Mass Properties Calculation and Fuel Analysis in the Conceptual Design of Uninhabited Air Vehicles

By

Osgar John Ohanian, III

Thesis Submitted to the Faculty of
Virginia Polytechnic Institute and State University
In partial fulfillment of the requirements for the degree of

Master of Science

In

Mechanical Engineering
Virginia Polytechnic Institute & State University

Arvid Myklebust, Chairman
Jan Helge Bøhn
Sam Wilson
Paul Gelhausen

December 1, 2003
Blacksburg, Virginia

Keywords: mass properties, geometry, fuel, conceptual design, UAV, OAV, thin shell, polygonal, polyhedron, slicing, partitioning.

Abstract

Mass Properties Calculation and Fuel Analysis in the Conceptual Design of Uninhabited Air Vehicles

By Osgar John Ohanian, III

The determination of an aircraft's mass properties is critical during its conceptual design phase. Obtaining reliable mass property information early in the design of an aircraft can prevent design mistakes that can be extremely costly further along in the development process.

In this thesis, several methods are presented in order to automatically calculate the mass properties of aircraft structural components and fuel stored in tanks. The first method set forth calculates the mass properties of homogenous solids represented by polyhedral surface geometry. A newly developed method for calculating the mass properties of thin shell objects, given the same type of geometric representation, is derived and explained. A methodology for characterizing the mass properties of fuel in tanks has also been developed. While the concepts therein are not completely original, the synthesis of past research from diverse sources has yielded a new comprehensive approach to fuel mass property analysis during conceptual design. All three of these methods apply to polyhedral geometry, which in many cases is used to approximate NURBS (Non-Uniform Rational B-Spline) surface geometry. This type of approximate representation is typically available in design software since this geometric format is conducive to graphically rendering three-dimensional geometry.

The accuracy of each method is within 10% of analytical values. The methods are highly precise (only affected by floating point error) and therefore can reliably predict relative differences between models, which is much more important during conceptual design than accuracy. Several relevant and useful applications

of the presented methods are explored, including a methodology for creating a CG (Center of Gravity) envelope graph.

Acknowledgements

"Trust in the Lord with all your heart, and lean not on your own understanding, in all you ways acknowledge Him, and He shall direct your paths." Proverbs 3:5-6

"The fear of the Lord is the beginning of knowledge, but fools despise wisdom and instruction." Proverbs 1:7

"Delight yourself in the Lord, and He will give you the desires of your heart." Psalm 37:4

First and foremost, I would like to thank my loving Creator for making me a curious being who loves to explore His creation and for giving me the opportunity to write this thesis. Without Him, I can do nothing.

I would also like to thank my supportive wife, Robin, who has made this arduous journey much more pleasant. Her love and helpful spirit have motivated me to achievements beyond my own expectations.

I would like to thank Arvid Myklebust, my advisor and Masters committee chairman, for his help and guidance all along the way. To the other members of my committee, Jan Helge Bøhn, Paul Gelhausen, and Sam Wilson, I sincerely appreciate your time and input. And to all those at AVID LLC who helped in one way or another, "thanks."

Table of Contents

List of Figures

List of Tables

List of Algorithm Pseudo Code Samples

Chapter 1: Introduction

The reliable calculation of the mass properties of an aircraft during its conceptual design phase is of great importance. The total weight, center of gravity, and moments of inertia affect almost every calculation used in evaluating whether a proposed design is optimal or even viable. In this paper, methodologies and algorithms for analyzing the mass properties of UAV aircraft will be developed and explained. There are diverse and complex considerations that must be addressed to correctly plan the mass distribution in an aircraft. Consequently, in most companies that design aircraft, there are "weight engineers," whose sole purpose is to ensure that the weight and balance requirements of an aircraft are met and are well designed. The developments set forth in this paper will hopefully aid in this effort to produce robust aircraft.

Conceptual Design of Aircraft

The design of aircraft is an inherently complex process. Consequently, there are three generally recognized phases to this development. They are: conceptual design, preliminary design, and detail design. Conceptual design is arguably the most important phase in the whole design process. This is due to the fact that the overall configuration, shape, and performance of the vehicle are determined during this phase [1].

Conceptual design is characterized by the creation and evaluation of many different configurations and geometries to determine the best possible fulfillment of the requirements set out for the aircraft. Many high level questions must be answered during this stage: How much will it cost? What will it look like? How much will it weigh? How well will it perform? At the same time, not much effort is put into investigating the details of the aircraft. The focus is not on individual components or systems, but rather on their interactions and overall efficiency at

meeting the design requirements. Due to the “back of the envelope” nature of this process, throughout this thesis the allowable percentage error in calculations will be assumed to be roughly 10%. However, the precision of the calculations is more important at this point, since reliably predicting relative differences between models is how design decisions are made. Studies are performed to analyze “trade-offs” between requirements and performance to see if the benefits of a change in configuration outweigh the costs.

The art of conceptual design is a very dynamic process, where the design is always being changed to incorporate newly learned information or much needed improvements. Drastically diverse designs should be considered during this phase to ensure that the full realm of possibilities has been explored and to avoid the tendency of pre-conceived ideas preventing the discovery of the optimal design. I referred to this process as the “art” of conceptual design because it requires imagination and innovative thinking that cannot be reduced to a set of principles listed in a book. This is attested to by the unconventional developments like the “flying wing”, V/STOL aircraft, and forward swept wings.

Computer-aided design tools created specifically for the conceptual design phase should be tailored to the fast-paced evolution of an aircraft during its infancy. A designer should be able to create and evaluate a new configuration roughly in the span of one day [1]. To aid in this goal the tools should be intuitive and easy to use, while at the same time extremely productive. Since specific details are not the focus in conceptual design, a user of conceptual design software should not have to worry about the details either. The software should be “smart” enough to fill in some details as the designer wrestles with the larger questions of configuration, shape, size, cost and performance. For instance, if the engine size is increased, the software should recognize that the total weight, fuel consumption, and thrust generated have also been affected. Another example would be that if the propeller diameter were changed, the geometry surrounding it would automatically be modified to ensure that there are no collisions.

Parametric geometric modeling lends itself to this kind of application. Geometric parameters can be linked to other parameters, so that when one changes the other changes accordingly. This kind of system has been implemented in AVID OAV, the software program in which the algorithms developed in this thesis have been implemented.

What is an OAV?

This thesis is specifically concerned with the conceptual design of Uninhabited Air Vehicles, or UAV's for short. A subclass of UAV's under development for the U.S. Army is that of Organic Air Vehicles (OAV). This version of a UAV is a vertical takeoff and landing (VTOL) vehicle, which allows it to land in hard-to-reach places. These vehicles are small enough (6" to 18" in diameter) to be carried by a soldier and be owned and controlled at the platoon or soldier level. Hence, the vehicle is "organic" to a small division of troops instead of being controlled by a remote operator at central command. These vehicles' main objective is to enable troops to gather reconnaissance information, i.e. live video or infrared imaging, while not putting a human life in harm's way. One particular advantage of an OAV is that it should be able to "perch and stare," or in other words, land and perform some task while idle. It can act as a movable sensor, a forward scout, or a laser-targeting device [2].

Another aspect of an OAV that distinguishes it from the larger class of UAV's is that it is "autonomous." This classification implies that there is no remote control of the vehicle. Instead, the vehicle is given a mission and then the onboard computers and navigation systems control the vehicle and guide it to its destination to perform its objective. This is particularly valuable in a battlefield situation, since no human resources are needed to control the vehicle remotely, thus freeing up an extra soldier for tactical operations while still providing invaluable information.

An example may show the worth of this kind of vehicle. Consider a group of special operations soldiers engaged in urban warfare. They can see down the street they are presently on, but would like to know if the crossing street one block away has enemy forces controlling it. An OAV is taken out of a soldier's back pack and programmed to fly to the building top on the next city block and stare down the street, identifying enemy and friendly troops in the unknown territory. The OAV flies into position and perches atop the building while the soldiers are preparing to move. The OAV gives the troops live video of an enemy ambush setup on the next city block, and they change their plans accordingly. The troops pull out, and an air strike is called in to eliminate the entrenched enemy. The OAV targets the enemy with a laser that guides a smart bomb to the correct location, and the friendly troops can now move in. Needless to say, the military would love to have this kind of ability, and hence the "OAV" project was born.

Currently, the most prevalent configuration in OAV design is a ducted-fan VTOL vehicle. The duct is a torus-like geometry with an airfoil cross-section. An example of a duct is shown below in Figure 1. To give a better understanding of what a typical OAV might look like, an example of a complete OAV conceptual design is shown in Figure 2.

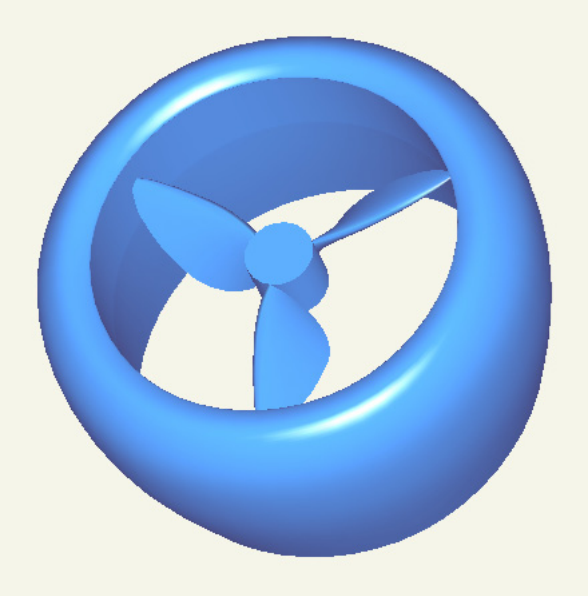

Figure 1: Example of Ducted-Fan Geometry

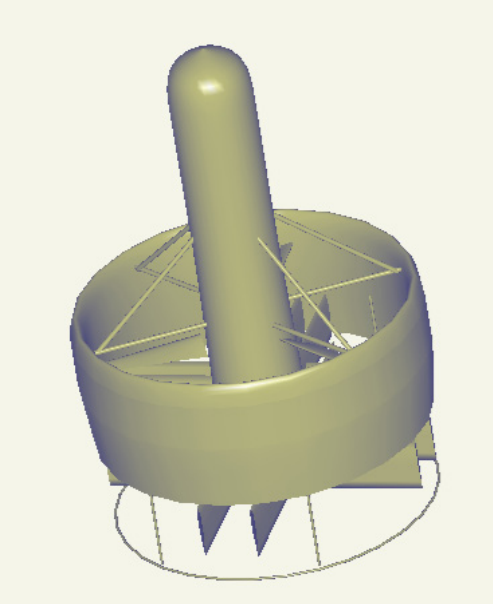

Figure 2: Example of OAV Geometry

Weight and Balance

One of the critical areas of conceptual design is that of “weight and balance.” As can be imagined, the total weight of the vehicle will determine if it can fly or not,

but there are much more subtle aspects of weight and balance that play a vital role in the conceptual design of an aircraft.

In general the term “weight and balance” refers to the mass properties of an aircraft and the resulting stability or lack thereof as a consequence of its mass properties. The term “mass properties” usually includes the following values: volume (or mass or weight), center of mass (or center of gravity), and the moments and products of inertia [3]. The definitions and equations to calculate these values will be discussed in a later chapter, but their usefulness must be explained first.

The weight of the vehicle is trivial to calculate, but can be the difference between the success and failure of a mission. It is essential for calculating the thrust needed for flight by way of derived equations from a “free body” diagram. A free body diagram is used to visually sum all of the forces and moments on an object to determine its static equilibrium or it’s dynamic response due to a particular loading [4]. From these equations the thrust and angular position of the vehicle can be determined to accomplish a desired constant velocity or acceleration. It is also important to note that a vehicle’s weight will change throughout its mission due to fuel burn and payload release.

The distribution of the vehicle’s weight is also of major concern. The locations of the aircraft’s components and their corresponding weights will determine where the aircraft’s center of gravity is located. To further explain the importance of this point in space, the definition must be well understood. In common terms, the center of gravity (CG) can be thought of as the location of the point where an object would balance on the head of a pin, regardless of the object’s orientation. This point is where the object’s gravity force (or weight) acts. Also, it is around this point that the object naturally rotates if a torque is applied. If a force is applied to an object and the force vector does not intersect the CG, then a torque is also created about the CG proportional to the moment arm (perpendicular

distance from CG to the force vector). In the case of aircraft, as it flies through the air, each of its components experiences aerodynamic forces (not at the aircraft CG), thus creating torques about the CG. To balance these torques, generally a control surface is deflected a certain amount to produce a counter torque to nullify the overall moment felt at the CG. For this reason, the CG location is critical for balancing the vehicle.

Finally, the moments and products of inertia (MOI and POI) of the aircraft are also determined by the distribution of the mass throughout the vehicle. Each component's MOI and distance from the aircraft CG affect the overall MOI. The MOI are used in dynamic analysis of the aircraft and also to size the control surfaces. While they are not used in many conceptual design calculations, they are still important and should be calculated during the conceptual design phase. One example of a study where MOI are critical would be to determine the dynamic response of the aircraft to a sudden gust of wind.

Problem Statement

Fast approximation methods for calculating the mass properties of solids defined by arbitrary surface geometry are needed for the analysis of OAV conceptual designs. The allowable error percentage in each calculation is roughly 10%, but the precision of the method and its ability to identify relative trends is more important. The arbitrary surface geometry will represent either a solid component or a thin shell object; therefore, the methods must be developed to handle both of these possibilities. The mass properties (or geometric properties) of interest are primarily:

- Total weight (or volume)
- Location of center of gravity
- Moments and product of inertia about the center of gravity

A method for addressing the special qualities of fuel mass properties must also be developed. The effects of fuel burn throughout a mission and the fuel's deformation due to aircraft rotation must also be accounted for in the method developed. The moments of inertia of the fuel must also be handled specially, since the fuel cannot be treated as a solid body.

The geometry supplied will come in the form of a tessellated (or triangulated) approximation of a B-Spline surface. It may be assumed that the surface geometry supplied represents a closed boundary.

Possible applications of the methods developed should also be explored. Specifically, a procedure for constructing a "CG envelope" graph must be derived and explained using the methods presented.

Chapter 2: Literature Review

There are many areas of research that the objectives of this thesis depend upon. They vary from pure mathematics, to aircraft design, to solid modeling, and even computer graphics. Each area, its relevance, and some selected sources will be discussed below.

Mass Properties: Mathematics

The derivation of equations of the mass properties of a three-dimensional solid are readily available in any calculus text book, such as Stewart's [5], where equations are first formulated for center of mass (or balancing point) of a line, then a plane, and finally a volume. Equations for moments of inertia are developed in a similar manner. However, a text oriented more towards engineering like Beer and Johnston's [4] gives a better explanation of how to calculate and apply mass properties to static and dynamic mechanics problems. Beer and Johnston explain the parallel axis theorem and how to calculate CG and MOI for an assembly of shapes. They list mass properties for common geometric shapes, and also provide useful information on how to calculate a moment of inertia of a mass about an arbitrary axis.

The Society of Allied Weight Engineers (SAWE) Weight Engineer's Handbook [3] is also a good resource for a broad range of mass properties related information. It covers calculating and measuring mass properties, as well as how these values affect aircraft design. It has an extensive list of mass properties formulae for known geometric shapes, and procedures for almost every imaginable mass property calculation (some practical, some obscure). However, one drawback of the information in the handbook is the ambiguity of English units used in their tables and calculations.

There are numerous articles that address particular aspects of mass properties calculation. Matrix transformations applied to mass properties are very well addressed in [6] and [7]. They both represent the CG as a vector and the MOI and POI as an inertia tensor, to which linear transformations can be applied to rotate the coordinate system. Nakai's explanation builds from the ground up to produce a very clear understanding of tensors, cosine matrices, principle moment directions, and effects of POI. Strom goes on to prove that in general one MOI cannot be greater than the sum of the remaining two. Some very interesting alternate methods for calculating mass properties are set forth in [8] using "third moments", but are limited to swept volumes like revolutions and prisms, and are not applicable to the objectives of this thesis.

Geometric Representation

There are many ways to represent a particular object's geometry. It is the aim of computer-aided design systems to create and store model geometries that are useful for design and analysis. Books by Farin [9], Mortenson [10], and Faux and Pratt [11] cover extensively a wide range of geometric modeling topics including hermite, Bézier, and B-Spline curves and surfaces. Mäntyla [12] and Mortenson [10] handle the topological aspects of computer-aided design in their explanations of solid modeling and representation schemes.

Mäntyla goes on to say that there are three major categories of representation schemes: decomposition models, constructive models, and boundary models. He explains that decomposition models represent a set of points as a collection of simple objects from a fixed library of primitive shapes, which are then "glued" together. The "primitive instancing" scheme falls into this category [12].

Constructive models, similar decomposition models, represent a point set by a collection of primitive point sets. What makes this category distinct from decomposition is that each primitive is an instance of a primitive solid type, and

these primitives can be combined, subtracted and intersected through Boolean set operations. This provides much more flexibility than the single gluing operation used in decomposition. “Constructive solid geometry” or CSG is a representation scheme that is included in this category [12].

Finally, boundary models represent a point set merely by its boundary. The boundary of a three-dimensional object is a three-dimensional surface, which can be represented by a parametric equation of two parameters. Such surfaces usually are divided into a collection of “faces”. Each face can also be represented by its boundary, a closed curve defined by a single parameter. Therefore, boundary models can be composed of a hierarchy of simpler models. The representation scheme known as “B-Rep”, or boundary representation, falls into this category. A B-Rep uses the topological entities of vertices, edges, and faces to define the connectivity of its geometry that consists of three-dimensional points, curves and surfaces.

The geometry generated by any modeling scheme must eventually be rendered to a computer screen for visualization. The geometry of the faces of a model can consist of planes, conics, and even freeform surfaces like NURBS (Non-uniform Rational B-Splines). There must be a generalized way to process the varied geometry to facilitate efficient rendering algorithms. The most common method is to approximate the surface geometry with planar polygon facets, since there are numerous algorithms for rendering polygon data [13]. Luken [13, 14], Piegl and Tiller [15], and Sheng and Hirsch [16] all provide methods for creating such tessellations of surface models. Once this approximation has been made, all topological information has been lost and a set of disjoint planar polygons is left. This kind of tessellated or triangulated model, essentially a polyhedron, is still of great use to mass properties calculation, as will be seen in the next section.

Mass Properties: Computer Algorithms

Numerous algorithms have been written to calculate the mass properties of geometric shapes, dating back all the way into the 1960's when CAD research was getting started. Lee and Requicha give a good overview of many such algorithms in [17]. They recognize that certain geometric representation schemes lend themselves toward different mass properties analysis routines. The types of representation they address are: primitive instancing, quasi-disjoint decomposition, simple sweeps, boundary representation (B-Rep), and constructive solid geometry (CSG). In the second part of their article Lee and Requicha explain a new method for analysis of CSG models [18]. However, the boundary representation scheme is the most relevant to the methods devised in this thesis, so while mass properties calculation algorithms for the other representations are interesting, they will not be discussed further.

Lee and Requicha divide the category of methods for analyzing B-Reps in two: divergence theorem methods and direct integration methods. These methods have the advantage that they do not need to decompose the model into standard geometric forms, but only need the surface of the model on which to operate [19]. It seems that Messner pioneered the divergence theorem methods in the 1970's [19, 20]. He shows that for a general polyhedron, which approximates a curved surface, the mass properties of the volume can be evaluated by integrating over the surface by means of the Gauss divergence theorem. The theorem states: Given a solid region V, and its surface S,

$$\iiint_V \operatorname{div} \mathbf{F}\, dV = \iint_S \mathbf{F} \bullet \mathbf{n}\ \ dS \tag{1}$$

where **F** is a vector function with continuous partial derivatives, **n** is the unit vector normal to the surface, and

$$\operatorname{div}\mathbf{F} = \sum_i \frac{\partial}{\partial x_i} F_i \tag{2}$$

$$\mathbf{F} \bullet \mathbf{n} = \sum_i F_i n_i \tag{3}$$

as set forth in [5], [19], and [21]. It can easily be shown that if the surface were divided into disjoint sections without any gaps, the sum of the surface integrals over the sections would be equal to the integral over the whole surface,

$$\iint_S \mathbf{F} \bullet \mathbf{n} \; dS = \sum_i \iint_{P_i} \mathbf{F} \bullet \mathbf{n}_i \; dP_i \tag{4}$$

where P_i and n_i are the ith surface patch and normal vector, respectively [17]. In Messner's method, each surface patch is a polygonal face, which can be broken into triangles over which a numerical integration is performed. To attain the integral over the whole surface, the integrals of all the triangular faces are summed.

An example of how this approach is implemented will give a clearer understanding of the method. Consider the moment of inertia about the x-axis, I_{xx}. The usual equation for this value is:

$$I_{xx} = \iiint \rho\left(y^2 + z^2\right) dV \tag{5}$$

Suppose a vector function **F** is defined to be:

$$\mathbf{F} = \frac{\rho}{3}\left\{y^3\mathbf{j} + z^3\mathbf{k}\right\} \tag{6}$$

Where **j** and **k** are the unit vectors in the y and z directions, respectively. Therefore the divergence of function F would be:

$$\operatorname{div}\mathbf{F} = \frac{\rho}{3}\left\{\frac{\partial(y^3)}{\partial y} + \frac{\partial(z^3)}{\partial z}\right\} = \rho\left(y^2 + z^2\right) \tag{7}$$

Applying Gauss' divergence theorem, we attain:

$$\iiint \rho\left(y^2 + z^2\right)dV = \iint \frac{\rho}{3}\left\{y^3 n_y + z^3 n_z\right\}dS \tag{8}$$

At this point, the integration is carried out using a quadrature rule on each triangular face. This method basically evaluates the integral's function at specific points within the triangle, prioritizes each point's influence through weighting coefficients, and finally sums the values together. In the case of a triangle, the Gauss-Radau four-point rule prescribes that the function be evaluated at the centroid (p_1), and three other points (p_2 through p_4), each 40% of the distance from the centroid to each of the vertices [19]. In this way a function, f, can be integrated:

$$\iint f\ dA = area * \left[f(p_1)w_1 + f(p_2)w_2 + f(p_3)w_3 + f(p_4)w_4\right] \tag{9}$$

where w_1 = -27/48 and w_2 = w_3 = w_4 = 25/48 [20]. Applying this method to Equation (5) above and summing over all the triangles in the surface will give I_{xx} for the whole solid. Likewise this method is used to calculate each of the mass property values.

Cattani and Paoluzzi develop a method that also uses the divergence theorem in [22], but opt not to use numerical integration. Instead they transform the surface triangles into the unit triangle and perform integrations directly using a change of

variables. Gonzalez-Ochoa, McCammon and Peters present a similar method, except that instead of operating on a tessellated approximation of a surface, they apply the Gauss divergence theorem directly to polynomial freeform surfaces [21]. They change the variables of integration so that they are evaluated over the surface parameters u and v instead of the surface area.

The other branch of mass properties calculation methods for boundary representation, mentioned by Lee and Requicha, is direct integration. Lien and Kajiya propose a method in [23] that falls into this category. Instead of changing the integral from a volume integral to a surface integral, they develop a general method for breaking the solid volume into smaller volumes that can be efficiently integrated. More specifically, a central projection from each face of the solid to an "origin" is used to create a set of tetrahedra whose union is equivalent to the original solid. Integrals for volume, center of gravity, and moments of inertia can then be evaluated for each tetrahedron and summed to determine the total integral.

The methods developed in this thesis are based on Lien and Kajiya's method. The simplicity, robustness, and efficiency of this method were the main advantages that influenced this choice. More details of how this method works will be discussed in the next chapter.

Aircraft Design

Several authors address the general topic of mass properties in the design and balance of aircraft, such as in [1] and [24]. Raymer discusses two methods for weights estimation: the approximate group weights method, and the statistical group weights method. Once the weights of particular groups of components (such as propulsion, fuselage, wings, and landing gear) have been estimated, the CG for the whole craft can then be estimated. Roskam has devoted a whole book in his series on airplane design to estimating component weights, CG, and

MOI. Both of these authors present methods that are based on historical data from similarly sized aircraft. Unfortunately, when designing OAV's, the lack of historical examples of past designs makes these methods impractical.

Other more specific studies have been performed to address issues such as conceptual estimation of MOI by Scott [25], real-time monitoring of CG and fuel level by Adelson [26], as well as automated weight and balance systems as described by Quinlivan [27]. Hargrave shows how computer graphics can be a good tool for analyzing mass properties during the design process [28].

Wiegand does a good job of presenting the basic algorithms of mass properties analysis for aircraft [29]. He describes a method via spreadsheet to accomplish all of the accounting and summing of weights to generate the aircrafts overall weight, CG, and MOI. He also provides error analysis with standard deviations for each computed quantity.

Fuel Tanks / Fluid Dynamics

The presence of contained fluids in an aircraft dramatically increases the difficulty in reliably calculating its mass properties. This is due to the fact that, as a liquid, the fuel deforms continuously under shear loads [30]. In common speech, this means that a liquid will conform to its container. Therefore, as the aircraft changes its orientation with respect to the force of gravity, the fuel will slosh and find a new equilibrium shape that will minimize its potential energy. Not only that, but the quantity of fuel is constantly changing throughout the mission due to the fact that the propulsion system is constantly consuming fuel to keep the aircraft aloft. It is also difficult to quantify a moment of inertia for a body of liquid. As a fuel tank rotates or accelerates, the fuel therein will not experience the same rigid body motion since it will deform and slosh. These three characteristics of fluids make it extremely hard to give exact values for the mass properties of the fuel onboard the aircraft. For this reason, baffles and other

devices have been introduced in fuel tanks in certain cases to try to decrease these effects.

The mass properties of the fuel, which affect the mass properties of the total aircraft, are of great importance for the stability of the vehicle. To demonstrate this, consider if the fuel travels to an extreme portion of the fuel tank, it could cause the aircraft's CG to leave the acceptable CG envelope, which puts the stability of the craft in jeopardy. In another case, if the fuel sloshes to one side of the vehicle, it could result in an unfavorable reaction in the flight of the aircraft.

Boynton explores, through experimentation, the moments of inertia of fluids in a tank [31]. There are several parameters that he considered in his experiments: tank size, aspect ratio, fuel viscosity, and percent full. When quantifying the data he collected, he expresses the fluids moment of inertia as a fraction of what its solid equivalent would be.

With respect to the sloshing of propellant in tanks, there have been considerable research efforts to explore this region of aircraft design. Several NASA reports [32-35] investigate, in depth, the dynamics of fuel sloshing in tanks for the design of rockets. They start with the Navier-Stokes equations and show their assumptions for simplification. However, they only consider a sinusoidal forcing function when analyzing the steady state sloshing of the liquid. A transient analysis of how the fuel sloshes and comes to equilibrium during a stepped change in pitch would have been more applicable to OAV dynamic analysis, but research on such a topic could not be found.

Dodge [36] and Hassman [37] both present a method of representing the fuel as an equivalent mechanical system made of masses, springs, and dampers. This would be quite desirable since, the behavior of such dynamic systems governed by differential equations is well known. However, the formulae they present are

for specific tank geometries and do not apply to the general problem of fuel sloshing in a container of arbitrary geometry.

Boesch and Bell describe the behavior of fuel slosh between partitioned tanks [38]. Unfortunately, they do not address the dynamic nature of the sloshing or the change in shape of the fuel geometry, but they do provide an effective method for calculating the amount of time needed for the fuel to reach an equilibrium level after a pitch-altering maneuver.

Molczyk shows how a simulation of fuel CG versus vehicle attitude was used during the design of cruise missiles [39]. The method used was of a “brute force” nature and also did not calculate MOI, and therefore is not appropriate for OAV conceptual design.

Chapter 3: Mass Properties Calculation

Definitions and Equations

To start out, it would be a good idea to clearly define the equations for determining mass, center of gravity, and moments and products of inertia. Below in Equations (10) through (19) are the definitions adapted from reference [4]:

Mass: $$M = \rho V = \rho \int dV \tag{10}$$

X coordinate of CG: $$\bar{x} = \frac{\int x\, dV}{V} \tag{11}$$

Y coordinate of CG: $$\bar{y} = \frac{\int y\, dV}{V} \tag{12}$$

Z coordinate of CG: $$\bar{z} = \frac{\int z\, dV}{V} \tag{13}$$

X-axis MOI: $$I_{xx} = \rho \int \left(y^2 + z^2\right) dV \tag{14}$$

Y-axis MOI: $$I_{yy} = \rho \int \left(x^2 + z^2\right) dV \tag{15}$$

Z-axis MOI: $$I_{zz} = \rho \int \left(x^2 + y^2\right) dV \tag{16}$$

XY POI: $$I_{xy} = \rho \int xy\, dV \tag{17}$$

YZ POI: $$I_{yz} = \rho \int yz\, dV \tag{18}$$

ZX POI: $$I_{zx} = \rho \int zx\, dV \tag{19}$$

To gain a more intuitive understanding of these quantities, some explaining is in order. The general idea is to sum the contributions of each infinitesimal particle of mass multiplied by a particular function. For instance, in the mass equation (10), there is no function integrated over the volume, since only the volume is required to calculate mass when given a uniform density.

The equations for center of gravity, Equations (11) through (13), are a bit more complex. These equations can also be referred to as “first moment” equations. This is due to the fact that each infinitesimal volume is multiplied by a moment arm. This moment arm adds the concept of position of each particle into the equation. The process requires that the moments (instead of masses) are summed, and then divided by the total volume. In essence this is like taking an average of all the positions of the points in an object. The final result is the coordinates of the CG (or centroid). The moments summed about this point, due to the mass of the object, equal zero. Therefore this point represents a balancing point for the whole solid, and the total weight or gravity force can be represented as acting at this point.

The equations for moment of inertia, Equations (14) through (16), are also referred to as “second moment” equations. This is due to the squared moment arm that multiplies each infinitesimal volume during the integration. In the case of the I_{xx}, the distance from the x-axis is the moment arm to be squared, and due to the Pythagorean theorem, this squared distance is $y^2 + z^2$. The same method is used for the other moments of inertia. The usefulness of this squared moment arm is derived from the fact that the amount of time needed for a point mass at a particular radius to reach a given speed is proportional to its mass and the square of the radius [4]. The moments of inertia of an object can intuitively be understood as its resistance to rotation (angular accelerations and decelerations).

The equations for product of inertia, Equations (17) through (19), represent the imbalance of the object when rotated about the current coordinate system axes. An object will naturally spin about its principle axes. When the coordinate system lines up with the principle axes, the products of inertia are zero. The principle moments of inertia, the MOI in the principle axes coordinate system, are intrinsic to the geometry of the object. Products of inertia, however, are not intrinsic to the object, but are merely a function of its alignment or misalignment with the principle axes [6]. For this reason, the moments and products of inertia are often represented as an "inertia tensor", which is illustrated below.

$$\mathbf{I} = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{zx} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{yz} & I_{zz} \end{bmatrix} \tag{20}$$

The eigenvectors of this inertia tensor matrix are the principle axes of the coordinate system in which the principle moments of inertia are defined. The principle moments of inertia are the eigenvalues of the same matrix. To visualize the situation a tool called Mohr's Circle is used. It is better known for visualizing stress tensors, but also applies to inertia tensors as well [4].

Difficulties of Implementation

For homogeneous objects (uniformly distributed mass), the equations laid out above are very straightforward; the integration performed is over simple functions. In most mathematical studies involving integrals, the domain is very simple and the integrand (function) is very complex. The converse is the case in mass properties calculation. While the integrand is almost trivial, the geometric domain can be extraordinarily complex [17]. The domains in geometric and solid modeling are usually of arbitrary shape. They can be convex or concave, have through-holes, and usually consist of more than one surface. What makes

integrating over this kind of domain even more difficult is that the boundary cannot usually be represented by a single equation [40, 41]. It must also be recognized that the computational error in mass properties analysis is mostly caused by the approximations in representing the geometric model, instead of the traditional sources of error such as round off, truncation, cancellation, and approximate integration techniques [17].

The geometry representation encountered in this study is of the boundary representation (B-Rep) form. This is the most prevalent representation scheme at this time; it is being used in the most prominent commercial modeling kernels ACIS and ParaSolid. Since the surfaces that make up the faces of solid models can be extremely complex polynomial or B-Spline surfaces, surface approximations are commonly used for graphical display and mass properties calculation. Generally, the surfaces are tessellated and represented by numerous polygons, which form one polyhedron. This type of representation will be used in this thesis for all mass properties calculation methodology. However, it is this approximation of the geometry that is the major source of inaccuracy in the results.

Central Projection Algorithm

The relations proposed by Lien and Kajiya was implemented in an algorithm by Lin and discussed in his paper, dissertation, and report [40, 41, 42]. It is also used here as a starting point for mass properties calculation. The algorithm is based on the concept of central projection. Three points on the surface of the model are connected with the origin to create a tetrahedron. A visual example of this is shown below in Figure 3.

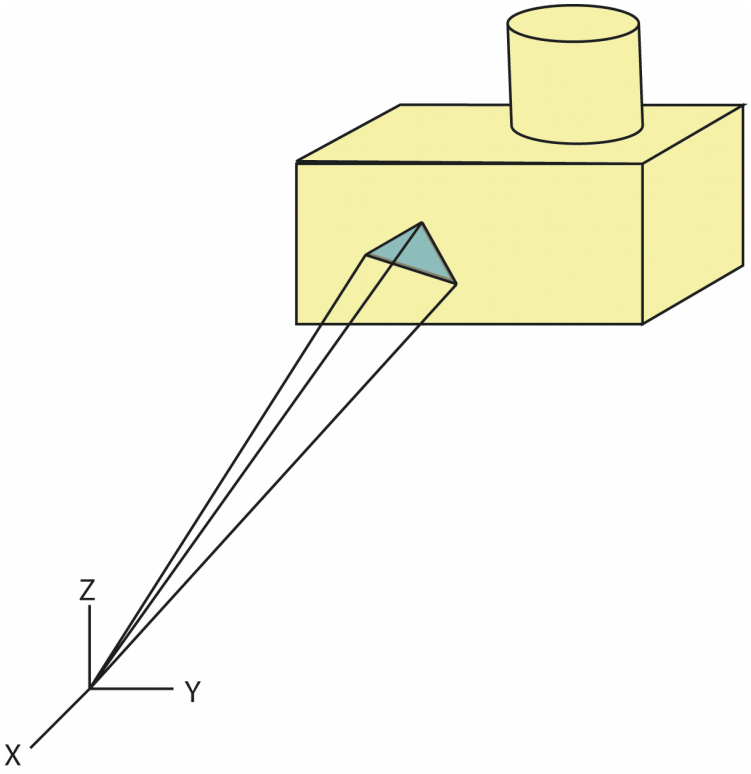

Figure 3: Tetrahedron Created by Central Projection

By tessellating the entire model surface and projecting to the origin, it can be decomposed it into a finite set of tetrahedra. Integration of the mass properties Equations (10) through (19) over this set of domains is much simpler than trying to integrate over the original domain. To illustrate the concept of this method, let us first look at a two-dimensional case: calculating the area of a general polygon. Any polygon can be decomposed into a set of triangles. Figure 4 illustrates this:

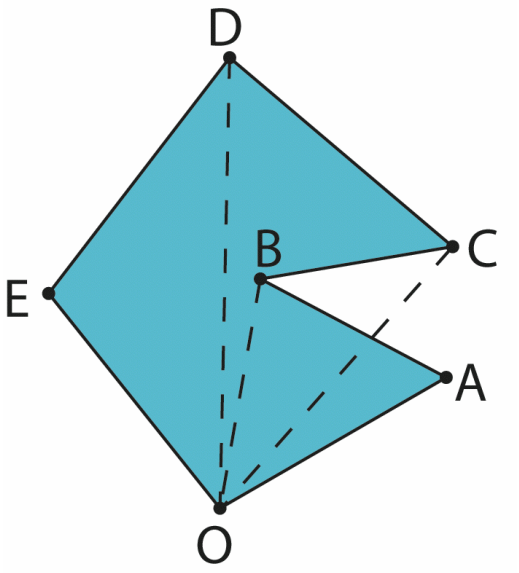

Figure 4: Polygon Decomposition into Triangles

If the points are traversed in order, creating triangles from adjacent points and the first point (acting as the origin), the triangles will cover the whole area of original polygon. One subtlety is that there are positively and negatively contributing triangles. Triangles that have positive angles at the origin (right-hand rule) are additive, and triangle that have negative angles at the origin point are subtractive. In mathematical terms, this is elegantly expressed through vector operations. The area of the triangle OAB in the figure above would be:

$$\begin{aligned} Area &= \tfrac{1}{2}\left\|\mathbf{r}_A \times \mathbf{r}_B\right\|, \\ \mathbf{r}_A &= \mathrm{A} - \mathrm{O}, \mathbf{r}_B = \mathrm{B} - \mathrm{O} \end{aligned} \tag{21}$$

In other words, the area of a triangle is one half of the magnitude of the cross product of its side vectors. Vector cross products, or outer products, return a vector that is perpendicular to the two original vectors, which also obeys the "right-hand rule". This means that if you curl your right-hand fingers around from the first vector to the second vector, your thumb will be pointing in the direction of the resultant vector. So for positive angles (counterclockwise) between vectors, their cross product will have a positive magnitude, and the opposite for negative

angles. Therefore, in Figure 4, we can see that triangle OAB has positive area, while triangle OBC has negative area (in the context of the polygon). In this way, we can see that additive area outside the boundary is cancelled out by subtractive area (as is duplicated additive interior area), thus leaving only the area of the interior of the polygon. Another way to understand this phenomenon is to look at the normal vectors of each segment. Figure 5 shows an enhanced Figure 4 with segment normal vectors.

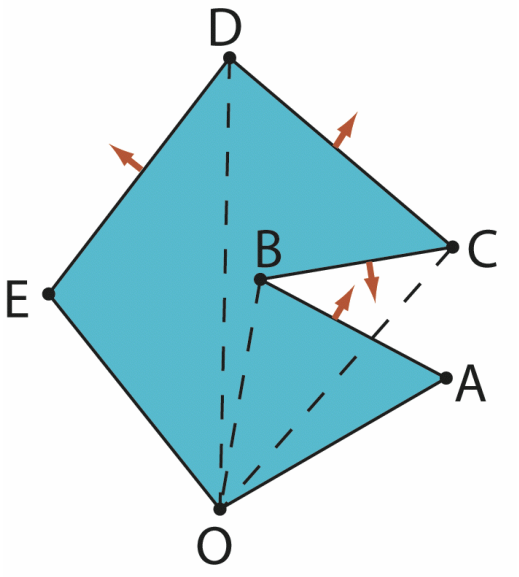

Figure 5: Polygon Decomposition with Segment Normal Vectors

As we saw from Figure 4, triangle OAB was additive and triangle OBC was subtractive. Notice here in Figure 5, that the normal vector of segment AB is pointing away from the origin. Mathematically, this could be shown by a positive dot product (or inner product), OA•Normal(AB) > 0. Conversely, the normal vector of segment BC is pointing towards the origin, OB•Normal(BC) < 0. Therefore, we can conclude that triangles with normal vectors pointing away from the origin are additive and triangles with normal vectors pointing towards the origin are subtractive. As we will see, this concept can be extended to three-dimensional polyhedra.

To further simplify the integration, Lien and Kajiya transform each tetrahedron into a unit tetrahedron before integration. This is accomplished by first considering each mass property equation to be of the form:

$$I = \int_{Q} f(x, y, z)\, dV \tag{22}$$

where Q is a polyhedron, and f is a polynomial function. A linear transformation can be defined:

$$\begin{aligned} x &= g_x(u, v, w) \\ y &= g_y(u, v, w) \\ z &= g_z(u, v, w) \end{aligned} \tag{23}$$

so that with a change of variables, Equation (22) becomes:

$$I = \iiint_{Q} f(g_x, g_y, g_z)\,|J|\, du dv dw \tag{24}$$

where the Jacobian is the determinant:

$$J = \begin{vmatrix} \frac{\partial g_x}{\partial u} & \frac{\partial g_x}{\partial v} & \frac{\partial g_x}{\partial w} \\ \frac{\partial g_y}{\partial u} & \frac{\partial g_y}{\partial v} & \frac{\partial g_y}{\partial w} \\ \frac{\partial g_z}{\partial u} & \frac{\partial g_z}{\partial v} & \frac{\partial g_z}{\partial w} \end{vmatrix} \tag{25}$$

The Jacobian relates a differential element in the original coordinate system to a differential element in the new coordinate system. It can also be thought of as a scaling factor between the two different Euclidean spaces.

Since the integrand in Equation (22) is polynomial in nature, it can be expressed in the general form:

$$f(x,y,z)=\sum_{i,j,k} x^i y^j z^k \tag{26}$$

where i, j, and k are integers. Looking at only one term from the summation and integrating yields:

$$I=\iiint_Q x^i y^j z^k \; dxdydz \tag{27}$$

Consider a simple case where Q is a tetrahedron with four vertices (v_0, v_1, v_2, v_3), where v_0 coincides with the origin. The coordinates of the vertices are:

$$\begin{aligned} v_0 &= (0,0,0) \\ v_1 &= (x_1,y_1,z_1) \\ v_2 &= (x_2,y_2,z_2) \\ v_3 &= (x_3,y_3,z_3) \end{aligned} \tag{28}$$

A linear transformation matrix, **T**, may be defined:

$$\mathbf{T}=\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{bmatrix} \tag{29}$$

which relates the old coordinate system (x,y,z) to a new coordinate system (u,v,w) in the following way:

$$\begin{Bmatrix} x \\ y \\ z \end{Bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} \tag{30}$$

such that the tetrahedron Q is transformed into a unit tetrahedron U, which has vertices:

$$\begin{aligned} v_0{}' &= (0,0,0) \\ v_1{}' &= (1,0,0) \\ v_2{}' &= (0,1,0) \\ v_3{}' &= (0,0,1) \end{aligned} \tag{31}$$

Performing this transformation on Equation (27) yields:

$$I = \|\mathbf{T}\| \iiint_U (x_1u + x_2v + x_3w)^i (y_1u + y_2v + y_3w)^j (z_1u + z_2v + z_3w)^k \, dudvdw \tag{32}$$

where the Jacobian ||**T**|| equals the absolute value of the determinant of the transformation matrix **T**. Next, Lien and Kajiya present a method for evaluating such equations:

$$\int_U u^i v^j w^k \, dV$$

$$= \int_0^1 \int_0^{1-w} \int_0^{1-w-v} u^i v^j w^k \, du dv dw$$

$$= \frac{1}{i+1} \int_0^1 \int_0^{1-w} (1-w-v)^{i+1} v^j w^k \, dv dw$$

$$= \frac{1}{i+1} \int_0^1 \int_0^{1-w} (1-w)^{i+1} \left(1 - \frac{v}{1-w}\right)^{i+1} v^j w^k \, dv dw$$

$$Let\ Y = \frac{v}{1-w},\ v = (1-w)Y,\ then$$

$$dv = (1-w)dY$$

$$I = \frac{1}{i+1} \int_0^1 \int_0^{1-w} (1-w)^{i+1} (1-Y)^{i+1} (1-w)^j Y^j w^k (1-w)\, dY dw$$

$$= \frac{1}{i+1} \int_0^1 \int_0^{1-w} (1-Y)^{i+1} Y^j (1-w)^{i+j+2} w^k \, dY dw$$

$$= \frac{\beta(j+1, i+2)}{i+1} \int_0^1 (1-w)^{i+j+2} w^k \, dw$$

$$= \frac{\beta(j+1, i+2)}{i+1} \beta(k+1, i+j+3)$$

$$= \frac{1}{i+1} \frac{j!\,(i+1)!}{(i+j+2)!} \frac{k!\,(i+j+2)!}{(i+j+k+3)!} \qquad (33)$$

$$= \frac{i!\,j!\,k!}{(i+j+k+3)!}$$

where the beta function has the following standard definition:

$$\beta(i+1, j+1) = \int_0^1 x^i (1-x)^j dx$$

$$\beta(i, j) = \frac{\Gamma(i)\Gamma(j)}{\Gamma(i+j)} = \frac{(i-1)!\,(j-1)!}{(i+j-1)!} \qquad (34)$$

$$where\ \ \Gamma(n+1) = \int_0^\infty e^{-x} x^n dx = n!$$

Now Equation (32) can be evaluated:

$$\begin{aligned} I &= \|\mathbf{T}\| \int_U (x_1 u + x_2 v + x_3 w)^i (y_1 u + y_2 v + y_3 w)^j (z_1 u + z_2 v + z_3 w)^k dV \\ &= \|\mathbf{T}\| \sum_m \sum_n \sum_p c(m,n,p) \int_U u^m v^n w^p dV \\ &= \|\mathbf{T}\| \sum_m \sum_n \sum_p c(m,n,p) \frac{m!\, n!\, p!}{(m+n+p+3)!} \end{aligned} \tag{35}$$

The function c(m,n,p) represents the coefficient of a term $u^m v^n w^p$ in the expansion of the integrand, which Lien and Kajiya describe as:

$$\begin{aligned} &(x_1 u + x_2 v + x_3 w)^i (y_1 u + y_2 v + y_3 w)^j (z_1 u + z_2 v + z_3 w)^k \\ &= \sum_{m+n+p=i+j+k} c(m,n,p) u^m v^n w^p \end{aligned} \tag{36}$$

Applying this technique, the integrals for mass, center of gravity, and moments of inertia of a tetrahedron can be evaluated. The formulas, as provided by Lien and Kajiya, except for the addition of density here, are shown in Equations (37) through (46). The mass is:

$$M = \rho \int_Q dV = \rho \|\mathbf{T}\| \frac{0!}{3!} = \rho \frac{\|\mathbf{T}\|}{6} \tag{37}$$

The three coordinates of the center of gravity are:

$$\bar{x} = \frac{\rho}{M}\int_Q x\, dV$$

$$= \frac{\rho}{M}\|T\|\iiint_U (x_1 u + x_2 v + x_3 w)\, du dv dw \tag{38}$$

$$= 1/4(x_1 + x_2 + x_3)$$

$$\bar{y} = 1/4(y_1 + y_2 + y_3) \tag{39}$$

$$\bar{z} = 1/4(z_1 + z_2 + z_3) \tag{40}$$

The moments of inertia are:

$$I_{xx} = \rho\int_Q (y^2 + z^2)\, dV = I_y + I_z \tag{41}$$

$$I_{yy} = \rho\int_Q (x^2 + z^2)\, dV = I_x + I_z \tag{42}$$

$$I_{zz} = \rho\int_Q (x^2 + y^2)\, dV = I_x + I_y \tag{43}$$

where,

$$I_x = \rho\int_Q x^2 dV = \frac{M}{10}\left({x_1}^2 + {x_2}^2 + {x_3}^2 + x_1x_2 + x_1x_3 + x_2x_3\right)$$

$$I_y = \rho\int_Q y^2 dV = \frac{M}{10}\left({y_1}^2 + {y_2}^2 + {y_3}^2 + y_1y_2 + y_1y_3 + y_2y_3\right)$$

$$I_z = \rho\int_Q z^2 dV = \frac{M}{10}\left({z_1}^2 + {z_2}^2 + {z_3}^2 + z_1z_2 + z_1z_3 + z_2z_3\right)$$

And the products of inertia are:

$$I_{xy} = \rho\int_Q xy\, dV = \frac{M}{20}\left[2(x_1y_1 + x_2y_2 + x_3y_3) + x_1y_2 + x_2y_1 + x_1y_3 + x_3y_1 + x_2y_3 + x_3y_2\right] \tag{44}$$

$$I_{yz} = \rho \int_Q yz \, dV = \frac{M}{20}\left[2\left(z_1 y_1 + z_2 y_2 + z_3 y_3\right) + z_1 y_2 + z_2 y_1 + z_1 y_3 + z_3 y_1 + z_2 y_3 + z_3 y_2\right] \tag{45}$$

$$I_{zx} = \rho \int_Q xy \, dV = \frac{M}{20}\left[2\left(x_1 z_1 + x_2 z_2 + x_3 z_3\right) + x_1 z_2 + x_2 z_1 + x_1 z_3 + x_3 z_1 + x_2 z_3 + x_3 z_2\right] \tag{46}$$

With the fundamental equations for this method having been derived, there is still the concern of a tetrahedron's positive or negative contribution to the overall mass properties of a polyhedron. As in the two-dimensional example, the concept of additive and subtractive triangle areas will be extended here to three-dimensional tetrahedra. Triangular facets on the surface of the model that create a projective tetrahedron with the origin and have normal vectors pointing away from the chosen origin will be defined as additive and those with normal vectors pointing towards the origin are subtractive. It is important to mention here how the normal vector of a triangle patch is determined: the points on the surface of the model must be consistently ordered so that the vertices of each triangle proceed counterclockwise when viewed from the outside of the model. Using the right-hand rule, this causes a normal vector to point outward from the model at each triangle facet. Determining this orientation is more conveniently calculated using the sign of the triple product of the vectors to each vertex of the triangle facet:

$$\mathbf{v}_1 \bullet \left(\mathbf{v}_2 \times \mathbf{v}_3\right) \tag{47}$$

Notice that the cross product, which was also present in the two-dimensional case, is the source of the sign applied to the tetrahedron's contribution. Whereas, in the two-dimensional case the magnitude of the cross product vector was related to the area of the triangle, the triple product results in a scalar value (due

to the dot product) that is either positive or negative and relates to the volume of the tetrahedron. A triple product is defined as:

$$\mathbf{v}_1 \bullet \left(\mathbf{v}_2 \times \mathbf{v}_3\right) = \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} \tag{48}$$

where x_1, y_1, and z_1, are the coordinates of $\mathbf{v}_1$, and so on. It can easily be shown (by multiplying out and matching terms), that the determinant of a 3 x 3 matrix, |M|, equals the determinant of its transpose, $|M^T|$. Therefore, this triple product is the same as the determinant of the transformation matrix **T**:

$$\mathbf{v}_1 \bullet \left(\mathbf{v}_2 \times \mathbf{v}_3\right) = \begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{vmatrix} = \left|\mathbf{T}\right| \tag{49}$$

Conveniently, by removing the absolute value operation on ||**T**|| in Equation (37) for the mass of the tetrahedra, the sign of its contributions can intrinsically be included in all calculations:

$$M = \rho \frac{\left|\mathbf{T}\right|}{6} \tag{50}$$

The following example will demonstrate this concept. Consider the simplest polyhedron, a tetrahedron, R in Figure 6, with vertices {v_1, v_2, v_3, v_4}.

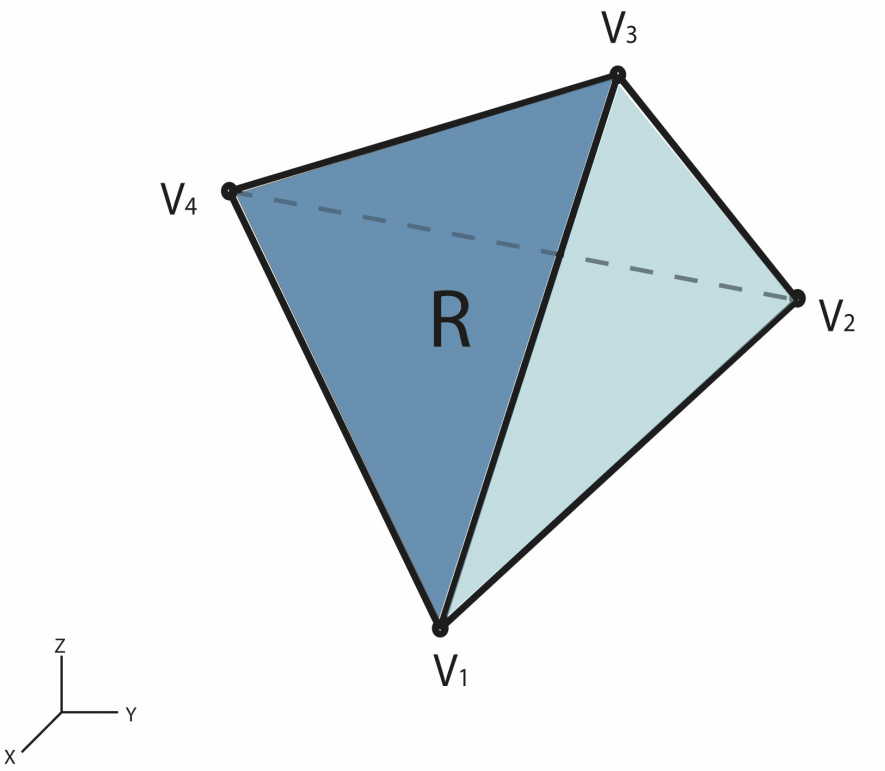

Figure 6: Example Polyhedron

This polyhedron can be decomposed into four separate tetrahedra projected from the origin (v_0), R_1, R_2, R_3, R_4, which is illustrated in Figure 7. The vertices of each new tetrahedron are shown in Table 1:

Table 1: Vertices for Decomposition Tetrahedra

Tetrahedron	Vertices
R_1	$\{v_0, v_1, v_2, v_3\}$
R_2	$\{v_0, v_1, v_3, v_4\}$
R_3	$\{v_0, v_2, v_4, v_3\}$
R_4	$\{v_0, v_1, v_4, v_2\}$

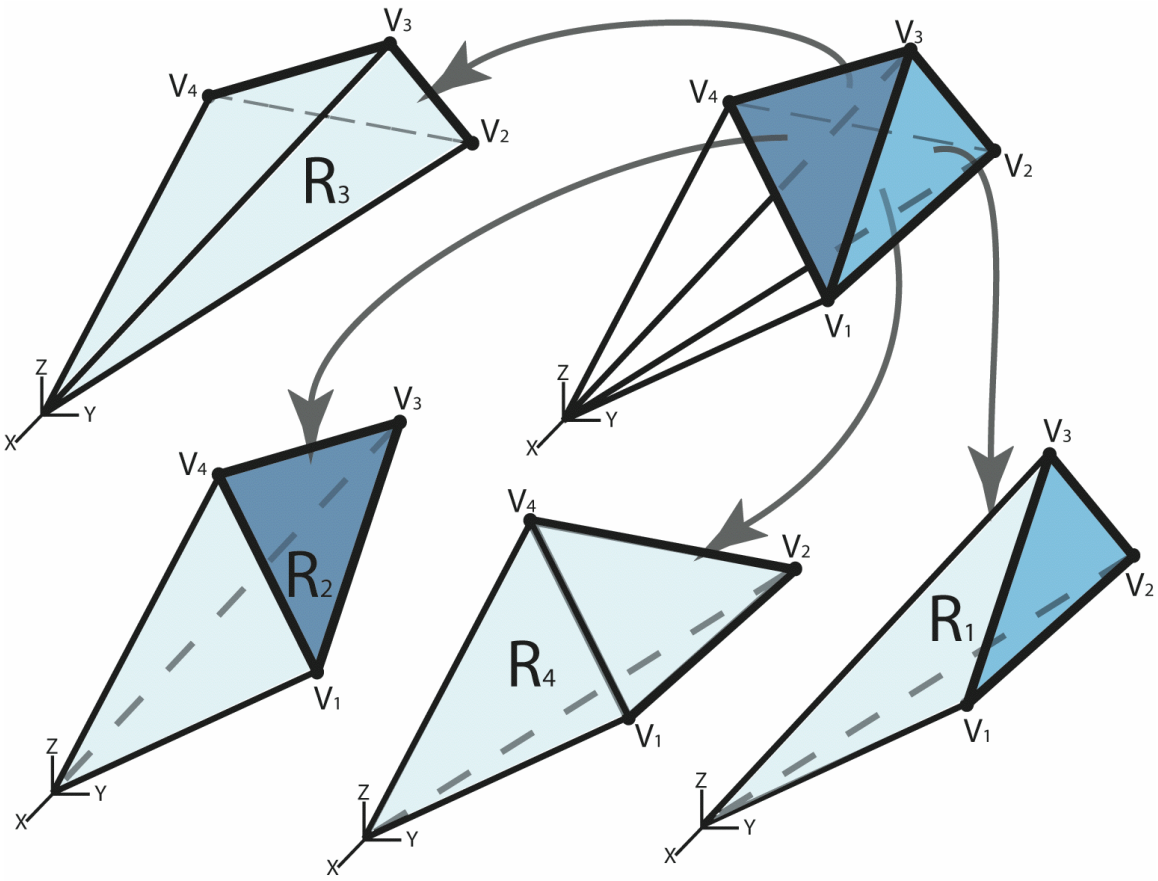

Figure 7: Polyhedron Decomposed into Projected Tetrahedra

In this example, the total mass of the polyhedron could be represented by the following equation:

$$M_{total} = M_1 + M_2 + M_3 - M_4 \tag{50}$$

A similar signed summation technique would be used for the moments and products of inertia also. The center of gravity summation is a bit more complicated. Each tetrahedron's calculated center of gravity coordinates must be weighted by that tetrahedron's signed mass, thus creating a moment about the origin. Once all of the pieces have been summed together, the coordinates are divided by the total mass. This is due to the fact that the moment generated by the total mass at a particular moment arm (distance from origin to polyhedron

CG) must equal the sum of the moments caused by each tetrahedron. This finally yields the center of gravity for the polyhedron [4].

A pseudo code version of this algorithm is shown below in Listing 1:

Listing 1: Pseudo Code for Homogeneous Solid Algorithm

```
Create variables for total Mass, CG, and MOI
Obtain list of Facets of polyhedron

For each facet in Facets list
       For each triangle in facet
          Create origin-projected tetrahedron from facet
          Calculate mass and cg of tetrahedron
          Add mass to total Mass
          Add cg*mass to total CG
       End loop
End loop

Divide total CG by total Mass

Move origin to CG

For each facet in Facets list
       For each triangle in facet
          Create origin-projected tetrahedron from facet
          Calculate moi of tetrahedron
          Add moi to total MOI
       End loop
End loop
```

There are several points to be noted about this algorithm:

1. The numbering of the vertices in the triangular facets on the model's surface must be ordered in a counterclockwise fashion when looking from the outside of the model. This ensures an outward pointing normal vector for each facet.
2. The accuracy of the method can be improved by shifting the origin to the center of gravity of the object. This reduces the presence of elongated tetrahedra when the model is far from the origin.
3. The location of the polyhedron affects the calculation of the moments of inertia. Moments of inertia are generally assumed to be about the center of gravity of the object. Hence, it is wise to translate the coordinate system to the center of gravity of the model before calculating the moments and

products of inertia. The parallel axis theorem may also be used to calculate moments of inertia about axes other than those passing through the CG.

4. The time complexity of this algorithm is proportional to the number of vertices in the model [23].
5. Mass properties of any arbitrary geometry can be calculated as long as the points on the boundary can be obtained, which is the case in a B-Rep scheme [41].

Thin Shell Algorithm

The algorithm discussed above is for homogenous solids that are represented by their boundary geometry. The need arose in the conceptual design of OAV's for the calculation of mass properties of thin shell components. However, the geometric representation scheme had not changed. If the surface of a solid was offset a certain distance inward, and then the rest of the original solid was hollowed out, a thin shell object would result. If this were possible, the algorithm for solid polyhedra above would evaluate the mass properties efficiently. However, this type of geometric operation was not available in the OAV software's infrastructure, so an alternate method was formulated. The following method is a new adaptation of the algorithm implemented by Lin, which was based on the relations set forth by Lien and Kajiya.

In this algorithm, each triangular facet is evaluated to compute mass properties as well as an offset triangular facet, which is scaled along the projection vectors from the origin. Figure 8 illustrates the two related facets.

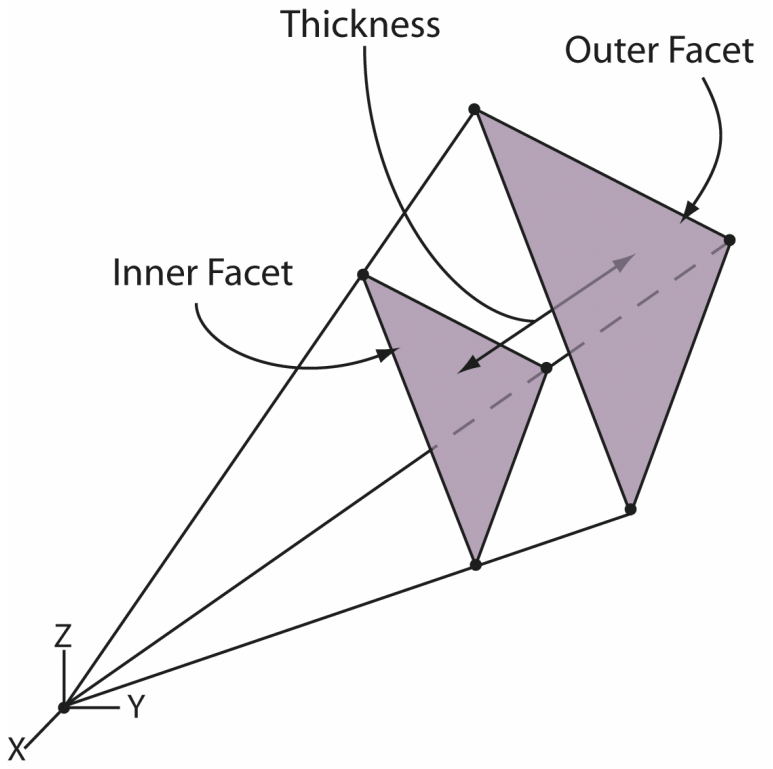

Figure 8: Thin Shell Triangular Facets

The original facet represents the outer surface of the thin shell, and the new offset facet represents the inner surface. Evaluating the solid mass properties of the outer surface facets and subtracting the solid mass properties of the inner facets will compute the thin shell's mass properties.

The most critical aspect of this method is the determination of the inner facet's vertices. The inner facet should be offset perpendicularly a set distance so that the new facet and original facet are parallel. To accomplish this, vector analysis is quite handy. A vector represents each vertex in the original facet. The vertices of the offset facet must also lie along these vectors so that the offset surface is closed (no gaps/overlaps), just like the original. The offset facet's vertices will be determined by scaling the original facet's vertex vectors. The scaling factor for each vertex vector must be determined such that the entire offset facet is parallel to the original and offset a specified distance.

Referring to Figure 9 below: **n** is the unit normal vector of the original facet; **t** is a vector representing the thickness and is collinear with **n**; **v** is the vector to the first vertex; **v'** is **v** projected on the normal vector **n**; **t'** is the vector along **v** that projects exactly onto **t**. The length of **t'** must be determined to correctly scale the vertices.

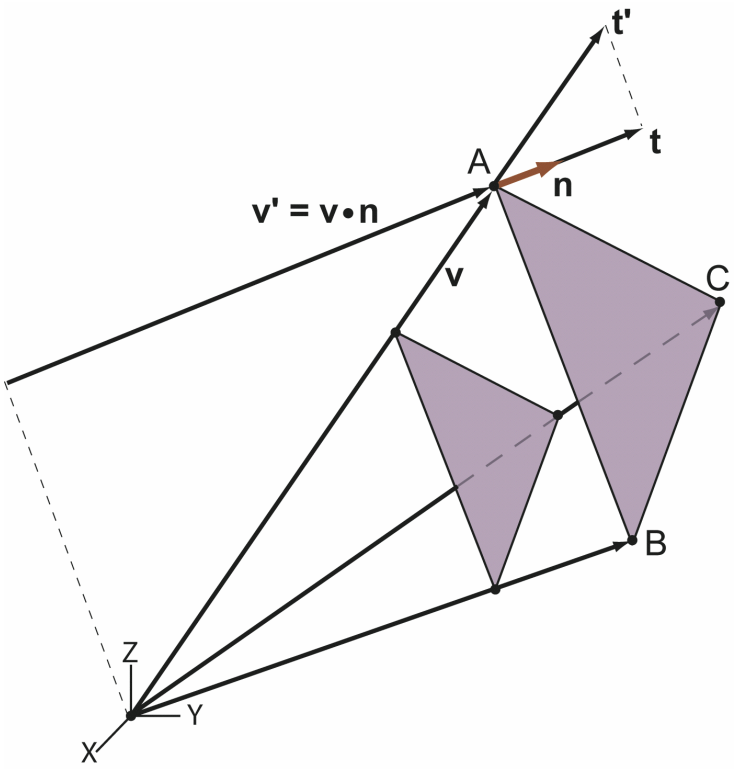

Figure 9: Vectors Related to Shell Facets

The length of **t'** is calculated by the properties of similar triangles in the following fashion:

$$\|\mathbf{t}'\| = \frac{\|\mathbf{t}\|\,\|\mathbf{v}\|}{\|\mathbf{v}'\|} = \frac{\|\mathbf{t}\|\,\|\mathbf{v}\|}{\mathbf{v} \bullet \mathbf{n}} \tag{51}$$

where the notation ||**x**|| denotes the norm or length of the vector **x**, the length of **t** is simply the thickness, and the dot product of **v•n** projects **v** onto **n**. The unit normal vector **n** can be found by taking the cross product of vectors in the facet's plane. Such vectors are most readily available by subtracting vertices from each other. In this case of Figure 9, (**B**-**A**) x (**C**-**A**) would create the normal vector with correct orientation. Care must be taken in choosing vertex order to ensure the right-hand rule convention is maintained. With the length of **t'** now known the new vector to the offset vertex may be found by scaling the original vector by this factor:

$$\frac{\|\mathbf{v}\| - \|\mathbf{t}'\|}{\|\mathbf{v}\|} \tag{52}$$

The length of **t'** can be positive or negative, depending on whether the offset facet is closer or farther from the origin, respectively. This process would be repeated for each vertex in the triangular facet. Therefore, for each triangular facet on the model's surface the mass properties would be calculated for the outer facet and would be contributed to the total and the inner facet's mass properties would be contributed with the opposite sign (if the outer facet causes positive volume, the inner causes negative volume, and vice versa).

There are some weaknesses to this technique. To understand the root of the problems, observe Figure 10 where a cube is being viewed from the side:

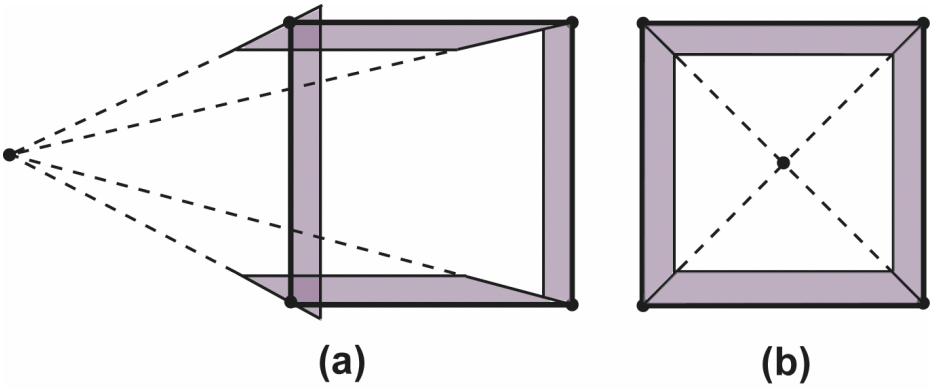

Figure 10: Distortions of Thin Shell Geometry

When the origin is far from the model (a), more specifically outside the model, there are inaccuracies wherever two facets' edges meet. These inaccuracies can be overcome by moving the origin to the center of gravity of the object, as seen in (b). This begets another problem: How can the origin be shifted to the CG if the CG itself cannot be calculated accurately due to the distortions? To overcome this problem an assumption was made: the shells are thin enough to consider the CG of the model to be the same as the centroid of the surface areas of the facets. To show the difference between the CG of a solid volume and the centroid of its surface, a worst-case scenario is shown in Figure 11.

Figure 11: Comparison of Solid CG vs. Surface Centroid

In above figure, the location of the geometric center of the solid is significantly different from that of just the surface. This implies some limiting assumptions inherent in this method. First, the thicknesses of the shell should be much less than the overall dimensions of the geometry (roughly 20 times smaller than largest dimension). Secondly, the thickness must be less than 1/2 the smallest

span in the geometry, otherwise overlapping portions of mass would be included in the calculations.

The area of each triangle in the bounding surface can be computed using Equation (21), and the centroid can be determined by averaging the three vertices. The same method for calculating the CG of multiple bodies that was used for the tetrahedra can be used for determining the centroid of the model's surface. Each triangle's centroid is weighted (multiplied) by its area and then summed with the rest. Finally, the summed coordinates of the centroid are divided by the total surface area, yielding the overall approximate CG of the model. The origin can be shifted to this point, and now the mass and moments of inertia can be accurately calculated for the thin shell geometry.

A pseudo code implementation of this thin shell algorithm is shown below in Listing 2:

Listing 2: Pseudo Code for Thin Shell Algorithm

```
Create variables for total Mass, Surface Area, CG, and MOI
Obtain list of Facets of polyhedron

For each facet in Facets list
       For each triangle in facet
          Calculate surface area and centroid from triangle
          Add surface area to total Surface Area
          Add centroid*surface area to total CG
       End loop
End loop

Divide total CG by total Surface Area

Move origin to CG

For each facet in Facets list
       For each triangle in facet
          Create origin-projected tetrahedron from facet
          Calculate mass and moi of tetrahedron
          Add mass to total Mass
          Add moi to total MOI

          Calculate offset inner facet

          Create origin-projected tetrahedron from inner facet
          Calculate mass and moi of tetrahedron
          Subtract mass to total Mass
```

```
            Subtract moi to total MOI
        End loop
End loop
```

Results and Accuracy

Several geometries were used to study the accuracy of the methods set forth in this chapter. Three simple geometric shapes were analyzed for the homogeneous solid and shell methods: a block, a cylinder, and a sphere. These geometries were chosen since their analytical mass properties are commonly known and easily calculated. The representation of these geometries was accomplished by creating faceted approximations of each surface. Graphical output of these geometries is shown below in Figure 12:

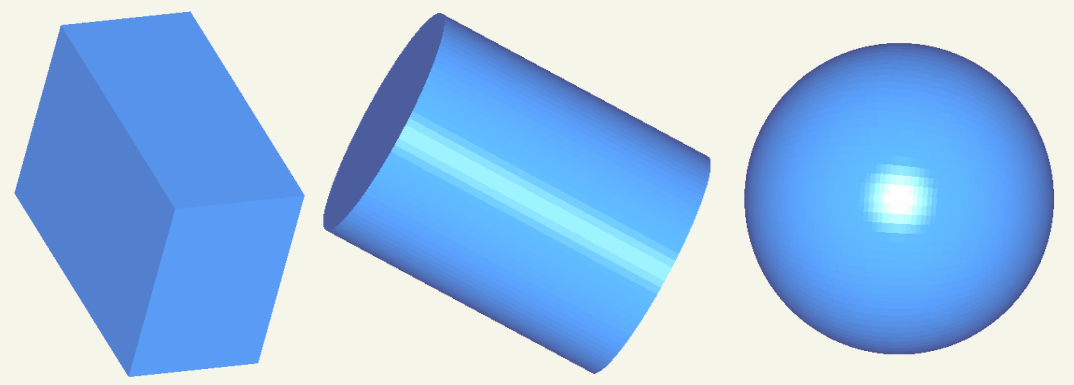

Figure 12: Block, Cylinder, and Sphere Facet Models

The facets of each model were created by determining points that were evenly spaced on the shape's surface and making triangles from the points while keeping a counterclockwise sense to ensure an outward normal vector. For instance, the sphere's points were generated by varying the spherical coordinate angles θ and ϕ at set increments while keeping the radius constant. The results of the analysis for the cylinder and sphere are shown below in Table 2:

Table 2: Results of Mass Properties Calculation

	Quantity	Analytical Value	Fine Mesh Value	% Error	Coarse Mesh Value	% Error
Solid Sphere Radius = 5" Density = 1	Mass [slug]	523.6	523.1	0.1	477.8	8.8
	CG X [in]	0	3.84e-10	0.0	3.69e-10	0.0
	CG Y [in]	0	-6.14e-16	0.0	-5.03e-17	0.0
	CG Z [in]	0	-2.29e-15	0.0	4.70e-16	0.0
	I_{xx} [slug-in^2]	5235.99	5228.67	0.14	4550.6	13
	I_{yy} [slug-in^2]	5235.99	5228.67	0.14	4550.6	13
	I_{zz} [slug-in^2]	5235.99	5226.95	0.2	4401.46	16
	I_{xy} [slug-in^2]	0	-4.66e-14	0.0	-6.33e-14	0.0
	I_{yz} [slug-in^2]	0	-1.64e-13	0.0	4.69e-14	0.0
	I_{zx} [slug-in^2]	0	1.82e-13	0.0	-3.73e-14	0.0
Shell Sphere Radius = 5" Density = 1 Thickness = .1"	Mass [slug]	30.79	30.78	0.04	29.4	4.5
	I_{xx} [slug-in^2]	503.08	520.57	0.1	455.4	9.5
	I_{yy} [slug-in^2]	503.08	520.57	0.1	455.4	9.5
	I_{zz} [slug-in^2]	503.08	520.57	0.1	444.78	11.6
Solid Cylinder Radius = 2" Height = 5" Density = 1	Mass [slug]	62.83	62.79	0.1	58.78	6.4
	I_{xx} [slug-in^2]	193.73	193.56	0.1	177.49	8.3
	I_{yy} [slug-in^2]	193.73	193.56	0.1	177.49	8.3
	I_{zz} [slug-in^2]	125.66	125.50	0.12	110.07	12.4
Shell Cylinder Radius = 2" Height = 5" Density = 1 Thickness = .1"	Mass [slug]	8.39	8.38	0.17	8.12	3.2
	I_{xx} [slug-in^2]	40.08	40.1	0.16	37.61	6.2
	I_{yy} [slug-in^2]	40.08	40.1	0.16	37.61	6.2
	I_{zz} [slug-in^2]	27.4	27.36	0.14	24.9	9.1

The values calculated for the cube were exact, and are consequently not shown in the above table. The CG and POI values are only shown for the first case, since they differ from the exact value of zero by magnitudes on the scale of floating point error, and are similar for the remaining cases. Two different mesh densities were analyzed to provide a good estimate of the error bounds. The precision of the calculations was perfect, replicating calculated values repeatedly for identical geometry. Some of the values calculated for the very coarse mesh are outside the stated allowable error percentages of 10%. However, this does not pose much threat to the validity of the algorithms since simply improving the

mesh quality can diminish the error to within acceptable bounds. The fine mesh comprised of 10,000 facets (100 x 100 grid of two parameters incremented). The coarse mesh had 100 facets (10 x 10 grid of two parameters incremented). To visually demonstrate the difference between these two triangulations, Figure 13 illustrates both sphere test geometries.

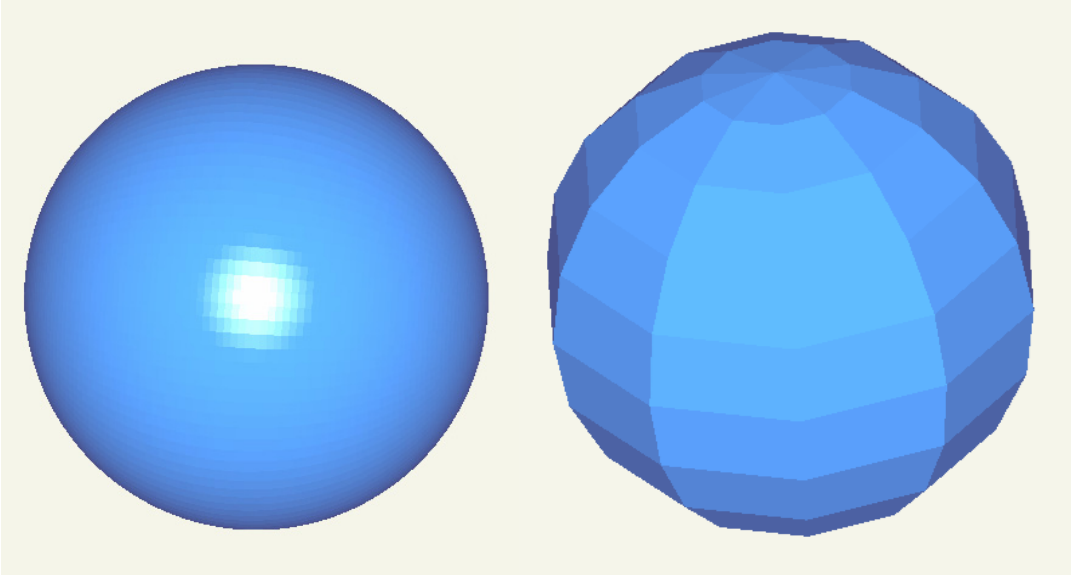

Figure 13: Comparison of Fine and Coarse Mesh Densities

One obvious conclusion that may be drawn from the above results is that the mesh density is the dominating parameter in the accuracy of the method. As the mesh density increases, the results approach their analytical values demonstrated by the fact that all of the fine mesh values had less than 1% error. Another important fact learned from these experiments is that planar surfaces are analyzed exactly using this method. This is implied by the fact that the block's mass properties were calculated exactly regardless of the mesh density. When a model that only has planar faces is triangulated, the resulting model has equivalent geometry; when a model with curved surfaces is triangulated, the resulting model is an approximation of that original geometry. This approximation of curved geometry is the main source of error in this method. The sphere, whose original surface was more curved than the block or cylinder, had the largest relative errors.

Quantifying or predicting this error is tricky business. There is definitely a correlation between surface curvature and accuracy. There is also a correlation between mesh density and accuracy. To show how the solid sphere calculations vary as the mesh density is refined a graph of average % error vs. the number of facets in the sphere approximation is shown below in Figure 14.

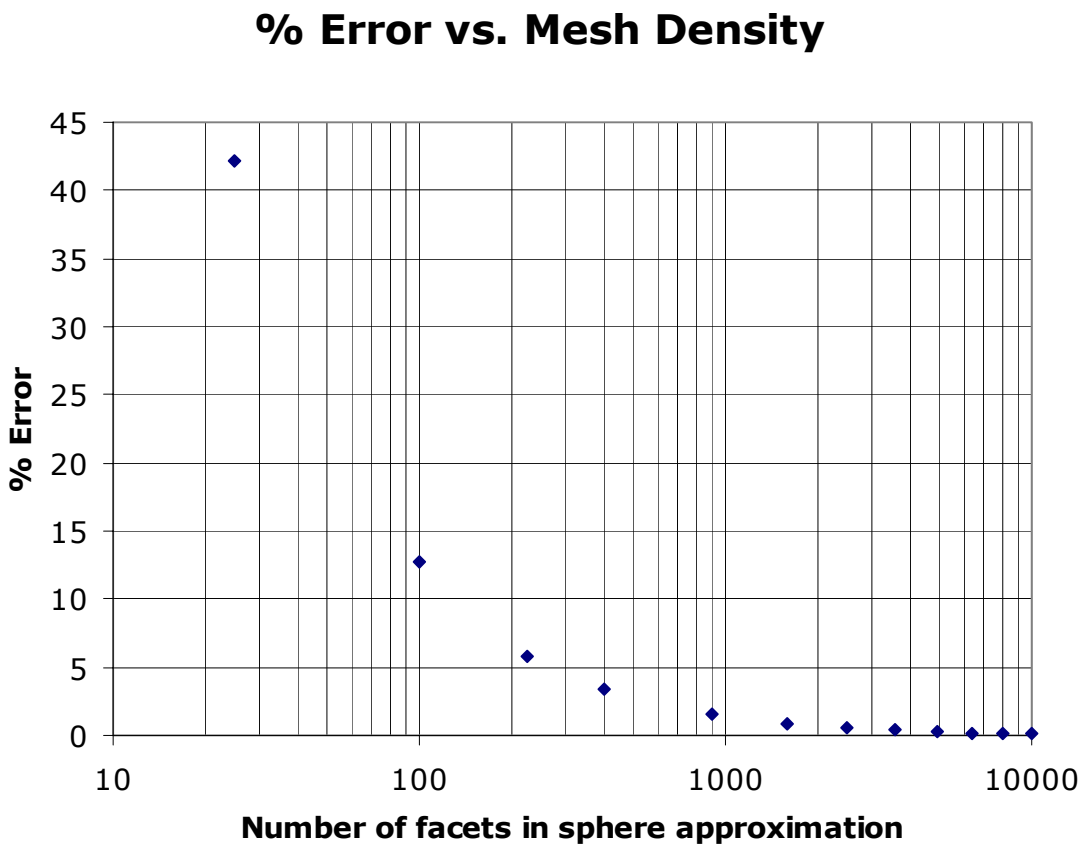

Figure 14: Percent Error in Sphere Mass Properties vs. Mesh Density

The above figure demonstrates that for the sphere geometry, meshes of greater than 200 facets provide adequate accuracy, while increasing the mesh fineness above 1000 facets shows diminishing returns in accuracy while increasing computational time. Unfortunately, the “knee” in the curve will be different for geometries of varying complexity, and an across-the-board mesh density cannot be prescribed.

Chapter 4: Fuel Analysis

Problem Statement and Simplifying Assumptions

The mass properties of a fuel tank continuously change during an aircraft's mission due to the constant burn of fuel. The fact that the fuel changes shape when the aircraft pitches or rolls during maneuvers further complicates mass properties analysis. The stability of the aircraft rests on the ability to predict or control the mass properties of the entire vehicle. Therefore, understanding the nature of fuel mass properties is an important study, especially for vehicles where up to 50% or more of the weight of the vehicle is due to fuel. Some OAV configurations require these kinds of fuel percentages. Therefore, this kind of analysis is very helpful during the conceptual design phase of OAV's to ensure a good and stable design.

A complete analysis of the mass properties of the fuel would require an extraordinary amount of computational power due to the dynamic behavior of fluids in motion. Consequently, all of the methods I have observed in my research make certain simplifying assumptions. Usually the assumptions are tailored toward acquiring a quantity of interest. For instance, if finding the center of gravity of the fuel in a tank is the objective, the assumption might be that the CG of the fuel is the same as that of a solid having the same shape. If one were trying to determine the resonant natural frequencies of the sloshing of the fuel, it could be assumed that the motion of the tank (more generally, the forcing function) is of a sinusoidal form. In my case the conditions are such that the CG is the quantity of greatest interest, but at the same time the MOI should not be oversimplified to the point of considering the fuel as if it were solid. Hence, the simplifying assumptions made in the following analysis are these:

1. The center of gravity of the fuel in the tank is the same as that of a solid having the same shape.
2. The moments and products of inertia will be expresses as a fraction of the MOI and POI of a solid having the same shape.
3. Sloshing of the fuel will be ignored.
4. The free surface of the fuel may be approximated as being planar.

Methodology and Algorithm

The general methodology adopted in this thesis is similar to that set forth by Daguia and Schumacher [43]. They detail the fuel analysis techniques used during the design of the X-47A Pegasus UAV. The main concept of their fuel tank analysis is to calculate the mass properties of the fuel at incremental fluid depth levels. They used Unigraphics, a solid modeling CAD system, to slice the tank at equally spaced parallel levels, and then evaluated the mass properties. They entered their data into a spreadsheet and plotted their results against actual data to test the method's accuracy. After adjusting for ullage (air space left for fuel expansion) and internal structure (pumps, ribs, piping), their prediction of CG movement was within 1% of actual values [43]. Daguia and Schumacher go on to explain that while the example they set forth was for a 0° pitch attitude of the vehicle, the method can be applied for a range of pitch angles to cover the full range of orientation that the aircraft might experience.

Since a commercial CAD system with the ability to easily perform the above calculations is not part of the AVID OAV conceptual design software, a new algorithm was created to accomplish the same goals. This new algorithm operates on polyhedra (polygonal geometry), which represent an approximation of more complex geometry. The objectives of the algorithm are to be able to evaluate the mass properties of fuel in a tank at different levels of volume. The tank may also have arbitrary orientation as long as the rotation angles about the x, y, and z-axes are specified.

The first building block of such an algorithm is a polygon partitioning or slicing algorithm. If a plane that is perpendicular to the gravity vector represents the surface of the liquid fuel, then the fuel in the tank may be represented as the portion of the tank geometry below that plane. In a commercial CAD system, this would be accomplished by a simple Boolean set operation, subtracting the half-space represented by the plane away from the tank geometry. An example of this operation, performed in IronCAD, is shown in Figure 15.

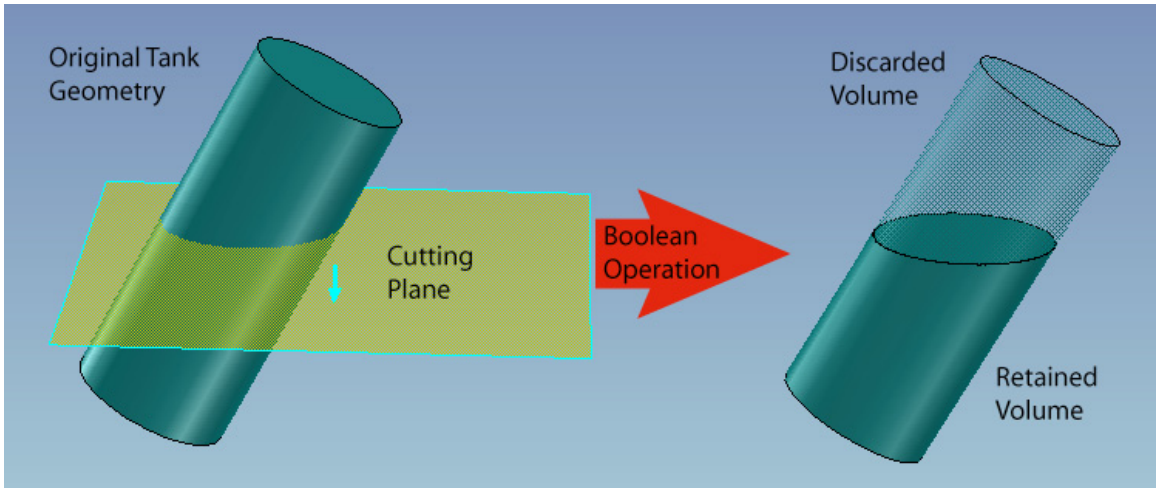

Figure 15: CAD Boolean Operation to Represent Remaining Fuel

A similar operation must be performed on the polygonal geometry representation that is used for mass properties calculation in this thesis. A similar process is discussed by Mäntyla in Chapter 14 of his book [12]. Mäntyla presents a more general process for a fully defined topological model having vertices, edges, faces and loops. The method developed here operates on polygons, and produces a polygonal result without explicit topological relationships. Segal and Sequin have also considered partitioning solid polyherdra [44], but the topological and geometric framework in which they were working differs from that of AVID OAV and renders their method inapplicable. Clipping of solid geometry for visualization of internal structure and interference is discussed by Rossignac, Megahed, and Schneider in [45]. However, the result of their clipping algorithm

was a two-dimensional pixel mask for graphical display, and also does not apply to the problem at hand.

In the method presented in this paper, individual polygons on the surface of a polyhedron will need to be sliced by the gravity plane, keeping the part of the polyhedron below the plane and discarding the other. In the algorithm, the gravity plane will be represented by a point on the plane with an accompanying normal vector (perpendicular to the plane). This vector will also indicate the positive side of the plane. An illustration of a polygon intersecting the gravity plane (cutting plane) is shown in Figure 16 below.

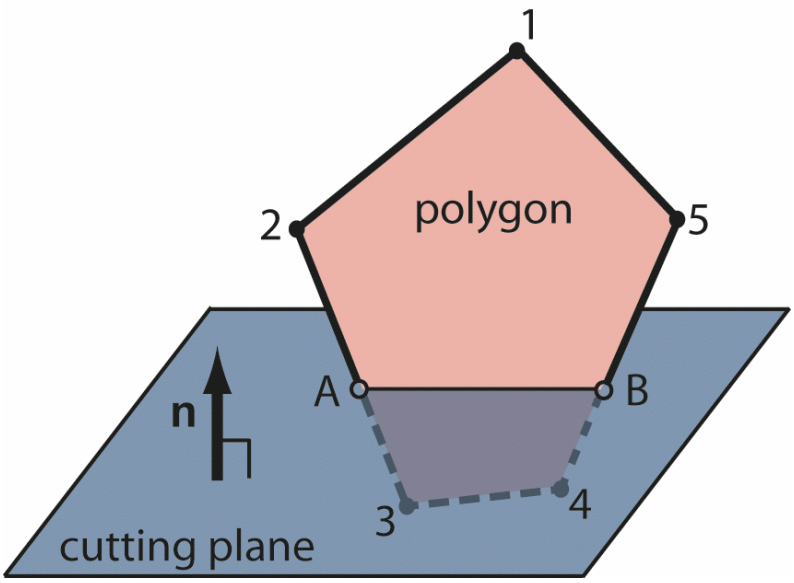

Figure 16: Polygon Intersection with Cutting Plane

In Figure 16 observe that the polygon with vertices {1,2,3,4,5} is divided into two polygons: {1,2,A,B,5} which will be kept and {A,3,4,B} which will be discarded. There are several things to note here; first of all, vertices on the positive side of the plane will be included in the polygon to be kept, and conversely points on the other side will be discarded. Secondly, vertices will need to be inserted where the polygon intersects the cutting plane and added to the new polygon while taking care to keep a counterclockwise orientation (unit normal outward convention). A simple rule can prescribe when and where to insert points: if a

segment along the border of the polygon has vertices of opposite sign (one on the positive side of the plane, the other on the negative side), then a new vertex must be inserted. The location of this vertex is the point where the segment intersects the cutting plane. Thirdly, the newly inserted points A and B would need to be included in another facet (polygon), one that caps the sliced tank polyhedron. Cap facets reside in the cutting plane and inside the original polygonal geometry. More will be said about cap facets later.

A clear definition for defining whether a point is on the positive or negative side of a plane, or even on the surface of the plane, must be set forth. A simple vector operation can solve this problem. Figure 17 shows a side view of the cutting plane and a typical vertex to be classified.

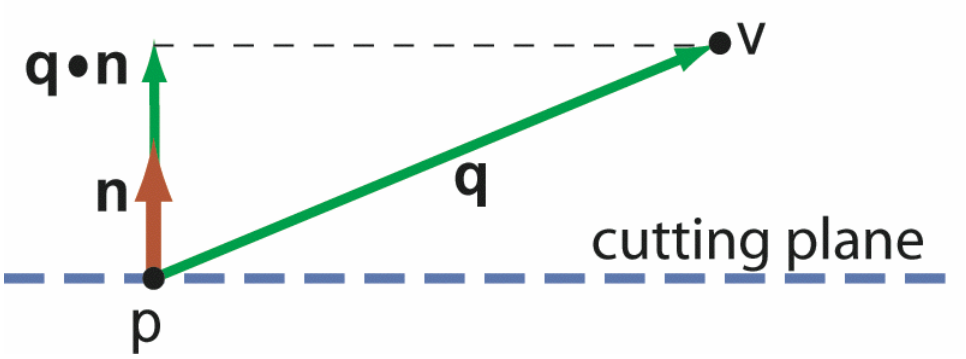

Figure 17: Vertex Classification by Vector Dot Product

In Figure 17, **q** is the vector from **p**, the point on the plane, to **v**, the vertex. If the vector **q** is projected onto the unit normal vector **n**, the sign of the magnitude will designate on which side of the plane the point resides. If the magnitude is sufficiently close to zero (within some predefined tolerance), the point may be considered to be in the plane.

Now that vertices can be classified as to their location relative to the cutting plane, a method for locating intersection points must be developed. Again, vector analysis provides an efficient way to determine the location of the intersection point. A typical situation is portrayed in Figure 18 below.

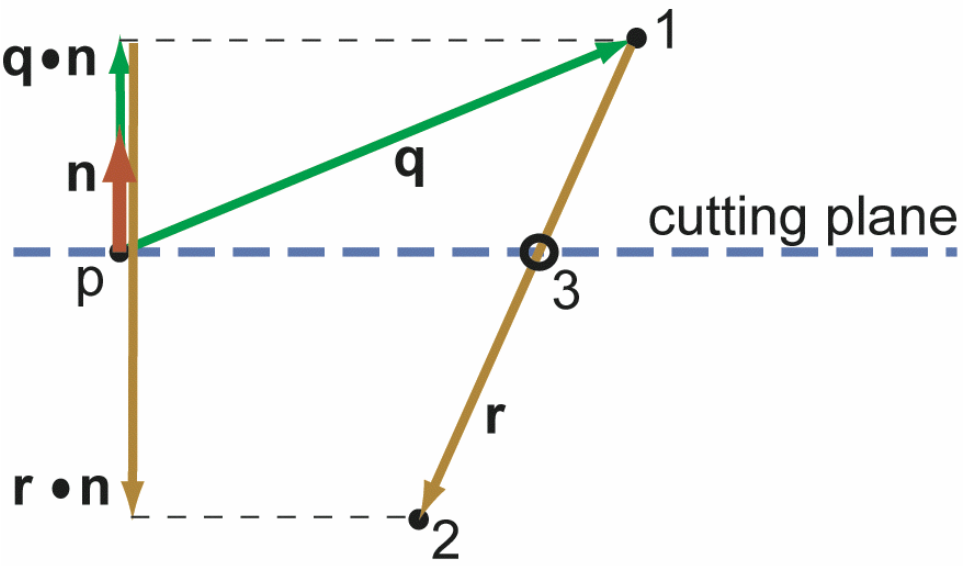

Figure 18: Inserting an Intersection Point on Cutting Plane

In Figure 18, vertex 1 is above the cutting plane and vertex 2 is below. An intersection point must be inserted at vertex 3. **q** is a vector from the plane point to vertex 1, and **r** is a vector from vertex 1 to vertex 2. A dot product is used to project **q** and **r** onto the unit normal vector **n**. Vertex 3 lies along vector **r**, so a normalized **r** vector could be scaled to determine vertex 3. The scaling factor would be equal to the ratio of lengths between vertices 1 to 3 and between vertices 1 to 2. The length of the vector from vertex 1 to 3 is unknown, but the ratio is also equal to the projection of vertex 1 (vector **q**) onto **n** to the projection of **r** onto **n**. If each vertex is considered as a vector from the origin (vertex 1 will be defined by $\mathbf{p}_1$, and so on), then the equation for the vector, $\mathbf{p}_3$, from the origin to the intersection point at vertex 3 is:

$$\mathbf{p}_3 = \mathbf{p}_1 + \left|\frac{\mathbf{q} \bullet \mathbf{n}}{\mathbf{r} \bullet \mathbf{n}}\right| \mathbf{r} \tag{53}$$

Fortunately the term q•n will have already been calculated during the vertex classification process, thus reducing the impact of this calculation on computational time.

The need for capping the sliced polyhedron affects the way the inserted points are processed. To illustrate this concept, Figure 19 is provided below, in which the polyhedron from Figure 6 is being cut and capped.

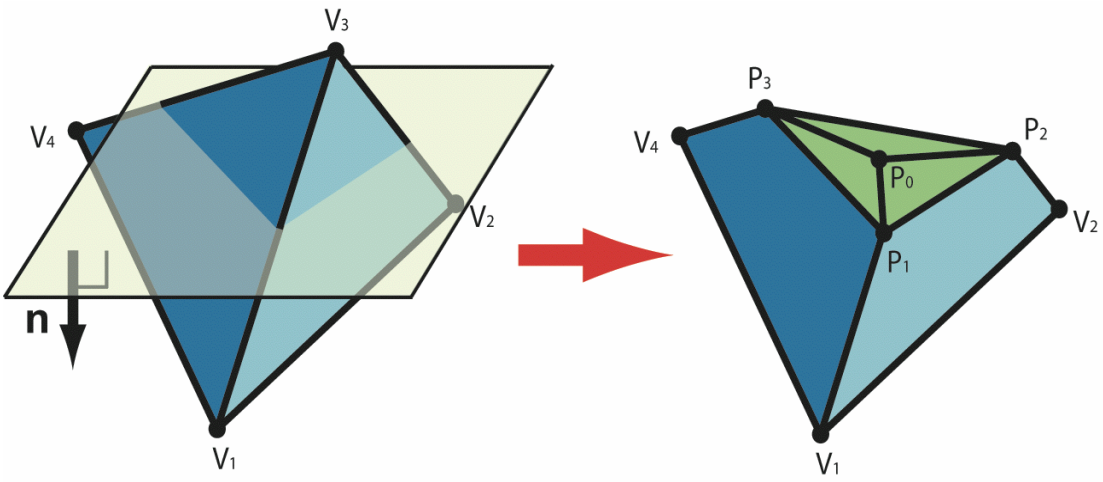

Figure 19: Capping of Sliced Geometry

Note that the cutting plane has a normal vector pointing down in this figure so that the points on the lower half of the model are actually considered "positive" and are kept. This allows for better observation of the cap facets. The ordered sets of points for each new cap facet are shown in Table 3, where p_0 represents the point used to define the cutting plane.

Table 3: Vertices of Cap Facets

Cap Facet	Vertices
F_1	$\{p_0, p_1, p_2\}$
F_2	$\{p_0, p_2, p_3\}$
F_3	$\{p_0, p_3, p_1\}$

Figure 19 shows the general concept of capping the sliced polygonal geometry, but a more specific method is required to systematically create these facets for the present algorithm. One important rule to understand is that cap facets are created from two intersection points on a sliced facet and one point originally used to define the cutting plane. Each of the two intersection points can be considered either an “exit” or an “entry”. The path of the polygon being sliced travels from positive space to negative space and back again. The points where these transitions occur will be call exit and entry points, respectively. The idea of using such “exit” and “entry” points was adapted from two-dimensional polygon clipping research by Greiner and Hormann [46]. An exit point is an intersection point whose previous point in the facet boundary was on the positive side of the cutting plane. An entry point is an intersection point whose previous point was on the negative side of the cutting plane. A cap facet must have only one exit point and one entry point, along with a starting point in the plane. In this way, every facet of the original polyhedron that gets sliced has a corresponding cap facet to ensure no breaks or openings in the resulting polyhedron. To ensure outward normal orientation of the cap facet, the general order in its vertex list will be {plane point, entry point, exit point}. The exit point used here will have preceded the entry point during the traversal of the polygon boundary vertex list.

The business of classifying exit and entry points and creating subsequent cap facets becomes increasingly complex when the concept of polygon vertices lying in the cutting plane is introduced. An example of some special cases is presented in Figure 20.

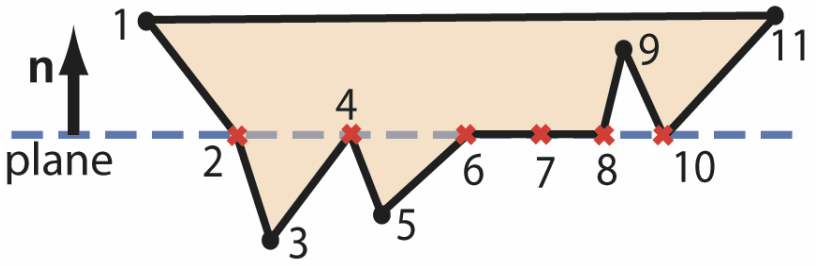

Figure 20: Exit and Entry Points Considering In-Plane Vertices

In Figure 20, all of the red crosshairs signify points on the boundary of the polygon that lie in the cutting plane. Point 2 is an exit point. Point 8 is an entry point. Points 4, 6, and 7 would all be discarded (along with 3 and 5, of course). Point 10 would be included in the newly cut polygon. From this visual example we can derive some rules for handling in-plane vertices. They are:

1. If the sign of an in-plane point's previous and next point are positive, keep the point (Point 10).
2. If the sign of the previous point is positive, and the sign of the next point is not positive, keep the point; it is an exit point (Point 2).
3. If the sign of the previous point is not positive, and the sign of the next point is positive, keep the point; it is an entry point (Point 8).
4. If the sign of the previous and next points are not positive, then discard the point (Points 4, 6, and 7).

It is important to note that points in the plane are not considered to have a positive or negative sign, and thus are included in the set "not positive" in the logical rules above.

With all of the above building blocks in place, an algorithm for calculating the mass properties of fuel geometry can be developed. Pseudo code for such an algorithm is shown below in Listing 1.

Listing 3: Pseudo Code for Polyhedron Slicing Algorithm

```
Obtain list of OldFacets of polyhedron
Create empty list for NewFacets
For each facet in OldFacets

       Obtain list of OldPoints from facet
       Create empty list for NewPoints
       For each point in OldPoints

          If current point is positive
             Add point to NewPoints
             If next point is negative
                Insert intersection point, set as exit
                If entry has also been set
```

```
                Create cap facet and add to NewFacets
             End if
          End if

       Else if negative
          If next point is positive
             Insert intersection point, set as entry
          End if
          Discard point

       Else if in plane
          If next is positive and previous is positive
             Add point to NewPoints
          Else if next is positive and previous is not
             Add point to NewPoints
             Set as entry
          Else if previous is positive and next is not
             Add point to NewPoints
             Set as exit
          Else
             Discard point
          End if
       End if

    End point loop
    If NewPoints is not empty, add it to NewFacets
    If exit and entry exist
       Create cap facet and add to NewFacets
    End if

End facet loop
NewFacets now contains the polygons of the cut polyhedron
```

Source code, which implements this pseudo code, is provided in Appendix A. The resulting sliced polyhedron represents the geometry of fuel at rest in a tank, with its free surface being perpendicular to the gravity vector. Mass properties of this geometry can easily be evaluated as a homogenous solid using the methods set forth in Chapter 3. This approach will render accurate values for weight and center of gravity, but moments of inertia will differ from the solid mass properties calculation.

Fluid Moment of Inertia

Daguia and Schumacher also included the moments of inertia in their fuel analysis, but assumed that the fuel will exhibit the same behavior as a solid. This is an invalid yet common assumption according to Boynton [31]. Boynton investigates experimentally the moments of inertia of fluids in tanks. He explores

the effects that viscosity, tank aspect ratio, angular acceleration, and baffles play in determining the moments of inertia of fluids in containers. He expresses his results in the form “percent of solid equivalent”, meaning that the MOI of the fuel is some percentage of the value that would be obtained if the liquid were a solid. Some of the conclusions he has drawn from his experiments are:

1. The roll MOI is very small for large cylindrical tanks. Decreasing the diameter increases the MOI.
2. The yaw and pitch MOI of cylindrical tanks and the MOI of rectangular tanks is strongly related to the aspect ratio of the tank.
3. MOI increases as viscosity increases. The data indicated that MOI is proportional to the square root of viscosity, but this fact is not firmly established.
4. The speed of oscillation or angular acceleration affects the MOI of small tanks.

Boynton also presents equations found in a document entitled “Analytical Expressions for the Moment of Inertia of a Filled Cylindrical Tank about Various Roll and Yaw-Pitch Axes” by Phillips and Bauer [47]. Boynton claims the data he collected agreed (he does not quantify how well) with the equations:

Roll Moment of Inertia:

$$\frac{I_{effective}}{I_{solid}} = 2\sqrt{2}\left(\frac{1-\frac{\sqrt{2\nu}}{4r\sqrt{\omega}}}{r\sqrt{\frac{\omega}{\nu}}+\frac{5\sqrt{\nu}}{64r\sqrt{\omega}}-\frac{3\sqrt{2}}{8}}\right) \qquad (54)$$

$$\frac{I_{effective}}{I_{solid}} = \frac{2\sqrt{2}}{r}\sqrt{\nu/\omega} \quad when \ \frac{\sqrt{\nu/\omega}}{r} < 1$$

Pitch and Yaw Moment of Inertia:

$$\frac{I_{effective}}{I_{solid}} = 1 - \frac{1/2 + 4\sum_{n=1}^{\infty}\left[\frac{\left(1 - \frac{2\tanh(\varepsilon_n a)}{\varepsilon_n a}\right)}{\varepsilon_n{}^2\left(\varepsilon_n{}^2 - 1\right)}\right]}{a^2/12 + 1/4} \tag{55}$$

where r is the radius of the cylindrical tank, a is the aspect ratio (height to diameter ratio), ν is the kinematic viscosity, ω is the forcing frequency, and ε_n is the n^{th} root of the derivative of the first order Bessel function. Boynton notes that the roll MOI is inversely proportional to radius and square root of forcing frequency, and directly proportional to the square root of the kinematic viscosity. For the pitch and yaw MOI, he notes that the aspect ratio critically affects the MOI. Regardless of the agreement with his experimental data that he observed, he was unable to specify under what circumstances the equations were valid, due to unavailability of the company's internal document where the derivation was shown. Unfortunately, Boynton and these equations only address *full* fuel tanks.

Some older research performed for NACA by Widmayer and Reese [48] suggests that the fullness of the tank does not significantly alter the moments of inertia. Widmayer and Reese studied the effects on moment of inertia effects of small pitching oscillation (a few degrees) applied to full and partially full tanks. They make reference, as many others in this field do, to Lamb [49] for an analytical prediction of fluid moment of inertia in ellipsoid containers. The relationship is this:

$$\frac{I_{effective}}{I_{solid}} = \left(\frac{a^2 - 1}{a^2 + 1}\right)^2 \tag{56}$$

where *a* is the aspect ratio of the cross-section perpendicular to rotation axis (if less than one, use the reciprocal1/a). Widmayer and Reese also mention further work in this area by Miles [50], extending to arbitrary cross-section tanks. It shows that there is a general connection between the torsional modulus of the tank cross-section and the fluid inertia of the full tank:

$$\frac{I_{effective}}{I_{solid}} = 1 - \frac{\rho J_p}{I_{solid}} \tag{57}$$

where ρ is the mass density of the fuel and J_p is the torsional modulus of the solid section. Widmayer and Reese show that their experimental data agree with these equations (not specifying to what degree), and therefore concludes that the equations are a good basis for estimating the moment of inertia for new tank configurations. Another report by NASA [51] gives an equation for percent solid equivalent moments of inertia of fluids in rectangular tanks. The equation comes from an article by Graham & Rodriguez [52]:

$$\frac{I_{effective}}{I_{solid}} = 1 - \frac{4a^2}{1+a^2} + 2.510\left(\tanh\frac{\pi}{2a} + 0.0045\right)\left(\frac{a^3}{1+a^2}\right) \tag{58}$$

Widmayer and Reese display results for testing tanks with varying fill levels showing that the values do vary, but not very much as can be seen below in Figure 21.

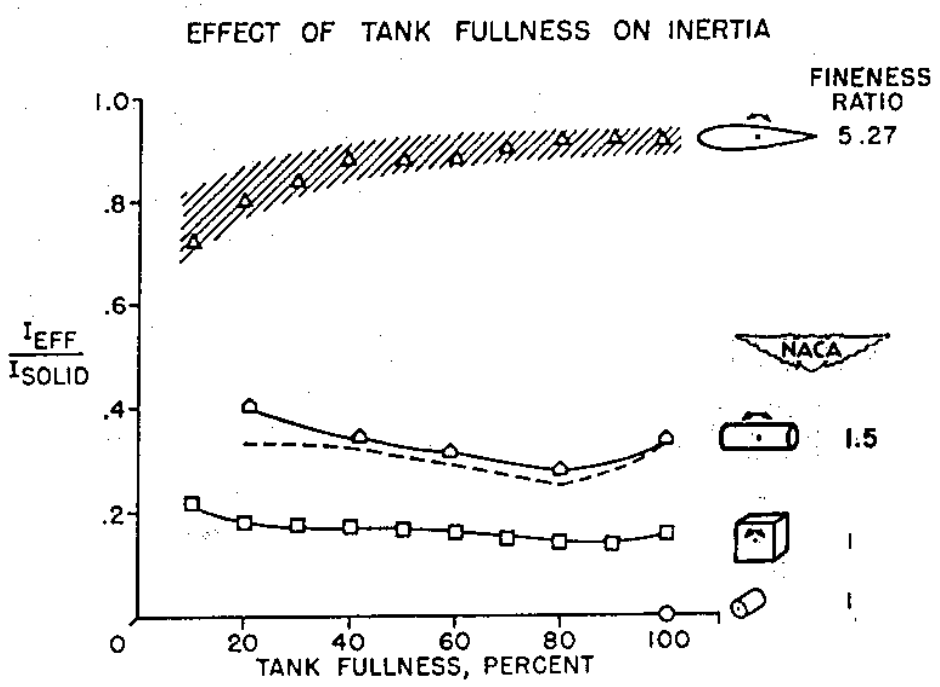

Figure 21: Widmayer and Reese's Data on Effect of Tank Fullness [48]

From this data, they conclude that using the moment of inertia value for the full tank is a good approximation for tanks 10% to 100% full. Since an empty tank's fuel MOI's are equal to zero, a linear interpolation between zero and full MOI will be used as an approximation for tanks in the region of 0% to 10% fullness in the methodology proposed in this paper.

Since one of the objectives of this thesis is to devise methods for fuel analysis during *conceptual design*, which in its nature is based on estimations, the above assumptions made by Widmayer and Reese will be adopted here. Equation 56, the simplest equation for percent solid equivalent MOI, while still being in agreement with experimental data (roughly within 10% of actual value), will be used for moment of inertia calculations in this thesis. The main reason the simplest equation was chosen was to maximize the speed of the algorithms developed for calculating the mass properties of the fuel.

In summary, a fuel tank can be analyzed at various fill levels and angles of orientation with the algorithm presented in this chapter. Such analysis could be used to develop an interpolation map for volume and center of gravity, given specific values of fuel volume and orientation angle. Moments of inertia of fuel in tanks 10% to 100% full can be calculated by applying a percent solid-equivalent (according to the tank's aspect ratio) to the MOI of a solid 100% full tank MOI. For tanks 0% to 10% full, the MOI can be interpolated between the empty tank and full tank values. Therefore, the MOI values do not need to be included in the interpolation map. The need for an interpolation map is due to the need for speed when accessing the CG value during pitch trim stability analysis, which will be discussed in the next chapter. Otherwise, the mass properties of a partially full tank could be calculated directly when they are needed.

Results and Accuracy

The method presented was used to predict the CG and MOI of fuel in a cylindrical tank and the results were compared to analytical solutions. An illustration of the test case is shown below in Figure 22.

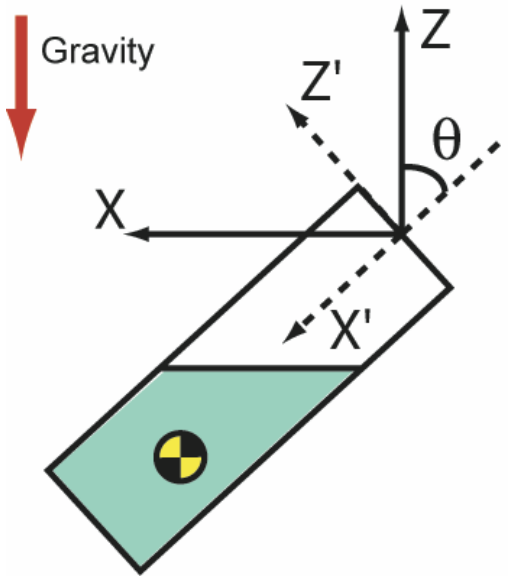

Figure 22: Fuel Mass Properties Calculation Test Case

Figure 22 shows a side view of a cylindrical tank, rotated by angle θ, with two different coordinate systems. The axes depicted with solid lines represent the global coordinate system, where gravity is in the negative z direction. The dotted axes represent the fuel tank's local coordinate system. The mass properties will be calculated in this local coordinate system. The cylinder is 10" long with a radius of 1". The density of the contained fuel is 0.0001 slug/in^3, and the tank is assumed to have 0% ullage and 0% internal structures. This yields a maximum tank capacity of 0.9933 lbs. of fuel. The interpolation map generated for this test case contained a sampling of the mass properties at nine different angles of rotation about the y-axis (θ in Figure 22) and nine different fuel levels. The fineness of the mesh triangulation used had 20 points per cross-section, placing this mesh somewhere between those used for the "fine mesh" and "coarse mesh" in the Results and Accuracy section of Chapter 4. Mass property results were obtained for several combinations of fuel weight and tank rotation. Results of the CG location are shown below in Table 4. They are compared to analytical values developed by integrating the governing equations. Successful analytical integration is only reasonable for simple boundaries, and that is why a cylindrical tank was chosen. The derivation of the analytical solution is shown in Appendix B. The relations developed apply only to the case where the entire free surface of the liquid is not touching either of the tank's end caps. This is the reason for several entries of N/A in Table 4.

Table 4: Comparison of Calculated and Analytical Values of CG

		CG X Coordinate			CG Y Coordinate			CG Z Coordinate		
Fuel Weight [lb]	**Y Rotation [deg]**	Results [in]	Analytical [in]	Error (% of X length)	Results [in]	Analytical [in]	Error (% of Y length)	Results [in]	Analytical [in]	Error (% of Z length)
0.9933	0	5.000	5.000	0.000	-4.15E-09	0.000	0.00	0.0000	0.0000	0.00
0.7500	30	6.217	6.287	0.702	-3.76E-09	0.000	0.00	-0.0199	-0.0195	0.02
0.5000	30	7.472	7.520	0.481	-3.56E-09	0.000	0.00	-0.0298	-0.0292	0.03
0.2500	30	8.718	8.747	0.291	-2.92E-09	0.000	0.00	-0.0626	-0.0584	0.21
0.0500	30	6.381	N/A		4.43E-10	N/A		-0.1635	N/A	
0.7500	55	6.175	6.259	0.832	-3.05E-09	0.000	0.00	-0.0523	-0.0482	0.21
0.5000	55	7.406	7.477	0.709	-2.59E-09	0.000	0.00	-0.0796	-0.0722	0.37
0.2500	55	8.580	8.661	0.812	-7.25E-10	0.000	0.00	-0.1730	-0.1445	1.43
0.0500	55	9.122	N/A		-1.18E-10	N/A		-0.4809	N/A	
0.6370	49.2	6.757	6.825	0.683	-3.19E-09	0.000	0.00	-0.0488	-0.0460	0.14

An explanation for the error computation in the above table in needed. Usually, an error percentage is determined by dividing the absolute error (actual value minus the computed) by the actual value. In this case, the absolute error is divided by a characteristic length instead. The reference length used was the total length of the tank in the direction corresponding to the CG coordinate calculated. Describing the error in this way presents the information in a more relevant format. For instance, if the exact CG x-component value of a 10' long tank was 1/16" and the computed value was 1/8", it would normally yield a percent error of 100%. But in the context of a 10' tank, a 1/16" error in CG is not particularly significant. Therefore a percent error in terms of the characteristic length is more relevant, and in this case would be roughly 0.1%.

The MOI of the fuel are estimated from the full tank MOI, which is not dependent upon tank rotation or fuel level (with the exception of 0% to 10% full tanks). Therefore those calculations would yield constant values in the Table 4 above and have been omitted for that reason. MOI test results are shown in Table 5 to illustrate the linear interpolation between empty and full tank values in the range of fill levels from 0% to 10%.

Table 5: MOI Values For Complete Range of Tank Fill Levels

Fuel Weight [lb]	Y Rotation [deg]	Ixx [lb-in-s^2]	Iyy [lb-in-s^2]	Izz [lb-in-s^2]	Ixy [lb-in-s^2]	Ixz [lb-in-s^2]	Iyz [lb-in-s^2]
0.9933	30	4.32E-06	0.02038	0.020367	2.40E-20	-1.07E-19	-4.23E-14
0.75	30	4.32E-06	0.02038	0.020367	2.40E-20	-1.07E-19	-4.23E-14
0.5	30	4.32E-06	0.02038	0.020367	2.40E-20	-1.07E-19	-4.23E-14
0.25	30	4.32E-06	0.02038	0.020367	2.40E-20	-1.07E-19	-4.23E-14
0.05	30	2.17E-06	0.01026	0.0102526	1.21E-20	-5.38E-20	-2.13E-14
0.0	30	0	0	0	0	0	0

Since this test case uses a tank whose full weight is close to 1 lb., the entry with fuel weight of 0.05 lbs. represents a tank that is 5% full. Observe that the MOI values corresponding to 0.05 lbs. are roughly one half of those for fill levels above 10%, and that for an empty tank the MOI are non-existent as would be expected. The POI for this case should be identically zero due to tank symmetry. However, the calculated POI values do show very small non-zero values, which can be explained as floating point error generated by the computer. Finally, the amount of error in the test calculations is well beneath the stated goal of 10%, and the method consistently yields reproducible results. This test case demonstrates that the proposed methodology is appropriate for conceptual design software.

Chapter 5: Applications

CG Envelope Graph

The term “CG envelope” refers to the range of CG values of an aircraft with which the vehicle can fly stably. If the CG travels outside of the envelope due to component movement, fuel slosh, or deployment of stores, then keeping the aircraft airborne may be difficult or even impossible under some circumstances. An example of a CG envelope graph is shown below in Figure 23.

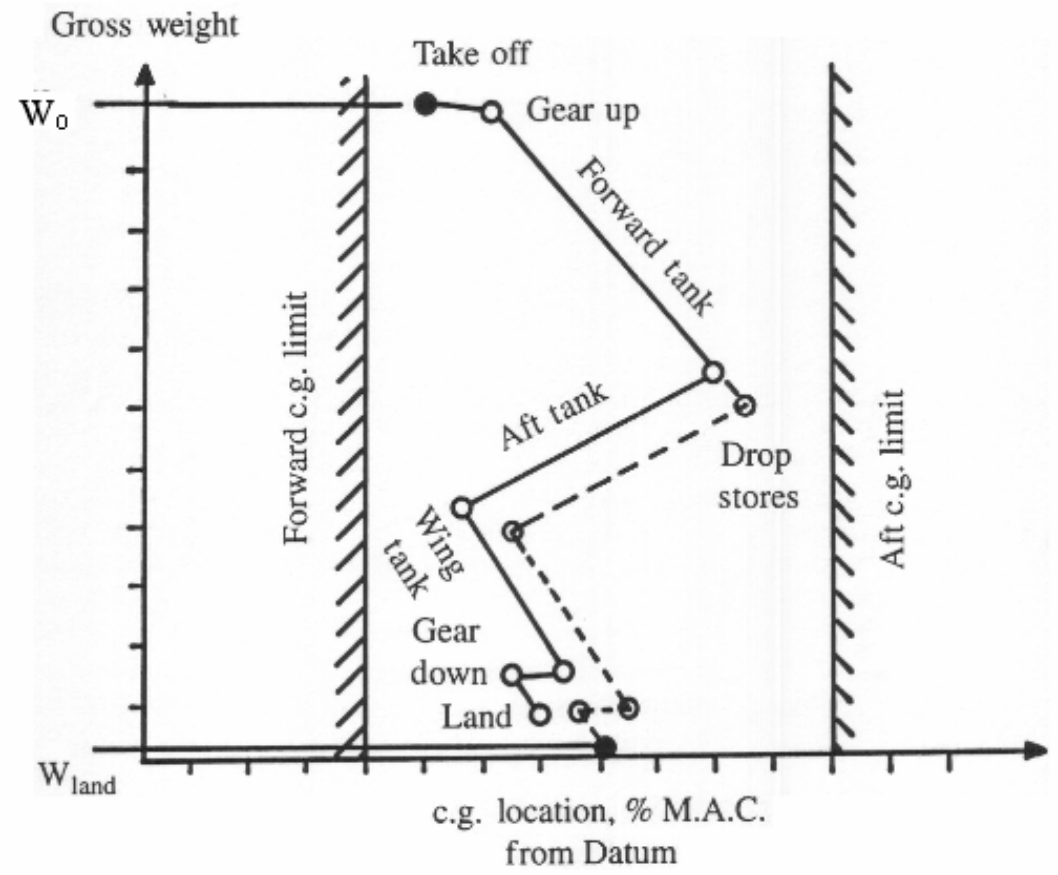

Figure 23: Example CG Envelope Graph

In Figure 23, the weight at take off is W_O and the CG at take off is signified by the large black dot near the top of the graph. The line that proceeds from this point to the final landing weight, W_{land}, represents the travel of the CG during the mission. This graph focuses on the CG movement along the length of the aircraft (i.e. along the x-axis). As the landing gear are raised and lowered, fuel is consumed, and stores are deployed, the CG x-coordinate changes. The CG should never cross the dashed boundaries, which signify the minimum and maximum CG locations for stable flight. As long as the CG stays within this range, it is within the "CG envelope". With the algorithms presented in this thesis, this type of graph can be accurately generated.

To accomplish this, the mass properties of each aircraft component would be calculated using the solid or thin shell algorithms, and these values could be accumulated to determine the aircraft's overall center of gravity. This would be the take off weight. Currently, OAV designs do not include retractable landing gear, but such a situation could be handled by calculating the landing gear's mass properties in extended and retracted positions. The affects of fuel burn on the CG envelope could be determined by evaluating the mass properties of the fuel at decreasing fuel levels. Traditionally, the orientation of the fuel tank is not taken into account in CG envelope graphs. However, for OAV's that will experience large changes in pitch, a variety of aircraft orientations could be analyzed to determine maximum CG travel due to fuel deformation. Finally, deployment of stores could be handled by calculating the mass properties of the vehicle with and without the store's mass present. Lines would be added to connect all of the data points, and fore and aft limits for CG could be added to form the "envelope".

Pitch Trim Stability Analysis

Aerodynamic forces, such as lift and drag, experienced by wings, propellers, ducts, etc. cause moments (torques) to be felt at the CG of the aircraft. These

moments are calculated by multiplying each force by its perpendicular distance to the aircraft CG. If the sum of the moments does not equal zero, then the aircraft will begin to rotate. Therefore, to achieve static stability of the aircraft, the moments about the CG must equal zero. When an aircraft achieves this state it is referred to as “trimmed”. To achieve this cancellation of moments, generally control surfaces such as vanes or elevators in the tail of an aircraft are used to generate a counteracting moment. When a tail elevator is deflected to produce a balancing pitching moment due to its lift, it also has just changed the total lift on the whole vehicle. The total lift experienced by the vehicle must be equal to its weight, or else it will rise or descend. This change in total lift causes a need for change in the angle of attack (angle with the oncoming air) of the vehicle. Therefore, to find the stability state where the lift equals the weight of the vehicle and the pitching moment is equal to zero, an iterative approach is needed [1].

The location of the CG is critical in this process. It affects the moments generated by each component of the craft. By moving the CG, the pitching moment can be nullified or even change sign. The fact that the orientation of the vehicle changes during the iterative stability solving process means that the CG of the vehicle will be changing due to fuel movement, which can have a significant effect on the aircraft’s stability. This is where the usefulness of an interpolation map of CG values for a given fuel weight and orientation angle becomes apparent. Quick access to CG information is needed, since the iterative process could run many times before finding the stability point. If the mass properties were recalculated every time the aircraft’s angle of attack was modified per iteration, the trim analysis would take much longer.

Dynamic Stability

During aeronautical maneuvers, there are six degrees of freedom for the aircraft: roll, pitch, and yaw rotations, and lateral, vertical, and longitudinal changes in velocity. The changes in roll, pitch, and yaw are directly related to the moments

applied to the vehicle. In a static analysis the moments are all equal to zero. However, in a dynamic analysis, the sum of the moments about the CG of the aircraft is equal to the MOI times the angular acceleration of the craft. Conceptually, the moment of inertia represents a body's resistance to angular acceleration, and is comparable to the idea of mass in translational motion. In dynamic simulation, the moments of inertia are vitally important. The algorithms presented in this paper allow for accurate moment of inertia values to be calculated and utilized in the earliest stages of an aircrafts design.

Chapter 6: Conclusions

In this thesis, several methods have been set forth in order to calculate the mass properties of aircraft structural components and liquid fuel in tanks. The first method, applicable to homogenous solids, is an implementation of a previously published algorithm. The second method, which addresses thin shell objects, is a newly formulated algorithm adapted from the homogenous solid method. A methodology for characterizing the mass properties of fuel in tanks has also been developed. While the concepts therein are not completely original, the synthesis of past research from diverse sources has yielded a new comprehensive approach to fuel mass property analysis during conceptual design. More generally, all of these methods apply to polyhedral geometry, which in many cases is used to approximate NURBS surface geometry. The accuracy of each proposed method is within the acceptable range for the conceptual design of aircraft, and is extremely precise. Several relevant and useful applications of the presented methods were explored, including a methodology for creating a CG envelope graph. The methods presented in this thesis have been implemented in C++ and tested for validity in the software system, AVID OAV. The source code containing the algorithms may be found in Appendix A.

Future research could explore several areas related to this work that are beyond the scope of this thesis. A performance comparison of the central projection method and the divergence theorem methods for mass properties calculation would be needed to definitively determine which algorithm is more efficient and/or accurate, and therefore superior. Alternatives for thin shell mass properties calculation could be investigated to determine if a more efficient algorithm exists that produces equivalent or better results than the algorithm set forth in this paper. Also, much research has been done in the realm of inertia calculation of fluid in tanks, but more experimental research could be performed to verify current findings or develop new relationships.

References

1. Raymer, Daniel P., *AIRCRAFT DESIGN: A Conceptual Approach*, 3rd Ed., American Institute of Aeronautics and Astronautics (AAIA), Washington, D.C., 1999.
2. DARPA, "FACT FILE: A Compendium of DARPA Programs", Revision 1, http://www.darpa.mil/body/pdf/final2003factfilerev1.pdf, August 2003.
3. *Weight Engineer's Handbook May 2002*, Society of Allied Weight Engineers (SAWE), Los Angeles, CA, 2002.
4. Beer, Ferdinand P. and E. Russell Johnston, Jr., *Vector Mechanics for Engineers*, 5th Ed., McGraw-Hill, New York, 1988.
5. Stewart, James, *Calculus*, 3rd Ed., Brooks/Cole Publishing, Pacific Grove, 1995.
6. Nakai, John H., "Coordinate Transformation of Inertias Using Tensors", SAWE Paper 2116, Proceedings from 51st Annual Conference of SAWE, Hartford, CT, 1992.
7. Strom, George J., "Matrix Methods for Mass Properties", SAWE Paper 1946, Proceedings from 49th Annual Conference of SAWE, Mesa, AZ, May 1990.
8. Strom, George J., "Mass Properties of Solids Using Third Moments", SAWE Paper 1474, Proceedings from 41st Annual Conference of SAWE, San Jose, CA, 1982.
9. Farin, Gerald, *Curves and Surfaces for CAGD*, 5th Ed., Morgan Kaufmann Publishers, 2002.
10. Mortenson, Michael E., *Geometric Modeling*, 2nd Ed., John Wiley and Sons, New York, 1997.
11. Faux, I. D. and M. J. Pratt, *Computational Geometry for Design and Manufacture*, John Wiley and Sons, New York, 1979.
12. Mäntyla, Martti, *An Introduction to Solid Modeling*, Computer Science Press, Rockville, MD, 1988.
13. Luken, William L., "Tessellation of Trimmed NURB Surfaces", *Computer Aided Geometric Design*, Vol. 13, 1996, p 163-177.
14. Luken, William L., "Comparison of Surface and Derivative Evaluation Methods for the Rendering of NURB Surfaces", *ACM Transactions in Grpahics*, Vol. 15, No. 2, 1996, p 153-178.
15. Piegl, Les A. and Wayne Tiller, "Geometry-based Triangulation f Trimmed NURBS Surfaces", *Computer-Aided Design*, Vol. 30, No. 1, 1998, p 11-18.

16. Sheng, X. and B. E. Hirsch, “Triangulation of Trimmed Surfaces in Parametric Space”, *Computer-Aided Design*, Vol. 24, No. 8, 1992, p 437-444.

17. Lee, Yong Tsui and Aristides A.G. Requicha, “Algorithms for Computing the Volume and Other Integral Properties of Solids. I. Known Methods and Open Issues”, *Comm. ACM,* Vol. 25, No. 9, Sept. 1982, p 635-641.

18. Lee, Yong Tsui and Aristides A.G. Requicha, “Algorithms for Computing the Volume and Other Integral Properties of Solids. II. A Family of Algorithms Based on Representation Conversion and Cellular Approximation”, *Comm. ACM,* Vol. 25, No. 9, Sept. 1982, p 642-650.

19. Messner, Adrian M., “A Surface Integral Method for Computer Calculation of Mass Properties”, SAWE Paper 852, Proceedings from 29th Annual Conference of the SAWE, Washington D.C., May 1970.

20. Messner, Adrian M. and G.Q. Taylor, “Algorithm 550, Solid Polyhedron Measures [Z]”, *ACM Transactions on Mathematical Software*, Vol. 6, No. 1, March 1980, p 121-130.

21. Gonzalez-Ochoa, Scott McCammon and Jörg Peters, “Computing Moments of Objects Enclosed by Piecewise Polynomial Surfaces”, *ACM Transaction on Graphics*, Vol. 17, No. 3, July 1998, p 143-157.

22. Cattani, C. and A. Paoluzzi, “Boundary Integration Over Linear Polyhedra”, *Computer-Aided Design*, Vol. 22, Issue 2, March 1990, p 130-135.

23. Lien, Sheue-ling and James T. Kajiya, “A Symbolic Method for Calculating the Integral Properties of Arbitrary Non-convex Polyhedra”, *Computer Graphics and Applications*, IEEE, October 1984, p 35-41.

24. Roskam, Jan, *Airplane Design, Part V: Component Weight Estimation*, Roskam Aviantion and Engineering Corporation, Ottawa, Kansas, 1989.

25. Scott, Paul W., “Conceptual Estimation of Moments of Inertia”, SAWE Paper 2171, Proceedings from 52nd Annual Conference of SAWE, Biloxi, Mississippi, May 1993.

26. Adelson, Richard L., “Airplane Center of Gravity and Fuel Level Advisory System”, SAWE Paper 1828, Proceedings from 47th Annual Conference of the SAWE, Plymouth, Michigan, May 1998.

27. Quinlivan, Patrick J., “Automated Weight and Balance System”, SAWE Paper 2049, Proceedings from 50th Annual Conference of SAWE, San Diego, CA, May 1991.

28. Hargrave, John, “3-D Graphics as a Mass Properties Working Tool”, SAWE Paper 1646, Proceedings from 44th Annual Conference of SAWE, Arlington, TX, May 1985.

29. Wiegand, Brian Paul, “The Basic Algorithms of Mass Properties Analysis and Control (Accounting, Uncertainty, and Standard Deviation)”, SAWE Paper 2067, Proceedings from 41st Annual Conference of SAWE, Hartford, CT, May 1992.

30. Sabersky, Rolf H., Allan J. Acosta, Edward G. Hauptmann, and E.M. Gates, *Fluid Flow: A First Course in Fluid Mechanics*, 4th Edition, Prentice-Hall, Upper Saddle River, NJ, 1999, pp 1.

31. Boynton, Richard and Robert Bell, “The Moment of Inertia of Fluids—Part 2”, SAWE Paper 3006, Proceedings from 59th Annual Conference of the SAWE, Berlin, Connecticut, June 2000.

32. Silverman, Sandor and H. Norman Abramson, “Lateral Sloshing in Moving Containers”, NASA Paper N67 15886, NASA, 1967.

33. Silverman, Sandor and H. Norman Abramson, “damping of Liquid Motions and Lateral Sloshing”, NASA Paper N67 15888, NASA, 1967.

34. Anonymous, “Slosh Suppression”, NASA Paper SP-8031, NASA, 1969.

35. Lomen, D. O., “Liquid Propellant Sloshing In Mobile Tanks of Arbitrary Shape”, NASA Paper CR-222, NASA, 1965.

36. Dodge, Franklin T., “Analytical Representation of Lateral Sloshing by Equivalent Mechanical Models”, NASA Paper N67 15891, NASA, 1967.

37. Hassman, Lewis T., “Mass Properties of Fluids in Large Containers”, SAWE Paper 1317, Proceedings from 38th Annual Conference of the SAWE, New York, NY, 1979.

38. Boesch, Colleen M. and Bell C. Lee, “A Method for Calculating Fuel Slosh Time Lag”, SAWE Paper 2259, Proceedings from 54th Annual Conference of the SAWE, Huntsville, Alabama, May 1995.

39. Molczyk, Gerald Jon, “IMMP – A Computer Simulation of Fuel CG Versus Vehicle Attitude”, SAWE Paper 1801, Proceedings from 46th Annual Conference of SAWE, Seattle, WA, May 1987.

40. Lin, W. H. and A. Myklebust, “A Constraint Driven Solid Modeling Open Environment”, presented at and published in the *Proceedings of the Second ACM/IEEE Symposium on Solid Modeling and Applications*, Montreal, Canada, May 19-21, 1993.

41. Lin, W. H., “Object-Oriented Software Development Environment for Geometric Modeling in Intelligent Computer Aided Design”, Ph.D. Thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1992.

42. Lin, W. H., “A Research Report to IBM Corporation: An Object-Oriented Software Development Environment for Geometric Modeling in Intelligent Computer Aided Design”, Report 436023-5, July 15, 1992.

43. Daguia, Marie and Ray Schumacher, “Fuel Analysis: Methods, Techniques, and Calibration”, SAWE Paper 3233, Proceedings from 61st Annual Conference of SAWE, Virginia Beach, Virginia, May 2002.

44. Segal, Mark and Carlos H. Sequin, “Partitioning Polyhedral Objects into Nonintersecting Parts”, *Computer Graphics and Applications*, IEEE, Vol. 8, Issue 1, Jan. 1988, p 53-67.

45. Rossignac, Jarek, Abe Megahed, and Bengt-Olaf Schneider, “Interactive Inspection of Solids: Cross-sections and Interferences”, *ACM Transactions on Computer Graphics*, Vol. 26, No. 2, July 1992, p 353-360.

46. Greiner, Günther and Kei Hormann, “Efficient Clipping of Arbitrary Polygons”, *ACM Transactions on Graphics*, Vol. 17, No. 2, April 1998, p 71-83.

47. Phillips, M.S. and H.F. Bauer “Analytical Expressions for the Moment of Inertia of a Filled Cylindrical Tank about Various Roll and Yaw-Pitch Axes”, SPACO Report No. 212-3.2.2 March 1966.

48. Widmayer, Edward, Jr. and James R. Reese, “Moment of Inertia and Damping of Fluid in Tanks Undergoing Pitching Oscillations”, NACA Research Memorandum L53E01a, June 1953.

49. Lamb, Horace, *Hydrodynamics*, 6th Ed., Cambridge University Press, Cambridge, 1932.

50. Miles, John W., “An Analogy Among Torsional Rigidity, Rotating Fluid Inertia, and Self-Inductance for an Infinite Cylinder”, Journal of Aerodynamic Science, Vol. 13, No. 7, July 1946, p 377-380.

51. Anonymous, “The Calculation of the Mass Moment of Inertia of a Fluid in a Rotating Rectangular Tank”, NASA Paper CR 197777, NASA, 1977.

52. Graham, E.W. and A. M. Rodriguez, “The Characteristics of Fuel Motion Which Affect Airplane dynamics”, *Journal of Applied Mechanics*, September 1952.

Appendix A: Source Code

Homogenous Solid Mass Properties Code:

```
static double Determinant(double mtx[3][3]){
  return mtx[0][0] * mtx[1][1] * mtx[2][2] + mtx[1][0] * mtx[2][1] * mtx[0][2]
       + mtx[2][0] * mtx[0][1] * mtx[1][2] - mtx[2][0] * mtx[1][1] * mtx[0][2]
       - mtx[1][0] * mtx[0][1] * mtx[2][2] - mtx[0][0] * mtx[2][1] * mtx[1][2];
}

class cMassProperties
{
    public:
        double Mass;
        double Volume;
        double SurfaceArea;
        double WettedSurfaceArea;
        double ContainerVolume;
        double Thickness; // only valid for thin shells
        double Density;
        double Xoff; // CG x coordinate from local geometry origin
        double Yoff;
        double Zoff;
        double Ixx; // MOI are about local geometry coordinate axes
        double Iyy;
        double Izz;
        double Ixy;
        double Ixz;
        double Iyz;
}


// cVisibleComponent is basically a list of polygon patched called cBorderSections
static cMassProperties CalcMassPropsFromGeometry(const cVisibleComponent & vc,
                                 double density,
                                 bool bexcludemoments=false);
static cMassProperties CalcMassPropsFromShellGeometry(const cVisibleComponent & vc,
                                double density,
                                double thickness);

////////////////////////////////////////////////////////////////////////
//  static CalcMassPropsFromGeometry(const cVisibleComponent & vc, double density, bool
bexcludemoments = false)
//  Reporter method
//  Purpose: Calculates shape's mass properties from its geometry (only applies
//            to visible comps) as if it were a homogeneous solid
//  Parameters:
//      const cVisibleComponent vc: the component whose values are being calculated
//      double density: density of component (slugs/in3)
//      bool bexcludemoments = false - optional param that specifies whether
//             to exclude moments of inertia.  The cFuelTank is the only
//             object to use this option currently
//  Pre: None.
//  Post:  none  returns cMassProperties object with new values
//  Coded by: John Ohanian 12/19/2002
//  Modified by:
//  Modifications:
cMassProperties CalcMassPropsFromGeometry(const cVisibleComponent & vc, double density, bool
bexcludemoments)
```

```
{
  cMassProperties mp;
  mp.Density = density;
  mp.Thickness = 0.0;

  // values to be calculated
  double dvolume=0.0;
  double dsurfacearea=0.0;
  double dwettedsurfacearea=0.0;
  double dcentroid[3]={0.0, 0.0, 0.0};
  double dixx=0.0;
  double diyy=0.0;
  double dizz=0.0;
  double dixy=0.0;
  double diyz=0.0;
  double dizx=0.0;

  // get the list of patches (BorderSections)
  list<cBorderSection>::const_iterator ipatch, endpatch;
  endpatch = vc.GetVisibleComponent().end();

  int numloops = bexcludemoments? 1 : 2;

  // need to sweep through the patches twice, first to calc volume & centroid
  // second, move origin to centroid and calc inertias
  // there's a lot of prep work in common so I made it a for loop, sorry for the
  // confusing code, but it keeps down the copy/paste maintanence
  int bdoinginertia=0;
  while(bdoinginertia<numloops){

    // cycle thru patches and calculate volume and centroid
    ipatch = vc.GetVisibleComponent().begin();
    while(ipatch != endpatch){
      // get the list of points for this surface patch
      // cPoint is basically an X,Y,Z triple
      list<cPoint> points = (*ipatch).GetBorder();

      int trianglesperpatch = points.size()-2;
      bool bfirstsameaslast = (points.front().GetX() == points.back().GetX())
        && (points.front().GetY() == points.back().GetY())
        && (points.front().GetZ() == points.back().GetZ()) ;

      if (bfirstsameaslast){
        trianglesperpatch--; // subtract one off from number of triangles because last point is not
unique
      }

      // for each triangle, we'll calculate mass props
      int t=0;
      while(t<trianglesperpatch){
        // if only three points, will use 1,2,3
        // if four points or more (counterclockwise around rectangle),
        // break into triangles, always start with point 1
        // so triangle 2 would be 1,3,4, triangle 3 would be 1,4,5 and so on...
        // a tetrahedra is then created by connecting  0,0,0 (the origin) to the other 3 points
        double pts[3][3]; // first index is point, second is xyz
        list<cPoint>::const_iterator ipoint = points.begin();
        int i=0;
        while(i<3){
          // subtract centroid for inertia calculations
          pts[i][0] = (bdoinginertia)? (*ipoint).GetX() - dcentroid[0]: (*ipoint).GetX();
          pts[i][1] = (bdoinginertia)? (*ipoint).GetY() - dcentroid[1]: (*ipoint).GetY();
          pts[i][2] = (bdoinginertia)? (*ipoint).GetZ() - dcentroid[2]: (*ipoint).GetZ();
```

```
          ipoint++;
          if (i==0){ // after first point, bump up to 2nd point depending on what number triangle (t)
            int j=0;
            while(j<t){ipoint++; j++;}
          }
          i++;
        }

        // calculate the determinant of the jabcobian (changed the sign to get correct sense)
        double d = Determinant(pts);

        // now the calculation of the values, first the volume and centroid
        // on second sweep, inertias
        if (!bdoinginertia){
          // calculate volume of this tetrahedra
          double dtetravolume = d/6;

          // add this incremental volume to the component's
          dvolume += dtetravolume;

          // calculate the surface area of this triangle
          // the cross product of 2 sides divided by 2, since |a x b| = |a||b|sin(theta)
          cPoint vec1(pts[1][0]-pts[0][0], pts[1][1]-pts[0][1], pts[1][2]-pts[0][2]);
          cPoint vec2(pts[2][0]-pts[0][0], pts[2][1]-pts[0][1], pts[2][2]-pts[0][2]);
          cPoint cross = vec1*vec2; // cross product
          double crossmagnitude = sqrt( pow(cross.GetX(), 2) + pow(cross.GetY(), 2) +
pow(cross.GetZ(), 2));
          dsurfacearea += crossmagnitude/2;

          // calculate centroid
          double dtetrax = (pts[0][0] +pts[1][0] + pts[2][0])/4;
          double dtetray = (pts[0][1] +pts[1][1] + pts[2][1])/4;
          double dtetraz = (pts[0][2] +pts[1][2] + pts[2][2])/4;
          // add this value times the tetra's volume to the component's X,Y,Z, later to be divided by
total volume
          dcentroid[0] += dtetrax*dtetravolume;
          dcentroid[1] += dtetray*dtetravolume;
          dcentroid[2] += dtetraz*dtetravolume;
        }
        else{
          // calculate inertias
          double dix = d/60 * (pts[0][0]*pts[0][0] + pts[1][0]*pts[1][0] + pts[2][0]*pts[2][0] +
                       pts[0][0]*pts[1][0] + pts[0][0]*pts[2][0] + pts[1][0]*pts[2][0]);
          double diy = d/60 * (pts[0][1]*pts[0][1] + pts[1][1]*pts[1][1] + pts[2][1]*pts[2][1] +
                       pts[0][1]*pts[1][1] + pts[0][1]*pts[2][1] + pts[1][1]*pts[2][1]);
          double diz = d/60 * (pts[0][2]*pts[0][2] + pts[1][2]*pts[1][2] + pts[2][2]*pts[2][2] +
                       pts[0][2]*pts[1][2] + pts[0][2]*pts[2][2] + pts[1][2]*pts[2][2]);
          double dtetraIxx = diy + diz;
          double dtetraIyy = dix + diz;
          double dtetraIzz = dix + diy;
          double dtetraIxy = d/120 * (2*(pts[0][0]*pts[0][1] + pts[1][0]*pts[1][1] +
pts[2][0]*pts[2][1])
                              + (pts[0][0]*pts[1][1] + pts[1][0]*pts[0][1] + pts[0][0]*pts[2][1]
                               + pts[2][0]*pts[0][1] + pts[1][0]*pts[2][1] + pts[2][0]*pts[1][1]));
          double dtetraIyz = d/120 * (2*(pts[0][2]*pts[0][1] + pts[1][2]*pts[1][1] +
pts[2][2]*pts[2][1])
                              + (pts[0][2]*pts[1][1] + pts[1][2]*pts[0][1] + pts[0][2]*pts[2][1]
                               + pts[2][2]*pts[0][1] + pts[1][2]*pts[2][1] + pts[2][2]*pts[1][1]));
          double dtetraIzx = d/120 * (2*(pts[0][0]*pts[0][2] + pts[1][0]*pts[1][2] +
pts[2][0]*pts[2][2])
                              + (pts[0][0]*pts[1][2] + pts[1][0]*pts[0][2] + pts[0][0]*pts[2][2]
                               + pts[2][0]*pts[0][2] + pts[1][0]*pts[2][2] + pts[2][0]*pts[1][2]));
```

```
          // add these values to the component's
          dixx += dtetraIxx;
          diyy += dtetraIyy;
          dizz += dtetraIzz;
          dixy += dtetraIxy;
          diyz += dtetraIyz;
          dizx += dtetraIzx;
        }

        t++;
      }

      ipatch++;
    }

    // if this is the end of 1st time thru, calculate total volume and centroid
    // for data collected in first pass
    if (!bdoinginertia){
      // check for invalid volume
      if (dvolume == 0.0){
        cout << "Zero volume! What happened?" << endl;
        return mp;
      }

      // now divide X,Y,Z by total volume to get correct values
      dcentroid[0] = dcentroid[0]/dvolume;
      dcentroid[1] = dcentroid[1]/dvolume;
      dcentroid[2] = dcentroid[2]/dvolume;

      dwettedsurfacearea = dsurfacearea;

    }

    bdoinginertia++;
  }

  // create cMassProperties with all values
  mp.Mass = dvolume*density;
  mp.Volume = dvolume;
  mp.SurfaceArea = dsurfacearea;
  mp.WettedSurfaceArea = dwettedsurfacearea;
  mp.Xoff = dcentroid[0];
  mp.Yoff = dcentroid[1];
  mp.Zoff = dcentroid[2];
  mp.Ixx = dixx*density;
  mp.Iyy = diyy*density;
  mp.Izz = dizz*density;
  mp.Ixy = dixy*density;
  mp.Iyz = diyz*density;
  mp.Ixz = dizx*density;

  // all values in returned cMassProperties should be correct by this point
  return mp;
}
```

Thin Shell Mass Property Code

```
////////////////////////////////////////////////////////////////////////
//  static CalcMassPropsFromShellGeometry(const cVisibleComponent & vc, double density,
//                                        double thickness)
//  Reporter method
```

```
//  Purpose: Calculates shapes's mass properties from its geometry (only applies
//           to visible comps) as if it were a thin shell
//  Parameters:
//      const cVisibleComponent vc: the component whose values are being calculated
//      double density: density of component
//      double thickness: thickness of shell, must be greater than zero
//  Pre: None.
//  Post:  none  returns cMassProperties object with new values
//  Coded by: John Ohanian 1/10/2002
//  Modified by:
//  Modifications:
//  Note: This method was derived from CalcMassPropsFromGeometry, but is substantially
//        different since it computes centroid differently, and must use an inner and
//        outer surface due to the thickness of the shell when calculating volume and inertias
cMassProperties CalcMassPropsFromShellGeometry(const cVisibleComponent & vc, double density, double
thickness)
{
  cMassProperties mp;
  mp.Density = density;
  mp.Thickness = thickness;

  if (thickness <= 0.0){
    cout << "Thickness <= 0 ! Returned empty result..." << endl;
    return mp;
  }

  // values to be calculated
  double dvolume=0.0; // volume of the solid
  double dcontainervolume=0.0; // how much the empty shell could hold
  double dsurfacearea=0.0;
  double dwettedsurfacearea=0.0;
  double dcentroid[3]={0.0, 0.0, 0.0};
  double dixx=0.0;
  double diyy=0.0;
  double dizz=0.0;
  double dixy=0.0;
  double diyz=0.0;
  double dizx=0.0;
  // subtracted volume/moi
  double dnegvolume=0.0;
  double dnegixx=0.0;
  double dnegiyy=0.0;
  double dnegizz=0.0;
  double dnegixy=0.0;
  double dnegiyz=0.0;
  double dnegizx=0.0;

  // get the list of patches (BorderSections)
  list<cBorderSection>::const_iterator ipatch, endpatch;
  endpatch = vc.GetVisibleComponent().end();

#ifdef DEBUG_MASSPROPS
  cout << endl << endl << "***********Mass Properties***************" << endl;
  cout << "Number of patches: " << vc.GetVisibleComponent().size() << endl;
#endif

  // need to sweep through the patches twice, first to calc volume & centroid
  // second, move origin to centroid and calc inertias
  // there's a lot of prep work in common so I made it a for loop, sorry for the
  // confusing code, but it keeps down the copy/paste maintanence
  int bdoinginertia=0;
  while(bdoinginertia<2){
```

```
    // if this is the beginning of second time thru, calculate total volume and centroid
    // from data collected in first pass
    if (bdoinginertia){
      // check for invalid surface area
      if (dsurfacearea == 0.0){
        cout << "Zero surface area! What happened?" << endl;
        return mp;
      }

      // now divide X,Y,Z by total surface area to get correct values
      // usually would divide by volume here, but I weighted the triangle centroids by area
      dcentroid[0] = dcentroid[0]/dsurfacearea;
      dcentroid[1] = dcentroid[1]/dsurfacearea;
      dcentroid[2] = dcentroid[2]/dsurfacearea;

    }

    // cycle thru patches
    ipatch = vc.GetVisibleComponent().begin();
    while(ipatch != endpatch){
      // get the list of points for this surface patch
      list<cPoint> points = (*ipatch).GetBorder();

      int trianglesperpatch = points.size()-2;
      bool bfirstsameaslast = (points.front().GetX() == points.back().GetX())
         && (points.front().GetY() == points.back().GetY())
         && (points.front().GetZ() == points.back().GetZ()) ;

      if (bfirstsameaslast){
        trianglesperpatch--; // subtract one off from number of triangles because last point is not
unique
      }

      // for each triangle, we'll calculate props
      int t=0;
      while(t<trianglesperpatch){
        // if only three points, will use 1,2,3
        // if four points or more (counterclockwise around rectangle),
        // break into triangles, always start with point 1
        // so triangle 2 would be 1,3,4, triangle 3 would be 1,4,5 and so on...
        // a tetrahedra is then created by connecting  0,0,0 (the origin) to the other 3 points
        double pts[3][3]; // first index is point, second is xyz
        list<cPoint>::const_iterator ipoint = points.begin();
        int i=0;
        while(i<3){
          // subtract centroid for inertia calculations
          pts[i][0] = (bdoinginertia)? (*ipoint).GetX() - dcentroid[0]: (*ipoint).GetX();
          pts[i][1] = (bdoinginertia)? (*ipoint).GetY() - dcentroid[1]: (*ipoint).GetY();
          pts[i][2] = (bdoinginertia)? (*ipoint).GetZ() - dcentroid[2]: (*ipoint).GetZ();
          ipoint++;
          if (i==0){ // after first point, bump up to 2nd point depending on what number triangle (t)
            int j=0;
            while(j<t){ipoint++; j++;};
          }
          i++;
        }

        if(!bdoinginertia){ // calculate surface area, centroid on first pass

          // calculate the surface area of this triangle
          // the cross product of 2 sides divided by 2, since |a x b| = |a||b|sin(theta)
          cPoint vec1(pts[1][0]-pts[0][0], pts[1][1]-pts[0][1], pts[1][2]-pts[0][2]);
          cPoint vec2(pts[2][0]-pts[0][0], pts[2][1]-pts[0][1], pts[2][2]-pts[0][2]);
```

```
          cPoint cross = vec1*vec2;
          double crossmagnitude = sqrt( pow(cross.GetX(), 2) + pow(cross.GetY(), 2) +
pow(cross.GetZ(), 2));
         double trianglearea = crossmagnitude/2.0;
          dwettedsurfacearea += trianglearea;
          dsurfacearea += trianglearea*2.0; // want both sides of the triangle, this is an
approximation

          // calculate centroid of triangle and weight by triangle area, later to be divided by total
area
          dcentroid[0] += ((pts[0][0] + pts[1][0] + pts[2][0])/3)*(trianglearea*2.0); //
trianglearea*2.0
          dcentroid[1] += ((pts[0][1] + pts[1][1] + pts[2][1])/3)*(trianglearea*2.0); // because
thats what's used
          dcentroid[2] += ((pts[0][2] + pts[1][2] + pts[2][2])/3)*(trianglearea*2.0); // above to
calc dsurfacearea
        }
        else { // calculate inertias on second pass

          // use tetrahedra method, but do once positive (outer surface) and once negative (inner
surface)
          int io=0;
          while(io<2){ // io=0 means outer surface, io=1 means inner surface
            if (io==1){ // doing inner surface
              // need to determine lenght of vectors to points for scaling
              double lengths[3];
              int k=0;
              while(k<3){
                lengths[k] = sqrt( pow(pts[k][0], 2) + pow(pts[k][1], 2) + pow(pts[k][2], 2) );
                k++;
              }
              // also will need normal vector to triangle, normalized (length 1)
              cPoint vec1(pts[1][0]-pts[0][0], pts[1][1]-pts[0][1], pts[1][2]-pts[0][2]);
              cPoint vec2(pts[2][0]-pts[0][0], pts[2][1]-pts[0][1], pts[2][2]-pts[0][2]);
              cPoint normalvec = vec1*vec2; // cross product
              normalvec.Normalize();

              // find inner surface, made of points scaled back from outer surface by thickness
              // here we find length of point vector that projects onto thickness length normal
vector
              // this is done by use of a dot product
              bool bok = true;
              double mapped_t[3];
              int j=0;
              while(j<3){
                double dot = normalvec.GetX()*(pts[j][0])
                       + normalvec.GetY()*(pts[j][1])
                       + normalvec.GetZ()*(pts[j][2]);
                mapped_t[j] = (thickness*lengths[j]) / dot;
                  if(fabs(mapped_t[j]) >= fabs(lengths[j])){
                  bok = false; // The origin is within the thickness, and will give bogus values
                  break;
                }
                j++;
              }
              if (bok){

                // calculate inner points by multiplying each vector component by scale factor
                int m=0;
                while(m<3){ // point index
                  int n=0;
                  while(n<3){ // xyz index
                    if (lengths[m] != 0.0){
```

```
                pts[m][n] = ((lengths[m] - mapped_t[m])/lengths[m]) * pts[m][n];
              }
              n++;
            }
            m++;
          }
        }
      }

      // calculate the determinant of the jabcobian
      double d = Determinant(pts);

      // calculate volume
      if (io==0){
        dvolume += d/6;
      } else{
        dnegvolume -= d/6; // positive for outer, negative for inner surface

        // caluclate container volume, sum of inner tetrahedra
        dcontainervolume += d/6;
      }

      // calculate inertias
      double dix = d/60 * (pts[0][0]*pts[0][0] + pts[1][0]*pts[1][0] + pts[2][0]*pts[2][0] +
                   pts[0][0]*pts[1][0] + pts[0][0]*pts[2][0] + pts[1][0]*pts[2][0]);
      double diy = d/60 * (pts[0][1]*pts[0][1] + pts[1][1]*pts[1][1] + pts[2][1]*pts[2][1] +
                   pts[0][1]*pts[1][1] + pts[0][1]*pts[2][1] + pts[1][1]*pts[2][1]);
      double diz = d/60 * (pts[0][2]*pts[0][2] + pts[1][2]*pts[1][2] + pts[2][2]*pts[2][2] +
                   pts[0][2]*pts[1][2] + pts[0][2]*pts[2][2] + pts[1][2]*pts[2][2]);
      double dtetraIxx = diy + diz;
      double dtetraIyy = dix + diz;
      double dtetraIzz = dix + diy;
      double dtetraIxy = d/120 * (2*(pts[0][0]*pts[0][1] + pts[1][0]*pts[1][1] +
pts[2][0]*pts[2][1])
                           + (pts[0][0]*pts[1][1] + pts[1][0]*pts[0][1] + pts[0][0]*pts[2][1]
                            + pts[2][0]*pts[0][1] + pts[1][0]*pts[2][1] + pts[2][0]*pts[1][1]));
      double dtetraIyz = d/120 * (2*(pts[0][2]*pts[0][1] + pts[1][2]*pts[1][1] +
pts[2][2]*pts[2][1])
                           + (pts[0][2]*pts[1][1] + pts[1][2]*pts[0][1] + pts[0][2]*pts[2][1]
                            + pts[2][2]*pts[0][1] + pts[1][2]*pts[2][1] + pts[2][2]*pts[1][1]));
      double dtetraIzx = d/120 * (2*(pts[0][0]*pts[0][2] + pts[1][0]*pts[1][2] +
pts[2][0]*pts[2][2])
                           + (pts[0][0]*pts[1][2] + pts[1][0]*pts[0][2] + pts[0][0]*pts[2][2]
                            + pts[2][0]*pts[0][2] + pts[1][0]*pts[2][2] + pts[2][0]*pts[1][2]));

      // add these values to the component's
      if (io==0){ // outer
        dixx += dtetraIxx;
        diyy += dtetraIyy;
        dizz += dtetraIzz;
        dixy += dtetraIxy;
        diyz += dtetraIyz;
        dizx += dtetraIzx;
      } else{ // inner
        dnegixx -= dtetraIxx; // positive for outer, negative for inner surface
        dnegiyy -= dtetraIyy;
        dnegizz -= dtetraIzz;
        dnegixy -= dtetraIxy;
        dnegiyz -= dtetraIyz;
        dnegizx -= dtetraIzx;
      }

      io++;
```

```
          }
        }

        t++;
      }

      ipatch++;
    }

    bdoinginertia++;
  }

  // Now do a reality check to make sure the subtract part's actually negative
  // this sounds silly, but in some cases it can be positive.
  // This will rule out problem cases where thickness is greater
  // than 1/2 the smallest span in very thin parts like
  // (which could actually be heavier as a thin shell than a solid)
  // if it is positive, just don't use it, we'll just return the solid results
  if (dnegvolume < 0.0){
    dvolume += dnegvolume;
    dixx += dnegixx;
    diyy += dnegiyy;
    dizz += dnegizz;
    dixy += dnegixy;
    diyz += dnegiyz;
    dizx += dnegizx;
  }
  else{
    cout << "Thin shell weight would have been greater than solid!!!" << endl;
    cout << "Subtractive weight = " << dnegvolume*density*32.2 << "Lb, supposed to be negative"
<<endl;
  }

  // create cMassProperties with all values
  mp.Mass = dvolume*density;
  mp.Volume = dvolume;
  mp.ContainerVolume = dcontainervolume;
  mp.SurfaceArea = dsurfacearea;
  mp.WettedSurfaceArea = dsurfacearea;
  mp.Xoff = dcentroid[0];
  mp.Yoff = dcentroid[1];
  mp.Zoff = dcentroid[2];
  mp.Ixx = dixx*density;
  mp.Iyy = diyy*density;
  mp.Izz = dizz*density;
  mp.Ixy = dixy*density;
  mp.Iyz = diyz*density;
  mp.Ixz = dizx*density;

  // all values in returned cMassProperties should be correct by this point
  return mp;
}
```

Fuel Analysis Code:

```
#ifndef FUEL_TANK_HEADER
#define FUEL_TANK_HEADER
//////////////////////////////////////////////////////
// cFuelTank.h
// Author: John Ohanian
// Creation Date: 7/9/03
```

```
//
//////////////////////////////////////////////////////

#include <vector>
#include "cPoint.h"
#include "cMassProperties.h"
#include "cVisibleComponent.h"
#include "cTransformationMatrix.h"

#include <stdio.h>
#include <string>
using namespace std;


// helper class, more like a structure
class cWeightCG
{
 public:
  cWeightCG() {Initialize();}
  ~cWeightCG(){}

  double WeightLb;
  double X,Y,Z;

  void Initialize() { WeightLb = X = Y = Z = 0.0;}

  cWeightCG(const cWeightCG& other){ Initialize(); *this = other;}
  const cWeightCG& operator=(const cWeightCG& right)
  {
    if (this != &right)
    {
      // Check for self assignment
                  WeightLb = right.WeightLb;
                  X = right.X;
                  Y = right.Y;
                  Z = right.Z;
    }
          return *this;
  }
};


class cFuelTank
{
 public:
  // default constructor
  cFuelTank();
  // copy constructor
  cFuelTank(const cFuelTank& orig);

  void Initialize();
  // default destructor
  virtual ~cFuelTank();

  // = operator, needed for standard library use
  const cFuelTank& operator=(const cFuelTank& right);

  void Copy(const cFuelTank&);

private:

  /********* Member Attributes **********/
```

```
  // these will determine the size of the map
  int NumberOfThetaIncrements; // includes 0 and 180 degrees
  int NumberOfFuelLevels; // includes empty and full

  // map of cWeighCGs with density=1.0, a vector of vectors
  // of cWeightCGs at differing fuel levels is calculated
  // for a fixed value of theta, and then added to this vector whose
  // entries have differing theta (in other words, first index is
  // theta, second index is fuel level)
  vector<vector<cWeightCG> > UnitycWeightCGsByThetaAndFuelLevelMap;

  // vector of cMassPropertiess with density=1.0, represents mass properties
  // of a full tank at varying degrees of theta, used for MOI
  // will have same size as UnitycWeightCGsByThetaAndFuelLevelMap
  vector<cMassProperties> UnityFullMassPropertiesByTheta;

  // density of the fuel
  double FuelDensitySlugdivIncub;

  // max fuel weight that can be contained in this tank
  double MaxFuelWeightLb;

  // geometry of the tank in local component space (no transforms)
  cVisibleComponent VisCompInLocalSpace;

  // transformation from local space to aircraft space
  cTransformationMatrix LocalToAircraftTransformation;

  // aspect ratios of the tank geometry for each plane
  // typically width/height, but if less than 1, take reciprocal
  double AspectXY, AspectYZ, AspectZX;

  // Should the map be regenerated during Validate() if geometry or
  // orientation of the tank have changed?
  bool bUpdateMapOnGeometryOrTransformationChange;

  // dirty bit to say geometry or orientation have changed
  bool bDirty;

  // Ullage and Internal Structure and Subsystems, affect
  // the maximum capacity of tank
  double UllageFraction;
  double InternalStructureFraction;

  // cache this value since it's used in UI
  double MaxCapacityLb;

//  string strObjectName;
  /********* End Member Attributes **********/


 public:
  /********* Public Methods **********/

  void SetNumberOfThetaIncrements(int);
  void SetNumberOfFuelLevels(int);

  void SetFuelDensitySlugdivIncub(double val);

  void SetGeometryInLocalSpace(const cVisibleComponent& vc);

  // to only set particular values leave other parameters NULL
  void SetLocalToAircraftTransformation(cTransformationMatrix l2a);
```

```
 void SetbUpdateMapOnGeometryOrTransformationChange(bool val);

 void SetUllageFraction(double);
 void SetInternalStructureFraction(double);


 int GetNumberOfThetaIncrements() const;
 int GetNumberOfFuelLevels() const;

 double GetFuelDensitySlugdivIncub() const;

 cVisibleComponent GetGeometryInLocalSpace() const;

 cTransformationMatrix GetLocalToAircraftTransformation() const;

 bool GetbUpdateMapOnGeometryOrTransformationChange() const;

 double GetUllageFraction() const;
 double GetInternalStructureFraction() const;

 double GetMaxCapacityLb() const;

 double GetFractionFull(double fuelweightlb) const; // returns 50% as .5

 void UpdateMaxCapacityLb();

 // will generate map from member data, therefore the geometry,
 // transformation, and density should be correctly set before calling
 int GenerateFuelMassPropertiesMap();

 // fast interpolation from map
 int CalculateFuelMassPropertiesForThetaAndWeight(double thetadeg,
                                      double fuelweightlb,
                                      cMassProperties& toreturn) const;


 // updates all info, returns true if success
 bool Validate();

 // convert a solid cMassProperties into a fluid one (modify the MOI)
 cMassProperties ConvertSolidToFluidMassProperties(cMassProperties solid,
                                   double aspectxy,
                                   double aspectyz,
                                   double aspectzx) const;

 /********* End Public Methods **********/


private:
 /********* Private Helper Functions **********/
 int SlicePolygonGeometry(list<cBorderSection>& origfacets,
                   cPoint planepoint,
                   cPoint planenormal,
                   list<int>& facetflags,
                   bool badvancing,
                   cVisibleComponent& toreturn) const;

 void HandleEntryPoint(cPoint& entry,
                cPoint planepoint,
                cPoint*& punmatchedexit,
                cPoint*& punmatchedentry,
                list<cBorderSection>& acceptedfacets) const;
```

```
  list<cPoint>::iterator CalcAndInsertIntersection(cPoint cur,
                                        cPoint next,
                                        double curdot,
                                        cPoint planenormal,
                                        list<cPoint>::iterator& p,
                                        list<cPoint>& points) const;

  int PrepFuelMassProperties(double thetadeg,
                       double densityslugdivincub,
                       list<cBorderSection>& origfacets,
                       cTransformationMatrix& transform,
                       cMassProperties& localfullmassprops,
                       double& zmin,
                       double& zmax) const;

  int SampleFuelMassProperties(double thetadeg,
                         int numfuellevels,
                         vector<cWeightCG>& weightcgstoreturn,
                         cMassProperties& fulllocalmasspropstoreturn);

  /********* End Private Helper Functions **********/
};
// inline string cFuelTank::ObjectName() const {return strObjectName;}

inline void cFuelTank::SetNumberOfThetaIncrements(int val){NumberOfThetaIncrements=val;}
inline void cFuelTank::SetNumberOfFuelLevels(int val){NumberOfFuelLevels=val;}
inline void cFuelTank::SetFuelDensitySlugdivIncub(double val){FuelDensitySlugdivIncub=val;}
inline void cFuelTank::SetbUpdateMapOnGeometryOrTransformationChange(bool
val){bUpdateMapOnGeometryOrTransformationChange=val;}
inline void cFuelTank::SetLocalToAircraftTransformation(cTransformationMatrix
mtx){LocalToAircraftTransformation=mtx; bDirty=true;}
inline void cFuelTank::SetUllageFraction(double val) {UllageFraction = val; UpdateMaxCapacityLb();}
inline void cFuelTank::SetInternalStructureFraction(double val) {InternalStructureFraction = val;
UpdateMaxCapacityLb();}

inline int cFuelTank::GetNumberOfThetaIncrements() const {return NumberOfThetaIncrements;}
inline int cFuelTank::GetNumberOfFuelLevels() const {return NumberOfFuelLevels;}
inline double cFuelTank::GetFuelDensitySlugdivIncub() const {return FuelDensitySlugdivIncub;}
inline cVisibleComponent cFuelTank::GetGeometryInLocalSpace() const {return VisCompInLocalSpace;}
inline cTransformationMatrix cFuelTank::GetLocalToAircraftTransformation() const {return
LocalToAircraftTransformation;}
inline bool cFuelTank::GetbUpdateMapOnGeometryOrTransformationChange() const {return
bUpdateMapOnGeometryOrTransformationChange;}
inline double cFuelTank::GetUllageFraction() const {return UllageFraction;}
inline double cFuelTank::GetInternalStructureFraction() const {return InternalStructureFraction;}


#endif // FUEL_TANK_HEADER
```

```
//////////////////////////////////////////////////////
// cFuelTank.cpp
// Author: John Ohanian
// Creation Date: 7/9/03
//
//////////////////////////////////////////////////////
const double epsilon = 1.0e-8; // numbers smaller than this are considered zero
                    // also used for determining if point is in the cutting plane
```

```
#define NEGATIVE -1 // on discard side of cutting plane
#define INTERSECT 0 // intersected cuttiong plane
#define POSITIVE  1 // on keep side of cutting plane

#include <iostream>
#include "cFuelTank.h"
#include "NumericalConstants.h" // pi and deg to radian conversion



////////////////////////////////////////////////////////////////////
//  Initialize()
//  Initializer
//  Purpose: To provide a consistent mechanism to initialize values,
//              regardless of constructor used.
//  Parameters: none
//  Pre: Must be called from a constructor.  DO NOT CALL FROM ANY OTHER
//      METHOD
//  Post: new attributes initialized
//  Coded by: John Ohanian
//  Modified by:
//  Modifications:
//
void cFuelTank::Initialize()
{

  NumberOfThetaIncrements = 0;
  NumberOfFuelLevels = 0;
  UnitycWeightCGsByThetaAndFuelLevelMap.clear();
  UnityFullMassPropertiesByTheta.clear();
  FuelDensitySlugdivIncub = 0.001122; // density of water
  MaxFuelWeightLb = 0.0;
  MaxCapacityLb = 0.0;
  AspectXY = AspectYZ = AspectZX = 1.0;
  bUpdateMapOnGeometryOrTransformationChange = false;
  bDirty = false;
  UllageFraction = 0.0;
  InternalStructureFraction = 0.0;
}


cFuelTank::cFuelTank()
{
    Initialize();
}


////////////////////////////////////////////////////////////////////
//  const & operator=(const cFuelTank & f)
//  assignment operator
//  Purpose: sets this to f
//  Parameters: none
//  Pre: f must exist
//  Post:  this is made an independent copy of f
//  Coded by: John Ohanian
//  Modified by:
//  Modifications:
const cFuelTank &  cFuelTank::operator=(const cFuelTank &right)
{
  if (&right != this)
  {
    Copy(right);
  }
```

```
  return *this;
}


//////////////////////////////////////////////////////////////////////
// cFuelTank (const cFuelTank & f)
//  Copy Constructor
//  Purpose: makes a copy of f
//  Parameters: none
//  Pre: f must exist
//  Post:  this is made an independent copy of f
//  Coded by: John Ohanian
//  Modified by:
//  Modifications:
cFuelTank::cFuelTank(const cFuelTank &right)
{
   Initialize();
   Copy(right);
}


void cFuelTank::Copy(const cFuelTank& right)
{
  if (&right == this)
    return;

  NumberOfThetaIncrements = right.NumberOfThetaIncrements;
  NumberOfFuelLevels = right.NumberOfFuelLevels;
  UnitycWeightCGsByThetaAndFuelLevelMap =
    right.UnitycWeightCGsByThetaAndFuelLevelMap;
  UnityFullMassPropertiesByTheta =
    right.UnityFullMassPropertiesByTheta;
  FuelDensitySlugdivIncub = right.FuelDensitySlugdivIncub;
  MaxFuelWeightLb = right.MaxFuelWeightLb;
  MaxCapacityLb = right.MaxCapacityLb;
  VisCompInLocalSpace = right.VisCompInLocalSpace;
  LocalToAircraftTransformation = right.LocalToAircraftTransformation;
  AspectXY = right.AspectXY;
  AspectYZ = right.AspectYZ;
  AspectZX = right.AspectZX;
  bUpdateMapOnGeometryOrTransformationChange =
    right.bUpdateMapOnGeometryOrTransformationChange;
  bDirty = right.bDirty;
  UllageFraction = right.UllageFraction;
  InternalStructureFraction = right.InternalStructureFraction;

  return;
}

cFuelTank::~cFuelTank()
{
}



//////////////////////////////////////////////////////////////////
// Author:  John Ohanian
// Date:    7/30/2003
// Purpose: Set the geometry of the fuel tank, will also extract
//            the aspect ratios in local space, will also mark
//            the tank's map dirty for regen during next validate()
// Input:   viscomp - a cVisibleComponent, representing the
```

```
//            geometry of the tank in local space
//
// Output:  none
////////////////////////////////////////////////////////////////
void cFuelTank::SetGeometryInLocalSpace(const cVisibleComponent& viscomp)
{
  VisCompInLocalSpace = viscomp;

  // need to find out the bounding box of this geometry
  double xmin, xmax, ymin, ymax, zmin, zmax;
  // start inverted so box fits tightly to the geometry
  xmin = ymin = zmin = 10e15;
  xmax = ymax = zmax = -10e15;

  list<cBorderSection> facets = VisCompInLocalSpace.GetVisibleComponent();
  list<cBorderSection>::iterator f = facets.begin();
  while (f != facets.end()){
    list<cPoint> border = (*f).GetBorder();
    list<cPoint>::iterator p = border.begin();
    while (p != border.end()){
      double x = (*p).GetX();
      if (x > xmax){
        xmax = x;
      } else if (x < xmin){
        xmin = x;
      }
      double y = (*p).GetY();
      if (y > ymax){
        ymax = y;
      } else if (y < ymin){
        ymin = y;
      }
      double z = (*p).GetZ();
      if (z > zmax){
        zmax = z;
      } else if (z < zmin){
        zmin = z;
      }
      p++;
    }
    f++;
  }
  double xextent = xmax-xmin;
  double yextent = ymax-ymin;
  double zextent = zmax-zmin;

  // get aspect ratios for MOI calculation from bounding box
  AspectXY = ((xextent==0.0)||(yextent==0.0)) ? 1.0 : xextent/yextent;
  AspectYZ = ((yextent==0.0)||(zextent==0.0)) ? 1.0 : yextent/zextent;
  AspectZX = ((zextent==0.0)||(xextent==0.0)) ? 1.0 : zextent/xextent;

  // need aspect ratio to be > 1, use reciprical
  if (AspectXY < 1.0) AspectXY = 1.0/AspectXY;
  if (AspectYZ < 1.0) AspectYZ = 1.0/AspectYZ;
  if (AspectZX < 1.0) AspectZX = 1.0/AspectZX;

  bDirty = true;
}



////////////////////////////////////////////////////////////////
// Author:  John Ohanian
```

```
// Date:    8/1/2003
// Purpose: Returns how much fuel can go in this tank, taking
//             into consideration Ullage and Internal Structure
// Input:   none
//
// Output:  Weight in Lb that tank can hold
//////////////////////////////////////////////////////////////////
double cFuelTank::GetMaxCapacityLb() const
{
  return MaxCapacityLb;

}

//////////////////////////////////////////////////////////////////
// Author:  John Ohanian
// Date:    8/1/2003
// Purpose: Returns what fraction tank is full, given a fuel weight
//            NOTE: takes into account Ullage and Internal Structure!
// Input:   fuelweightlb - amount of fuel in tank
//
// Output:  fraction fuelweightlb/MaxCapacityLb, can be > 1
//////////////////////////////////////////////////////////////////
double cFuelTank::GetFractionFull(double fuelweightlb) const
{
  double capacitylb = GetMaxCapacityLb();

  if (capacitylb <= 0.0){
    return 1000;// surely more than a hundred percent full
  }

  return fuelweightlb/capacitylb;
}




//////////////////////////////////////////////////////////////////
// Author:  John Ohanian
// Date:    6/16/2003
// Purpose: Performs a single mass property calculation on geometry
//              that represents a partially full fuel tank
// Input:   origfacets - polygon patches "facets" representing the geometry
//          planepoint - a point on the cutting plane
//          planenormal - the normal vector of the plane
//          facetflags - record of what facets were classified as on
//              the last run
//          badvancing - goes with facetflags, says which direction
//              the cutting plane is moving with respect to last run
//              true = moving towards full, false = moving towards empty
//          toreturn (return value) - cut geometry
//
// Output:  0 if succeeded, negative if error
//////////////////////////////////////////////////////////////////
int cFuelTank::SlicePolygonGeometry(list<cBorderSection>& origfacets,
                       cPoint planepoint,
                       cPoint planenormal,
                       list<int>& facetflags,
                       bool badvancing,
                       cVisibleComponent& toreturn) const
{
  // the list of accepted facets for this coming run
  // they will be used to calculate mass properties
  list<cBorderSection> acceptedfacets;
```

```
// loop through all the facets to check whether to keep, chuck, or cut
list<cBorderSection>::iterator f = origfacets.begin();
list<int>::iterator flagiter = facetflags.begin();
while (f != origfacets.end()){

  // check facetflags to see if we can optimize on this facet
  // if last time it was positive and we are advancing in the plane,
  // then we will definitely keep it this time, don't analyze it
  // if last time it was negative and we are retreating the plane,
  // then we will definitely discard it this time, don't analyze it
  bool bskip = false;
  if ((badvancing == true) && ((*flagiter) == POSITIVE)){
    acceptedfacets.push_back(*f); // include it again on this run
    bskip = true;
  }
  else if ((badvancing == false) && ((*flagiter) == NEGATIVE)){
    // exclude it again on this run
    bskip = true;
  }

  if (bskip == false){
    // get points for this facet (list will be modified)
    list<cPoint> points = (*f).GetBorder();
    list<cPoint>::iterator p = points.begin();
    if (p != points.end()){ // handles empty list

    // cPoint's of interest
    cPoint next, cur = (*p);

    // need to make sure first point is same as last (complete loop)
    list<cPoint>::iterator last = points.end(); last--;
    if ((cur.GetX() != (*last).GetX()) ||
      (cur.GetY() != (*last).GetY()) ||
      (cur.GetZ() != (*last).GetZ()))
    {
      points.push_back(cur);
    }

    // We need to cap the cut goemetry, so for each polygonal facet that intersects
    // the cutting plane, we need to trim it and also create a new cap facet that
    // includes the newly created segment in the cutting plane.
    // When a facet intersects the cutting plane, there will be an exit intersection
    // and an entry intersection point.  The cap facet will take the form
    // {planepoint, entrypoint, exitpoint}, this order keeps outward normal
    // warning: the entry and exit used to create a cap should be found as
    // you go around the edge of the facet, finding the exit first then the entry
    cPoint* punmatchedexit = NULL; // holds exit until you find a corresponding entry
    cPoint* punmatchedentry = NULL;// if an entry is found first, you won't be able
    // to match it until you get to the end, so hold it

    // values of point relative to plane dotted with the normal
    // determines on which side of the plane the point is
    double prevdot = 0.0, nextdot = 0.0, curdot = planenormal.Dot(cur - planepoint);

    // if there is at least one positive point, then we need to keep the facet
    int numpos = 0, numneg = 0, numpoints = points.size()-1;//only count unique points

    while (++p != points.end()){
      next = (*p);
      nextdot = planenormal.Dot(next - planepoint);
```

```
// Okay, here are the rules for including/excluding points and
// inserting intersection points:
// 1. Points on the plane -
//    a) if prev and next are positive -> include it
//    b) if prev pos, next !pos -> include it, set as exit
//    c) if prev !pos, next pos -> include it, handle entry
//       d) otherwise exclude it
// 2. Point has positive dot (on the right side of the plane)
//       -> include it
//       if next neg -> insert intersection point, set as exit
// 3. Point has negative dot (on the wrong side)
//       -> exclude it
//       if next pos -> insert intersection point, handle entry
//
// "handle entry" means: if there is a corresponding exit -> create a cap
// but if there isn't, save it in the entry pointer for the cleanup at the end

// 1. point on the cutting plane
if (fabs(curdot) < epsilon){

  if (prevdot > epsilon){
    if (nextdot > epsilon){
      // a) let it stay in the list
    }
    else {
      // b) let it stay in the list, but set as exit point
      list<cPoint>::iterator onplane = p;
      onplane--; // p is actually next point
      punmatchedexit = &(*onplane);
    }
  }
  else {
    if (nextdot > epsilon){
      // c) let it stay in the list, and handle entry point
      list<cPoint>::iterator onplane = p;
      onplane--;
      HandleEntryPoint((*onplane), planepoint,
                  punmatchedexit, punmatchedentry, acceptedfacets);
    }
    else {
      // d) remove it from the list
      list<cPoint>::iterator remove = p; // p is actually the next point
      remove--;
      points.erase(remove);
    }
  }
}
// 2. point in positive space
else if (curdot > 0.0){
  numpos++; // one positive point is enough to justify keeping the facet

  if (nextdot < -epsilon){ // there is a change in sign -> intersection point
    // calculate intersection point and insert it
    list<cPoint>::iterator inserted = CalcAndInsertIntersection(cur, next,
                                                    curdot, planenormal,
                                                    p, points);

    // now the intersection should be considered the current point
    curdot = 0.0;
    cur = *inserted;

    // set it as an exit point
    punmatchedexit = &(*inserted); // hope this is kosher
```

```
    }
  }
  // 3. point in negative space
  else {
    numneg++;

    list<cPoint>::iterator remove = p;
    remove--; // since p is really the next point

    // check to see if we need to insert a point before removing it
    if (nextdot > epsilon){
      // calculate intersection point and insert it
      list<cPoint>::iterator inserted = CalcAndInsertIntersection(cur, next,
                                                    curdot, planenormal,
                                                    p, points);

      // now the intersection should be considered the current point
      curdot = 0.0;
      cur = *inserted;

      // this is an entry point so...
      HandleEntryPoint((*inserted), planepoint,
                  punmatchedexit, punmatchedentry, acceptedfacets);
    }

    // remove the negative point
    points.erase(remove);
  }

  // increment the values for reuse
  prevdot = curdot;
  curdot = nextdot;
  cur = next;
} // end point loop

// since last point is same as first, but was never cur with the
// chance to be removed, check now that the loop is finished
// if we need to remove it, the 2 cases are
// 1) if it is below the plane
// 2) if it is on the plane, and its prev point was not positive
if ((curdot < -epsilon) ||
  ((fabs(curdot) < epsilon) && (prevdot < epsilon)))
{
  points.pop_back();
}

// if facet meets requirements, add it
if (numpos > 0){
  cBorderSection facet;
  facet.SetBorder(points);

  acceptedfacets.push_back(facet);
}

// remember what kind of facet this was for future runs
if (numpos == numpoints){
  (*flagiter) = POSITIVE; // the whole facet was included
} else if (numneg == numpoints){
  (*flagiter) = NEGATIVE; // the whole facet was excluded
} else {
  (*flagiter) = INTERSECT; // the facet must have been cut
}
```

```
      // may need to perform some clean up tasks related to cap facets
      if ((punmatchedexit != NULL) && (punmatchedentry != NULL)){
        // This case may occur if an entry was found first and the
        // corresponding exit lied on the other side of the beginning
        // point.  Create a cap facet from these two points
        HandleEntryPoint(*punmatchedentry, planepoint,
                    punmatchedexit, punmatchedentry, acceptedfacets);
      }
      else if (fabs(nextdot) < epsilon){
        // if the last (first also) point was in the plane, it may not
        // have been correctly classified as an exit/entry during the loop
        // because of the discontinuity at the beginning/end.  So if there
        // is either an unmatched exit or entry, create a cap from the points
        if (punmatchedentry != NULL){
          punmatchedexit = &next;
          HandleEntryPoint(*punmatchedentry, planepoint,
                      punmatchedexit, punmatchedentry, acceptedfacets);
        }
        else if (punmatchedexit != NULL){
          HandleEntryPoint(next, planepoint,
                      punmatchedexit, punmatchedentry, acceptedfacets);
        }
      }// end cleanup of special cases

    }

    f++;
    flagiter++;
  }// end facet loop

  // create a new visible component that represents the liquid
  cVisibleComponent fuelviscomp;
  toreturn.SetVisibleComponent(acceptedfacets);

  return 0;
}


// helper function for SlicePolygonGeometry
void cFuelTank::HandleEntryPoint(cPoint& entry,
                 cPoint planepoint,
                 cPoint*& punmatchedexit,
                 cPoint*& punmatchedentry,
                 list<cBorderSection>& acceptedfacets) const
{
  // There are two possible cases here.  punmatchedexit can exist or not exist
  // If an exit exists, we need to create a cap facet.  The order of points
  // will be {planepoint, entry, exit} to keep outward normal orientation
  // which is vital for mass props calculation.  The other case is that
  // punmatched exit is NULL.  This means that we found an entry first and that
  // the corresponding exit is on the other side of the beginning point, so
  // we hold onto this entry in punmatchedentry until the last exit is found
  if (punmatchedexit != NULL){
    cBorderSection facet;
    list<cPoint> border;
    border.push_back(planepoint);
    border.push_back(entry);
    border.push_back(*punmatchedexit);
    border.push_back(planepoint);
    // not going to set normals, since this doesn't get rendered

    facet.SetBorder(border);
```

```
    acceptedfacets.push_back(facet);

    punmatchedexit = NULL;  // reset it, no need to delete
  }
  else {
    punmatchedentry = &entry;
  }
}

// helper function for SlicePolygonGeometry
list<cPoint>::iterator cFuelTank::CalcAndInsertIntersection(cPoint cur,
                          cPoint next,
                          double curdot,
                          cPoint planenormal,
                          list<cPoint>::iterator& p,
                          list<cPoint>& points) const
{
  cPoint cur2next = next - cur; // vector from cur to next
  double cur2nextdot = cur2next.Dot(planenormal);
  cPoint intersection;
  if (fabs(cur2nextdot) > epsilon){
    // the intersection point is located along the vector between cur and next
    // and the scaling of that vector is the ratio of the point cur projected
    // onto the normal to the vector cur2next projected onto the normal
    intersection = cur + cur2next*fabs(curdot/cur2nextdot);
  }
  else {// degenerate case
    intersection = cur;
  }

  return points.insert(p, intersection);
}


////////////////////////////////////////////////////////////////
// Author:  John Ohanian
// Date:    6/18/2003
// Purpose: Setup and check validity of inputs to CalcFuelMassProperties
//              and SampleFuelMassProperties
// Input:   viscomp - cVisibleComponent being analyzed
//          x, y, z - position
//          xrotdeg, yrotdeg, zrotdeg - rotations
//          theadeg - The orientation of the vehicle (Theta)
//          origfacets (return value) - the facets which will be used
//              in the two functions mentioned above
//          transform (return value) - the transform that puts the
//              comp's goemetry into global space
//          localfullmassprops (return value) - mass props if comp was solid
//              in local space (transform is not applied to it)
//          zmin & zmax (return values) - lower & upper extents of
//              facets in global z direction
//
// Output:  0 if succeeded, negative if error
////////////////////////////////////////////////////////////////
int cFuelTank::PrepFuelMassProperties(double thetadeg,
                          double densityslugdivincub,
                          list<cBorderSection>& origfacets,
                          cTransformationMatrix& transform,
                          cMassProperties& localfullmassprops,
                          double& zmin,
                          double& zmax) const
{
```

```
  // total volume
  localfullmassprops =
       CalcMassPropsFromGeometry(VisCompInLocalSpace,
                                 densityslugdivincub);

  // check for problem cases
  if (localfullmassprops.GetMass() < epsilon){
    cout << "GetMass epsilon" << endl;
    return -1;
  }


  // need to transform all points into global space
  cTransformationMatrix aircrafttoglobal; // premultiply -> aircraft to global

  // incorporate theta and 90deg difference b/w aircraft and global (gravity)
  aircrafttoglobal.SetRotation(0.0, (90-thetadeg)*degtorad, 0.0);

  // transform that puts points in global space
  transform = aircrafttoglobal * LocalToAircraftTransformation;

  // now tranform the points
  cVisibleComponent viscomp = VisCompInLocalSpace;
  viscomp.PointTransformation(transform);

  // list of original Facets coming from the visible component, don't modify
  origfacets = viscomp.GetVisibleComponent();
  list<cBorderSection>::iterator f = origfacets.begin();

  // need to determine extents of the facets in the z direction
  zmax = -10e10; // start inverted so box fits tightly around
  zmin = 10e10; // points, if started at zero could get bad result

  while (f != origfacets.end()){
    list<cPoint> border = (*f).GetBorder();
    list<cPoint>::iterator p = border.begin();
    while (p != border.end()){
      double z = (*p).GetZ();
      if (z > zmax){
        zmax = z;
      } else if (z < zmin){
        zmin = z;
      }
      p++;
    }
    f++;
  }
  double zextent = zmax - zmin; // size of geometry in z direction
  if (zextent <= 0.0){
    return -1;
  }

  // everything is setup and ok
  return 0;
}


////////////////////////////////////////////////////////////////
// Author:  John Ohanian
// Date:    7/10/2003
// Purpose: Generates an interpolation map of fuel mass props
//              by varying theta and fuel weight independently
// Input:
```

```
//          none
// Pre:     Need to have set VisCompInLocalSpace, LocalToAircraftTransformation
//            FuelDensitySlugdivIncub, NumberOfThetaIncrements,
//            NumberOfFuelLevels
//
// Output:  0 if succeeded, negative if error
////////////////////////////////////////////////////////////////
int cFuelTank::GenerateFuelMassPropertiesMap()
{
  if ((NumberOfThetaIncrements<2) || (NumberOfFuelLevels<2)){
    return -1;
  }

  // clear the dirty bit
  bDirty = false;

  // clear the Map
  UnitycWeightCGsByThetaAndFuelLevelMap.clear();
  UnityFullMassPropertiesByTheta.clear();

  // cycle through the thetas, get vectors of cMassProperties at different fuel levels
  int i=0;
  while (i<NumberOfThetaIncrements){
    // NumberOfThetaIncrements include 0 and 180
    double thetadeg = (180.0 / (NumberOfThetaIncrements-1) ) * i;

    vector<cWeightCG> tempweightcgs;
    cMassProperties tempinertia;
    int error = SampleFuelMassProperties(thetadeg,
                                NumberOfFuelLevels,
                                tempweightcgs,
                                tempinertia);
    if (error!=0){
      cout<< "Error during SampleFuelMassProperties!"<<endl;
      return error;
    }

    // these will always be the same size
    UnitycWeightCGsByThetaAndFuelLevelMap.push_back(tempweightcgs);
    UnityFullMassPropertiesByTheta.push_back(tempinertia);

    i++;
  }

  if (UnitycWeightCGsByThetaAndFuelLevelMap[0].size() == 0) return -1;

  // set MaxFuelWeightLb now that we've finished the map
  MaxFuelWeightLb = UnityFullMassPropertiesByTheta[0].GetMass()*FuelDensitySlugdivIncub;// Unity Mass
(really weight) needs to be multplied by density

  // update max capacity
  UpdateMaxCapacityLb();

  return 0;
}

////////////////////////////////////////////////////////////////
// Author:  John Ohanian
// Date:    7/10/2003
// Purpose: Interpolate mass props from the generated map
// Input:   thetadeg - the attitude of the aircraft
//          fuelweightlb - the current weight of fuel in tank
//          toreturn (return value) - the interpolated mass props
```

```
// Output:  0 if succeeded,
//          -1 if error
//          1 if desired fuel weight exeeds MaxFuelWeightLb
//
// Pre:     It's a good idea to call Validate before this
////////////////////////////////////////////////////////////////
int cFuelTank::CalculateFuelMassPropertiesForThetaAndWeight(double desiredthetadeg,
                                          double desiredfuelweightlb,
                                          cMassProperties& toreturn) const
{
  toreturn.SetMass(desiredfuelweightlb);

  int thetasize = UnitycWeightCGsByThetaAndFuelLevelMap.size();
  int weightsize = (thetasize>0)?
    UnitycWeightCGsByThetaAndFuelLevelMap[0].size()
    : 0;

  // detect degenerate cases
  if ((thetasize<2) || (weightsize<2)){
    cout<<"Degenerate fuel map size! theta:"<<thetasize<<", weight:"<<weightsize<<endl;
    return -1;
  }
  if (desiredfuelweightlb > MaxFuelWeightLb){
    cout<<"desired fuel weight is higher than MaxFuelWeightLb"<<endl;
    desiredfuelweightlb = MaxFuelWeightLb;
  }

  double incrementtheta = 180.0/(thetasize-1);
  // determine bounding theta columns and values
  int indexthetalow = (int)(desiredthetadeg/incrementtheta);
  double thetalow = indexthetalow*incrementtheta;
  int indexthetahigh = indexthetalow + 1;
  double thetahigh = thetalow + incrementtheta;

  // calculate interpolation factor for theta
  // so that desired value = low value + factor*high value
  double interptheta = (desiredthetadeg - thetalow)/(thetahigh - thetalow);

  // prepare variables for CG interpolation
  double weightlow, weighthigh;
  weightlow = weighthigh = 0.0;
  double x[2]={0.0,0.0};
  double y[2]={0.0,0.0};
  double z[2]={0.0,0.0};

  int n=0;
  while (n<2){
    const cWeightCG* plow = NULL;
    const cWeightCG* phigh = NULL;
    const vector<cWeightCG> & column =
      UnitycWeightCGsByThetaAndFuelLevelMap[indexthetalow + n];
    unsigned int i=0;
    while (i<column.size()){
      double weight = (column[i].WeightLb)*FuelDensitySlugdivIncub;
      if (weight < desiredfuelweightlb){
    plow = &(column[i]);
    weightlow = weight;
      } else if (weight >= desiredfuelweightlb){
    phigh = &(column[i]);
    weighthigh = weight;
    break; // should have plow by now since starts empty
      }
```

```
    i++;
  }
  if ((plow==NULL) || (phigh==NULL)){
    cout<<"Couldn't find bounding weights for fuel interpolation!"<<endl;
    return -1;
  }

  double interpweight = (desiredfuelweightlb - weightlow)/(weighthigh-weightlow);

  x[n] = plow->X + interpweight*(phigh->X - plow->X);
  y[n] = plow->Y + interpweight*(phigh->Y - plow->Y);
  z[n] = plow->Z + interpweight*(phigh->Z - plow->Z);

  n++;
}


// finally interpolate CG
toreturn.SetXoff(x[0] + interptheta*(x[1] - x[0]));
toreturn.SetYoff(y[0] + interptheta*(y[1] - y[0]));
toreturn.SetZoff(z[0] + interptheta*(z[1] - z[0]));

// assumption here that full tank mass props is good
// for partially full tanks is only valid between 10% and 100%
// check for 0% to 10% case here and set linear interpolation factor
double lowfuelfactor = 1.0;
double fullweight = UnityFullMassPropertiesByTheta[0].GetMass()*FuelDensitySlugdivIncub;
if (desiredfuelweightlb/32.2 < 0.1*fullweight){
  lowfuelfactor = desiredfuelweightlb/(0.1*fullweight);
}

// interpolate MOI
cMassProperties fulllow = UnityFullMassPropertiesByTheta[indexthetalow];
cMassProperties fullhigh = UnityFullMassPropertiesByTheta[indexthetahigh];
toreturn.SetIxx((fulllow.GetIxx()
           + interptheta*(fullhigh.GetIxx() - fulllow.GetIxx()))
          * lowfuelfactor
          * FuelDensitySlugdivIncub);// multiply by density since was unity
toreturn.SetIyy((fulllow.GetIyy()
           + interptheta*(fullhigh.GetIyy() - fulllow.GetIyy()))
          * lowfuelfactor
          * FuelDensitySlugdivIncub);
toreturn.SetIzz((fulllow.GetIzz()
           + interptheta*(fullhigh.GetIzz() - fulllow.GetIzz()))
          * lowfuelfactor
          * FuelDensitySlugdivIncub);
toreturn.SetIxy((fulllow.GetIxy()
           + interptheta*(fullhigh.GetIxy() - fulllow.GetIxy()))
          * lowfuelfactor
          * FuelDensitySlugdivIncub);
toreturn.SetIyz((fulllow.GetIyz()
           + interptheta*(fullhigh.GetIyz() - fulllow.GetIyz()))
          * lowfuelfactor
          * FuelDensitySlugdivIncub);
toreturn.SetIxz((fulllow.GetIxz()
           + interptheta*(fullhigh.GetIxz() - fulllow.GetIxz()))
          * lowfuelfactor
          * FuelDensitySlugdivIncub);

toreturn.SetDensity(FuelDensitySlugdivIncub);

// success!
return 0;
```

```
}


////////////////////////////////////////////////////////////////
// Author:  John Ohanian
// Date:    6/19/2003
// Purpose: Evaluate the mass props of the fluid at equally spaced heights
//              which can be used to create a curve fit mapping
// Input:   viscomp - cVisibleComponent being analyzed
//          x, y, z - position
//          xrotdeg, yrotdeg, zrotdeg - rotations
//          theadeg - The orientation of the vehicle (Theta)
//          numfuellevels - how many equally spaced planes to use
//          weightcgstoreturn (return value) - vector of weight&cg of fluid
//              at the parallel levels, starting from empty and going to full
//              (currently approximated as solid ->moi is inaccurate)
//          fullinertiatoreturn - full tank's cMassProperties
//
// Output:  0 if succeeded, negative if error
////////////////////////////////////////////////////////////////
int cFuelTank::SampleFuelMassProperties(double thetadeg,
                           int numfuellevels,
                           vector<cWeightCG>& weightcgstoreturn,
                           cMassProperties& fulllocalinertiatoreturn)
{
  weightcgstoreturn.clear();

  list<cBorderSection> origfacets; // the list of facets (geometry)
  cTransformationMatrix localtoglobal; // transforms geometry into global space
  double zmin; // minimum z value in global space
  double zmax; // maximum ""

  // setup everything
  int error = PrepFuelMassProperties(thetadeg,
                             1.0, // unity density for the map
                             origfacets, // returned in aircraft
                             localtoglobal,
                             fulllocalinertiatoreturn, // in local space
                             zmin,
                             zmax);

  // convert this to fluid
  fulllocalinertiatoreturn
    = ConvertSolidToFluidMassProperties(fulllocalinertiatoreturn,
                               AspectXY,
                               AspectYZ,
                               AspectZX);

  if (error != 0){
    cout << "Error in location 1" << endl;
    return error;
  }
  cTransformationMatrix localtoglobalinverse = localtoglobal.Inverse();

  // first entry is always empty, no need to transform, since all 0's
  cWeightCG empty;
  weightcgstoreturn.push_back(empty);

  // full tank
  cWeightCG full;
  full.WeightLb = fulllocalinertiatoreturn.GetMass();
  full.X = fulllocalinertiatoreturn.GetXoff();
  full.Y = fulllocalinertiatoreturn.GetYoff();
```

```
full.Z = fulllocalinertiatoreturn.GetZoff();

// trivial case
if (numfuellevels < 3){
  weightcgstoreturn.push_back(full);
  return 0;
}

// get size of facet list
int numfacets = origfacets.size();
// keep corresponding list of ints to say what facets were used in last run
list<int> facetflags;
int i=0;
while(i<numfacets){
  facetflags.push_back(NEGATIVE);
  i++;
}

// create definition of cutting plane (a point on the plane and a vector normal
cPoint localcg(fulllocalinertiatoreturn.GetXoff(),
          fulllocalinertiatoreturn.GetYoff(),
          fulllocalinertiatoreturn.GetZoff());
cPoint globalcg = localtoglobal * localcg; // global cg x,y will be used to keep origin point for
                                           // cap facets close to center of part
cPoint planepoint(globalcg.GetX(), globalcg.GetY(), 0.0); // z will be set below
cPoint planenormal(0.0,0.0,-1.0);  // negative because we want to get points below

double offset = (zmax - zmin)/(numfuellevels-1);

// cycle through the liquid levels and record in array
int j=0;
while(j<(numfuellevels-2)){

  // determine and set liquid level
  double znew = zmin + (j+1)*offset;
  planepoint.SetZ(znew);

  // now perform slice and mass props
  cVisibleComponent temp;
  int error = SlicePolygonGeometry(origfacets,
                           planepoint,
                           planenormal,
                           facetflags,
                           true, // always advancing
                           temp);
  if (error != 0){
    cout << "Error in location 2" << endl;
    return error;
  }
  bool bexcludemoments = true;  //moments are taken from full tank
  cMassProperties massprops =
      CalcMassPropsFromGeometry(temp,1.0,bexcludemoments);

  // transform and add to list
  massprops.Transformation(localtoglobalinverse);

  cWeightCG tempweightcg;
  tempweightcg.WeightLb = massprops.GetMass();
  tempweightcg.X = massprops.GetXoff();
  tempweightcg.Y = massprops.GetYoff();
  tempweightcg.Z = massprops.GetZoff();
  weightcgstoreturn.push_back(tempweightcg);
```

```
    j++;
  }

  // add full cWeightCG to end of list
  weightcgstoreturn.push_back(full);

  return 0;
}


////////////////////////////////////////////////////////////////
// Author:  John Ohanian
// Date:    8/1/2003
// Purpose: Validate the fuel tank, updating if anything is dirty
// Input:   none
//
// Output:  bool - true if success, false if failed
////////////////////////////////////////////////////////////////
bool cFuelTank::Validate()
{
  if ((bUpdateMapOnGeometryOrTransformationChange==true) && (bDirty==true)){

    bDirty = false; // we're in the process of cleaning

    if ((NumberOfThetaIncrements <= 0) || (NumberOfFuelLevels <= 0))
      return true; //nothing to update

    // this function resets the dirty bit
    int error = GenerateFuelMassPropertiesMap();
    if (error != 0){
      return false;
    }
  }

  return true;
}

////////////////////////////////////////////////////////////////
// Author:  John Ohanian
// Date:    8/7/2003
// Purpose: Convert a mass properties for a solid into the mass
//              properties of a fluid of the same shape
//              Only affects the moments of inertia
// Input:   cMassProperties solid - the solid mass properties
//          aspectxy - aspect ratio of the geometry in xy plane
//          aspectyz - aspect ratio of the geometry in yz plane
//          aspectzx - aspect ratio of the geometry in zx plane
//          ^all aspect ratios should be greater than 1
//
// Output:  the fluid mass properties
////////////////////////////////////////////////////////////////
cMassProperties cFuelTank::ConvertSolidToFluidMassProperties(cMassProperties solid,
                                  double aspectxy,
                                  double aspectyz,
                                  double aspectzx) const
{
  if (aspectxy < 1){
    aspectxy = (aspectxy <= 0.0)? 1.0 : 1.0/aspectxy;
  }
  if (aspectyz < 1){
    aspectyz = (aspectyz <= 0.0)? 1.0 : 1.0/aspectyz;
  }
  if (aspectzx < 1){
```

```
    aspectzx = (aspectzx <= 0.0)? 1.0 : 1.0/aspectzx;
  }

  // according to Lamb, the fluid/solid ratio is related...
  // Ifluid/Isolid = ((a^2 - 1)/(a^2 + 1))^2 , where a is the aspect
  // ratio of the tank in the plane perpendicular to the rotation axis

  // start with solid
  cMassProperties toreturn = solid;

  // modify the MOI
  double factorxy = (aspectxy*aspectxy - 1)/(aspectxy*aspectxy + 1);
  double factoryz = (aspectyz*aspectyz - 1)/(aspectyz*aspectyz + 1);
  double factorzx = (aspectzx*aspectzx - 1)/(aspectzx*aspectzx + 1);
  toreturn.SetIxx(solid.GetIxx()*factoryz);
  toreturn.SetIyy(solid.GetIyy()*factorzx);
  toreturn.SetIzz(solid.GetIzz()*factorxy);
  toreturn.SetIxy(solid.GetIxy()*factorxy);
  toreturn.SetIyz(solid.GetIyz()*factoryz);
  toreturn.SetIxz(solid.GetIxz()*factorzx);

  return toreturn;
}


//////////////////////////////////////////////////////////////
// Author:  John Ohanian
// Date:    9/25/2003
// Purpose: Recalculate the max capacity
// Input:   none
//
// Pre:     MaxFuelWeightLb, UllageFraction, and
//          InternalStructureFraction have been set
// Post:    The MaxCapacityLb is up to date
//////////////////////////////////////////////////////////////
void cFuelTank::UpdateMaxCapacityLb()
{
  // update max capacity
  double fractionfull = 1.0;
  if (UllageFraction > 0.0){
    fractionfull -= UllageFraction;
  }
  if (InternalStructureFraction > 0.0){
    fractionfull -= InternalStructureFraction;
  }
  MaxCapacityLb = MaxFuelWeightLb * fractionfull;
}
```

Appendix B: Analytical Solution to Cylindrical Tank CG

Governing equations for CG:

$$\bar{x} = \frac{1}{M}\int x\, dm$$

$$\bar{y} = \frac{1}{M}\int y\, dm$$

$$\bar{z} = \frac{1}{M}\int z\, dm$$

where M is the total mass, and dm is a differential mass.

Need to change variables since tank is rotated by theta:

$$x = -z'\sin\theta + x'\cos\theta$$

$$y = y'$$

$$z = z'\cos\theta + x'\sin\theta$$

$$J = \begin{vmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{vmatrix} = \cos^2\theta + \sin^2\theta = 1$$

which yields:

$$\bar{x} = \frac{1}{M}\int \left(-z'\sin\theta + x'\cos\theta\right)\rho dx'\, dy'\, dz'$$

$$\bar{y} = \frac{1}{M}\int y'\,\rho dx'\, dy'\, dz'$$

$$\bar{z} = \frac{1}{M}\int \left(z'\cos\theta + x'\sin\theta\right)\rho dx'\, dy'\, dz'$$

where ρ is the density of the fuel.

We will use limits of integration:

$x : from - R\ to + R$

$y : from - \sqrt{R^2 - x^2}\ to \sqrt{R^2 - x^2}$

$z : from - L\ to - L + h - x\tan\theta$

where R is the radius of the cylinder, L is the length, and h is the height of the fluid (distance from the bottom cap to where the free surface intersects centerline of cylinder, which is also the x' axis). $-L + h - x\tan\theta$ is the equation of the free surface of the fuel.

The weight of fuel in the tank is given by

$W_f = g\rho V$

where g is the gravitational constant and V is the total volume. The volume can also be expressed as:

$V = \pi R^2 h$

which yields the relationship:

$$h = \frac{W_f}{g\rho\pi R^2}$$

evaluating the x equation with the new limits of integration:

$$\bar{x} = \frac{\rho}{M}\int_{-R}^{R}\int_{-\sqrt{R^2-x^2}}^{-\sqrt{R^2-x^2}}\int_{-L}^{-L+h-x\tan\theta}\left(-z'\sin\theta + x'\cos\theta\right)dx'\,dy'\,dz'$$

$$= \frac{\rho}{M}\int_{-R}^{R}\int_{-\sqrt{R^2-x^2}}^{-\sqrt{R^2-x^2}}\left[-\frac{1}{2}z'^2\sin\theta + z'x'\cos\theta\right]\Bigg|_{-L}^{-L+h-x\tan\theta}dx'\,dy'$$

$$= \frac{\rho}{M}\int_{-R}^{R}\int_{-\sqrt{R^2-x^2}}^{-\sqrt{R^2-x^2}}\left[\left(Lh - \frac{h^2}{2}\right)\sin\theta + \left((h-L)\tan\theta\sin\theta + h\cos\theta\right)x + \left(\frac{-\tan^2\theta\sin\theta}{2} - \tan\theta\cos\theta\right)x^2\right]dx'\,dy'$$

Now to simplify, let the quantities multiplying powers of x (starting with x^0) be A, B, C and convert to polar coordinates.

$$x = r\cos\phi,\ \ y = r\sin\phi$$

$$\bar{x} = \frac{\rho}{M}\int_0^R\int_0^{2\pi} Ar + Br^2\cos\phi + Cr^3\cos^2\phi\, d\phi dr$$

$$= \frac{\rho}{M}\int_0^R \left(rA\phi + Br^2\sin\phi + Cr^3\left(\frac{1}{2}\phi + \frac{1}{4}\sin 2\phi\right)\right)\Bigg|_0^{2\pi} dr$$

$$= \frac{\rho}{M}\int_0^R \left(2\pi rA + \pi Cr^3\right)dr = \frac{\rho}{M}\left(\pi AR^2 + \frac{\pi C}{4}R^4\right)$$

Substituting back in for A, C, and h yields the final result:

$$\underline{\bar{x} = \sin\theta\left(L - \frac{W_f}{2g\rho\pi R^2} + \frac{g\rho\pi R^4}{W_f}\left(\frac{-\tan^2\theta}{2} - 1\right)\right)}$$

This result matches calculations by Mathematica.

A similar process was used for the y and z coordinates of the CG, yielding equations:

$$\underline{\bar{y} = 0}$$

$$\underline{\bar{z} = \cos\theta\left(-L + \frac{W_f}{2g\rho\pi R^2}\right) + \frac{g\rho\pi R^4}{4W_f}\left(-\frac{1}{2}\tan\theta\sin\theta\right)}$$

which also check with Mathematica.

The quatities calculated by these equations are in the global space (x,y,z, not x',y',z'). To transform the results into local coordinates the following transformation may be used:

$$x' = z\sin\theta + x\cos\theta$$
$$y' = y$$
$$z' = z\cos\theta - x\sin\theta$$

Vita

John Ohanian is currently pursuing a Ph.D. at Virginia Polytechnic Institute and State University. His main focus of research is on the use, evaluation, and optimization of hybrid propulsion systems in VTOL UAV's. Other areas of interest include stability and control of VTOL UAV's, conceptual aircraft design, computer-aided geometric design, software development, and computational fluid dynamics.

However, John's academic pursuits are only one aspect of his life. First and foremost, he tries in everything that he does to glorify his Lord and Savior, Jesus Christ (but fails more often than not). The second highest priority in his life is to love his beautiful wife, Robin. There is also a new family member due May 9th 2004 that John is tremendously excited to meet. John's family, friends, and church come next in his life, followed by school and work.

In his free time John enjoys playing basketball, volleyball, chess, and spending time outdoors hiking and camping. He is also a sucker for puzzles (can't put one down until he's figured it out). He also enjoys spending time with his wife watching movies, dancing, and reading books.

In the future, John plans to become a university professor or possibly work in industry. In the long term, he and his wife would like to go overseas as short-term missionaries.