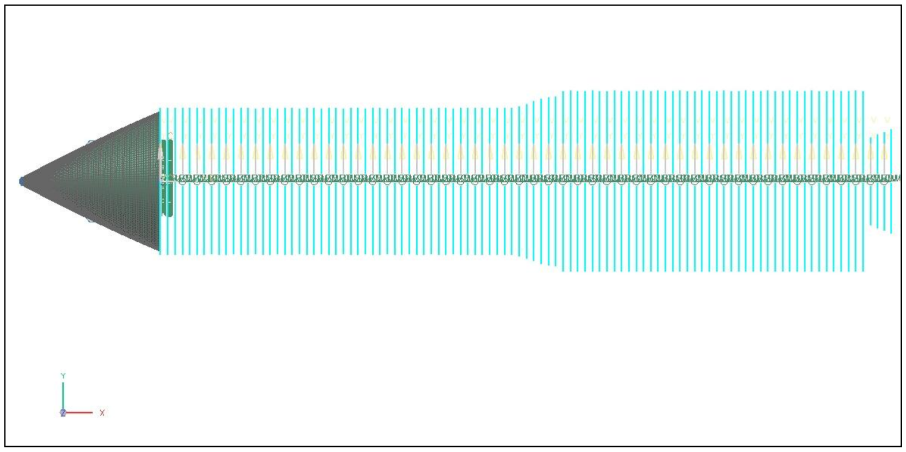


Figure 4-39: Section Maker Model in NX6 Uniting 2D Mesh and Beam Model

By doing these two analyses simultaneously, the time was reduced and the difficulty of transferring data from one model to another was eliminated. Also, there was no need to make an inferior fairing beam model that would provide less accurate geometry and forces before proceeding to the higher fidelity model. The fairing sheet is given an input density, meshed, and is still parametric so its size can be changed programmatically and interactively and the mesh will update. This includes the ability to change from one type of fairing to another. This process needs only to edit the original expressions, and then to run an update command for the model. This applies a mesh again to the new surface as it was previously applied to the first surface. This is much faster than running this Section Maker tool over and over again in the loop.

The meshing surface also has a radius control (as did the CFD model) for more accurate representation of the tip. The sheet is meshed, using a quad mesh that is controlled based on a pre-specified density.

The beam element information was gathered earlier in the OML Slicer Dicer tool and is now applied here. Each of the beam elements has its own Ixx, Iyy, Izz, CG, Moment, Material type, as well as two Inner Radii, and two Outer Radii associated with it.

4.1.16 Apply Forces

The next task is to apply the forces and pressures, as created previously in Fluent and exported to the text files (.csv files). These are applied around and along the fairing and the rocket body beam model. It is difficult to display a pressure that is applied around the rocket body when the beam model is only displayed as a point. NX6 displays the pressures applied around the beam as an arrow with a magnitude. The stresses are calculated based on these pressures. This is done using NX6 and C++. This is shown here in Figure 4-40.

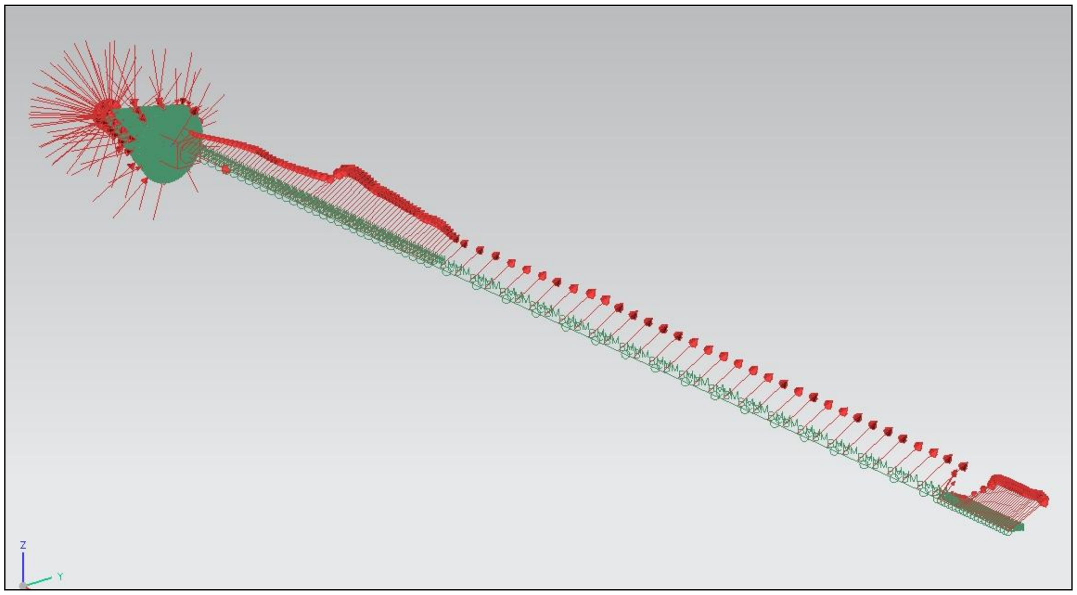
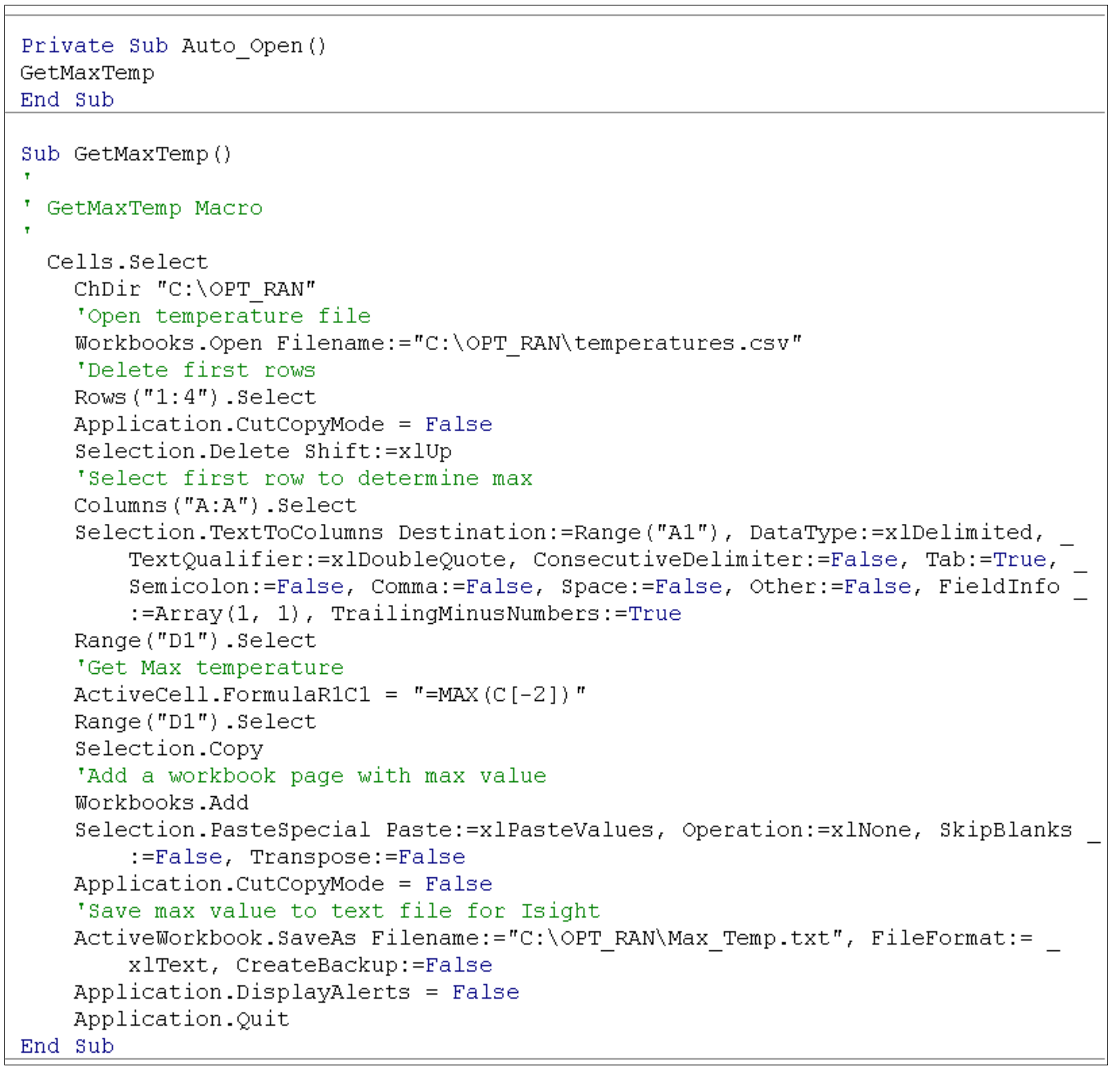


Figure 4-40: NX Nastran Model with Forces Applied

4.1.17 Apply Temperatures

Because a more accurate description of the fairing is desired, greater accuracy can be maintained by applying the temperatures to the structural elements of the fairing sheet model. The temperatures were determined previously in Fluent and exported to .csv files. The temperatures in this file can also serve as an immediate pass or fail test to determine if the material will melt or has hit a critical pre-specified cutoff point. The text file is parsed and the maximum temperature location is determined. This functionality is performed using Visual Basic for Applications (VBA) as shown in Figure 4-41. This value is compared against the user defined iSIGHT-FD limits.



```
Private Sub Auto_Open()
GetMaxTemp
End Sub

Sub GetMaxTemp()
'
' GetMaxTemp Macro
'
  Cells.Select
    ChDir "C:\OPT_RAN"
    'Open temperature file
    Workbooks.Open Filename:="C:\OPT_RAN\temperatures.csv"
    'Delete first rows
    Rows("1:4").Select
    Application.CutCopyMode = False
    Selection.Delete Shift:=xlUp
    'Select first row to determine max
    Columns("A:A").Select
    Selection.TextToColumns Destination:=Range("A1"), DataType:=xlDelimited, _
        TextQualifier:=xlDoubleQuote, ConsecutiveDelimiter:=False, Tab:=True, _
        Semicolon:=False, Comma:=False, Space:=False, Other:=False, FieldInfo _
        :=Array(1, 1), TrailingMinusNumbers:=True
    Range("D1").Select
    'Get Max temperature
    ActiveCell.FormulaR1C1 = "=MAX(C[-2])"
    Range("D1").Select
    Selection.Copy
    'Add a workbook page with max value
    Workbooks.Add
    Selection.PasteSpecial Paste:=xlPasteValues, Operation:=xlNone, SkipBlanks _
        :=False, Transpose:=False
    Application.CutCopyMode = False
    'Save max value to text file for Isight
    ActiveWorkbook.SaveAs Filename:="C:\OPT_RAN\Max_Temp.txt", FileFormat:= _
        xlText, CreateBackup:=False
    Application.DisplayAlerts = False
    Application.Quit
End Sub
```

Figure 4-41: Excel Automatic Max Temperature Calculation

4.1.18 Run Nastran Analysis

This section runs the Nastran model as specified above and exports results of the stresses around and along the length of the FEA model. The results can be viewed in NX6 and evaluated manually. The stresses and weight will be used by iSIGHT-FD for optimization decisions.

4.1.19 Setup of iSIGHT-FD Optimization

This section encapsulates the processes already described and manages the overall inputs and output from each process. Doing this optimization on multiple linked computers made it difficult to control. Though one computer was in charge of the optimization, it meant it had to wait for the other computers' routines to finish before continuing based on the obtained data. A few steps are performed to prepare the inputs before the optimization begins as well as to ensure that the process continues without failing. First, any secondary computers or batch clusters should be initialized to be waiting for the Fluent input files. They are initialized by running the *0-Check_Fluent_WAITING.bat* on a windows box which runs a C++ executable (*Check_Fluent_File_In.exe*) to have set a loop to wait, sleep and check every so often for an input from the host computer. To run on a Linux computer or batch cluster, the *watchcommand.sh* shell script is started and, similarly, it waits and sleeps while waiting for new inputs to show up. When an input file arrives at the shared folder location, these programs initiate Fluent execution. The file that will flag readiness for Fluent execution is not put in until the fourth step in the optimization (the *Put File in OK (4)* step) is executed, but it is still important to start these processes on the other computers so that they are ready when the files arrive.

Next, the *Slicer Dicer (1)* step is used to run the Slicer Dicer tool on the desired model with the desired increments. Following this, the *OML Maker (2)* step is executed and the external OML surfaces are selected for the CFD Parametric Sheet Tool. This is the OML tool built specifically for this optimization that makes the rocket model mesh for Fluent. Once these steps are done, the optimization process shown in Figure 4-42 is begun.

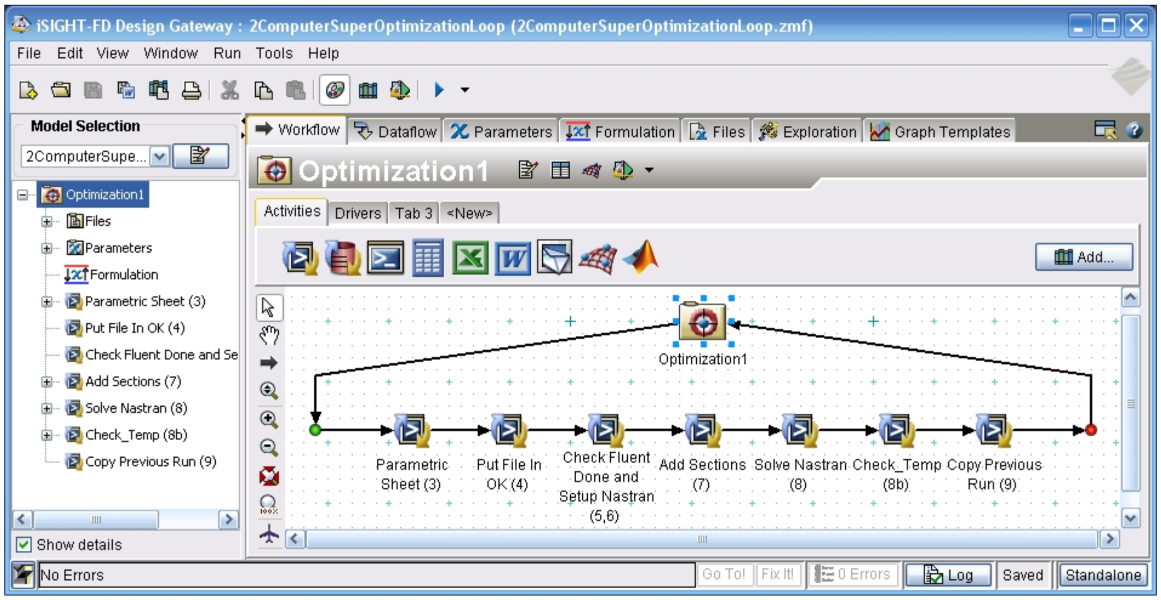
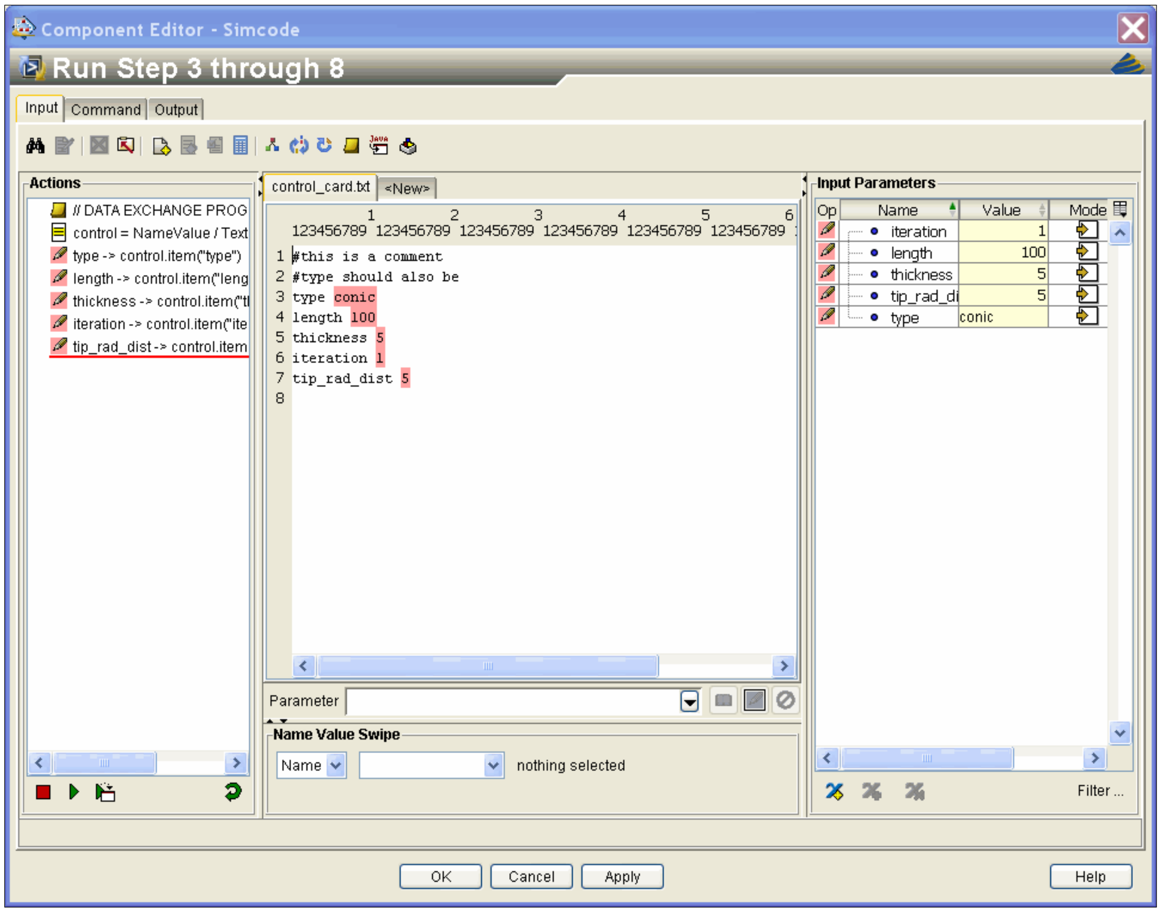


Figure 4-42: Optimization Scheme in iSIGHT-FD

The optimization tool is now run and the first step in the iteration, *Parametric Sheet (3)*, is run. This step creates a new run folder for this round of the execution. It runs the NX6 Journal File, which is named *make_cone_mesh_and_exportANSYS.macro.* This macro executes the *Expression_Iterator_NX6.dll* and then runs the journal steps for creating the law curve and linking the expressions. The values are controlled by the optimization and are written to the control_card.txt file for all the other steps to use. A generalized reduced gradient method was used that alternates the variables so that on each iteration only one is dependent and all the others are independent. This can be very time intensive. Time could possibly be improved through the use of Response Surface Methodology, but was not implemented in this thesis. Some common starting input control values are shown below in Figure 4-43. The macro then starts the *NX6_FLUENT_Parametric_Mesh.dll* which adds the nose cone to the sheet previously made by the OML tool and meshes the airsolid, the rocket and the mirror image of the airsolid for Fluent analysis.



```
// DATA EXCHANGE PROG
control = NameValue / Text
type -> control.item("type")
length -> control.item("leng
thickness -> control.item("t
iteration -> control.item("ite
tip_rad_dist -> control.item
```

```
#this is a comment
#type should also be
type conic
length 100
thickness 5
iteration 1
tip_rad_dist 5
```

Op	Name	Value	Mode
	iteration	1	
	length	100	
	thickness	5	
	tip_rad_di	5	
	type	conic	

Figure 4-43: Input Control Card for Parameterization Control

Next, the *Put File in OK (4)* step is run that informs the next computer that files are in and ready for the CFD Fluent run. This also begins the Check Fluent Done (5) step, which similarly waits and sleeps until a file is returned that flags to the optimization loops that the CFD results have all been copied back to the shared folder. Meanwhile, the other computer is running the Fluent program which deletes the non-important meshes, makes the boundary conditions and sets up the correct flow parameters. When the file is returned, the Check Fluent Done (5) step completes and the *Setup Nastran (6)* step is begun, which copies all files back to the optimization computer from the shared location and runs the *setup_nastran.macro*. This, in turn, executes the

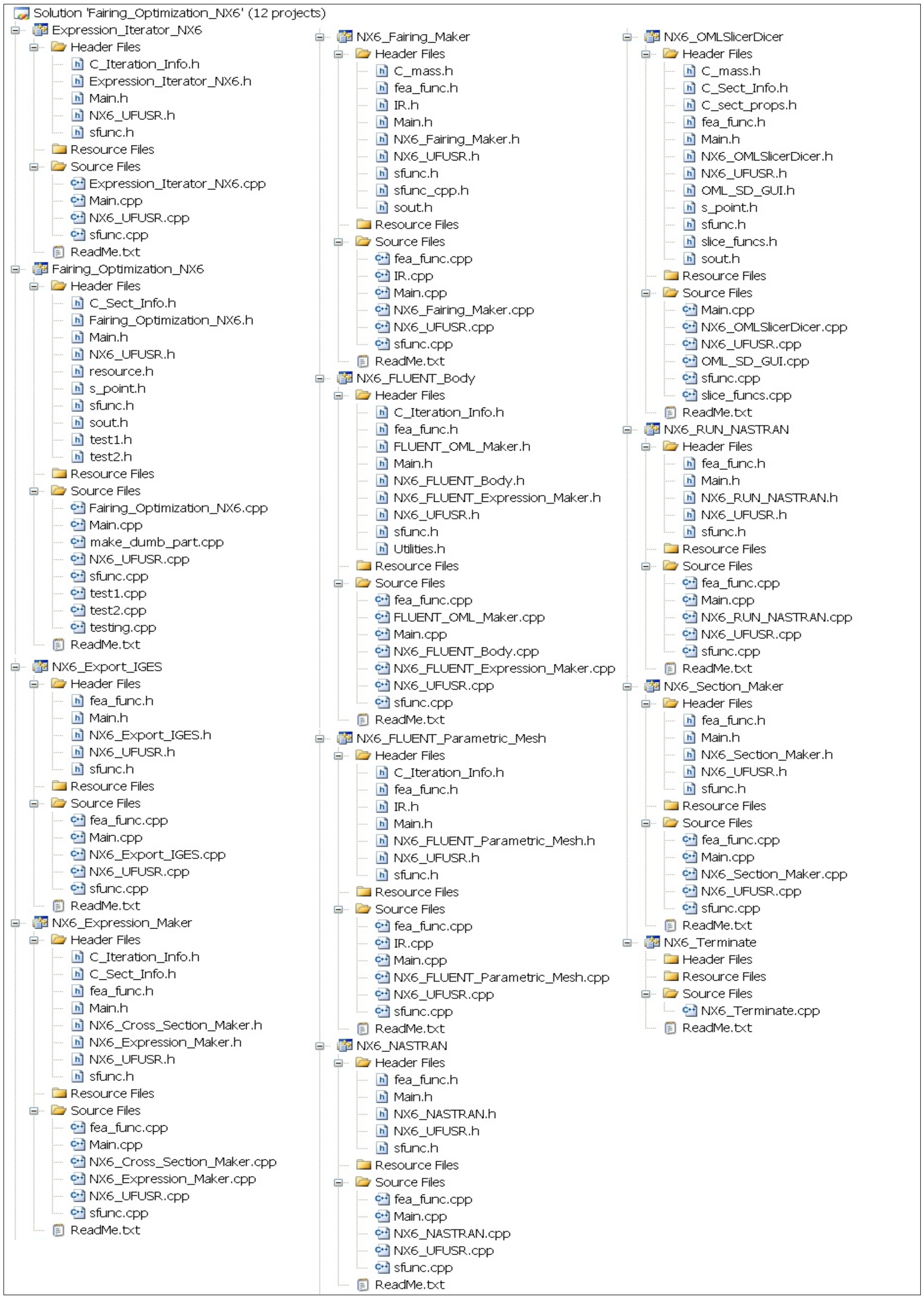
Expression_Iterator_NX6.dll, which has the journal steps for creating the law curve and linking the expressions. Then the *NX6_Fairing_Maker.dll* reads the Fluent outputs and creates the combined sheet cone and beam rocket model.

The *Nastran Solve (8)* step is then run, which inserts the pressures and sets up the solver conditions and begins the analysis. Following this, the *Check Temp (8b)* step is run, which runs the Microsoft Excel VBA program for determining the max temperature along the rocket.

The *Copy Previous Run (9)* step is then run, which zips all the files for storage and saves them into a folder where the zip files are all listed numerically. It then deletes the current run folder so that the next iteration can begin. All the runs are stored so that at the end of the optimization, a comprehensive history of all that has happened is catalogued. The program iSIGHT-FD itself also creates histories of the parameters being controlled to show how the optimization has progressed and how it arrives at its determined optimal values.

4.2 General Code Development

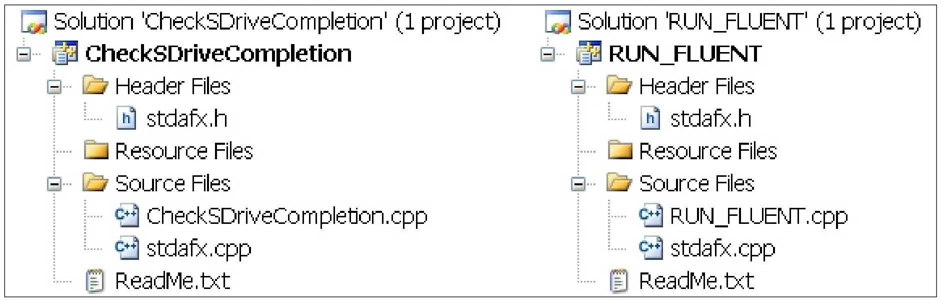
The main code developed for this project was created in Microsoft's Visual Studio .NET 2005 Professional and was written in C++. These main solution code files are listed here in Figure 4-44. As much code as was determined could be shared between each of these different projects has been. This sharing allows for bug fixes to propagate through all of the programs that use a function without having to change each one individually. Unfortunately, it can lead to difficulties, since changing one program can unknowingly lead to changes in another that may not have been intended. For this reason, extra care must be taken to ensure all the steps of the program remain intact.



```
Solution 'Fairing_Optimization_NX6' (12 projects)
  Expression_Iterator_NX6
    Header Files
      C_Iteration_Info.h
      Expression_Iterator_NX6.h
      Main.h
      NX6_UFUSR.h
      sfunc.h
    Resource Files
    Source Files
      Expression_Iterator_NX6.cpp
      Main.cpp
      NX6_UFUSR.cpp
      sfunc.cpp
    ReadMe.txt
  Fairing_Optimization_NX6
    Header Files
      C_Sect_Info.h
      Fairing_Optimization_NX6.h
      Main.h
      NX6_UFUSR.h
      resource.h
      s_point.h
      sfunc.h
      sout.h
      test1.h
      test2.h
    Resource Files
    Source Files
      Fairing_Optimization_NX6.cpp
      Main.cpp
      make_dumb_part.cpp
      NX6_UFUSR.cpp
      sfunc.cpp
      test1.cpp
      test2.cpp
      testing.cpp
    ReadMe.txt
  NX6_Export_IGES
    Header Files
      fea_func.h
      Main.h
      NX6_Export_IGES.h
      NX6_UFUSR.h
      sfunc.h
    Resource Files
    Source Files
      fea_func.cpp
      Main.cpp
      NX6_Export_IGES.cpp
      NX6_UFUSR.cpp
      sfunc.cpp
    ReadMe.txt
  NX6_Expression_Maker
    Header Files
      C_Iteration_Info.h
      C_Sect_Info.h
      fea_func.h
      Main.h
      NX6_Cross_Section_Maker.h
      NX6_Expression_Maker.h
      NX6_UFUSR.h
      sfunc.h
    Resource Files
    Source Files
      fea_func.cpp
      Main.cpp
      NX6_Cross_Section_Maker.cpp
      NX6_Expression_Maker.cpp
      NX6_UFUSR.cpp
      sfunc.cpp
    ReadMe.txt
  NX6_Fairing_Maker
    Header Files
      C_mass.h
      fea_func.h
      IR.h
      Main.h
      NX6_Fairing_Maker.h
      NX6_UFUSR.h
      sfunc.h
      sfunc_cpp.h
      sout.h
    Resource Files
    Source Files
      fea_func.cpp
      IR.cpp
      Main.cpp
      NX6_Fairing_Maker.cpp
      NX6_UFUSR.cpp
      sfunc.cpp
    ReadMe.txt
  NX6_FLUENT_Body
    Header Files
      C_Iteration_Info.h
      fea_func.h
      FLUENT_OML_Maker.h
      Main.h
      NX6_FLUENT_Body.h
      NX6_FLUENT_Expression_Maker.h
      NX6_UFUSR.h
      sfunc.h
      Utilities.h
    Resource Files
    Source Files
      fea_func.cpp
      FLUENT_OML_Maker.cpp
      Main.cpp
      NX6_FLUENT_Body.cpp
      NX6_FLUENT_Expression_Maker.cpp
      NX6_UFUSR.cpp
      sfunc.cpp
    ReadMe.txt
  NX6_FLUENT_Parametric_Mesh
    Header Files
      C_Iteration_Info.h
      fea_func.h
      IR.h
      Main.h
      NX6_FLUENT_Parametric_Mesh.h
      NX6_UFUSR.h
      sfunc.h
    Resource Files
    Source Files
      fea_func.cpp
      IR.cpp
      Main.cpp
      NX6_FLUENT_Parametric_Mesh.cpp
      NX6_UFUSR.cpp
      sfunc.cpp
    ReadMe.txt
  NX6_NASTRAN
    Header Files
      fea_func.h
      Main.h
      NX6_NASTRAN.h
      NX6_UFUSR.h
      sfunc.h
    Resource Files
    Source Files
      fea_func.cpp
      Main.cpp
      NX6_NASTRAN.cpp
      NX6_UFUSR.cpp
      sfunc.cpp
    ReadMe.txt
  NX6_OMLSlicerDicer
    Header Files
      C_mass.h
      C_Sect_Info.h
      C_sect_props.h
      fea_func.h
      Main.h
      NX6_OMLSlicerDicer.h
      NX6_UFUSR.h
      OML_SD_GUI.h
      s_point.h
      sfunc.h
      slice_funcs.h
      sout.h
    Resource Files
    Source Files
      Main.cpp
      NX6_OMLSlicerDicer.cpp
      NX6_UFUSR.cpp
      OML_SD_GUI.cpp
      sfunc.cpp
      slice_funcs.cpp
    ReadMe.txt
  NX6_RUN_NASTRAN
    Header Files
      fea_func.h
      Main.h
      NX6_RUN_NASTRAN.h
      NX6_UFUSR.h
      sfunc.h
    Resource Files
    Source Files
      fea_func.cpp
      Main.cpp
      NX6_RUN_NASTRAN.cpp
      NX6_UFUSR.cpp
      sfunc.cpp
    ReadMe.txt
  NX6_Section_Maker
    Header Files
      fea_func.h
      Main.h
      NX6_Section_Maker.h
      NX6_UFUSR.h
      sfunc.h
    Resource Files
    Source Files
      fea_func.cpp
      Main.cpp
      NX6_Section_Maker.cpp
      NX6_UFUSR.cpp
      sfunc.cpp
    ReadMe.txt
  NX6_Terminate
    Header Files
    Resource Files
    Source Files
      NX6_Terminate.cpp
    ReadMe.txt
```

Figure 4-44: Main Solution Showing Included Project Files

A few additional C++ coding projects are listed in Figure 4-45 which helped with the cyclic nature of iSIGHT-FD and with resetting of the adjoining computers to their ready states. Other programs involved included Java, Matlab, Visual Basic for Applications, and Batch Files. It was also easier to use journal or macro files from time to time in programs such as NX and Fluent when tasks were all but repetitive or could not be handled in C++ or other programming languages. All of these secondary languages played comparatively minor roles in the project and were only used when it was advantageous for the related program.

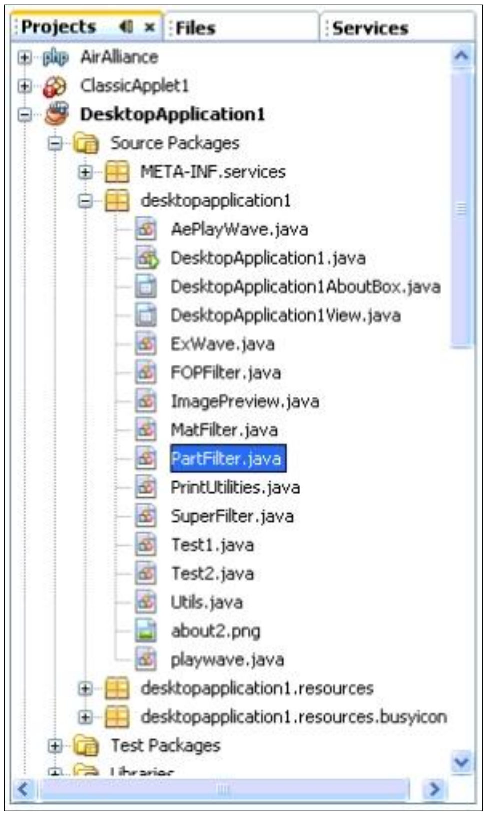


```
Solution 'CheckSDriveCompletion' (1 project)
  CheckSDriveCompletion
    Header Files
      stdafx.h
    Resource Files
    Source Files
      CheckSDriveCompletion.cpp
      stdafx.cpp
    ReadMe.txt
```

```
Solution 'RUN_FLUENT' (1 project)
  RUN_FLUENT
    Header Files
      stdafx.h
    Resource Files
    Source Files
      RUN_FLUENT.cpp
      stdafx.cpp
    ReadMe.txt
```

Figure 4-45: Additional Programming Solutions

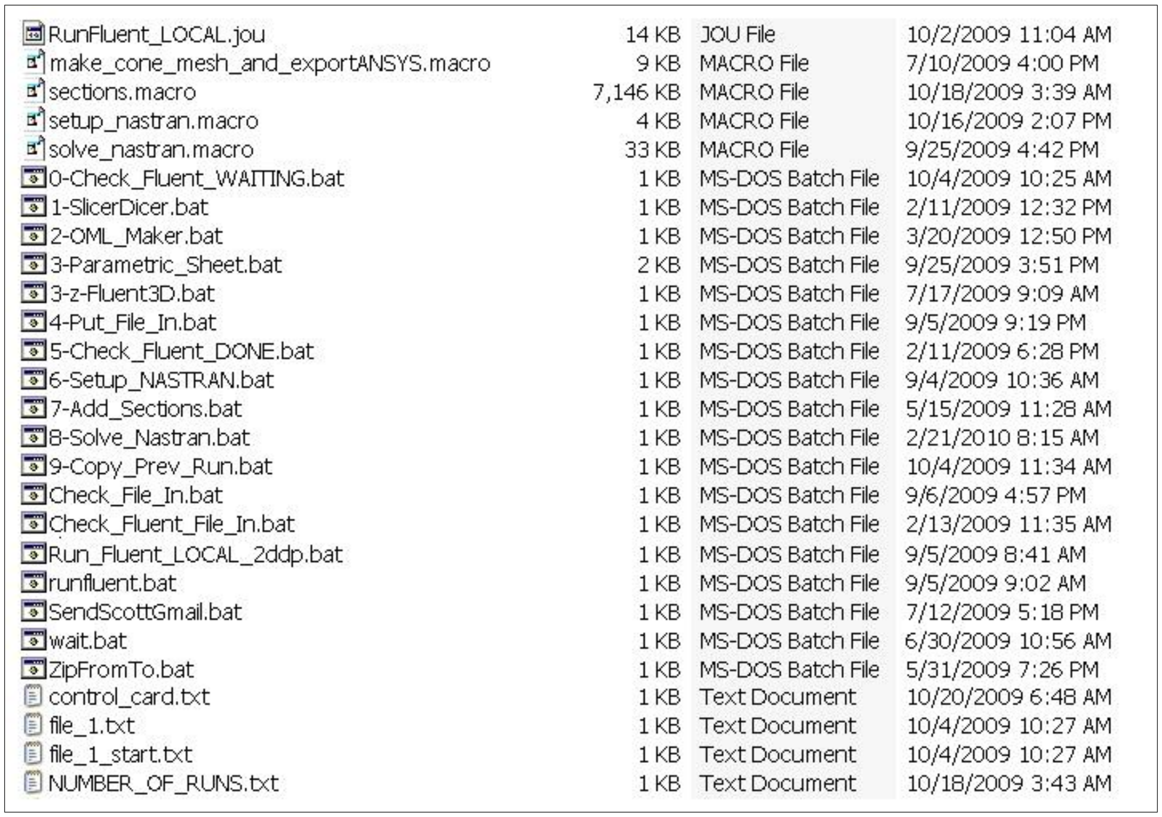
In Figure 4-46, the Java project created for the iSIGHT-FD Startup GUI is shown. This project was created with the iSIGHT-FD tool for ease of compatibility. It was determined unfortunately, that the version of Java that is compatible with NX6 is not compatible with the version of Java for iSIGHT-FD, so multiple types would have to be maintained for use with both systems.



```
Projects   Files   Services
AirAlliance
ClassicApplet1
DesktopApplication1
    Source Packages
        META-INF.services
        desktopapplication1
            AePlayWave.java
            DesktopApplication1.java
            DesktopApplication1AboutBox.java
            DesktopApplication1View.java
            ExWave.java
            FOPFilter.java
            ImagePreview.java
            MatFilter.java
            PartFilter.java
            PrintUtilities.java
            SuperFilter.java
            Test1.java
            Test2.java
            Utils.java
            about2.png
            playwave.java
        desktopapplication1.resources
        desktopapplication1.resources.busyicon
    Test Packages
    Libraries
```

Figure 4-46: Java Files Used for the iSIGHT-FD Startup GUI

Figure 4-47 shows the files used for actually controlling what is run in iSIGHT-FD. The majority of these control files are batch files and they, in turn, run the other macros and executables for the project. If the program is to be executed in NX, Visual Studio must export dll files upon compilation, which are run by the NX macro files. A few text files that store information for management between steps are also listed. Notice that the *sections.macro* file, which is generated by the C++ Cross Section Maker, is very large comparatively speaking to the other macros listed. This is because it cycles through each slice of the beam model and performs the tasks that C++ could not. Because the macro couldn't be cyclic in any way, the file gets very large.



RunFluent_LOCAL.jou	14 KB	JOU File	10/2/2009 11:04 AM
make_cone_mesh_and_exportANSYS.macro	9 KB	MACRO File	7/10/2009 4:00 PM
sections.macro	7,146 KB	MACRO File	10/18/2009 3:39 AM
setup_nastran.macro	4 KB	MACRO File	10/16/2009 2:07 PM
solve_nastran.macro	33 KB	MACRO File	9/25/2009 4:42 PM
0-Check_Fluent_WAITING.bat	1 KB	MS-DOS Batch File	10/4/2009 10:25 AM
1-SlicerDicer.bat	1 KB	MS-DOS Batch File	2/11/2009 12:32 PM
2-OML_Maker.bat	1 KB	MS-DOS Batch File	3/20/2009 12:50 PM
3-Parametric_Sheet.bat	2 KB	MS-DOS Batch File	9/25/2009 3:51 PM
3-z-Fluent3D.bat	1 KB	MS-DOS Batch File	7/17/2009 9:09 AM
4-Put_File_In.bat	1 KB	MS-DOS Batch File	9/5/2009 9:19 PM
5-Check_Fluent_DONE.bat	1 KB	MS-DOS Batch File	2/11/2009 6:28 PM
6-Setup_NASTRAN.bat	1 KB	MS-DOS Batch File	9/4/2009 10:36 AM
7-Add_Sections.bat	1 KB	MS-DOS Batch File	5/15/2009 11:28 AM
8-Solve_Nastran.bat	1 KB	MS-DOS Batch File	2/21/2010 8:15 AM
9-Copy_Prev_Run.bat	1 KB	MS-DOS Batch File	10/4/2009 11:34 AM
Check_File_In.bat	1 KB	MS-DOS Batch File	9/6/2009 4:57 PM
Check_Fluent_File_In.bat	1 KB	MS-DOS Batch File	2/13/2009 11:35 AM
Run_Fluent_LOCAL_2ddp.bat	1 KB	MS-DOS Batch File	9/5/2009 8:41 AM
runfluent.bat	1 KB	MS-DOS Batch File	9/5/2009 9:02 AM
SendScottGmail.bat	1 KB	MS-DOS Batch File	7/12/2009 5:18 PM
wait.bat	1 KB	MS-DOS Batch File	6/30/2009 10:56 AM
ZipFromTo.bat	1 KB	MS-DOS Batch File	5/31/2009 7:26 PM
control_card.txt	1 KB	Text Document	10/20/2009 6:48 AM
file_1.txt	1 KB	Text Document	10/4/2009 10:27 AM
file_1_start.txt	1 KB	Text Document	10/4/2009 10:27 AM
NUMBER_OF_RUNS.txt	1 KB	Text Document	10/18/2009 3:43 AM

Figure 4-47: Programming Control Files

For each one of the dll files created, the *NX6_UFUSR.cpp* code was used and is shown in Figure 4-48. This takes care of all the requirements for connecting to the NX6 libraries and sets up the initial error handling for all the NX6 dll files created. It is set up to reference a *Main.h* document and a *Main()* function, but each project has its own unique *Main.h* and *Main()* function that this references. From the unique *Main.cpp* files, the other codes and functions related to these projects are called respectively.

```
/***************************************************************************
**
** NX_UFUSR.cpp
**
** Description:
**     Contains Unigraphics entry points for the application.
**     This takes
***************************************************************************/
/* Include files */
#include "NX6_UFUSR.h"
#include "Main.h"
/***************************************************************************
**  Activation Methods
***************************************************************************/
/*  Explicit Activation
**      This entry point is used to activate the application explicitly, as in
**      "File->Execute UG/Open->User Function..." */
extern DllExport void ufusr( char *parm, int *returnCode, int rlen )
{
    /* Initialize the API environment */
    UgSession session( true );
    try
    {
        /* TODO: Add your application code here */
        time_t start_time;     time(&start_time);     //marks the time the program
starts
        Main();                       //This is the main that is unique to each project
        elapsed_time(start_time);             //tells how long the program takes to run
    }
    /* Handle errors */
    catch ( const UgException &exception )
    {
        processException( exception );
               std::cout<<"--- Error encountered ---"<<std::endl;
    }
        UgInfoWindow::write( "Ending");
}
extern int ufusr_ask_unload( void )
{
    return      (UF_UNLOAD_IMMEDIATELY //use unless you want to run something else from
the program);
}
/* processException
       Prints error messages to standard error and a Unigraphics
       information window. */
static void processException( const UgException &exception )
{
    /* Construct a buffer to hold the text. */
    ostrstream error_message;
    /* Initialize the buffer with the required text. */
    error_message << endl
                  << "Error:" << endl
                  << ( exception.askErrorText() ).c_str()
                  << endl << endl << ends;
    /* Open the UgInfoWindow */
    UgInfoWindow::open ( );
    /* Write the message to the UgInfoWindow.  The str method */
    /* freezes the buffer, so it must be unfrozen afterwards. */
    UgInfoWindow::write( error_message.str() );
    /* Write the message to standard error */
    cerr << error_message.str();
    error_message.rdbuf()->freeze( 0 );
}
```

Figure 4-48 NX6_UFUSR.cpp Code

4.2.1 Additional Code

This code has been the life blood of the thesis and has made it possible for the optimization of the fairing models. The code for the programs is lengthy enough to be difficult to include to its full extent. Still, an attempt to show the essence of their designs and uses has been made in the appendix for those who would like to see more about details of their executions (see APPENDIX A).

5 Results and Discussion of Results

This chapter shows verification and validation for the processes and then showcases the results obtained from optimizing multiple examples. The models generated for this thesis have been studied and sized for accuracy and for size maintenance. They have been compared to empirical data and are found to be representative of nose cones in use. Through these confirmations, this chapter will show how accuracy and time savings are achieved by adherence to the principles provided by these methods.

5.1 Verification and Validation

Verification, in its simplest words, is a demonstration of grid independence. In other words, verification ensures that the answer is not being poorly represented due to the mesh quality. To verify the mesh is to ensure that it is detailed enough to portray what the model truly represents. It is a test to see that what the software is setup to solve is truly solved and therefore, that necessary depth of detail is given. This does not ensure however that the software is taking into account all the possible influencing factors of the reality.

Validation is a test to ensure that all important factors are being accounted for in the model. This is done by comparing the model to empirical data. This can be more challenging because there would be no need for this project if empirical data already existed for the problem specified. By finding similar cases that have been done in the past and mimicking them, the resultant accuracy can be used as a measure of confidence in similar models.

As a simple example of these two tests, imagine a triangle being represented by a number of squares. One large square is a poor approximation of a triangle, but a billion small squares arranged in the shape of a triangle is a much better approximation. If the triangle is to be represented on a computer screen, a billion squares might be more representation than there are pixels on the screen. Perhaps, unbeknownst to the user, a triangle on the computer screen of this size and shape can at most be displayed as one million squares. This is the capability level of your screen and therefore the closest accuracy that is obtainable. Say it is also determined that the steps from those squares that represent a straight edge are more detail than the human eye can detect. Then if this was the goal, an adequate level of model can be achieved with merely a million squares on this device. Still, the human eye may not detect even a half million squares as being a poor representation. So if a model of a million squares (or even a half million) is perfectly adequate, why create a model that is a billion squares, which uses up excess memory and takes longer to create? The answer may be that the computer and user do not know what the human eye will be able detect and therefore the user may play the “safe side” of the fence.

The job of verification here is to ensure that a sufficient representation (the detail the human eye can detect) has been achieved, and also not to do more than is required. A common method of verification is to double your quantity and compare. So if, at first, your model had one thousand squares to then to try two thousand and compare. If a change can be seen, the model is not accurate enough. Looping through this method until the result is unchanged from the previous run (at least to the level desired) gives a closer acceptable approximation. Based on the numbers provided above, after ten iterations of doubling, this example would use 512,000 boxes to accurately represent the triangle. The process to determine the number of boxes is shown in Figure 5-1.

Model #	Verify if Visible Change
1.) 1,000 boxes	
	Change
2.) 2,000 boxes	
	Change
3.) 4,000 boxes	
	Change
4.) 8,000 boxes	
	Change
5.) 16,000 boxes	
	Change
6.) 32,000 boxes	
	Change
7.) 64,000 boxes	
	Change
8.) 128,000 boxes	
	Change
9.) 256,000 boxes	
	Change
10.) 512,000 boxes	
	No Change (Accept above)
11.) 1,024,000 boxes	

Figure 5-1: Verification Doubling Example for Triangle Approximated as Squares

Validation, with respect to the example above, might be finding a person who had determined that 500,000 boxes were all it took to accurately represent a triangle. It would still be important to verify these findings (by doubling and checking at one million boxes, and halving to check at 250,000 boxes). However, the trouble of multiplying by two from 1,000 to 512,000 would not have been required with the validation, and the model would be even more concise than verification provided alone. However, this example does not give validation its full value. Many situations are not as clear cut as a simple eye examination for accuracy. Things, such as a pressure, calculated on a computer need a reference to the real world to ensure that the verification values are not converging to a position due to a shortcoming in their model representation. Validation is what ties computational models to the real world and is therefore a key component to determining model accuracy.

Confidence in a model comes from both verifying and validating the models to help confirm that accurate and time efficient results have been achieved. In the following sections, verification and validation are given for the rocket models.

5.1.1 Fluent Verification

An example case was performed specifically on the conical section of the rocket to ensure that accurate results were being achieved. Meshes with different air volume size, cell shape, and fineness were used in order to establish grid independence. The first issue to resolve was how large a grid was needed to one, capture relevant flow information and two, to accurately describe the problem domain. While the lower boundary is fixed at the cone axis, the top boundary was originally set a distance away from the cone edge (equal to about two cone diameters away for a 30º cone). This case was compared to one where that boundary was moved closer, to about one cone diameter away. The cases yielded virtually identical results so it was determined that the smaller window size in the second case is acceptably large to model flow over the cone. Comparisons were performed to determine if the shockwave could penetrate the "farfield" condition without influencing results; the results remained unchanged.

Various grid cell sizes were also examined. For each cone type, the grid was refined until two grids whose cell widths differed, on average, by a factor of two were found to produce virtually the same solution. Knowledge gained from this step, about the required grid size, would be used later in the creation of the mesh around the full rocket body.

An additional method for creating appropriately adapted meshes is the grid adaption tool in Fluent. This tool finds locations in the fluid flow with sufficiently high pressure or velocity gradients, and allows the user to refine the grid in those locations, in order to capture the more detailed flow behavior there. Thus, one may start by running a calculation with a coarse grid and continually adapt the grid as necessary. Though great effectiveness was seen in grid independence studies, the grid adaption tool was not utilized in these simulations. This is because fine grids were already proven to be adequate, and it is one more thing that would need to be recursive. Also, it proved difficult to programmatically ensure sufficient resolution. Still, it certainly could prove to be a useful addition in future studies.

Four residuals were monitored to judge convergence. These were the normalized mass flow continuity residual, normalized energy residual, and the normalized x- and y- velocity continuity residuals. Many of the simulations converged to a very low level for all residuals (~10^{-14}), while others leveled off at a significantly higher level (~10^{-2}). Due to the large number of variables being considered in this study, it was assumed that if a residual level of 10^{-2} could be reached and remain constant for tens of thousands of iterations, the results were kept. When compared to the other results from similar nose cones, they were found to have very comparable values, especially in regions downstream on the rocket body where the geometry was of a consistent shape.

5.1.2 Fluent Model Validation

Meshes were created using both quadrangular (quad) and triangular (tri) shaped cells to determine which, if any, gave more valid results compared to known values of the drag coefficient. In all cases, tri meshes gave values much closer to empirical results so it was decided that tri meshes would be used on the full rocket body models. The reason for the difference may be that quads had poor cell alignment with the shockwave at many locations, where the tri cells were more randomly distributed and not subject to such misalignment.

Another design decision made against early work was to use the *Inviscid* setting. In all examined cases, the inviscid results fared much better than the laminar results for cases that were otherwise identical; specifically, the drag coefficients were much closer to accepted empirical values (Nielsen 1960). This also is reflected in the work of Bigarella et al, as can be seen in their discussions and the schlieren photographs with numerical density gradient contours of turbulent versus laminar and inviscid configurations. (Bigarella 2005) Bigarella et al also reported an increase of 25% to computation time from using a laminar instead of an inviscid calculation. The use of an inviscid setting is also widely justified in scholarly works for situations such as this, which have such a high Reynolds number. (Clark 1969)

This is also supported by Eremenko et al. “For certain vertex angles, the drag exhibits a minimum for a blunted, rather than sharp forebody geometry. This result, obtained by Euler computations, is confirmed by Navier-Stokes simulations at a Reynolds number of 270,000 performed in selected cases. The latter show that the frictional drag contribution to the total drag is negligible.” (Eremenko 2003)

Since the drag coefficients represent the most succinct validating data, they are summarized in Figure 5-2 below, superimposed on the graph that provided the known empirical data. The red squares are the Empirical data at their exact locations on the graph, the green triangles are the more coarse mesh, and the blue X’s are the data points with a more fine representation.

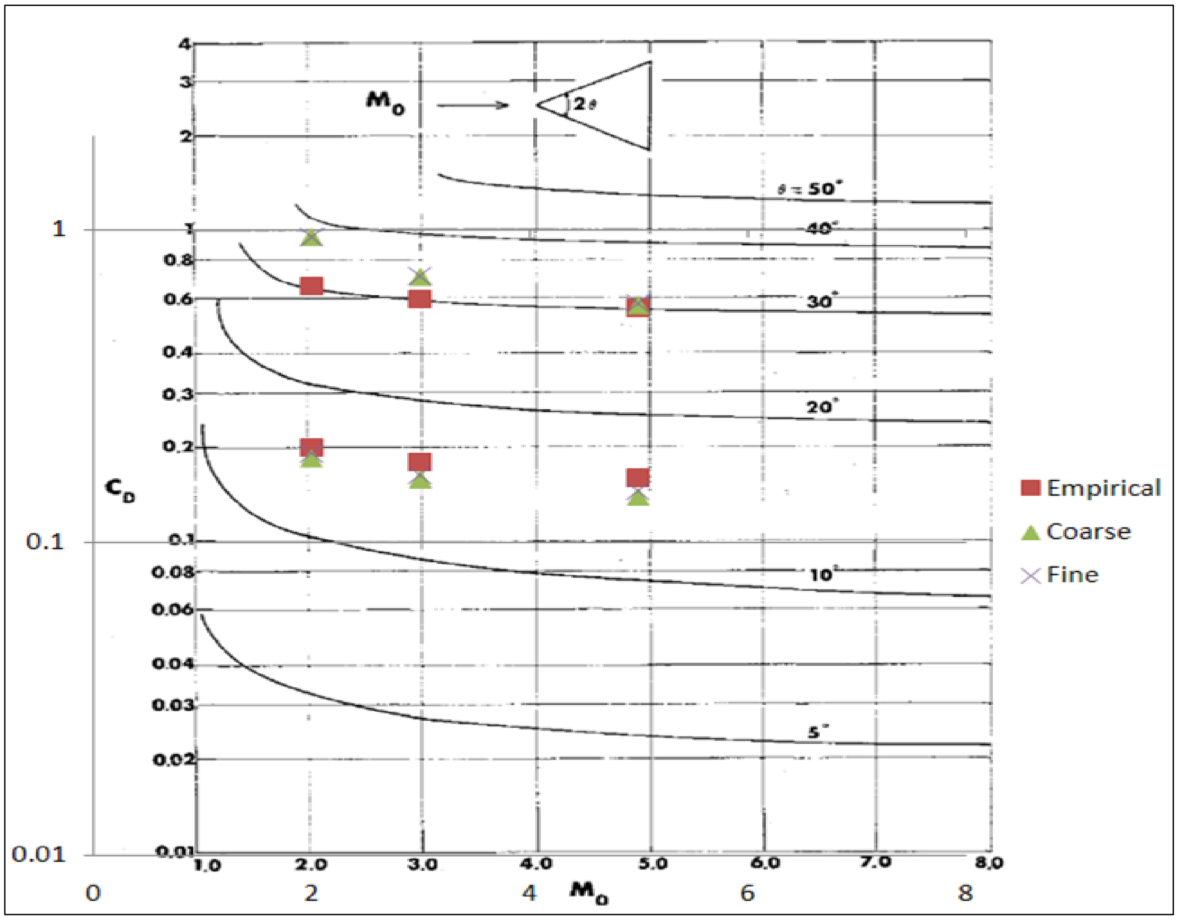


Figure 5-2: Supersonic Cone Drag Coefficients (Nielsen 1960) with Data Results Imposed

Clearly, there is a strong correlation between the established empirical data and those results obtained from Fluent. It is also worth mentioning that the similarity between the drag coefficients for grids of different mesh fineness further verifies grid independence.

Another exercise of validation was the comparison of the shockwave angles with known data. As a representative example case, a cone with a 15° angle was checked at Mach 3 and was found to have a shock angle of 24.45° off the axis, which closely matched the expected value of 25.14° to within 1° of accuracy. This is representative of many examples which were tested with comparable results.

5.2 Test Case I

The first test case is a simple rocket model that is completely fabricated but made to represent the types of assemblies that are used in a real rocket model. This rocket is put through the entire optimization process and provides results based on the application of the discussed method. This test case is referred to as the simple rocket in this thesis.

5.2.1 Simple Rocket Test Case Description

A variety of materials make up the internal components of rockets and they have many connections that combine in an assortment of complex ways. Below in Figure 5-3, a cut of the simple rocket model shows the interior components. These are a simple representation of the various shapes and materials that would be found overlapping in an assortment of ways throughout a rocket. This simple version allows for a faster computation time while still testing different forms of mating conditions.

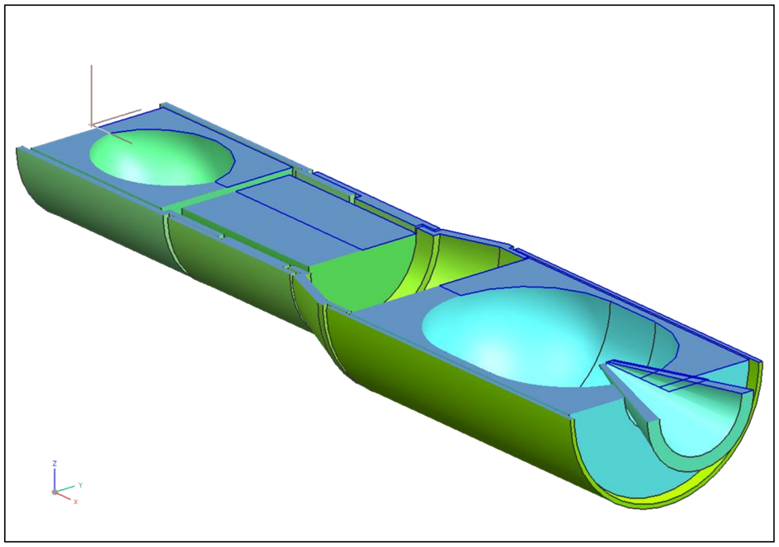


Figure 5-3 Picture Showing Simple Rocket Model Interior

5.2.2 Simple Rocket Optimization Results

For this rocket, three nose cone types were allowed as choices for the optimization. They were the Power, Parabolic and OGIVE. The iSIGHT-FD program iterated 60 times to come to an optimal solution that minimized the stress and weight on the rocket, while keeping the temperature below the specified limit. The parameters converged at the following values shown in Table 1.

Table 1: Optimal Parameters for Simple Rocket Model

Parameters	Value
Type	Parabolic
Optimal Length (in)	99.449069
Thickness (in)	4
tip_rad_dist (in)	4.5693695

The Simple Rocket was easily converted using the OML Maker tool into the template for CFD 2D meshing with a parametric nose cone. The optimized version of the model is shown here in Figure 5-4.

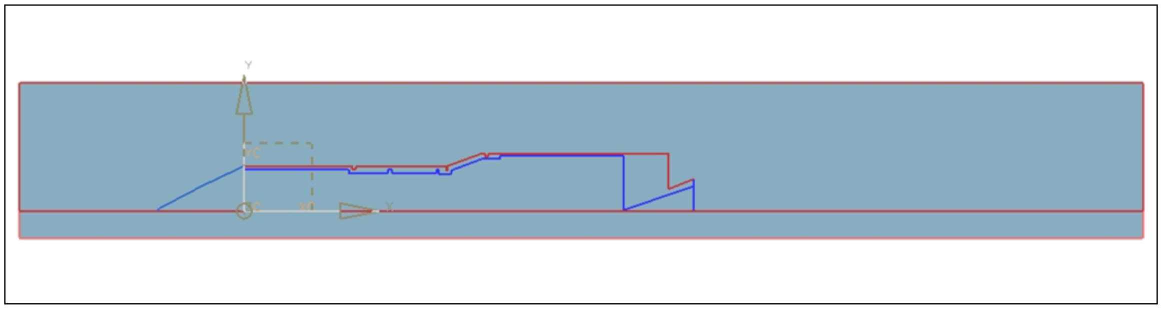


Figure 5-4: Optimized Simple Rocket in OML Maker

In Figure **5-5** is a picture of the optimized mesh with the reflected elements and the internal elements used for creation of boundary conditions in Fluent. Upon deletion of these two excessive groups of elements, the Fluent boundary conditions are made as shown in Figure 5-6.

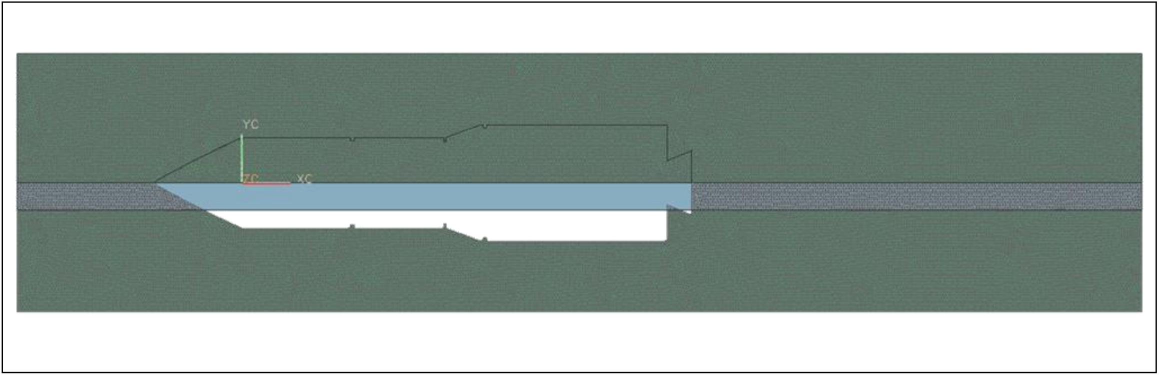


Figure 5-5**: Optimized Simple Rocket in NX Mesh Maker**

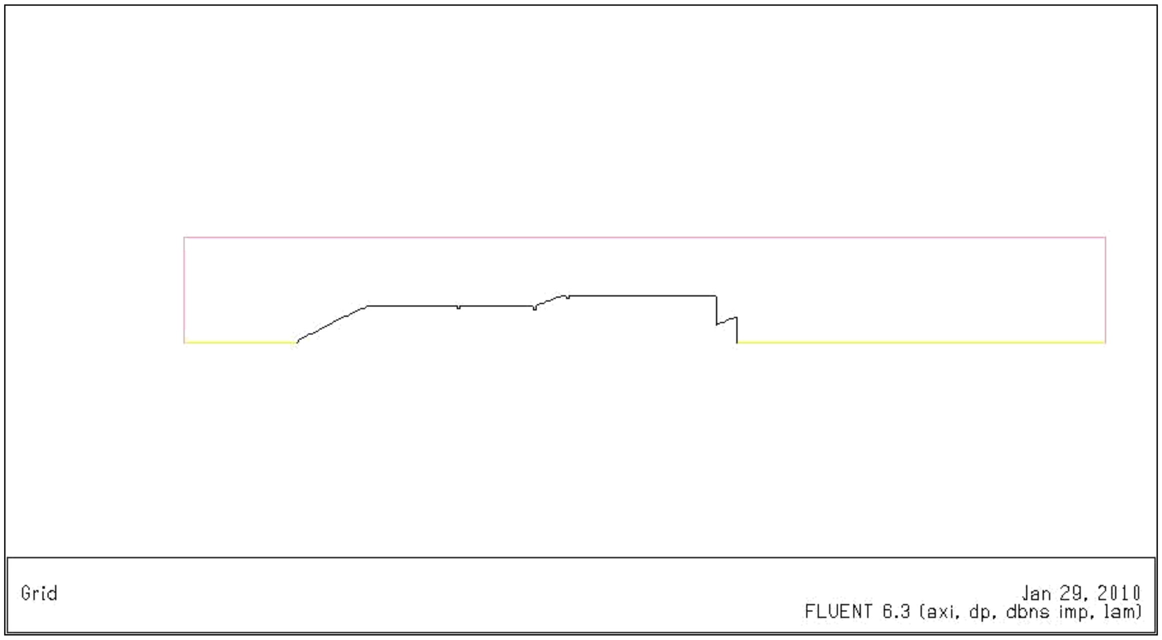


Figure 5-6: Optimized Simple Rocket Boundary Conditions in Fluent

The resulting ANSYS mesh with boundary conditions is ready for the flow study in Fluent and is shown here in Figure 5-7.

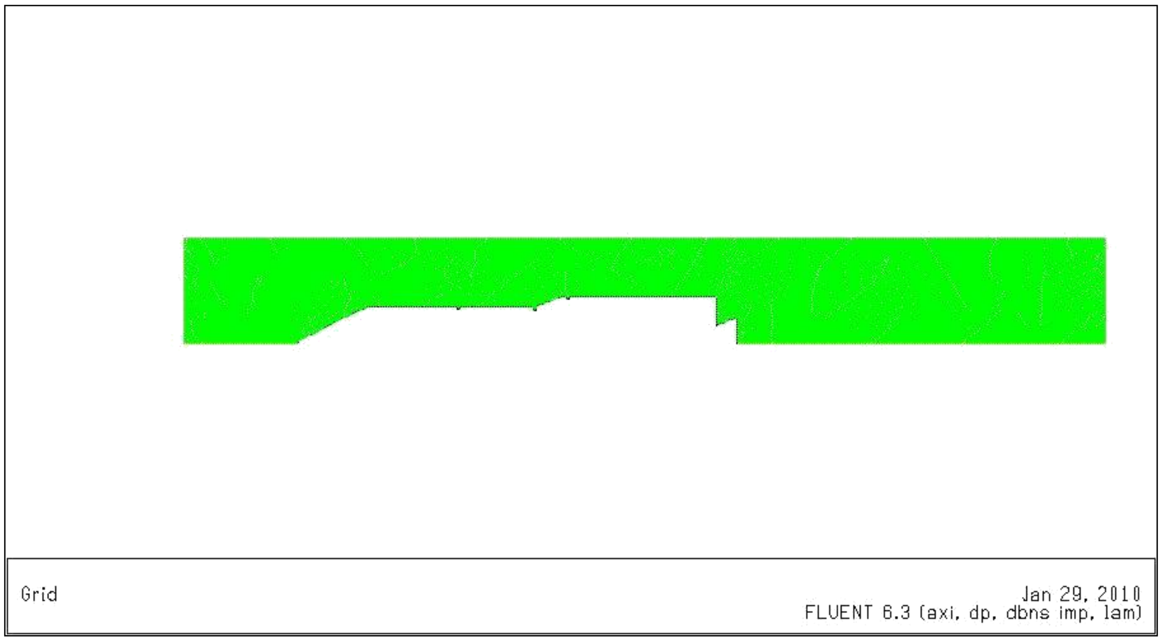


Figure 5-7: Optimized Simple Rocket Mesh in Fluent

After the script created for Fluent has been run, the picture in Figure 5-8 is automatically taken which shows how the pressure contours are forming around the rocket. The surface pressures are the main product of the analysis so those have been stored to a text file (.csv file). The temperatures along the wall surface are also stored to be programmatically sorted through later.

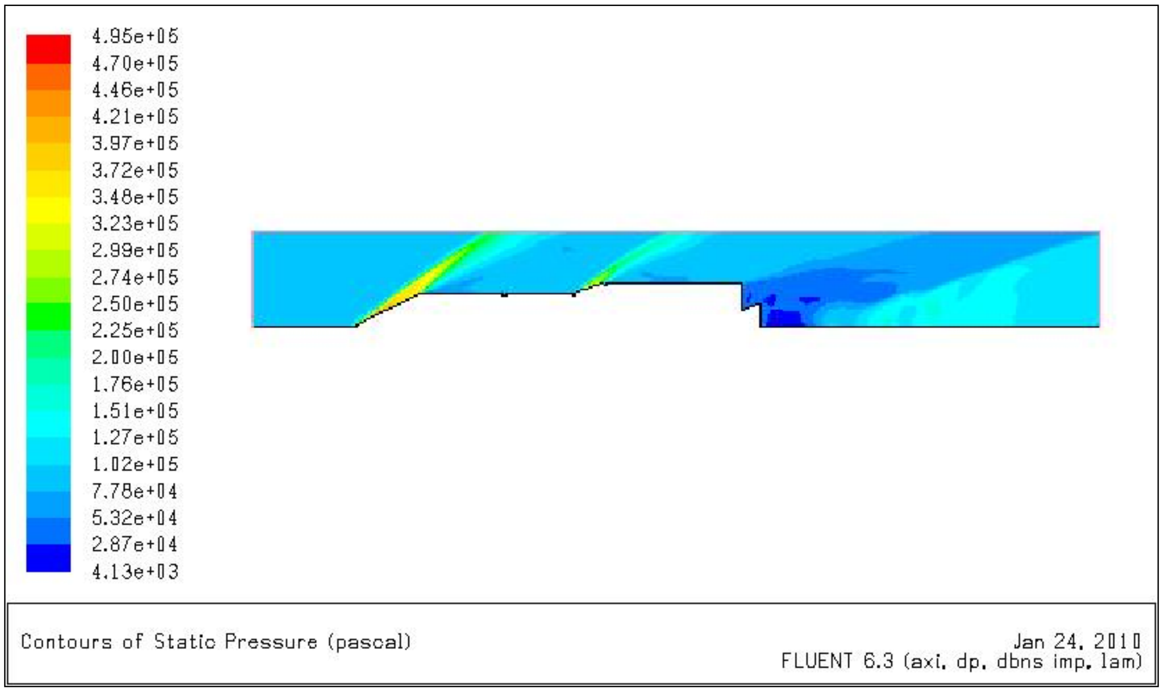
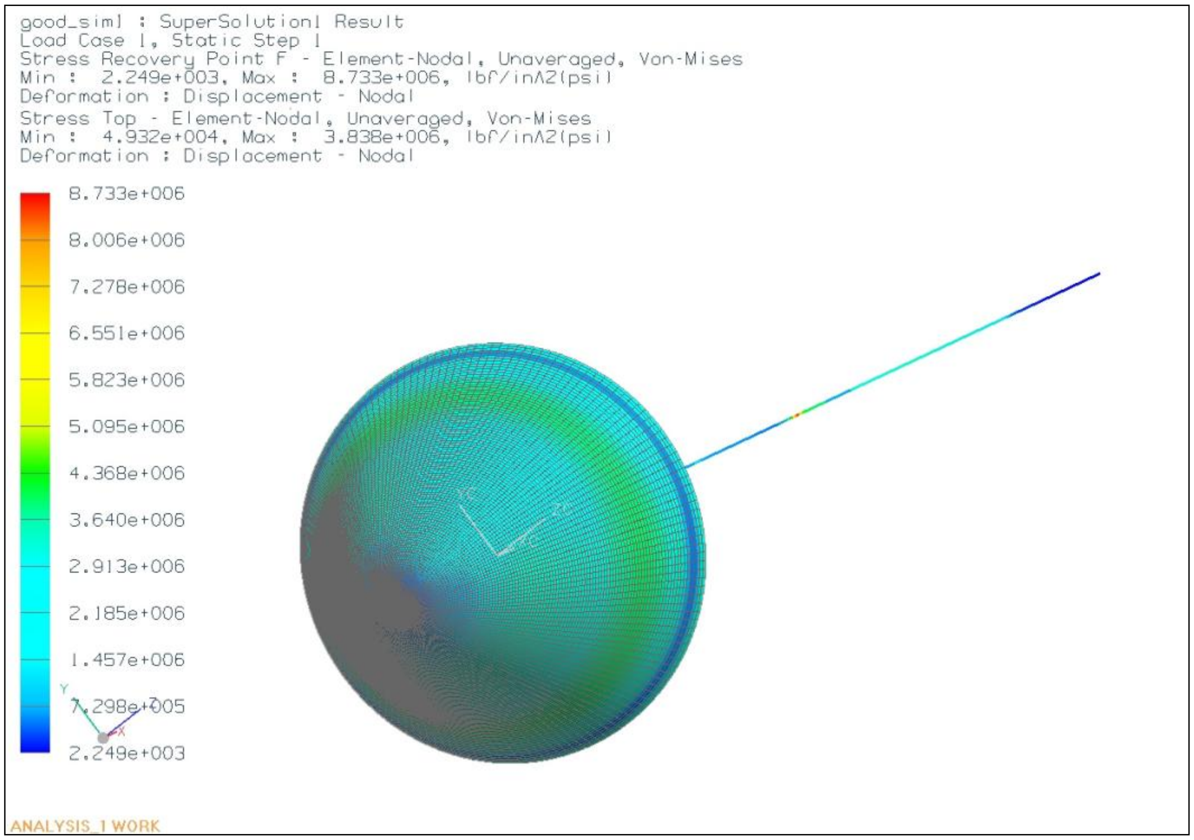


Figure 5-8: Optimized Rocket Pressure Contours in Fluent

The Nastran model is then made in NX6 and the stresses are calculated based on the outputs in the pressure file, as shown here in Figure 5-9. This picture is actually showing the deformation results for both the Von-Mises stress on the beams and the Von-Mises stress on the cone. These are usually shown independently in NX6, but have been superimposed here using the same scales for comparison. The original min and max scales for the two element types are shown at the top in the figure. As indicated, the beam model had far higher stresses than the cone.



```
good_sim1 : SuperSolution1 Result
Load Case 1, Static Step 1
Stress Recovery Point F - Element-Nodal, Unaveraged, Von-Mises
Min :  2.249e+003, Max :  8.733e+006, lbf/in^2(psi)
Deformation : Displacement - Nodal
Stress Top - Element-Nodal, Unaveraged, Von-Mises
Min :  4.932e+004, Max :  3.838e+006, lbf/in^2(psi)
Deformation : Displacement - Nodal
```

Figure 5-9: Optimized Simple Rocket Von-Mises Stress Plot

The model itself was simplistic and unfortunately shows that not just anyone can become a rocket scientist. More thought must be taken when throwing together even a simple rocket model. The model had weaknesses that led to failure even though the nose cone itself did not fail. Even by minimizing the maximum stress throughout the rocket as much as possible, the rocket will still break at this weak point.

The highest stress on the rocket is located at the red point about halfway down the visible rocket body beam model; this maximum stress (8.733e+006 psi) is well above the plastic deformation point of the steel. The nose cone itself had a max of 3.838e+006 which is still well within the elastic region of the metal and not deflecting significantly. Below, the ending results of the max temperature, max stress, and weight are shown in Table 2.

Table 2: Optimal Results for Simple Rocket Model

Results	Value
Max Temp (K)	1556.3
Max Stress (psi)	4.95 x e+05
Weight (lbs)	18958.6

All of these results are higher than would be desirable; but again, the nose cone is actually performing its best to minimize these values. In the end, the rocket is not sturdy enough to handle the conditions it was placed in, regardless of the nose cone. Below, in Figure 5-10 the pressures along the surface of the entire rocket are displayed. As can be seen, the nose cone is actually seeing the greatest pressures but has been engineered to withstand the increases. Unfortunately, the rest of the rocket was not engineered as well and would have to be reworked. At this point, the regions which performed poorly would have to be reinvestigated and redesigned to improve their results before the rocket could be improved. It would also be advisable to repeat this optimization study to ensure these components are brought up to a level that can withstand the specified flight conditions. A more optimal choice may be available if the structural integrity of the rocket body had not been so limiting to the overall design.

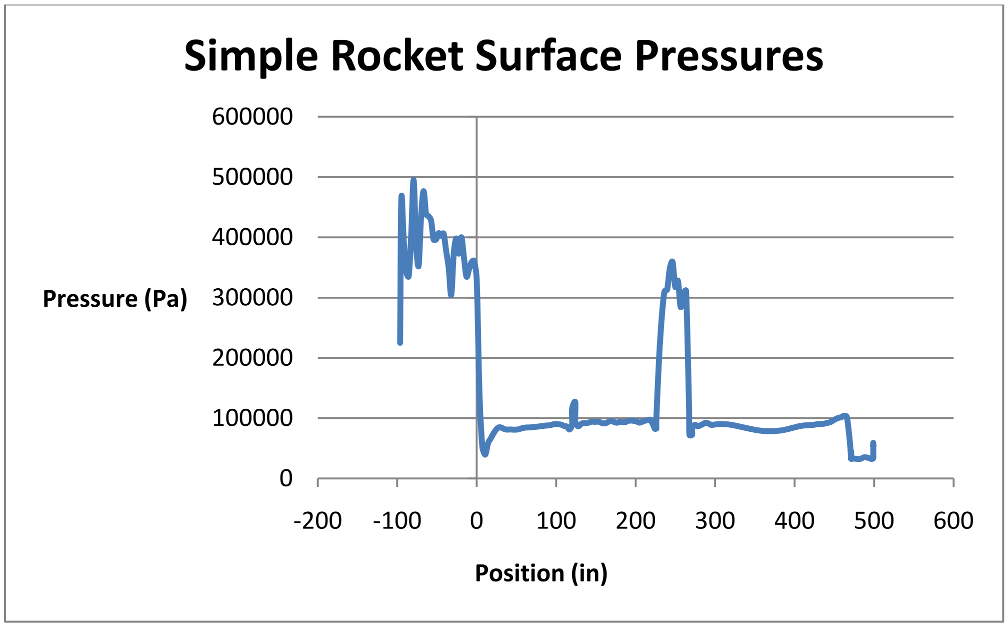


Figure 5-10: Simple Rocket Surface Pressure Plot

The temperature contour is shown over the whole rocket in Figure 5-11. The maximum temperatures are hard to see because they are all along the surface of the rocket and unfortunately, the surface is the place that really matters. The surface temperatures were therefore output to a text file (.csv) and from this, the plot in Figure 5-12 was created to show where the maximum temperature is located along the contour of the surface. The maximum temperature value is once again not on the nose cone, but peaks at a poor seam on a location midway along the rocket, as shown in Figure 5-12. This maximum temperature (1556.3 Kelvin) is high enough to melt steel over time and is another reason the rocket model design fails. A look into the redesign of this region would be recommended.

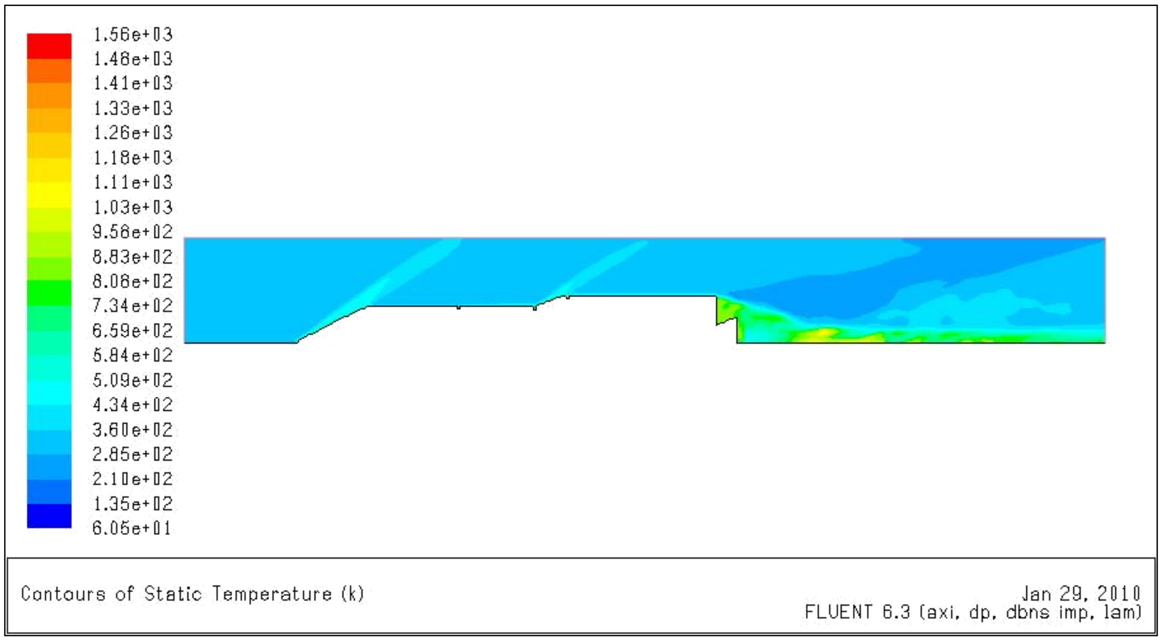


Figure 5-11: Optimized Simple Rocket Contours of Temperature

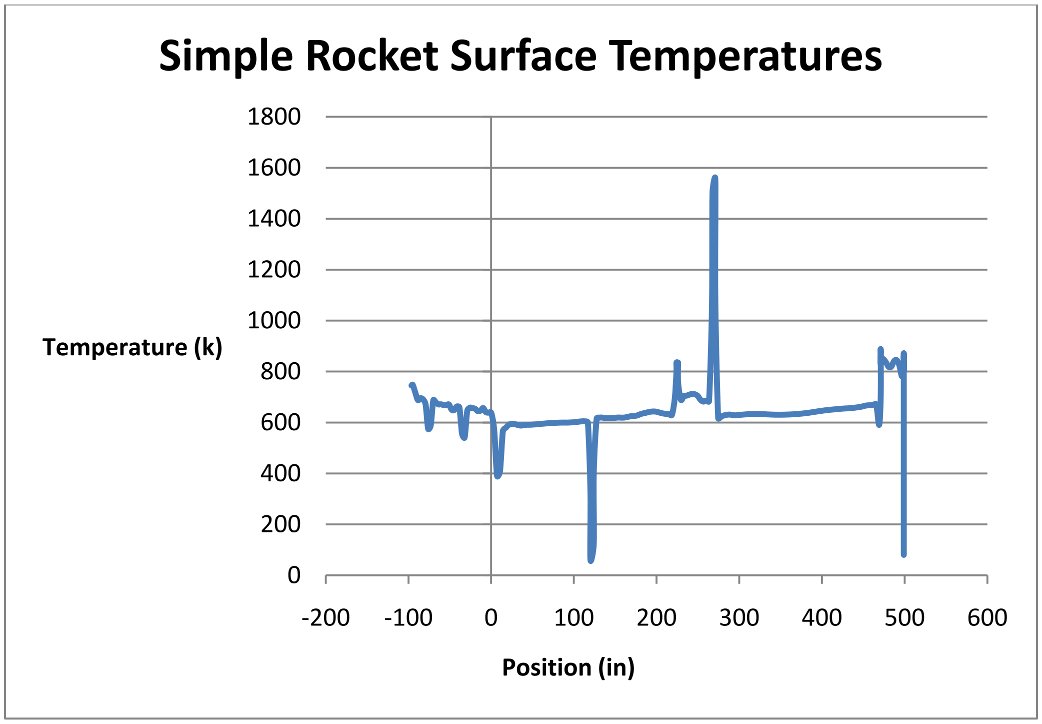


Figure 5-12: Simple Rocket Surface Temperatures

In summary of the results of the simple rocket model, the optimization tool performed well and the rocket did not. This tool could stop costly continuation of a product, which is not yet up to the specified standards required, and could avert disastrous results.

5.2.3 Test Case II Description

To more thoroughly test the tool, it became apparent that a new rocket case would be required and that this second case would have to be better engineered to withstand the extreme conditions that supersonic speeds place on a rocket.

In the following case, which will be referred to as Test Case II, the rocket has been made into a design that is similar in principle to the Delta IV Medium rocket. The Delta IV line of rockets is still in use today and served as a basic principle guide in the design of this simplified, school level rocket.

Below in Figure 5-13, the rocket is shown in its original design environment of SolidWorks. Another aspect for evaluation with this optimization tool is to see how well it handles parts brought in from another CAD package. This ability is useful since many companies employ outside entities to design certain components in their rockets. Being able to easily add these into their assemblies without recreating them is helpful.

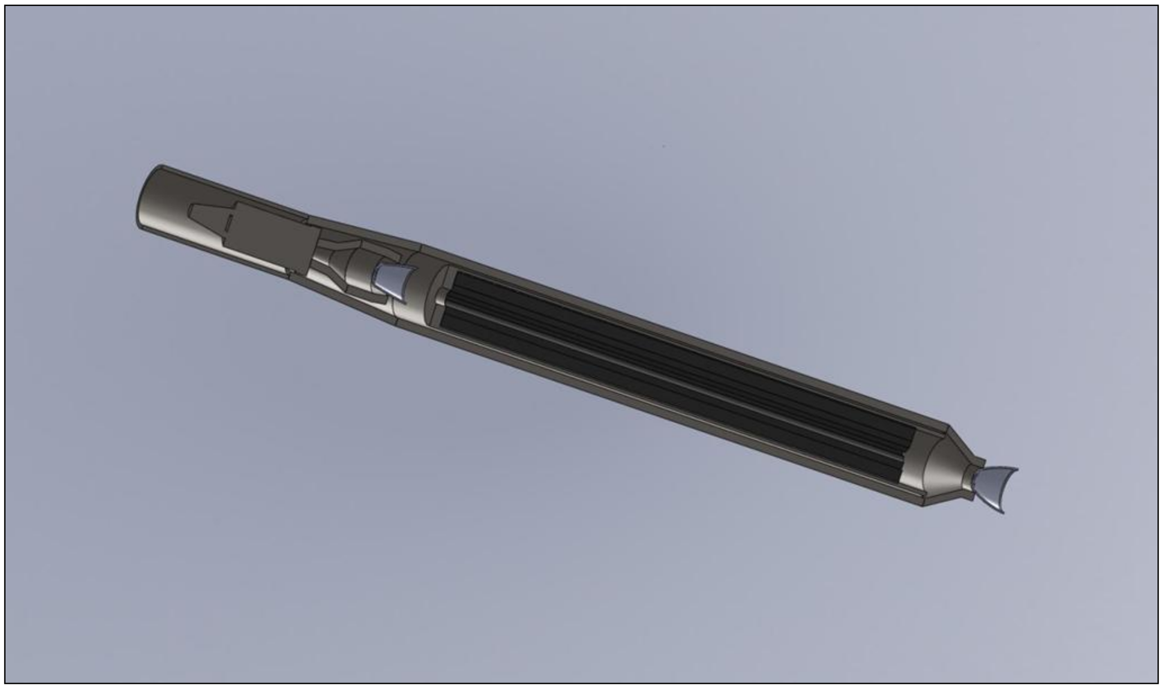


Figure 5-13: Test Case II Large Rocket in SolidWorks

The import to NX6 transferred all the geometry with excellent results. All the geometric data remained intact and solidified. The material properties had to be recreated in NX6, however, since the specific materials had not previously been made in NX6. It is necessary for NX6 to have an accurate database of all the materials so that it can use them in the creation of the slicer dicer mass properties and for the NX Nastran structural analyses. In Figure 5-14, the rocket model can be seen translated into NX6 with its updated material properties. Two properties which were added in NX6 were a custom steel for the exterior and a custom rubber for the solid fuel. Other components in the model have also been modeled as a basic standard 1020 steel.

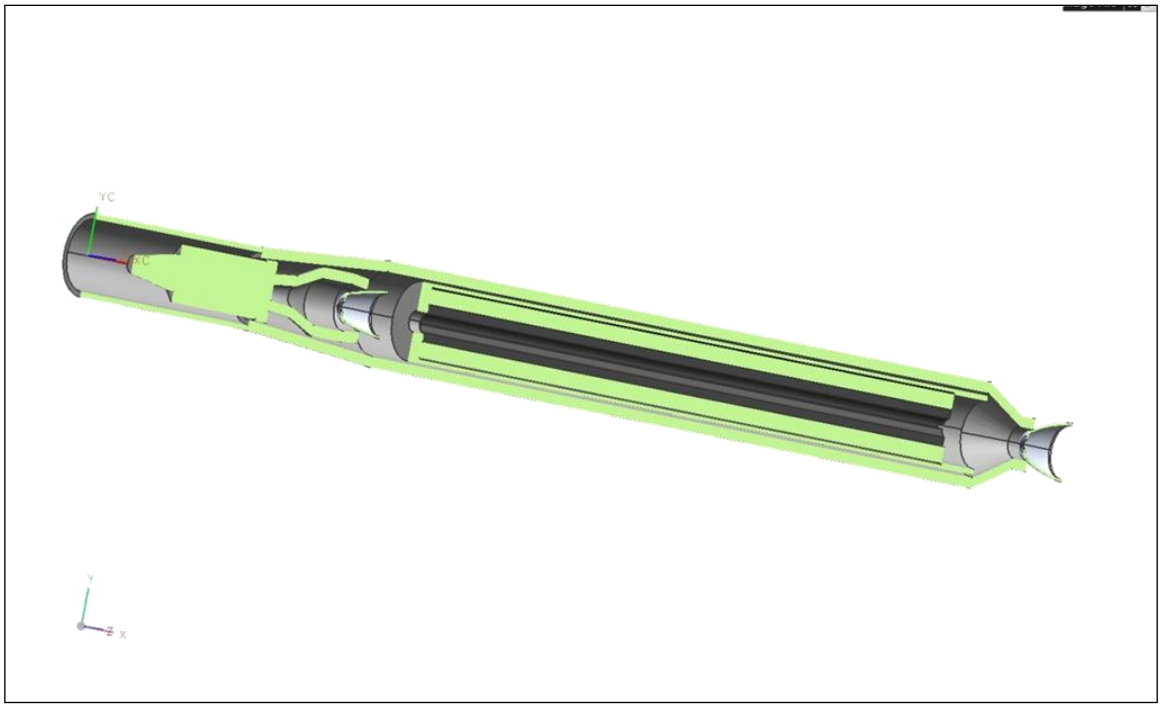


Figure 5-14: Test Case II Rocket Translated into NX6

The outer mold line was determined by the OML tool and is seen here in Figure 5-15 with the final iteration nose cone geometry. The orange surface is the airsolid and represents the total region of air to be included in the Fluent analysis. The airsolid ends against the outer mold line of the rocket. Notice also that the inner mold line can be seen contouring the OML of the rocket.

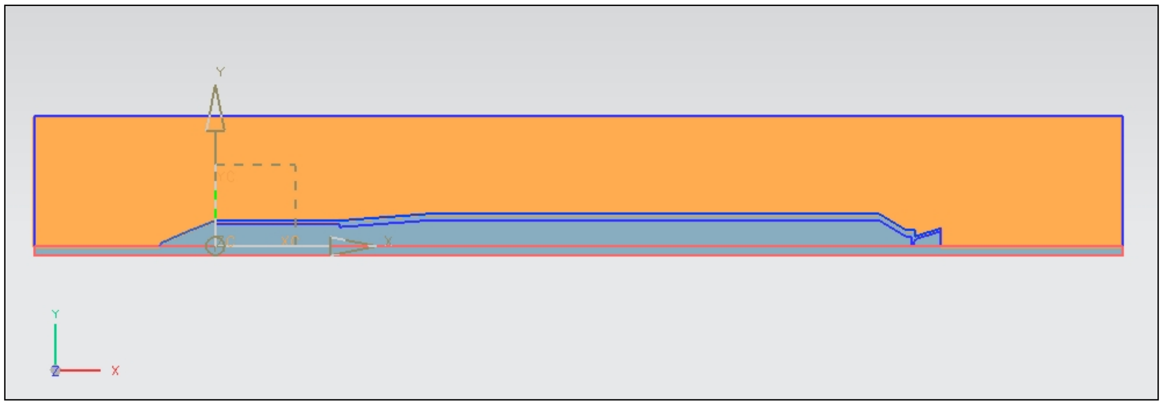


Figure 5-15: Test Case II OML Maker Resultant Air Flow Outline

In this study, it was found that the body maker tool was worth updating. This test case was the largest and longest rocket tested so far (at around 200 feet in length) and updating the program removed the possibility of a negative volume error, which occurs in Fluent, that relates to the convex shape of the elements. This error was caused by an extreme ratio of elements in the x-direction compared to the y-direction. The body maker tool now ensures that the element ratio in the x and y directions are held similar regardless of the rocket size. Before, the rocket only ensured that the airsolid was a multiple of the radius in the radial direction and a multiple of the rocket height in the upwind and downwind directions.

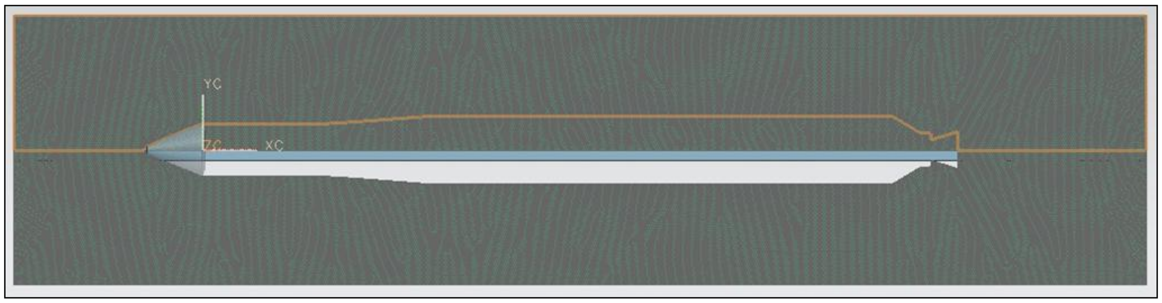


Figure 5-16: Test Case II Updated Mesh for use in Fluent

Because the Test Case II simplified design was based on the principles of the Delta IV Medium rocket, the speed and altitude for Fluent were also mimicked after it. They were set to the state at which the max pressure condition for that rocket had been, which was an altitude of 7.76 miles, and a velocity of 0.428 miles per second. This should be near the extreme condition for this rocket and a good test parameter for optimization of the nose cone.

In Figure 5-17, the surface pressure results from Fluent in the final iteration are displayed, showing how the pressures are distributed over the length of the rocket. The maximum pressure is, of course, at the nose cone itself (the region that is negative is the nose cone), and shows how extra emphasis on structure and stability is needed for this region of the rocket.

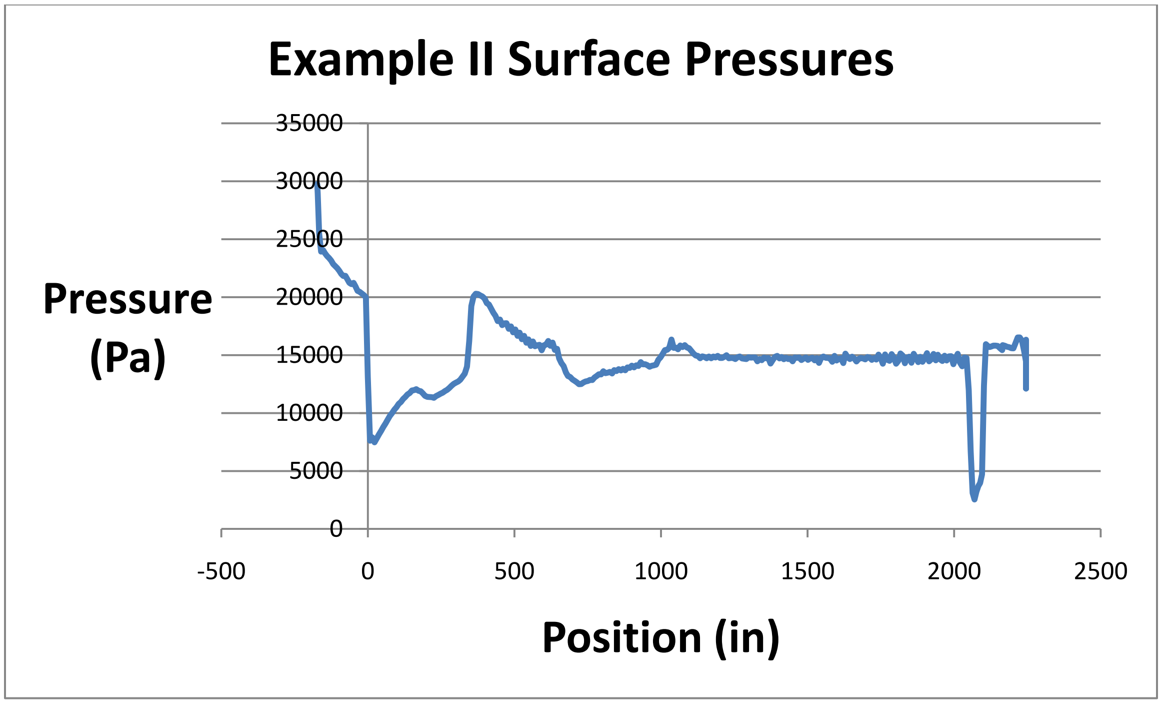


Figure 5-17: Test Case II Surface Pressures

The final pressures listed in this file are then input into the structural NX6 model as shown here in Figure 5-18.

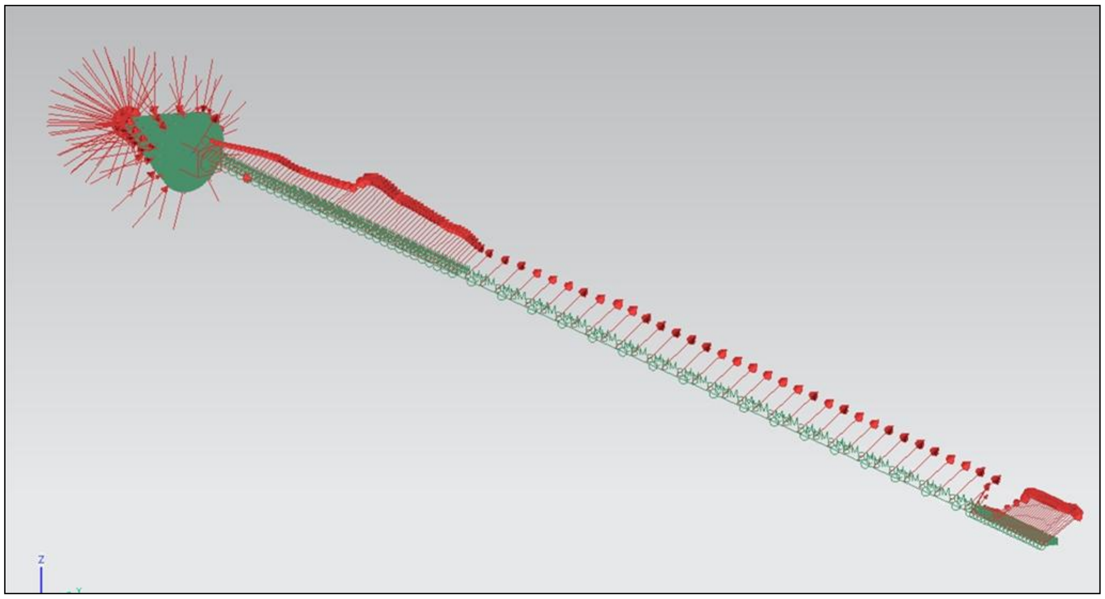
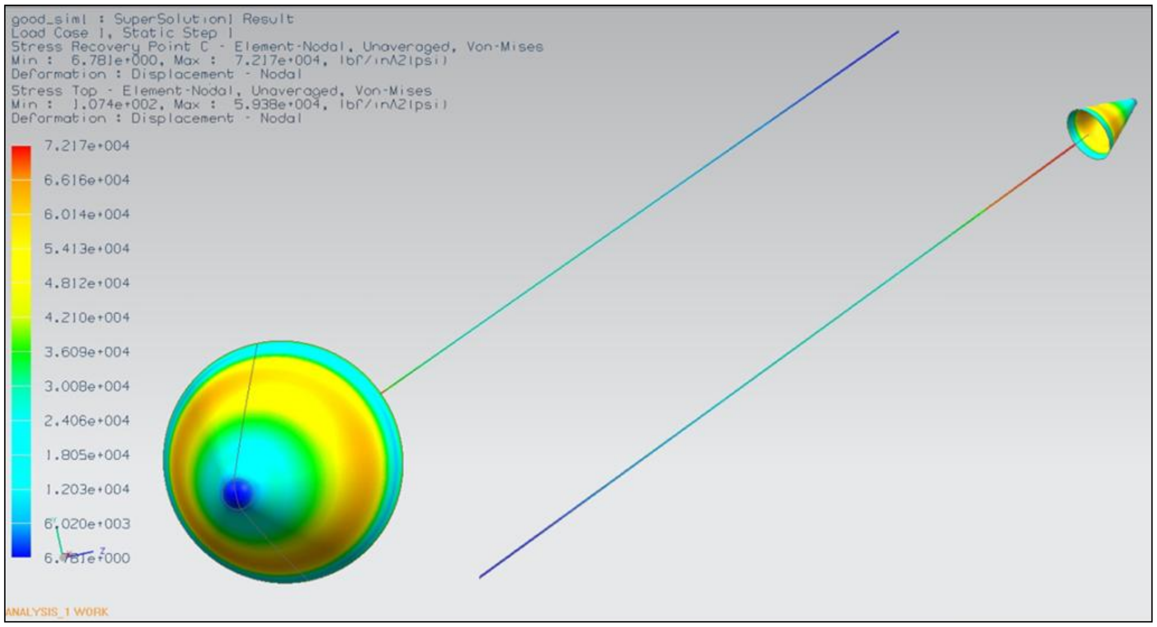


Figure 5-18: Structural Rocket Beam Model with Pressures

The pressures are used to analyze the model using NX Nastran inside of NX6. The combined Von-Mises stress results for both the beam and shell portions of the model are shown here in Figure 5-19. Two views are given to show the cone in detail and to show the maximum stress region right behind the cone. The max stress is on the front of the rocket body but is less than half the allowable before plastic deformation.



```
good_sim1 : SuperSolution1 Result
Load Case 1, Static Step 1
Stress Recovery Point C - Element-Nodal, Unaveraged, Von-Mises
Min :  6.781e+000, Max :  7.217e+004, lbf/in^2(psi)
Deformation : Displacement - Nodal
Stress Top - Element-Nodal, Unaveraged, Von-Mises
Min :  1.074e+002, Max :  5.938e+004, lbf/in^2(psi)
Deformation : Displacement - Nodal
```

Figure 5-19: Combined Beam and Shell Von Mises Stress Results (2 Views)

The final iteration surface temperatures shown in Figure 5-20 are well within acceptable parameters, but there is a peak that is significantly higher than everything else and it is on the harsh edge of the aft nozzle connection. From this result, the decision might be made to angle that edge to remove the disproportionate heat at this location and to help minimize the drag off the back at the same time.

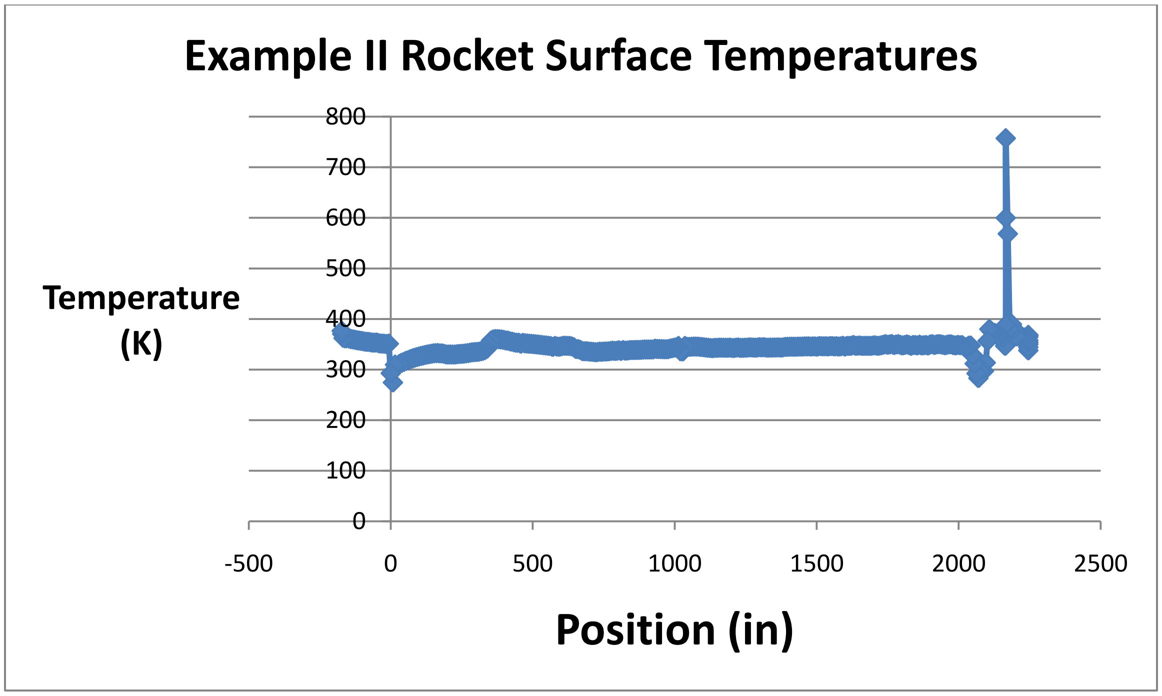


Figure 5-20: Test Case II Surface Temperatures

6 Conclusion

This thesis gives a detailed process for optimizing fairings that decreases time to design the rocket, and improves the reliability of the result by minimizing the weight and the drag on the nose cones. The Conclusion is divided into two basic parts. The first section is the process evaluation, which describes in detail the success of the tools in their native design environment as rocket optimization tools. It describes how implementing these tools can simplify the initial design process and streamline the design itself. The results of the tools have been clear improvements over any initial results that have been obtained. In addition, this process paves the way for additional advanced analyses, such as more advanced structural and thermal tests that can be performed on the parametric CAD and FEA models. These possible additions to the process are described in detail in the Future Work section. Also shown is how this thesis provides a framework for a library of tools that could be instigated from these functional building blocks.

6.1 Process Evaluation

The process for optimizing fairings in this thesis has a variety of improvements over the past segmented efforts of manually changing one aspect of design, only to learn that another area of analysis has been detrimentally impacted. The time savings and the functionality have been improved and the final models are more updateable and reusable than they have been in the past.

The following sections detail how the individual components have improved their respective areas.

6.1.1 OML Maker

The OML Maker tool has proven to be a quick way to sift through a large amount of information to obtain the specifics of the OML and IML of a rocket. It has proven accurate and capable of handling a large variety of rocket models and of passing the core features into a unified contour–ideal for flow over a surface. It handles the seamless connection of segmented parts in a way that no others have, according to all research done for this project. In its current state, this tool alone has a variety of applications in efforts of flow analysis for both internal and external flows. This tool is also currently capable of helping in simplification of geometry and design or for simplification of structural analyses. It could also be expanded to do these combinations more extensively, as will be described in the Future Work section. This tool has been averaging with a run time of approximately 10 seconds for all of the example rockets here, which took roughly half an hour to create by hand, resulting in a 180 times increase in speed! It should also be noted that in a real rocket, there are literally 1000's of components to sort through manually so a more significant amount of time will be saved.

6.1.2 Slicer Dicer

The Slicer Dicer tool has proven very effective at taking a model and disseminating it into the core sectional properties of mass, inertia, center of gravity and density. The user has full control of the division regions of the model and the authority over the usage format of the output. The outputs have proven accurate for describing the rocket design and allow for multiple tests and analyses to be performed ranging from quick weight checks, and static stress tests, to more dynamic loading and fatigue testing. This tool took what was roughly a dozen hours to do by hand and computed it in under a dozen minutes for over a 60 times increase in speed! Considering the much larger savings in hours that this represents, it may be considered of more value than the OML Maker despite the smaller factor of time savings.

6.1.3 Beam Modeler

The Beam Modeler has taken what took a week to design once for one rocket model and specific cone design and made a parametric and load updateable design that was created in a few minutes for approximately a 400 times increase in speed! This tool takes advantage of the aforementioned tools to seamlessly take what would be a painstakingly tedious and error prone process and automate it into an opportunity for more upfront design and analysis. This model helps with multiple analyses other than the basic stress tests performed in this thesis, such as more dynamic studies of vibration and flutter.

6.2 Future Work

The work done in this thesis had several tools that proved exceptional for advancement and repeatability of more conventional methods. More work can be done, however, to improve accuracy, speed and to provide more options to the overall package. Also, in addition to the tests performed in this thesis, there are many other tests that would be performed on rocket models. Many of the individual tools supplied could be developed to further improve the alternate aspects of the main design. The current process is the groundwork for these viable extensions.

Below, ideas are listed for future work that can be done to both improve the tools for their current tasks, and for branching into other areas of optimization, design, and analysis.

6.2.1 Slicer Dicer

The Slicer Dicer tool controls how the model is divided into discrete entities. A helpful addition, that is not currently available, would be controls for analyzing variable “rings” along the length of the rocket. This is an achievable addition that can be very useful. For example, there are those that run analyses that focus primarily on the exterior components of the rocket; being able to distinguish and divide out the interior objects from the exterior based on a variable interior controllable radius would be a quick way to make such a model. Another form of geometric exclusion that could be of value to others is to create a tool that takes an axisymmetric half or quarter of the rocket as a representative of the whole. This “pie slice” does not help for all forms of analysis, but can still be useful for certain structural, thermal and flow analyses with the right boundary conditions. The creation of these axisymmetric wedges could also be used to help better divide up complex rocket geometries that present issues for dividing regions into slices due to limitations in CAD systems as were mentioned in 4.1.1.

6.2.2 Beam Modeler

The current models produced, based on the Slicer Dicer, are too stiff at the joints between objects and could be more accurately modeled with a high stiffness spring (with a specified K value). This keeps the model from being more rigid than it really is. For the case of the analysis performed in this process, the excessive stiffness is, if anything, a safer model that would ensure that fracture occurs sooner than reality and it would do so consistently for all the models. For a vibrational analysis, however, the added stiffness could “help” or “hurt” the results at the various frequencies causing misleading results. To more accurately portray this scenario, an addition of springs should be input by the user, which model the joint regions. This would be input during the first preparation of the structural model and would then be used repetitively for all loops of the optimization in a vibrational study. The spring values would be pulled from a list that has one spring value, with reference to the junction it is related to by position or number. This would then be programmatically inserted into the beam model. A similar process was created in I-DEAS for the early version of the beam modeler before more detail into types of analyses to be performed was determined. This was therefore not implemented again for the NX6 version, but the package is certainly capable of this improvement. Some types of joint springs are shown attached in this I-DEAS generated model (Figure 6-1).

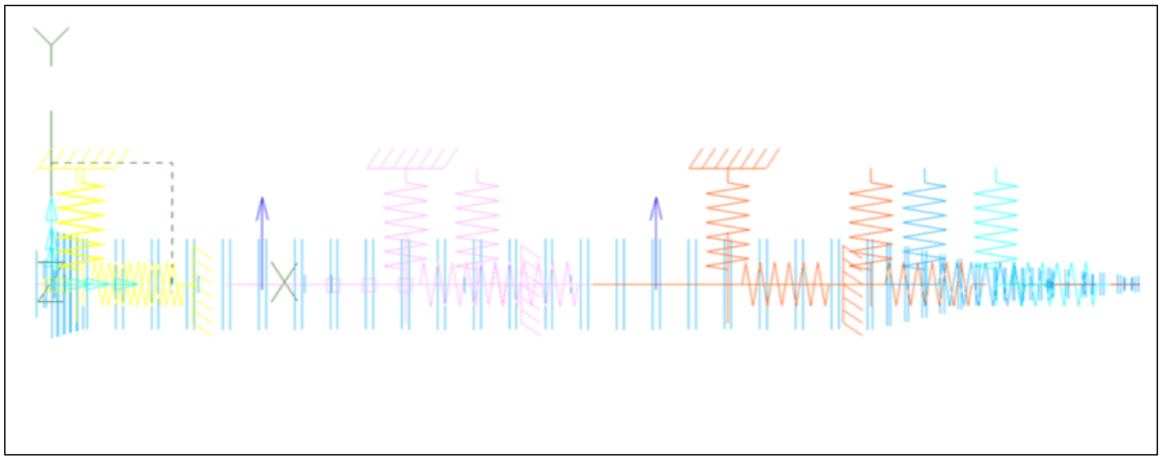


Figure 6-1: I-DEAS Automated Beam Modeler Example with Springs

In addition to the spring refinement, another perplexing problem can be the adjustment in weight due to fuel depletion. The stiffness change due to the fuel loss is another valid factor. It is often a known value of depletion as the rocket is propelled, so this could be incorporated, but it leads to the question of which moments in time are worth studying. Between fuel consumption and trajectory information the possibilities for calculation are immense.

Other areas of concern might also include the rocket sitting on the platform in windy conditions or stage separations. All of these situations can benefit from the tools in this thesis, help to improve quality and save time.

6.2.3 Mesh Maker

A unique option in Fluent that could greatly improve computation time is the grid refinement tool. This tool is very helpful at finding regions that are in need of additional mesh refinement, while also being able to remove excess elements in areas where they are not necessary. This process is cyclic, however, and it can be hard to quantify if the cycles have sufficiently refined the model; it is also hard to repeat programmatically. The journal files would have to be created by a C++ (or equivalent) program and the values would have to be determined through visual recognition on the screen. This is because Fluent cannot return the value to the C++ program automatically. This implementation becomes of more value if more than one study will be done on the mesh. Each time the nose cone changes, the mesh would be recreated in NX6 and the repetitive refinement process would have to be completed again in Fluent.

6.2.4 Non-Axisymmetric Meshes

Another valuable future work would be for the creation of non-axisymmetric meshes, which allow for the inclusion of fins and other defining features that are beyond the scope of the current project. This addition makes it possible for more rockets to be optimized that would otherwise need to be approximated. It also improves the flow analysis of said models and leads to greater accuracy in their results. This would benefit the flow analysis models (not recommended for the structural models) and could save some frustration in future analyses when seemingly small irregularities to the exterior conformity lead to large aerodynamic instabilities. At this point, there is an apparent need for a 3D model. Before, a 3D model would have only been a revolve of the 2D model about its center axis, with no irregularities. This current 2D to 3D revolve would only require higher computation times with no added knowledge about the performance of the exterior surface.

6.2.5 Startup Java Code

For a user familiar with all of the programs used it is easier to set up the steps manually, but for someone unfamiliar with the code, an interface is very helpful. This is a good future work item for a final user product and is already significantly prepared. Below in Figure 6-2, an initial GUI created for this purpose is shown. When the *Browse…* or *Open/Save Runs File* buttons are pressed, the program is designed to open a window that filters the selection options to provide only the relevant choices as options ensuring no incorrect data is entered. In Figure 6-3 an example for the Matlab Trajectory input file filter is shown in which only ".mat" files are allowed to be selected.

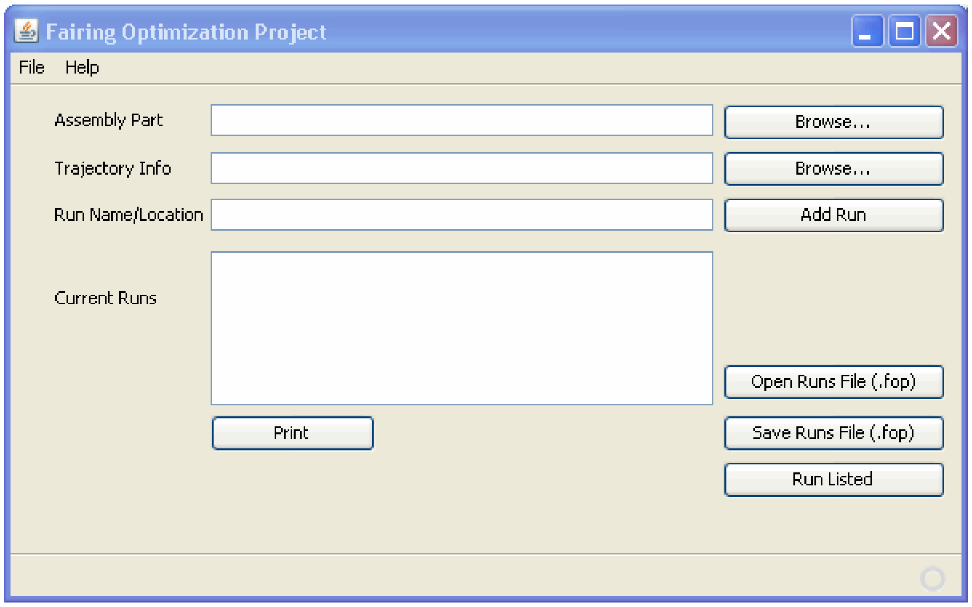


Figure 6-2: Java GUI Interface for iSIGHT-FD

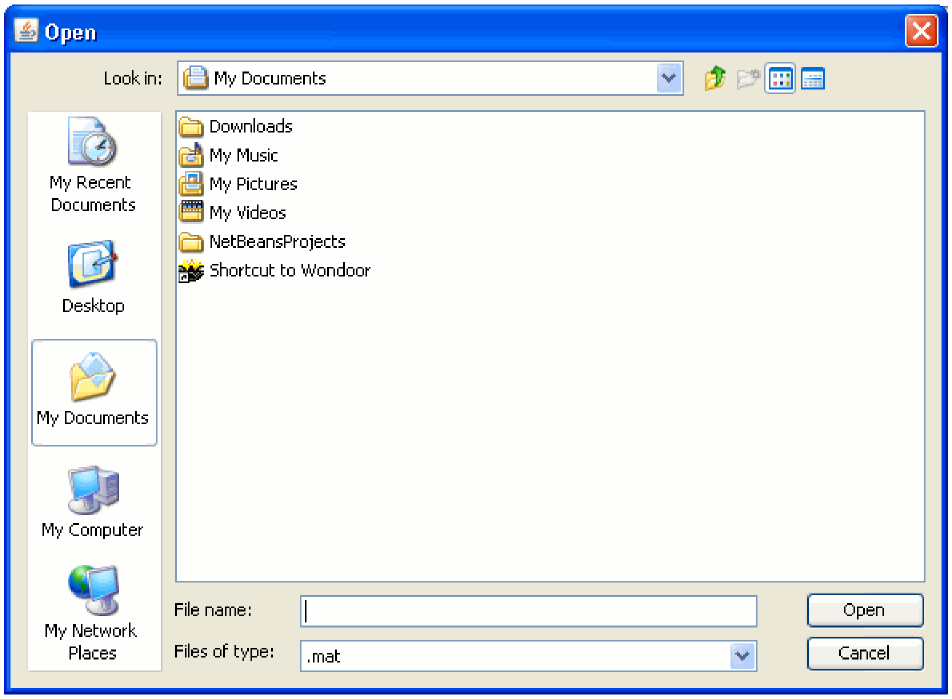


Figure 6-3: Example Filter for File Inputs

Some example inputs are given here in Figure 6-4 to show the layout. Multiple runs can be named and added to the list for execution based on a related Assembly and Trajectory. The runs can then be saved or loaded to the list based on an input ".fop" extension, which was made for this project. The print button writes all of the listed runs to a text file as well to show what runs have been organized in the past. When all the desired runs are listed, the *Run Listed* button is pressed to initialize iSIGHT-FD to start those projects.

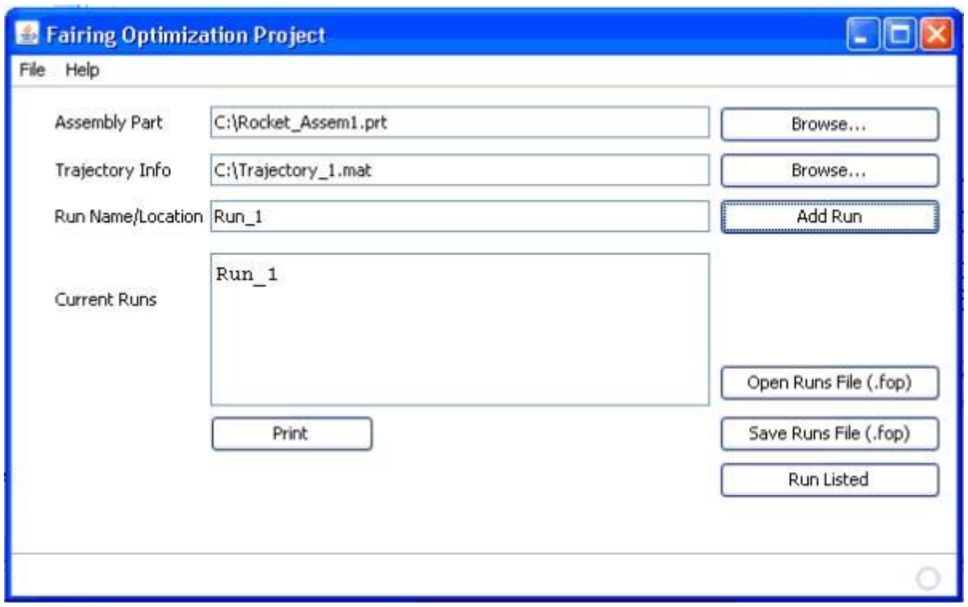


Figure 6-4: Example Inputs for an Optimization Run

6.2.1 Missile DATCOM

Early work done in Missile DATCOM was helpful, but was found to be inadequate to model the complexity of which the other tools of this thesis are capable. This is mostly because there is only a cone, a body, and an aft section to its inputs. So any multistage or unique geometry, such as joints, along the rocket body cannot be described or analyzed with this tool. The one advantage to this tool was the extreme time savings. If this tool could be used to approximate the pressures until a more concrete fairing design is established (and if the more comprehensive rocket design in Fluent could be used to do final iterations) it would be far faster. The computation of Missile DATCOM takes seconds versus the scale of hours for Fluent. A possible way to disseminate the design into manageable pieces is shown in a few steps, beginning with Figure 6-5 and continuing on through Figure 6-7. This early optimization will not be fully accurate, but could at least provide a better starting point for a detailed optimization. The principle divides the rocket in Figure 6-5 into its core pieces, as in Figure 6-6, and interprets each as a separate rocket with pressures of its own, like in Figure 6-7. These values would then need to be reconstructed into a common pressure mapping of the rocket and could be used for early analyses.

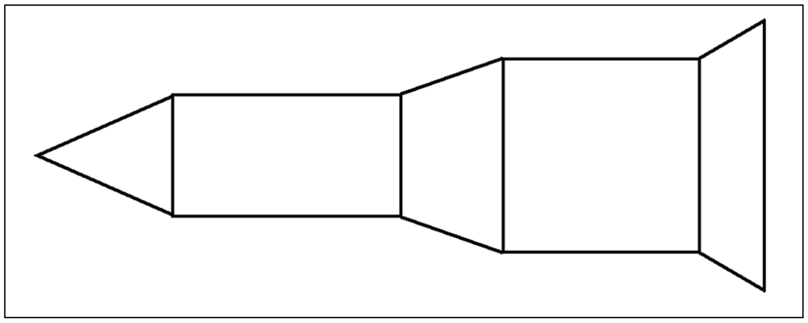


Figure 6-5: Multistage Rocket Beyond Missile DATCOM Capabilities

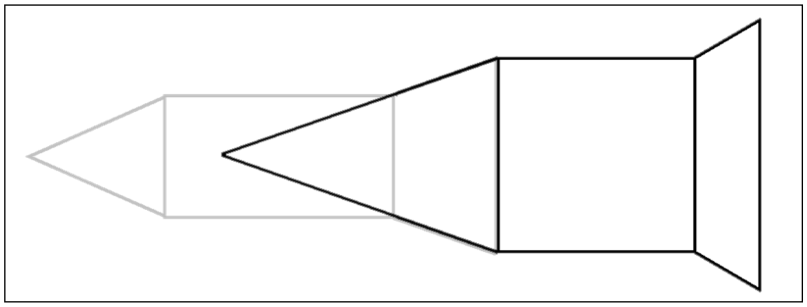


Figure 6-6: Possible Secondary Rocket Configuration

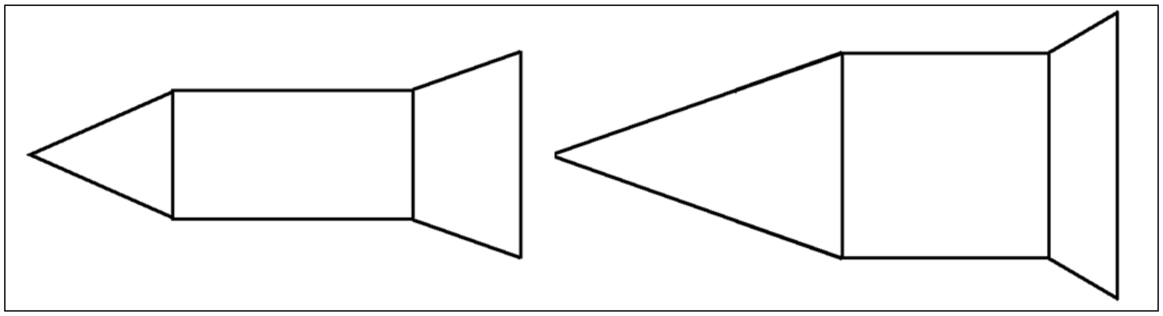


Figure 6-7: Disseminated Rockets for Missile DATCOM

6.3 Closing Remarks

This thesis has shown many unique and distinct methods for optimizing fairings that decrease design time and improve the result reliability. This is done by seamlessly connecting the various areas of design and analysis. The tools are both interactive and programmatic, depending on user needs. The options for future development are prevalent and applicable to a variety of fields. The methods converge to optimal designs accounting for the structural, thermal and aerodynamic aspects of any rocket model's trajectory. In short, these tools positively align the trajectories of the often opposing goals of minimizing cost while maintaining quality.

REFERENCES

Alexandrov, N.M., Lewis, R.M. "Analytical and Computational Aspects of Collaborative Optimization for Multidisciplinary Design." AIAA 40, no. 2 (February 2002): 301-309.

Alonso, J.J., LeGresley, P., van der Weide, E., Martins, J.R.R.A., Reuther, J.J. "pyMDO: A Framework for High-Fidelity Multi-Disciplinary Optimization." American Institute of Aeronautics and Astronautics, Inc. AIAA-2004-4480 (2004).

Bhaskaran, R. Flow over an Airfoil - Problem Specification. Cornell University. 2002. http://courses.cit.cornell.edu/fluent/wedge/index.htm (accessed May 1, 2009).

Bigarella, E.D.V., Azevedo, J.L.F. "Numerical Study of Turbulent Flows over Launch Vehicle Configurations." Journal of Spacecraft and Rockets 42, no. 2 (March-April 2005).

Clark, E.L. "Aerodynamic Characteristics of the Hemishpere at Supersonic and Hypersonic Mach Numbers." AIAA Journal , 1969.

Crawford, C.A., Haimes, R. "Synthesizing an MDO Architecture in CAD." American Institute of Aeronautics and Astronautics AIAA-2004-281-113 (2004).

Cummings, H.N. "Some Quantitative Aspects of Fatigue of Materials." WADD Technical Report, Curtiss-Wright Corporation, Propeller Division, Caldwell, New Jersey, 1960.

Deepak, N.R., Ray, T., Boyce, R. "Nose Cone Design Optimization for a Hypersonic Flight Experimental Trajectory." 14th AIAA/AHI Space Planes and Hypersonic Systems and Technologies Conference AIAA-2006-7998 (2006).

Dye, C, Staubach, J.B., Emmerson, D., Jensen, C.G. "CAD-Based Parametric Cross-Section Designer for Gas Turbine Engine MDO Applications." Computer-Aided Design & Applications 4, no. 1-4 (2007).

Eremenko, P., Mouton, C.A., Hornung, H.G. "The Pressure Drag of Blunted Cones in Supersonic Flow." American Institute of Aeronautics and Astronautics AIAA-2003-1269 (2003).

Fluent Inc. "FLUENT 6.3 User's Guide." September 20, 2006. http://hpce.iitm.ac.in/Manuals/Fluent_6.3/fluent6.3/help/html/ug/main_pre.htm (accessed May 1, 2009).

Gatzke, T., Dowgillo, R., Ikeda, Y. "The Design of an Aerodynamic Fairing Using Viscous Computational Fluid Dynamics." AIAA, 1998: 711-721.

Khalid, M., McIlwain, S., Chen, S. "Aerokinetic Heating on Projectiles in Hypsersonic Flows." Canadian Aeronautics and Space Journal Vol. 49, no 3 (September 2003): 107-116.

Kim, S-J, Jeon, K-S, Lee, J-W. "Optimal Design of a Space Launch Vehicle Using Response Surface Method." American Institute of Aeronautics & Astronautics, 2000.

King, M.L., Fisher, M.J., Jensen, C.G. "A CAD-centric Approach to CFD Analysis With Discrete Features." Computer-Aided Design & Applications 3, no. 1-4 (2006): 279-288.

Lange, C.H. "Probabilistic Fatigue Methodology and Wind Turbine Reliability." Stanford, CA, 1996.

Lee, J-W, Lee, Y-K, Byun, Y-H. "Design of Space Launch Vehicle Using Numerical Optimization and Inverse Method." Journal of Spacecraft and Rockets 38, no. 2 (March-April 2001): 212-218.

Lee, J-W, Min, B-Y, Byun, Y-H, Kim, S-J. "Multipoint Nose Shape Optimization of Space Launcher Using Response Surface Methodology." Journal of Spacecraft and Rockets 43, no. 1 (January-February 2006).

Lee, Y-K, Lee, J-W, Byun, U-H. "The Design of Space Launch Vehicle Using Numerical Optimization and Inverse Method." AIAA, 1999: 757-767.

Martin, M. P., Wright, M., Candler, G.V., Piomelli, U., Weirs, G., Johnson, H. Preliminary LES over a hypersonic elliptical cross-section cone. Annual Research Brief, Center for Turbulence Research, 2001.

Nielsen, J.N. In Missile Aerodynamics, Chap. 9 Sec.9-4 275-280. New York: McGraw-Hill Book Company Inc., 1960.

Roshanian, J., Keshavarz, Z. "Multidisciplinary design optimization aaplied to a sounding rocket." Indian Inst. Sci., July-Aug. 2006: 86, 363-375.

Samareh, J.A. "Use of CAD Geometry in MDO." Edited by Geometry Laboratory Computer Sciences Corporation. Symposium on Multi-Disciplinary Analysis and Optimization AIAA-96-3991 (1996).

Scott, N., Jensen, C.G. "High-level Operations to Streamline Associative Computer-Aided Design." Computer-Aided Design and Applications. Vol. 6. 2009. 317-327.

Sutherland, I. Sketchpad, a man-machine graphical communication system. PHD Thesis, Massachusetts Institute of Technology, 1963.

Tappeta, R.V. (Univ of Notre Dame), Nagendra, R. "Multidisciplinary design optimization approach for high temperature aircraft engine components." Structural Optimization 18 (1999): 134-135.

Taylor, D.L. Computer-Aided Design. Addison-Wesley, 1992.

Townsend, J.C. Salas, A.O. "Managing MDO Software Development Projects." American Institute of Aeronautics and Astronautics AIAA-2002-5442 (2002).

Unal, R., Lepsch, R.A., Engelund, W., Stanley, D.O. "Approximation Model Building and Multidisciplinary Design Optimization using Response Surface Methods." AIAA, 1996: 592-598.

Yang, M.Y., Wilby, J.F. "Derivation of Aero-Induced Fluctuation Pressure Environments for Ares I-X." American Institute of Aeronautics and Astronautics, Inc. AIAA-2008-2801 (May 2008).

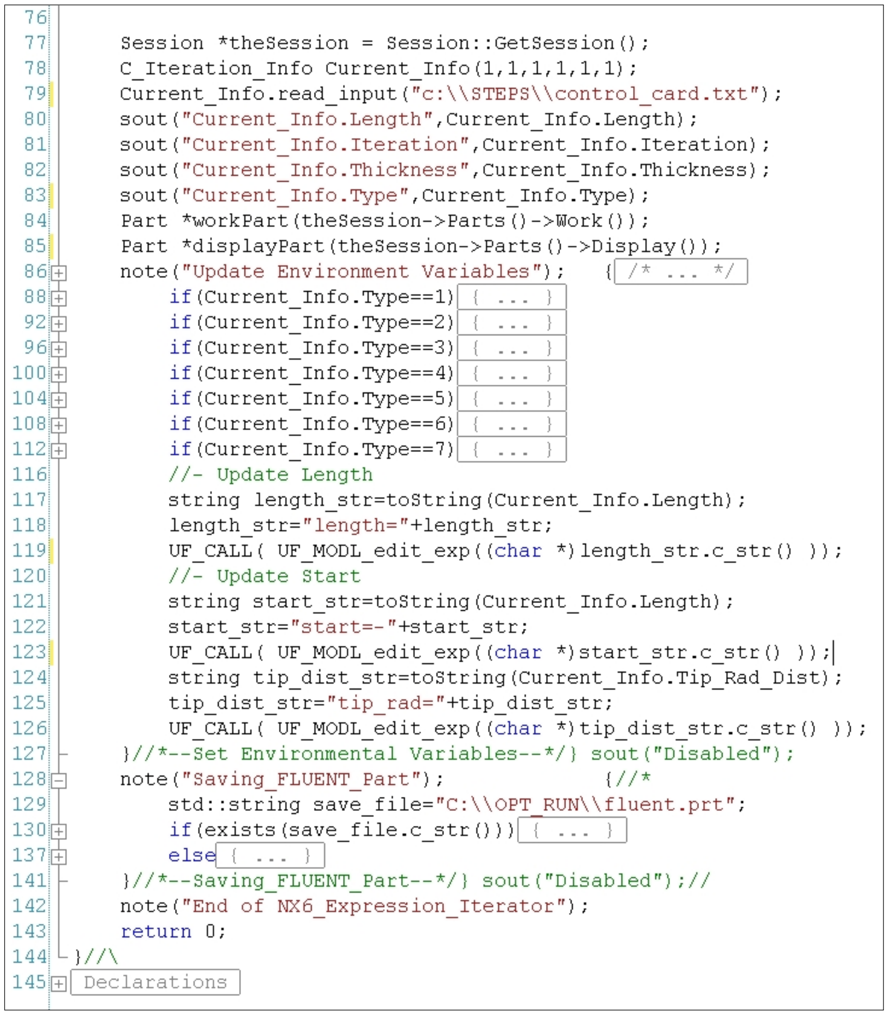
Zeid, I. Mastering CAD/CAM. Boston: McGraw Hill, 2005.

APPENDIX A. ADDITIONAL CODE

The following sections show a compressed version of some of the code that makes up the core efforts of this thesis. It is provided for those who are interested in following how some of the processes were designed to flow and shows in more technical terms the steps already described in both Chapters 3 and 4. First, the C++ Files are shown and then a few other language files are shown that handled some complexities for different programs.

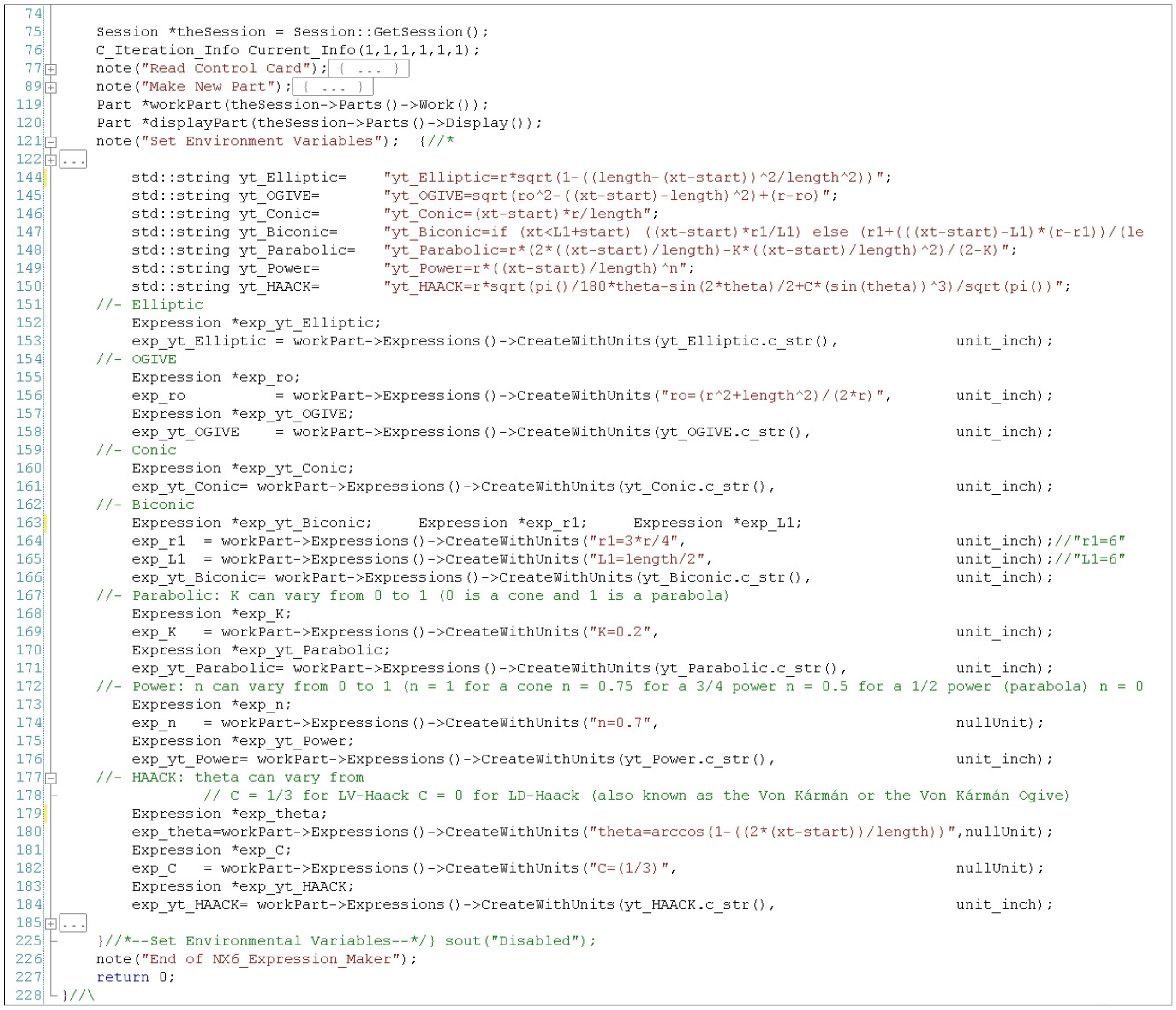
A.1 C++ Files

A.1.1 NX6_Iterator Code



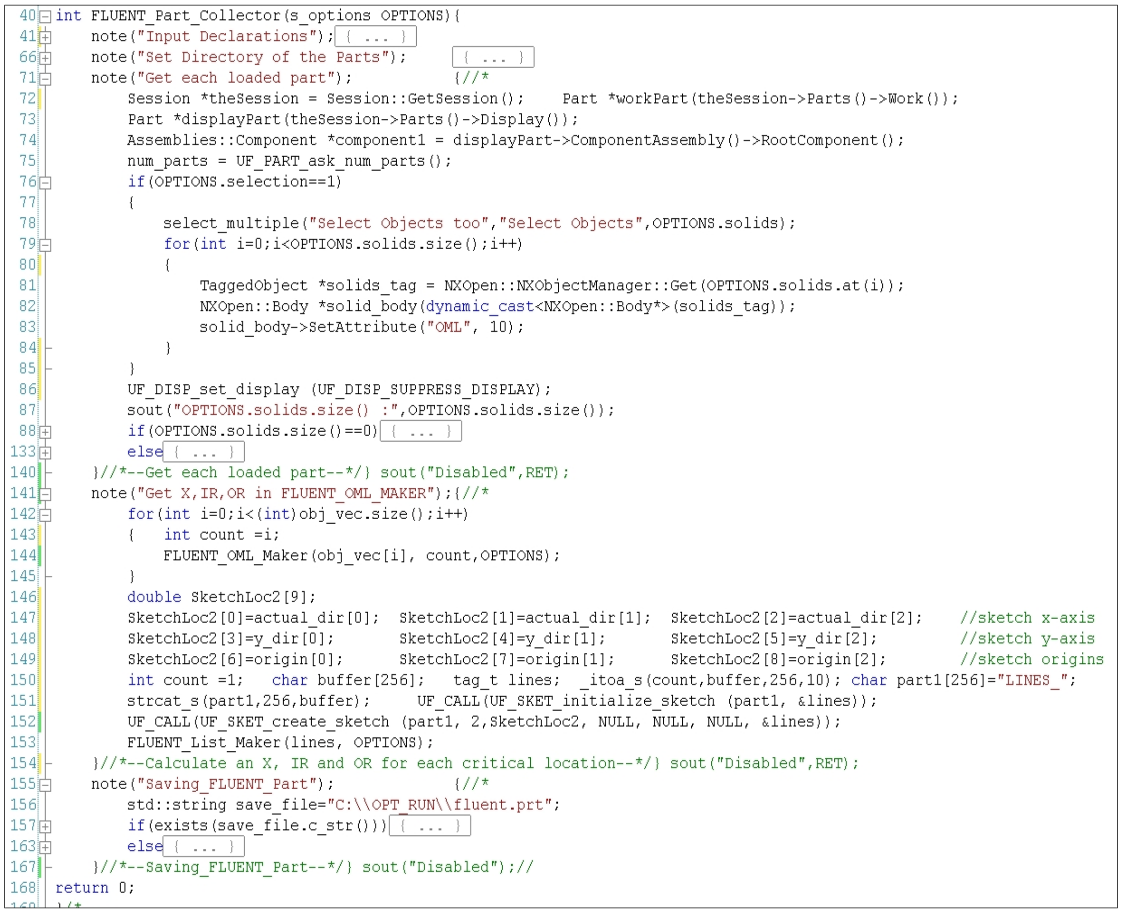
```cpp

    Session *theSession = Session::GetSession();
    C_Iteration_Info Current_Info(1,1,1,1,1,1);
    Current_Info.read_input("c:\\STEPS\\control_card.txt");
    sout("Current_Info.Length",Current_Info.Length);
    sout("Current_Info.Iteration",Current_Info.Iteration);
    sout("Current_Info.Thickness",Current_Info.Thickness);
    sout("Current_Info.Type",Current_Info.Type);
    Part *workPart(theSession->Parts()->Work());
    Part *displayPart(theSession->Parts()->Display());
    note("Update Environment Variables");    {/* ... */
        if(Current_Info.Type==1){ ... }
        if(Current_Info.Type==2){ ... }
        if(Current_Info.Type==3){ ... }
        if(Current_Info.Type==4){ ... }
        if(Current_Info.Type==5){ ... }
        if(Current_Info.Type==6){ ... }
        if(Current_Info.Type==7){ ... }
        //- Update Length
        string length_str=toString(Current_Info.Length);
        length_str="length="+length_str;
        UF_CALL( UF_MODL_edit_exp((char *)length_str.c_str() ));
        //- Update Start
        string start_str=toString(Current_Info.Length);
        start_str="start=-"+start_str;
        UF_CALL( UF_MODL_edit_exp((char *)start_str.c_str() ));
        string tip_dist_str=toString(Current_Info.Tip_Rad_Dist);
        tip_dist_str="tip_rad="+tip_dist_str;
        UF_CALL( UF_MODL_edit_exp((char *)tip_dist_str.c_str() ));
    }//*--Set Environmental Variables--*/} sout("Disabled");
    note("Saving_FLUENT_Part");              {//*
        std::string save_file="C:\\OPT_RUN\\fluent.prt";
        if(exists(save_file.c_str())){ ... }
        else{ ... }
    }//*--Saving_FLUENT_Part--*/} sout("Disabled");//
    note("End of NX6_Expression_Iterator");
    return 0;
}//\
Declarations
```

A.1.2 Expression_Maker Code



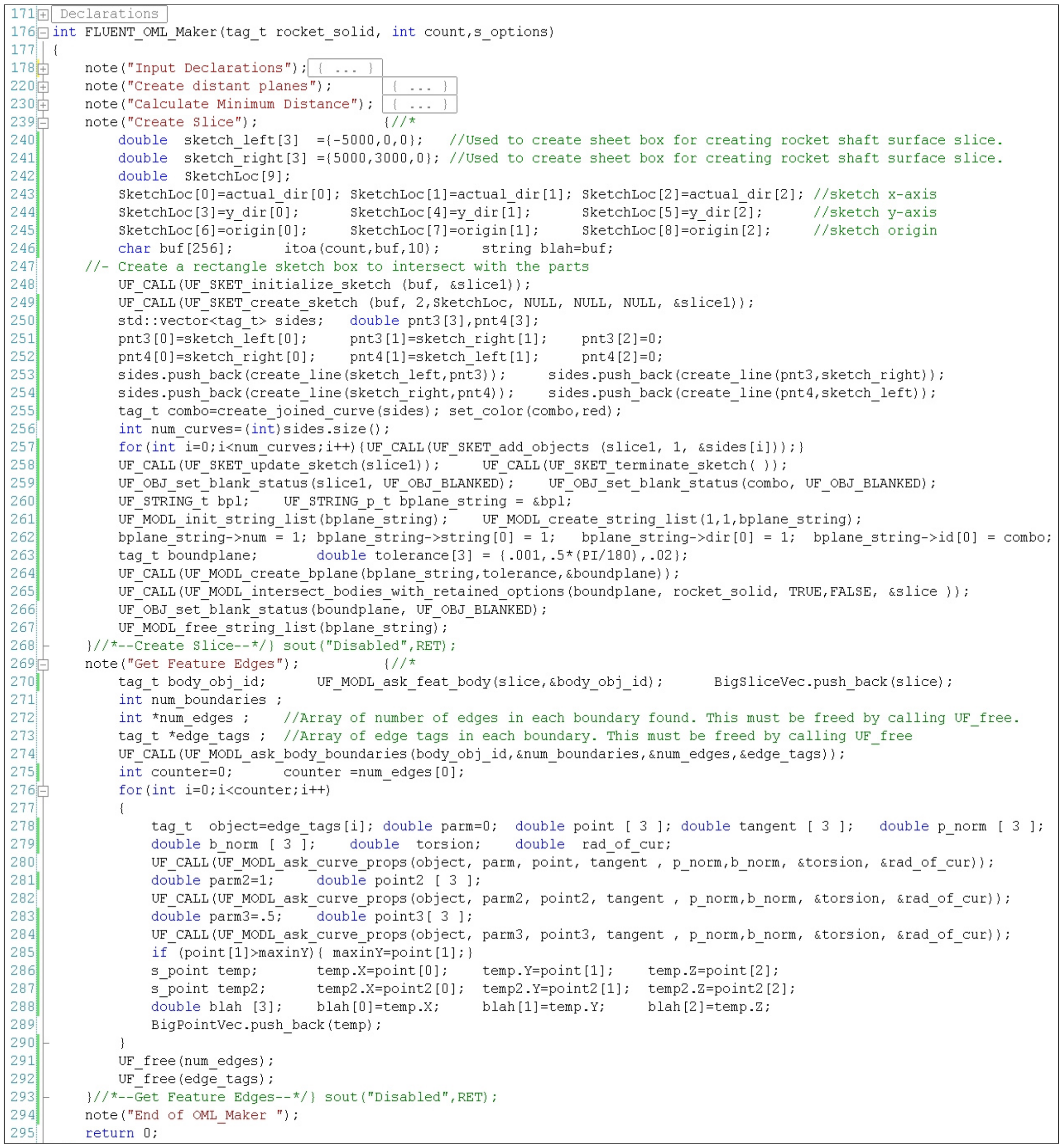
```

    Session *theSession = Session::GetSession();
    C_Iteration_Info Current_Info(1,1,1,1,1,1);
    note("Read Control Card");{ ... }
    note("Make New Part");{ ... }
    Part *workPart(theSession->Parts()->Work());
    Part *displayPart(theSession->Parts()->Display());
    note("Set Environment Variables");  {//*
...
        std::string yt_Elliptic=    "yt_Elliptic=r*sqrt(1-((length-(xt-start))^2/length^2))";
        std::string yt_OGIVE=       "yt_OGIVE=sqrt(ro^2-((xt-start)-length)^2)+(r-ro)";
        std::string yt_Conic=       "yt_Conic=(xt-start)*r/length";
        std::string yt_Biconic=     "yt_Biconic=if (xt<L1+start) ((xt-start)*r1/L1) else (r1+(((xt-start)-L1)*(r-r1))/(le
        std::string yt_Parabolic=   "yt_Parabolic=r*(2*((xt-start)/length)-K*((xt-start)/length)^2)/(2-K)";
        std::string yt_Power=       "yt_Power=r*((xt-start)/length)^n";
        std::string yt_HAACK=       "yt_HAACK=r*sqrt(pi()/180*theta-sin(2*theta)/2+C*(sin(theta))^3)/sqrt(pi())";
    //- Elliptic
        Expression *exp_yt_Elliptic;
        exp_yt_Elliptic = workPart->Expressions()->CreateWithUnits(yt_Elliptic.c_str(),             unit_inch);
    //- OGIVE
        Expression *exp_ro;
        exp_ro          = workPart->Expressions()->CreateWithUnits("ro=(r^2+length^2)/(2*r)",       unit_inch);
        Expression *exp_yt_OGIVE;
        exp_yt_OGIVE    = workPart->Expressions()->CreateWithUnits(yt_OGIVE.c_str(),                unit_inch);
    //- Conic
        Expression *exp_yt_Conic;
        exp_yt_Conic= workPart->Expressions()->CreateWithUnits(yt_Conic.c_str(),                    unit_inch);
    //- Biconic
        Expression *exp_yt_Biconic;     Expression *exp_r1;     Expression *exp_L1;
        exp_r1  = workPart->Expressions()->CreateWithUnits("r1=3*r/4",                              unit_inch);//"r1=6"
        exp_L1  = workPart->Expressions()->CreateWithUnits("L1=length/2",                           unit_inch);//"L1=6"
        exp_yt_Biconic= workPart->Expressions()->CreateWithUnits(yt_Biconic.c_str(),                unit_inch);
    //- Parabolic: K can vary from 0 to 1 (0 is a cone and 1 is a parabola)
        Expression *exp_K;
        exp_K   = workPart->Expressions()->CreateWithUnits("K=0.2",                                 unit_inch);
        Expression *exp_yt_Parabolic;
        exp_yt_Parabolic= workPart->Expressions()->CreateWithUnits(yt_Parabolic.c_str(),            unit_inch);
    //- Power: n can vary from 0 to 1 (n = 1 for a cone n = 0.75 for a 3/4 power n = 0.5 for a 1/2 power (parabola) n = 0
        Expression *exp_n;
        exp_n   = workPart->Expressions()->CreateWithUnits("n=0.7",                                 nullUnit);
        Expression *exp_yt_Power;
        exp_yt_Power= workPart->Expressions()->CreateWithUnits(yt_Power.c_str(),                    unit_inch);
    //- HAACK: theta can vary from
                // C = 1/3 for LV-Haack C = 0 for LD-Haack (also known as the Von Kármán or the Von Kármán Ogive)
        Expression *exp_theta;
        exp_theta=workPart->Expressions()->CreateWithUnits("theta=arccos(1-((2*(xt-start))/length))",nullUnit);
        Expression *exp_C;
        exp_C   = workPart->Expressions()->CreateWithUnits("C=(1/3)",                               nullUnit);
        Expression *exp_yt_HAACK;
        exp_yt_HAACK= workPart->Expressions()->CreateWithUnits(yt_HAACK.c_str(),                    unit_inch);
...
    }//*--Set Environmental Variables--*/} sout("Disabled");
    note("End of NX6_Expression_Maker");
    return 0;
}//\
```

A.1.3 Fluent Part Collector Code



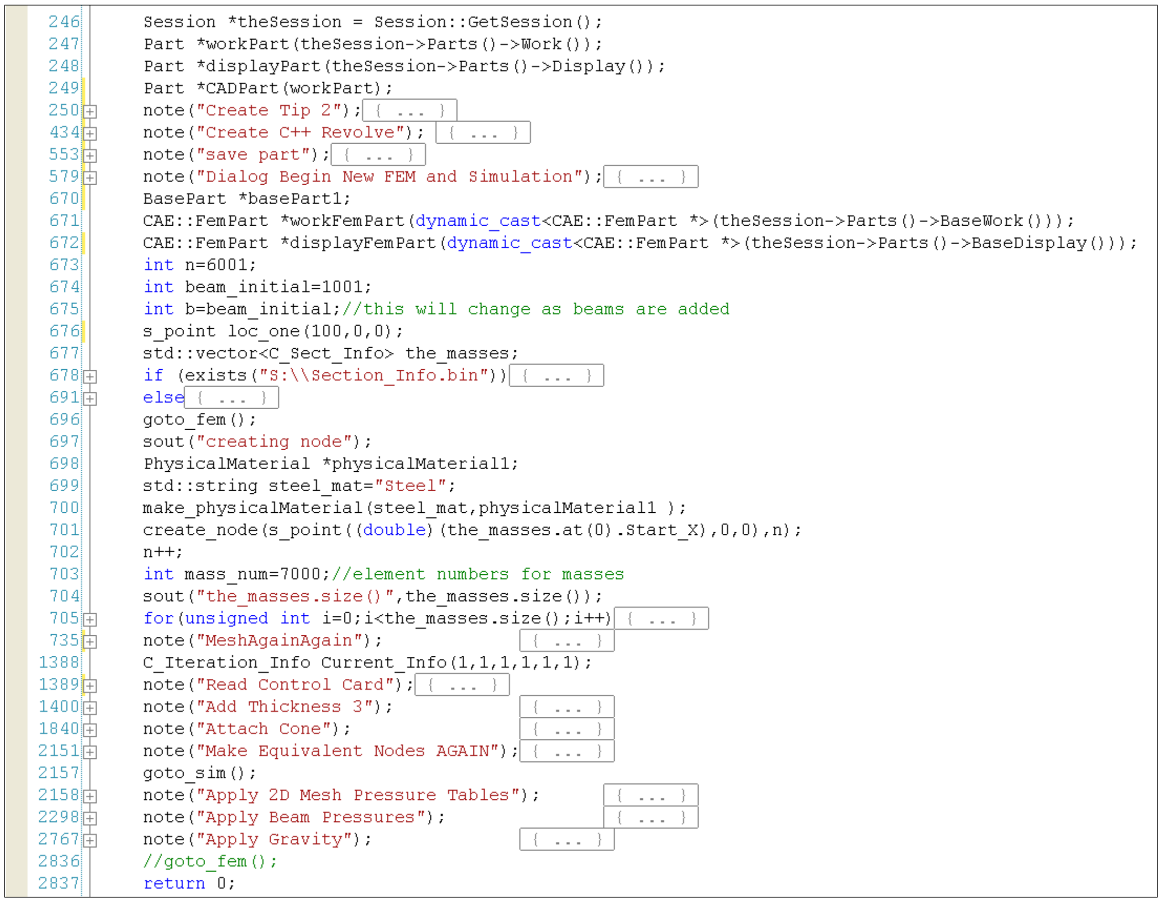
```
int FLUENT_Part_Collector(s_options OPTIONS){
    note("Input Declarations");{ ... }
    note("Set Directory of the Parts");     { ... }
    note("Get each loaded part");           {//*
        Session *theSession = Session::GetSession();    Part *workPart(theSession->Parts()->Work());
        Part *displayPart(theSession->Parts()->Display());
        Assemblies::Component *component1 = displayPart->ComponentAssembly()->RootComponent();
        num_parts = UF_PART_ask_num_parts();
        if(OPTIONS.selection==1)
        {
            select_multiple("Select Objects too","Select Objects",OPTIONS.solids);
            for(int i=0;i<OPTIONS.solids.size();i++)
            {
                TaggedObject *solids_tag = NXOpen::NXObjectManager::Get(OPTIONS.solids.at(i));
                NXOpen::Body *solid_body(dynamic_cast<NXOpen::Body*>(solids_tag));
                solid_body->SetAttribute("OML", 10);
            }
        }
        UF_DISP_set_display (UF_DISP_SUPPRESS_DISPLAY);
        sout("OPTIONS.solids.size() :",OPTIONS.solids.size());
        if(OPTIONS.solids.size()==0){ ... }
        else{ ... }
    }//*--Get each loaded part--*/} sout("Disabled",RET);
    note("Get X,IR,OR in FLUENT_OML_MAKER");{//*
        for(int i=0;i<(int)obj_vec.size();i++)
        {   int count =i;
            FLUENT_OML_Maker(obj_vec[i], count,OPTIONS);
        }
        double SketchLoc2[9];
        SketchLoc2[0]=actual_dir[0];  SketchLoc2[1]=actual_dir[1];  SketchLoc2[2]=actual_dir[2];    //sketch x-axis
        SketchLoc2[3]=y_dir[0];       SketchLoc2[4]=y_dir[1];       SketchLoc2[5]=y_dir[2];         //sketch y-axis
        SketchLoc2[6]=origin[0];      SketchLoc2[7]=origin[1];      SketchLoc2[8]=origin[2];        //sketch origins
        int count =1;   char buffer[256];   tag_t lines;  _itoa_s(count,buffer,256,10); char part1[256]="LINES_";
        strcat_s(part1,256,buffer);     UF_CALL(UF_SKET_initialize_sketch (part1, &lines));
        UF_CALL(UF_SKET_create_sketch (part1, 2,SketchLoc2, NULL, NULL, NULL, &lines));
        FLUENT_List_Maker(lines, OPTIONS);
    }//*--Calculate an X, IR and OR for each critical location--*/} sout("Disabled",RET);
    note("Saving_FLUENT_Part");             {//*
        std::string save_file="C:\\OPT_RUN\\fluent.prt";
        if(exists(save_file.c_str())){ ... }
        else{ ... }
    }//*--Saving_FLUENT_Part--*/} sout("Disabled");//
return 0;
}/*
```

A.1.4 NX6_OML_Maker Code



```
Declarations
int FLUENT_OML_Maker(tag_t rocket_solid, int count,s_options)
{
    note("Input Declarations");{ ... }
    note("Create distant planes");        { ... }
    note("Calculate Minimum Distance");   { ... }
    note("Create Slice");                 {//*
        double  sketch_left[3]  ={-5000,0,0};   //Used to create sheet box for creating rocket shaft surface slice.
        double  sketch_right[3] ={5000,3000,0}; //Used to create sheet box for creating rocket shaft surface slice.
        double  SketchLoc[9];
        SketchLoc[0]=actual_dir[0]; SketchLoc[1]=actual_dir[1]; SketchLoc[2]=actual_dir[2]; //sketch x-axis
        SketchLoc[3]=y_dir[0];      SketchLoc[4]=y_dir[1];      SketchLoc[5]=y_dir[2];      //sketch y-axis
        SketchLoc[6]=origin[0];     SketchLoc[7]=origin[1];     SketchLoc[8]=origin[2];     //sketch origin
        char buf[256];      itoa(count,buf,10);     string blah=buf;
    //- Create a rectangle sketch box to intersect with the parts
        UF_CALL(UF_SKET_initialize_sketch (buf, &slice1));
        UF_CALL(UF_SKET_create_sketch (buf, 2,SketchLoc, NULL, NULL, NULL, &slice1));
        std::vector<tag_t> sides;   double pnt3[3],pnt4[3];
        pnt3[0]=sketch_left[0];     pnt3[1]=sketch_right[1];    pnt3[2]=0;
        pnt4[0]=sketch_right[0];    pnt4[1]=sketch_left[1];     pnt4[2]=0;
        sides.push_back(create_line(sketch_left,pnt3));     sides.push_back(create_line(pnt3,sketch_right));
        sides.push_back(create_line(sketch_right,pnt4));    sides.push_back(create_line(pnt4,sketch_left));
        tag_t combo=create_joined_curve(sides); set_color(combo,red);
        int num_curves=(int)sides.size();
        for(int i=0;i<num_curves;i++){UF_CALL(UF_SKET_add_objects (slice1, 1, &sides[i]));}
        UF_CALL(UF_SKET_update_sketch(slice1));     UF_CALL(UF_SKET_terminate_sketch( ));
        UF_OBJ_set_blank_status(slice1, UF_OBJ_BLANKED);    UF_OBJ_set_blank_status(combo, UF_OBJ_BLANKED);
        UF_STRING_t bpl;    UF_STRING_p_t bplane_string = &bpl;
        UF_MODL_init_string_list(bplane_string);    UF_MODL_create_string_list(1,1,bplane_string);
        bplane_string->num = 1; bplane_string->string[0] = 1;   bplane_string->dir[0] = 1;  bplane_string->id[0] = combo;
        tag_t boundplane;       double tolerance[3] = {.001,.5*(PI/180),.02};
        UF_CALL(UF_MODL_create_bplane(bplane_string,tolerance,&boundplane));
        UF_CALL(UF_MODL_intersect_bodies_with_retained_options(boundplane, rocket_solid, TRUE,FALSE, &slice ));
        UF_OBJ_set_blank_status(boundplane, UF_OBJ_BLANKED);
        UF_MODL_free_string_list(bplane_string);
    }//*--Create Slice--*/} sout("Disabled",RET);
    note("Get Feature Edges");            {//*
        tag_t body_obj_id;      UF_MODL_ask_feat_body(slice,&body_obj_id);      BigSliceVec.push_back(slice);
        int num_boundaries ;
        int *num_edges ;    //Array of number of edges in each boundary found. This must be freed by calling UF_free.
        tag_t *edge_tags ;  //Array of edge tags in each boundary. This must be freed by calling UF_free
        UF_CALL(UF_MODL_ask_body_boundaries(body_obj_id,&num_boundaries,&num_edges,&edge_tags));
        int counter=0;      counter =num_edges[0];
        for(int i=0;i<counter;i++)
        {
            tag_t  object=edge_tags[i]; double parm=0;  double point [ 3 ]; double tangent [ 3 ];   double p_norm [ 3 ];
            double b_norm [ 3 ];    double  torsion;    double  rad_of_cur;
            UF_CALL(UF_MODL_ask_curve_props(object, parm, point, tangent , p_norm,b_norm, &torsion, &rad_of_cur));
            double parm2=1;     double point2 [ 3 ];
            UF_CALL(UF_MODL_ask_curve_props(object, parm2, point2, tangent , p_norm,b_norm, &torsion, &rad_of_cur));
            double parm3=.5;    double point3[ 3 ];
            UF_CALL(UF_MODL_ask_curve_props(object, parm3, point3, tangent , p_norm,b_norm, &torsion, &rad_of_cur));
            if (point[1]>maxinY){ maxinY=point[1];}
            s_point temp;       temp.X=point[0];    temp.Y=point[1];    temp.Z=point[2];
            s_point temp2;      temp2.X=point2[0];  temp2.Y=point2[1];  temp2.Z=point2[2];
            double blah [3];    blah[0]=temp.X;     blah[1]=temp.Y;     blah[2]=temp.Z;
            BigPointVec.push_back(temp);
        }
        UF_free(num_edges);
        UF_free(edge_tags);
    }//*--Get Feature Edges--*/} sout("Disabled",RET);
    note("End of OML_Maker ");
    return 0;
```

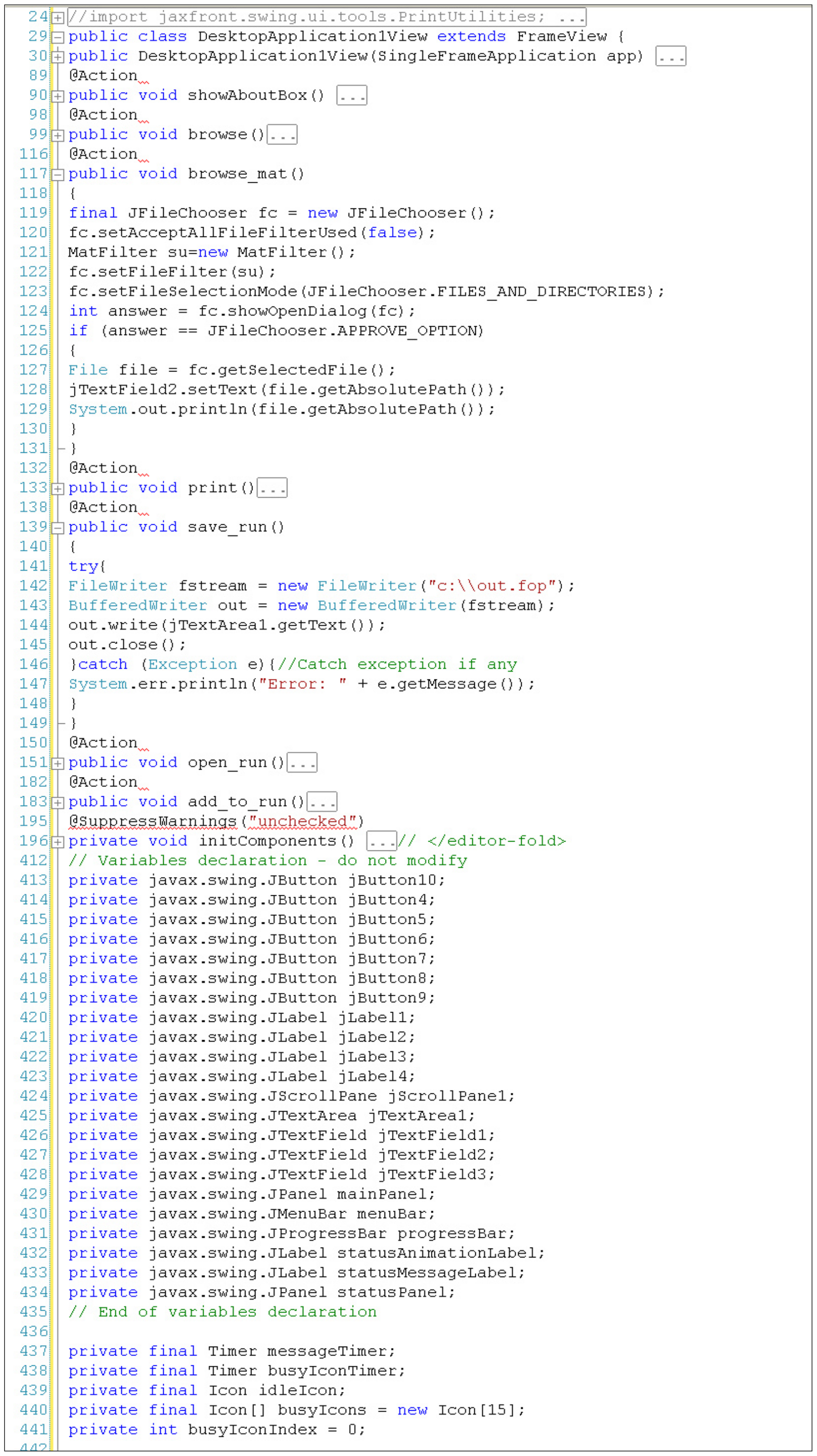
A.1.5 NX6_Fairing_Maker Code



```
Session *theSession = Session::GetSession();
Part *workPart(theSession->Parts()->Work());
Part *displayPart(theSession->Parts()->Display());
Part *CADPart(workPart);
note("Create Tip 2");{ ... }
note("Create C++ Revolve"); { ... }
note("save part");{ ... }
note("Dialog Begin New FEM and Simulation");{ ... }
BasePart *basePart1;
CAE::FemPart *workFemPart(dynamic_cast<CAE::FemPart *>(theSession->Parts()->BaseWork()));
CAE::FemPart *displayFemPart(dynamic_cast<CAE::FemPart *>(theSession->Parts()->BaseDisplay()));
int n=6001;
int beam_initial=1001;
int b=beam_initial;//this will change as beams are added
s_point loc_one(100,0,0);
std::vector<C_Sect_Info> the_masses;
if (exists("S:\\Section_Info.bin")){ ... }
else{ ... }
goto_fem();
sout("creating node");
PhysicalMaterial *physicalMaterial1;
std::string steel_mat="Steel";
make_physicalMaterial(steel_mat,physicalMaterial1 );
create_node(s_point((double)(the_masses.at(0).Start_X),0,0),n);
n++;
int mass_num=7000;//element numbers for masses
sout("the_masses.size()",the_masses.size());
for(unsigned int i=0;i<the_masses.size();i++){ ... }
note("MeshAgainAgain");              { ... }
C_Iteration_Info Current_Info(1,1,1,1,1,1);
note("Read Control Card");{ ... }
note("Add Thickness 3");             { ... }
note("Attach Cone");                 { ... }
note("Make Equivalent Nodes AGAIN");{ ... }
goto_sim();
note("Apply 2D Mesh Pressure Tables");       { ... }
note("Apply Beam Pressures");                { ... }
note("Apply Gravity");               { ... }
//goto_fem();
return 0;
```

A.2 Additional Coding Files

A.2.1 Java Input Code



```java
//import jaxfront.swing.ui.tools.PrintUtilities; ...
public class DesktopApplication1View extends FrameView {
public DesktopApplication1View(SingleFrameApplication app) ...
@Action
public void showAboutBox() ...
@Action
public void browse()...
@Action
public void browse_mat()
{
final JFileChooser fc = new JFileChooser();
fc.setAcceptAllFileFilterUsed(false);
MatFilter su=new MatFilter();
fc.setFileFilter(su);
fc.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
int answer = fc.showOpenDialog(fc);
if (answer == JFileChooser.APPROVE_OPTION)
{
File file = fc.getSelectedFile();
jTextField2.setText(file.getAbsolutePath());
System.out.println(file.getAbsolutePath());
}
}
@Action
public void print()...
@Action
public void save_run()
{
try{
FileWriter fstream = new FileWriter("c:\\out.fop");
BufferedWriter out = new BufferedWriter(fstream);
out.write(jTextArea1.getText());
out.close();
}catch (Exception e){//Catch exception if any
System.err.println("Error: " + e.getMessage());
}
}
@Action
public void open_run()...
@Action
public void add_to_run()...
@SuppressWarnings("unchecked")
private void initComponents() ...// </editor-fold>
// Variables declaration - do not modify
private javax.swing.JButton jButton10;
private javax.swing.JButton jButton4;
private javax.swing.JButton jButton5;
private javax.swing.JButton jButton6;
private javax.swing.JButton jButton7;
private javax.swing.JButton jButton8;
private javax.swing.JButton jButton9;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel2;
private javax.swing.JLabel jLabel3;
private javax.swing.JLabel jLabel4;
private javax.swing.JScrollPane jScrollPane1;
private javax.swing.JTextArea jTextArea1;
private javax.swing.JTextField jTextField1;
private javax.swing.JTextField jTextField2;
private javax.swing.JTextField jTextField3;
private javax.swing.JPanel mainPanel;
private javax.swing.JMenuBar menuBar;
private javax.swing.JProgressBar progressBar;
private javax.swing.JLabel statusAnimationLabel;
private javax.swing.JLabel statusMessageLabel;
private javax.swing.JPanel statusPanel;
// End of variables declaration

private final Timer messageTimer;
private final Timer busyIconTimer;
private final Icon idleIcon;
private final Icon[] busyIcons = new Icon[15];
private int busyIconIndex = 0;
```

A.2.2 Slicer Dicer Text Input Instructions

Slice_Points.txt Instructions

A # sign is a comment

Each line takes in the first three values

Each is a point X, Y, Z, Increment

This pattern repeats until there are no more points

The point inputs are actually carrying two values

The X coordinate is the slice plane region for the increment value.

The y and z are the vector direction to consider what direction should be the representative thickness.

Avoid doors and other large holes using this vector direction control.

If the default increment is used and not the input file option a input of (Total length, 1, 0, Default Increment) will be used.

A.2.3 Slice_Points.txt Example

```
# All “#” lines are comments

# The Zero is the far left of the model

# The 2,2 gives a 45 degree angle for getting OML

# This really only matters when the S:\Sections_Info

# check box is selected and beam sections are made.

# 2,2 is no different from 1,1 on following lines

# The vectors are unitized anyway

# The first line is the increment from 0 to 10.1

# The second is the increment from 10.1 to 26.65 etc.

0,2,2,8

10.1,3,3,12

26.65,1,1,8

37.4125,1,1,5

# The last 1,1,5 are not really used currently

# They still should be input to maintain format and may

# have future meaning as division type specifics
```