Pioneering Economical Space Propulsion

New Mexico
Supercomputing Challenge
Final Report
April 05, 2006

Team 81
Sandia Preparatory School

Team Members
Garrett Lewis
Zachary Rosenberg

Teacher
Neil McBeth

Mentor
Louis Friedman

THIS PAGE INTENTIONALLY LEFT BLANK

II. Table of Contents

III. **Executive Summary**

For decades mankind has strived to reach beyond the confines of Earth; over the past half-century, space travel has evolved from an overzealous idealistic fantasy to a hard reality. Yet in that time, the cost of exploration has remained astronomical due, in large part, to the price of propulsion. A number of systems have been proposed to remedy this problem, including chemical, solar, nuclear, anti-matter, and many other systems, but high entry costs have largely prevented mainstream entrance into new fields.

This project was designed to provide a computational model in C++ that demonstrates estimated costs of a hypothetical mission to Pluto using systems of the four propulsion methods described above. Data output from the developed program was then interpreted by two methods, both broad and specific, to determine the most effective system for the mission throughout the coming century, assuming typical research efforts.

The project was selected largely due to rising interest in antimatter systems, as well as the recent return-to-flight of the Space Shuttle and ever-extending exploration of the outer solar system.

IV. **Background**

1. Introduction

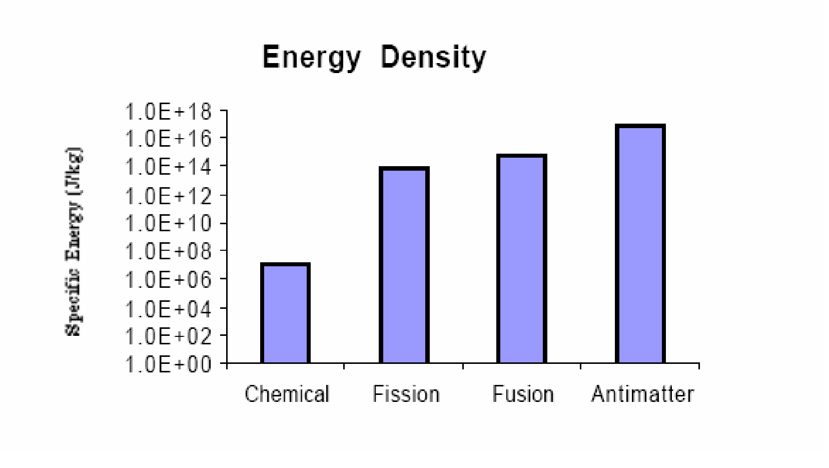
Through the past 47 years, since Sputnik was launched by the Soviet Union, the primary method of space propulsion has been through the use of chemical propellants. This method, however, is not necessarily the most efficient means, both because of the massive cost of towing the large amounts of fuel needed along with the payload and the relatively low efficiency of the energy produced from the fuel mass and resources put into the system. A number of recent projects, though, have given rise the possibility of implementation of alternative methods of propulsion in space flight.

The space shuttle is a recognizable example of chemical a propulsion system. Unfortunately, within this system, approximately 10% of the total mass of a craft is composed of the fuel used to propel it, making the cost to send one kilogram of payload into space about $10,000, creating astronomical final expenses. Recent developments of hybrid solid rockets have allowed for more control of the burning of the fuel, and a corresponding increase in efficiency, but, nonetheless, the problems persist.

Recently, the Planetary Society made an attempt to propel a craft, Cosmos I, using solar sails. Solar sails utilize the momentum of photons to accelerate through space and hypothetically reach nearly the speed of light if given enough time at sufficient proximity to a light source. However, despite the high efficiency of solar sail propulsion, the cost of the sails themselves are enormously high as a result of the need for low areal densities, only a few atoms thick and yet capable of withstanding innumerable micro meteor impacts while supporting surface areas of several hundred square kilometers. Another form of space-going sail has been proposed in the form of the so-called Magsail, which utilizes charged particles on the solar wind to propel the craft, but the much greater efficiency of the former type makes it the target of this project.

Space Propulsion 100 (SP-100), a futuristic Russian craft that utilized nuclear power as means of propulsion, has also spurred examination of new technologies with the high mass conversion ratios achieved within the propellant. Yet, nuclear propulsion raises environmental concerns over the necessity to cultivate radioactive materials on earth and potentially produces temperatures beyond those with which researchers are currently able to work with, reducing the desirability of such methods. Nonetheless, of the methods herein examined, nuclear systems such as that of Deep Space I, a recently-launched American vessel, are receiving the greatest following.

A rising star, antimatter-annihilation propulsion provides by far the most efficient means of space propulsion. Antimatter, as the name implies, is essentially the exact opposite of matter, of which the universe is almost universally composed at the current time. It has the same mass as matter, but the exact opposite charge and spin. As an example, a proton with a positive charge of one electron Volt will counter an antiproton with a negative charge of one electron Volt. When such particles collide, they instantaneously annihilate one another producing, by Einstein's Theory of General Relativity, massive amounts of pure energy.



Comparison of energy produced by four methods. [17]

In the split second after the annihilation of two such particles, numerous subatomic particles, including pions (π), are created.

For approximately 18 attoseconds, these pions carry a mass at approximately 97% of the speed of light and thus make an effective means to propel a craft. Additionally and ultimately, a strong series of gamma-rays is emitted which has no direct use as propellant but can be used to heat a secondary system and produce thrust by other means as well as provide power to craft subsystems. Although the production efficiency of antimatter is currently too small for practical use, the predicted exponential growth of the technology of production and storage provides hope that this one-day might be a viable option for propulsion systems.

V. Model

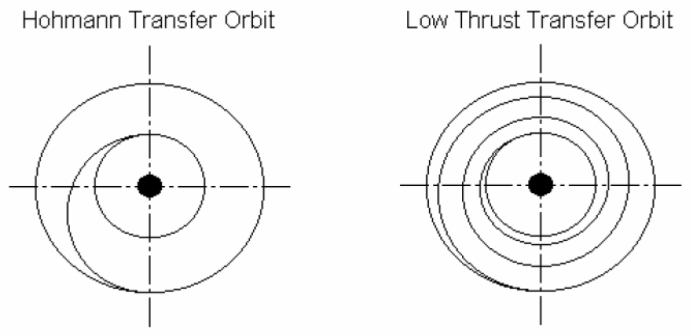
1. Purpose

Through the use of a computational model, this project has been designed to give an accurate series of predictions leading to the cost of an explorative mission in space for each of four propulsion devices. These devices individually represent the four general categories of propulsion by Chemical, Nuclear, Solar, and Antimatter Annihilation systems, with the precise mechanisms being defined by both practicality and appropriateness to the specific mission defined.

Through this model, the ultimate potential for each method to be used on a massive scale in future space exploration may be determined and, through extrapolation, allow for more effective distribution of resources to develop advanced systems for any proposed mission in the field.

2. Mission

For this model, a wholly hypothetical mission was devised to send a space probe toward the orbit of Pluto. This ultra light probe, weighing only 10 kg, is to be used for unspecified scientific research at the final orbit where, presumably, the planet lies. A

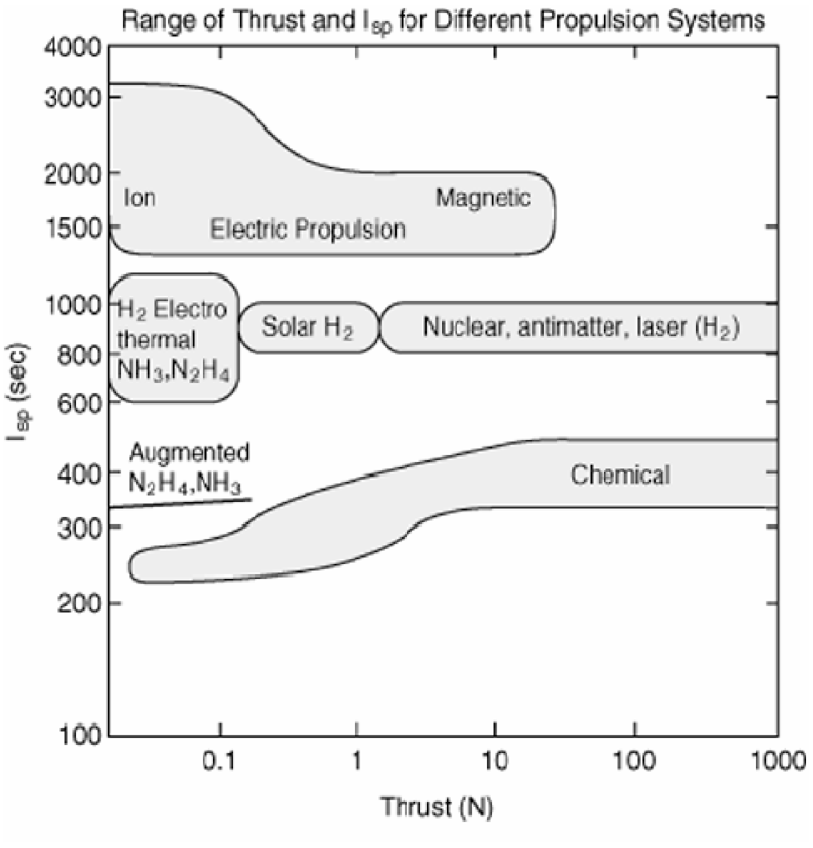


Two possible transfer orbits [24]

more complex system of modeling the planets' positions and gravitational fields was discarded from the model for ease of code construction and calculation within the orbits, rather, each method of propulsion was held solely responsible for producing the force necessary to establish a transfer ellipse from earth's orbit to that of Pluto. Likewise, the potential of gravity assist was not implemented in the mission design.

Without these interferences, each system was provided with an equal footing from which to make the transfer, reducing the primary target of the mission to a simple change in velocity. Thus, the sole remaining concern appears with the cost of achieving the thrust needed to complete the mission.

The selection of Pluto as the mission target was derived from the recent push to expand human knowledge of the outer reaches of the solar system by means of such missions as Deep Space I and New Horizons, which was launched shortly after this project. As we learn more and more about the inner solar system and the nature of other stars beyond our sun, there remains a large void in out knowledge from the areas immediately beyond the great Gas Giants of Jupiter, Saturn, Uranus, and Neptune. Here, beyond



Ranges of specific impulse and thrust for various systems [15] the reach of manned flight or real-time communication, the light of the sun shines so feebly that it would barely be differentiated from another star, and yet, these areas of Pluto, the Kuiper Belt, and the Oort Cloud are possibly the best places to find primordial samples of the materials that formed the Earth. For those reasons, it is becoming ever more important to explore these regions and develop the systems to enact this exploration.

To complete this mission, the probe will carry a propulsion system based on one of four published concepts—either a standard chemically propelled rocket device using multistage boosters, a fission-based nuclear electric propulsion (NEPP) system, immense solar sails, or a mechanism modeled for antimatter-initiated microfusion (AIM). Each of these systems seems to provide the most appropriate concept within the fundamental form that it represents for this mission, as will be described by more in-depth means in Section V.3.

3. Systems Utilized

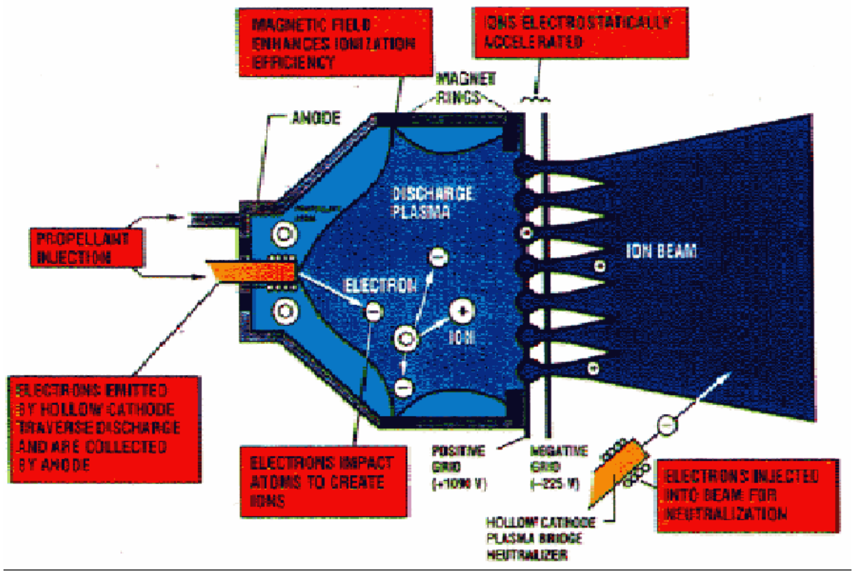
A. Chemical: Star 48 Rocket Motor

The chemical propulsion system used in the model is based largely on that used by the New Horizons mission launched in January, 2006 and destined for Pluto itself. This centers on the Star 48A rocket motor that has been implemented as a Payload Assistant Module on the third stage of a Boeing Delta Launch Vehicle (PAM-Delta). This device is used directly with the primary launch vehicle (LV) and thus does not require a separation in Earth orbit that is necessary for each of the other systems. Nonetheless, the decreased mass found by separation provides sufficient incentive to enact the process before the final burn is initiated.

As with all rocket motors, the Star class uses solid chemical propellants at maximum capacities of slightly over two Metric Tonnes (MT). This allows for controlled burnout over the duration of the flight, although this computational model does not initially have such capabilities. However, the extra control is countered by a slight loss of efficiency in the mass conversion ratio, which pulls down the total thrust available for the mission.

Ultimately, however, this is one of the best available systems on which to model the chemical device for this project. Thus, this system has been implemented according to specifications for the immediate years and adapted according to the research parameters for chemical systems to determine the efficacy of the system both today and in later years relative to the other systems.[28]

B. Nuclear: Nuclear-Electric Ion Thruster


MAGNETIC FIELD ENHANCES IONIZATION EFFICIENCY
IONS ELECTROSTATICALLY ACCELERATED
MAGNET RINGS
ANODE
DISCHARGE PLASMA
PROPELLANT INJECTION
ELECTRON
ION
ION BEAM
ELECTRONS EMITTED BY HOLLOW CATHODE TRAVERSE DISCHARGE AND ARE COLLECTED BY ANODE
ELECTRONS IMPACT ATOMS TO CREATE IONS
POSITIVE GRID (+1090 V)
NEGATIVE GRID (−225 V)
HOLLOW CATHODE PLASMA BRIDGE NEUTRALIZER
ELECTRONS INJECTED INTO BEAM FOR NEUTRALIZATION

Sample ion propulsion system [15]

Due to the immense heat and radioactivity that creates correspondingly high stresses on direct nuclear propulsion systems such as those that heat a gaseous or plasma propellant and those that use detonations of miniature warheads for thrust, an indirect nuclear-electric system was chosen to provide the basis of the nuclear propulsion in the model. More specifically, an ion accelerator described in [23] was selected for its near-term applicability and high Specific Impulse (Isp), a measure of the relative thrust produced by the mechanism.

This device uses energy acquired through the decay series of Uranium-235 to produce electricity by means similar to those implemented at earth-based power reactors, merely on a smaller scale. This electricity is then run through a series of solenoids to produce magnetic fields and accelerate ions as a typical rocket would accelerate heated combustion products.

On a limiting note, the relatively miniscule capacity of the thruster leads to very slight thrust at lower power levels. The only possibility to remedy this is a straightforward increase in the power that is achieved by increasing the fission rate of the Uranium and producing the heat to run the generator.

The most difficult portion of developing these systems, however, remains short lifetimes of power production units (PPUs) due to those temperatures described above, and it is these difficulties that continue to hold back advances in the field of nuclear propulsion, keeping the cost above that of chemical devices. The model seeks to find a remedy to this in research over the course of the century without compromising the accuracy of the analysis through a slightly varied research function relative to the other units. Further, environmental activism places the field on shaky ground that provides little room for advancement in the face of both political and economic pressure.[23]

C. **Solar: Solar Sail**

The solar-powered propulsion system that was modeled is the relatively simple concept of a solar sail, a device that utilizes the pressure of photons produced by a light-emitting object to accelerate through space. The light source may be anything, but the very slight order at which the light transfers momentum to the sail necessitates that the source be either the nearest star, in this case the sun, or a fantastic laser beyond the current capabilities of humans. As the latter is seemingly impractical in its requirements for size, precision, and power, the sole viable power source is the sun's light.

The photon pressure should not be confused with the solar wind, an incessant outward stream of ions, which, even under optimal usage by a magnetic sail, produce forces several orders of magnitude below those of the light reaching the sail.

Seemingly perfect due to the lack of need for a fuel and infinite Isp, solar sails are restrained by the immense sail areas and miniature areal densities that must be devised to create pressure sufficient to propel the craft. While areas must reach kilometers on a side for many missions, they must be only a few nanograms thick to prevent the build-up of insurmountable masses that must be accelerated. Further, the sail must stand up to the beatings of various space debris and retain functional sails throughout the mission to achieve the full orbital radius of the target.[19]

D. Antimatter: Antimatter-Initiated Microfusion

The ultimate mechanism for any device requiring energy, antimatter annihilation ultimately provides a system with pure energy. In the case of electron-positron interaction, this is in the form

$$\begin{array}{ll} & 2\pi^{0} \rightarrow 2(\gamma+\gamma) \\ \bar{p}+p\rightarrow & 1.5\pi^{+}\rightarrow 1.5(\mu^{+}+\upsilon_{\mu}) \\ & 1.5\pi^{-}\rightarrow 1.5(\mu^{-}+\bar{\upsilon}_{\mu}) \end{array}$$

$$\Downarrow$$

$$\begin{pmatrix} \mu^{+}\rightarrow \mathrm{e}^{+}+\bar{\upsilon}_{\mu}+\upsilon_{e} \\ \mu^{-}\rightarrow \mathrm{e}^{-}+\bar{\upsilon}_{e}+\upsilon_{\mu} \end{pmatrix} \Rightarrow \mathrm{e}^{+}+\mathrm{e}^{-}+\text{neutrinos} \rightarrow \gamma+\gamma+\text{neutrinos}$$

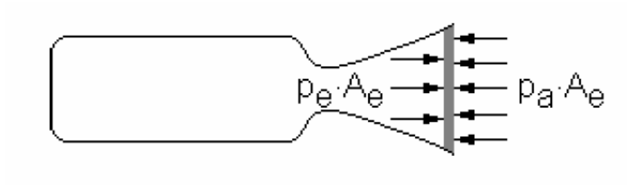
Transformation of antiproton and proton from matter to energy [17] of a high-intensity gamma-ray, while in the case of proton-anti-proton interaction there is a series of subatomic particles produced, including neutrinos and pions, with the latter being charged particles moving at approximately 97% of the speed of light that may be directed as any ion to produce thrust.

As in nuclear systems, however, this means of direct propulsion is a poor use of the energy derived of the reaction. Rather, a much more efficient use of the products of the reaction is found in Antimatter-Initiated Microfusion. This system uses the destruction of antimatter to catalyze fusion in a small pellet containing a few small nuclei, in this case Deuterium-Helium-3 combined with a solution of Uranium-238 that provides not only the high energy desired from an extended series of fusion reactions, but a bonus in the decay of the Uranium to heat a propellant and accelerate out of a typical nozzle while the charged byproducts of the nuclear reactions are electrically accelerated out of the craft in a system similar to that of the nuclear model.

The almost imperceptible volume of antimatter needed to complete a standard mission may be stored relatively easily in an advanced version of the Penning Trap, developed at Pennsylvania State University once it is produced and provide thrust orders of magnitude greater than the other methods while the volume remains orders lower.

The drawback for antimatter is the almost nonexistent production capacity for antimatter that leads to production efficiencies of very nearly zero for the immediate timeframe that create inversely high prices for even the minute amounts needed for a space mission. The only hope to make antimatter a competitive system is that it will follow in the path of liquid Oxygen and increase exponentially in production efficiency for an extended period of time.[17]

4. General Mathematics



Rocket diagram for equation [24]

Underlying each of the systems modeled herein except the solar sails, which serve as a special case due to their unique method of obtaining thrust, is the thrust equation that is derived of the pressure gradient in any rocket and, in a modified state, in electrical acceleration systems. The basic equation states:

$$F = \dot{m}_e V_e + (p_e A_e - p_0 A_e) \qquad [28]$$

Where F is force, m is mass, V is volume, p is pressure, and A is area of the exit to the reaction chamber. This equation governs almost all motion of spacecraft at some level, regardless of specific parameters, although many systems are better described by more specific parameters, such as the rocket equation, which is named non-creatively for the objects it describes:

$$\Delta v = g_0 I_{sp} \ln[1 + \frac{m_{propellant}}{m_{final}}] \qquad [28]$$

Here, the change in velocity is equated against the gravitational acceleration at Earth's surface, g, the Isp, and the natural log of the mass of the propellant relative to the empty mass of the system. By solving this equation, several steps are skipped that would otherwise slow the computation using only the thrust equation used above. This is particularly influential in this model, which relies heavily on the delta-v.

In the cases of the nuclear and antimatter systems modeled, where the propellant was accelerated to much greater speeds than in the chemical systems, this equation was transformed into a relativistic form:

$$\frac{\Delta v}{c} = \frac{\left(\frac{m_{initial}}{m_{final}}\right)^{\frac{2v_e}{c}} - 1}{\left(\frac{m_{initial}}{m_{final}}\right)^{\frac{2v_e}{c}} + 1} \qquad [17]$$

Here, the exhaust velocity is reintroduced along with the speed of light that define how the propellant's mass increases with increasing energy and thus how the craft will accelerate. If, though, the exhaust velocity or Isp and delta-v are known, the equations above may be reorganized to solve for the required propellant mass to be implemented in the cost function, as was the case with the three devices that fit these equation.

Of course, there was further calculation required to determine the delta-v and, particularly, the exhaust velocity. In the case of both the nuclear and chemical systems, this was an easy process due to the familiarity that the scientific community has built with each mechanism over the decades that has allowed for extensive simplification of advanced systems of equations to only a few short bullets.

Antimatter systems, though, are much less researched and therefore much less known. This results in the introduction of numerous abstract parameters used to define portions of the propulsion systems and cost functions that are not fully understood. More precise descriptions and definitions for this device are available in section X.1, as well as [16] and [22].

On a completely different system of equations, solar sails are much more direct in the standards that govern their motion. They rely on simple momentum transfer from the photons to the sail, and yet, their orbital motion quickly turns this straightforward analysis into a convoluted, extensive chain of solutions that only lead back to the same parameter later in the system. Ultimately, however, the system may be simplified to a small system for demonstration:

$$p = \frac{4.56E-6(1+R)}{r_{AU}^{2}} \qquad [12]$$

$$F = pA \qquad [14]$$

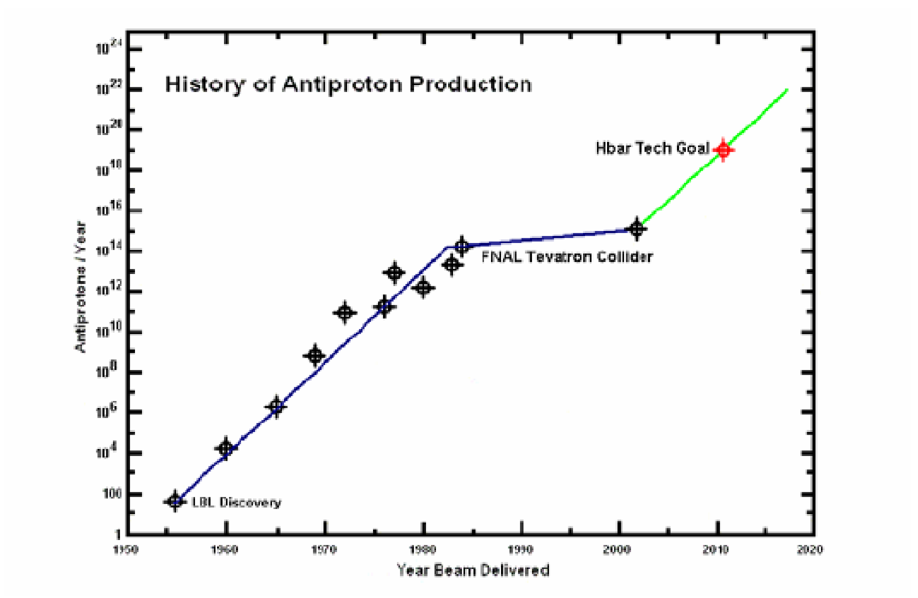
$$\beta = \frac{2Fr_{AU}^{2}}{m} \qquad [14]$$

$$\ddot{r} = r\dot{\theta}^{2} - \frac{\mu_{solar}}{r^{2}} + \frac{\beta\cos^{3}\alpha_{r}}{r^{2}} \qquad [14]$$

$$\ddot{\theta} = \frac{-2\dot{r}\dot{\theta}}{r} + \frac{\beta\cos^{2}\alpha_{r}\sin\alpha_{r}}{r^{3}} \qquad [14]$$

These five equations define the planar motion of the sail with p as pressure, R as reflectivity, and r as the radius in the first equation. The second is simply force as a product of pressure and area, but the third becomes slightly more obscure with β referring to characteristic acceleration and m to mass of the entire craft. The fourth and fifth equations use the polar coordinates P(r, θ) with the angle of the sail relative to the radial vector and μ as the heliocentric gravitational constant. Using variations on this system and basic Newtonian physics, the position of the sail may be easily determined.

The scientific and engineering advances that allow the model to enact variations in the price of various aspects of the mission for each system are produced as an exponential function of time remaining to launch, and while the precise values of each function vary throughout the model to account for variations in the properties of the parameters across functions as well as across files. The equations all involved some form of the basic equation graphed below with *k* on the vertical axis:

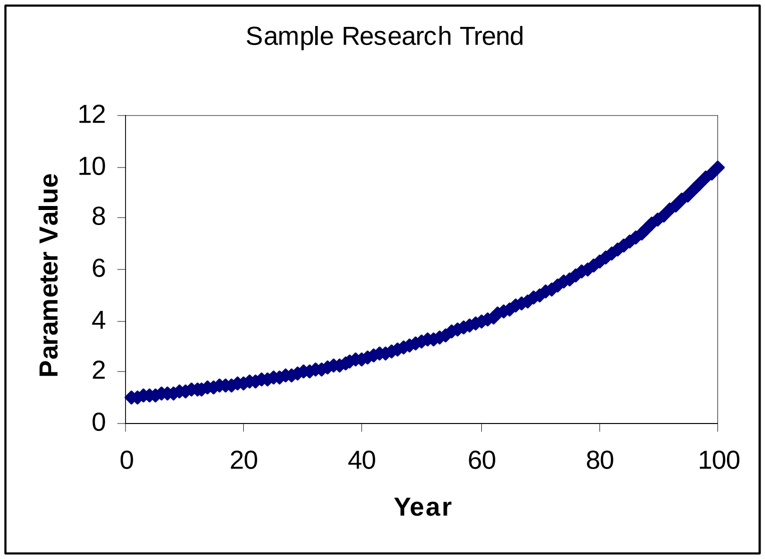


Antimatter production and furute predictions[13]

$$k = 10^{\frac{time*time}{time*100}}$$

]

Graph of the fundamental equation defining research



Thus, it can be seen that the mathematics within the model provide a strong foundation for the model that would be difficult to achieve on a manual level, especially in the case of the necessarily iterative solar sailing model. When calculating for the entire 100-year period over which the model runs, it becomes apparent at what level the mathematics is simplified through the use of pion4.exe.

5. Coding Process

Using Microsoft Visual C++ .NET as a platform, the code was assembled using a combination of standard C++ libraries and functions written specifically for the task at hand. To accommodate these newly written functions in fields ranging from data pertaining to the Hohmann transfer orbit to the pre-launch mission costs, a series of classes has been built to interact in a primary file, *pion4.cpp* (X.9), and ultimately, return the estimated price of each propulsion system.

This class structure is key to the flow of the code for several reasons, not the least of which is the organization it provides. By separating related functions, such as those related to antimatter and those pertaining to orbital mechanics, a short list of partial systems could easily be monitored and ordered within the primary functions without the clutter that would be associated with building the 2,500 lines in one file. Effectively, the classes allowed the solver to simply solve one case and then move on to the next while using only one small file at a time, freeing memory and making debugging easier. Rather than jumping through the equivalent of over 50 pages of text and back again, he had merely to track a steady descent through no more than approximately five pages and typically less than one.

The structure also provided ease of handling during stages of partial completion. In such cases, the programmer was able to retain relatively sloppy, half-completed portions of the code in separate areas while testing finished parts. Thus, there was no final build that resulted in vast numbers of errors, but only small, easily-remedied problems.

These classes also contained limited hierarchy that allowed them to inherit some fundamental functions in certain cases, particularly in the relationship between the *orbit* (X.7) and *hohmann* (X.5) classes.

Once the code produced functional output to the console (Primitive version of Method Two, X.9, described below), a process of reorganizing was initiated in a new project folder to expand the functionality of the model and provide usable data. This was ultimately achieved by developing a new central function (Method One, X.9, also described below) that implements the original model for each of 100 years and providing the user with a function to choose either the old or the new when first opening the program.

Method Two provided the initial functionality of the model by allowing the user to define an interval of time available prior to launch and implementing the selection into a *_tmain* function that called the appropriate class functions for each of the systems and solved for the total mission cost. During this process, it simultaneously output the data achieved at every step to the console to be perused by the user and permit easy identification of any sources of error in development. With the advent of Method One at later stages, however, this console output was expanded to a file stream that dropped into a *.doc file (Sample at XI.2) that could be resaved for future reference, allowing easier tracking of data.

If, though, the user should choose to follow Method One when prompted, the program begins an automatic process of iterating through the systems of equations to solve for the price of each propulsion means over the coming century. Herein, the user is provided with a **.xls* file (XI.1) that gives a table of each final value that may be once again resaved and analyzed. This method is particularly useful for the trend determinations that the project sought.

The code remains in an evolutionary state, awaiting further adjustments to improve the program as described in VII.2.

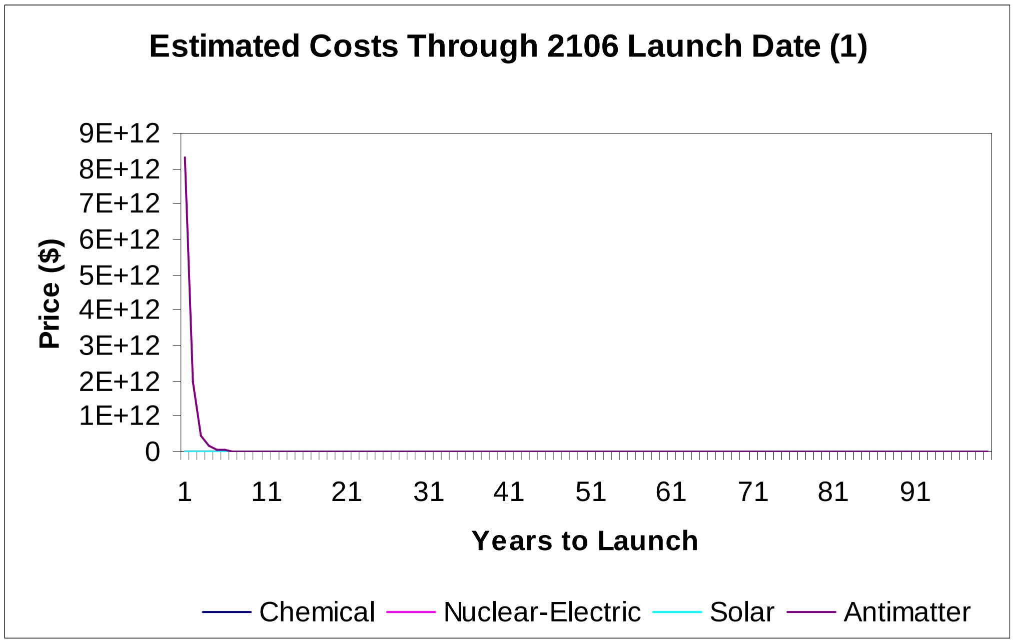
VI. Results

1. Graphs

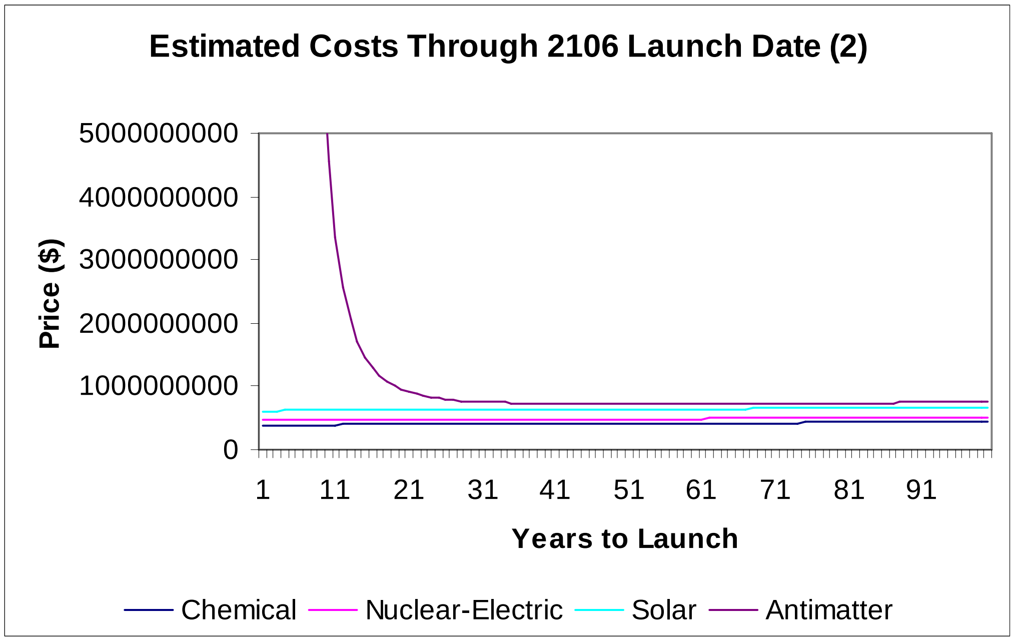
While Method Two provided only individual data points, the output of Method One was easily transposed into a series of graphs to demonstrate the estimated trends of the cost functions for the four systems analyzed. These graphs give four views of the trends from the very broad in the case of the first to a heavily magnified version in graph four, with intermediate levels between. This is achieved through manipulation of the scale of the vertical axis; the horizontal remains at the same level throughout, delineating the 100-year period to 2106.

The graphs will be further analyzed in Section 2.

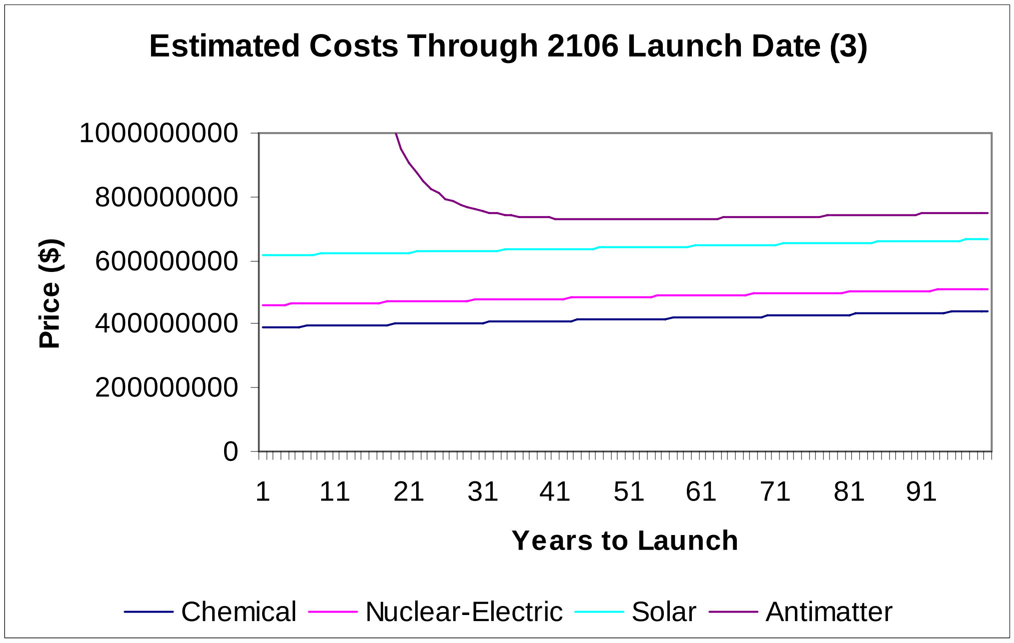
Graph 1:



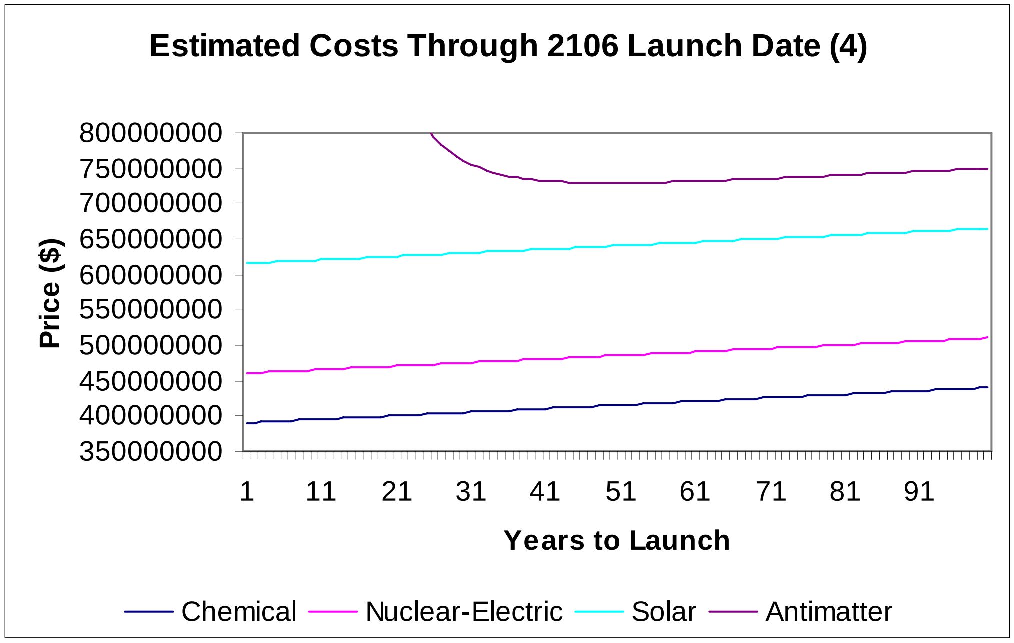
Graph 2:



Graph 3:



Graph 4:



2. General Observations

The data, both from Method I (Graphs above and Chart XI.1) and Method Two (Chart XI.2), indicated that among near-term missions, antimatter propulsion systems are wholly impractical with cost estimates not falling below US$1 billion until the 20th year. This presents costs over the immediate era that are several orders of magnitude greater than those of every other system. The three remaining concepts give cost estimates for the defined mission that move nearly in unison, with solar sails requiring approximately US$150 thousand in excess of the NEPP, which in turn presents prices of approximately US$75 thousand beyond those of a chemically propelled craft.

At 50 years, the AIM system reaches its lowest estimated cost of approximately US$729.3 million. From that point, it joins the uniform rise in required funding, remaining around US$90 million above the solar system. It does, however, show very slight trend toward the cost of the others.

The individual data points devised by the program to generate the costs for the mission indicate a heavy skewing of responsibility toward the cost of human operations rather than the fuels and propulsion devices. The sole exception to this is the noted in the price of antimatter for the first 50 years. At that point, the falling price changes fail to significantly affect the total mission cost, reflecting on the heavy impact of labor pricing. Further, this labor impact led to a direct increase in overall price for each system.

VII. Conclusions

1. General Conclusions

For near-term applications, the data provided by the program largely supports the estimates set forth by previous research for the systems. [5][17][23][28] Beyond that, the trends rise roughly in accordance with expectations at a general level. Although it is difficult to determine long-term costs, the trends do follow rough precedents set forth by science since the Renaissance.

The cost functions of the three higher systems were expected to ultimately converge and intersect with the chemical model, but they instead, with the exception of antimatter, follow this standard. Rather, they remained at a roughly constant level above the cost of a chemical thruster device throughout the 100 years modeled. This discrepancy, though, is not of such magnitude that the data may be seen as inconclusive. While nuclear and solar mechanisms greatly reduce fuel consumption in orbit, there is very little to restrain the costs of manpower and launch systems that are required to implant the craft in an initial orbit about the sun. When this is added to the price of research and development, the effects of advances in the latter are largely wiped out among the three less-expensive devices.

The AIM system modeled herein, however, demonstrates drastic change through the first half-century and a continuing, albeit less noticeable, approach toward the costs of the other systems beyond that. Instead of the nearly linear growth demonstrated in the lower three systems after manpower's addition, it retains an inversely exponential trend altered only slightly by the pricing pressures in the latter half of the century. Thus, although the model indicates that antimatter systems are not an effective pursuit for this mission, there are indications that missions of a sufficiently later date or increased payload may provide a base for antimatter systems.

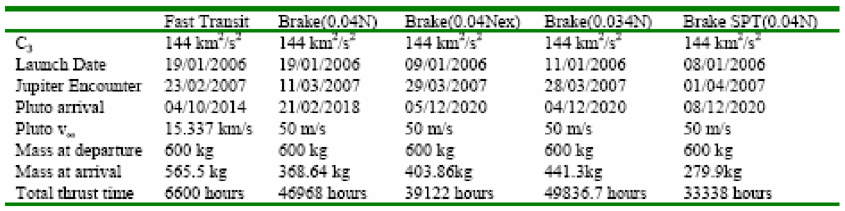
The increased payload necessitated for, perchance, a return mission or a manned mission would greatly increase the volume of fuel required for all systems and thus increase the effects of improved technology in any of these mechanisms. This could potentially pull these functions into sufficiently new paths to make any of the three more-advanced systems more desirable than a chemical version that is most desirable in this analysis.

2. Continuing Work and Recommendations

Although functional and effective as a means of demonstrating the potential of antimatter as a propulsion method, this model does not fully incorporate potential variations in the prices of labor or launch mechanisms. Nor does it provide the options of expanded changes in orbital velocity for farther-reaching missions or expanded payloads for return voyages, including potential manned exploration. Further work on the program over the coming months will attempt to a least partially remedy some of these shortcomings.

Primary consideration in this respect will go to the orbital and payload questions, as they seem to hold the greatest potential to significantly affect the systems beyond what the current model indicates. This will be achieved by a means similar to Method One that iterates over the period in question to determine the gradient produced by these attributes to a mission. Ideally, Calculus methods will be implemented on the output to further determine the affects of these mechanisms.

There also remains the potential of an expanded time frame undergoing analysis to attempt to better pinpoint any point of intersection experienced by the cost functions of the



	Fast Transit	Brake(0.04N)	Brake(0.04Nex)	Brake(0.034N)	Brake SPT(0.04N)
C_3	144 km^2/s^2	144 km^2/s^2	144 km^2/s^2	144 km^2/s^2	144 km^2/s^2
Launch Date	19/01/2006	19/01/2006	09/01/2006	11/01/2006	08/01/2006
Jupiter Encounter	23/02/2007	11/03/2007	29/03/2007	28/03/2007	01/04/2007
Pluto arrival	04/10/2014	21/02/2018	05/12/2020	04/12/2020	08/12/2020
Pluto v_∞	15.337 km/s	50 m/s	50 m/s	50 m/s	50 m/s
Mass at departure	600 kg	600 kg	600 kg	600 kg	600 kg
Mass at arrival	565.5 kg	368.64 kg	403.86kg	441.3kg	279.9kg
Total thrust time	6600 hours	46968 hours	39122 hours	49836.7 hours	33338 hours

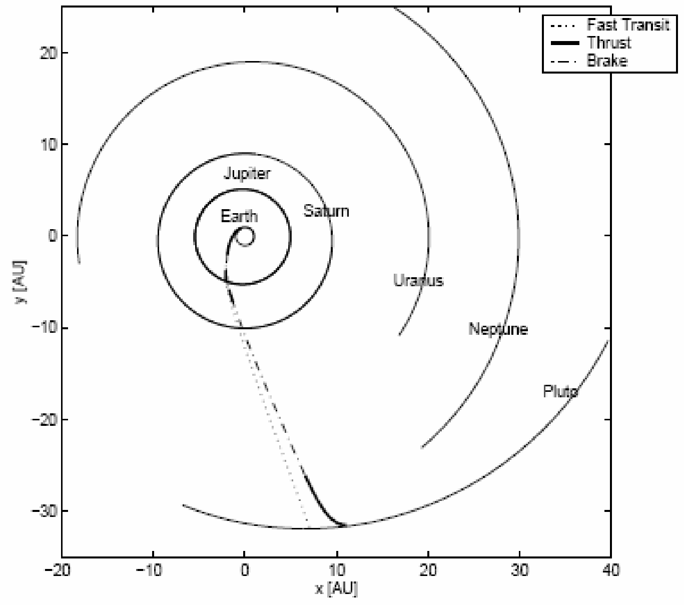


Figure 10.Earth-Jupiter-Pluto Option in 2006 with electric propulsion

Sample trajectory for a gravity assist mission [27]

mechanisms in question. If implemented, this will require only simple adjustments to some constants in the code.

A secondary goal of continued work entails the development of an expanded Graphical User Interface (GUI) for easier operation of the program by those less familiar with its development. Currently, the user must have a level of understanding of the program achieved by reading the section of this report on the project's development (Section V) to easily implement and interpret the program. It is hoped that extended work will allow for adaptation of this issue.

VIII. **Acknowledgements**

We would like to extend our sincere thanks to the faculty and staff of Sandia Preparatory School for their support and intrigue that has enabled us to compete in the Adventures in Supercomputing Challenge. In particular we would like to thank them for their financial support in transportation as well as access to and use of computer equipment.

We would also like to extend our gratitude and appreciation to our sponsor and advisor Neil McBeth who has helped us through our difficulties and supported us throughout the duration of the project. His devotion and commitment to helping us participate despite any personal problems has been greatly appreciated and without him our project would not have been completed.

Additionally, we extend our deepest thanks to Louis Friedman and John Suding for their advice that provided us with the essential information needed to complete our model.

IX. **References**

[1] 109th Congress of the United States of America. *National Aeronautics and Space Administration Authorization Act of 2005*. S.1281, January, 2005.

[2] Borowski, Stanley K. "Comparison of Fusion/Antiproton Propulsion Systems for Interplanetary Travel", AIAA-87-1814, July, 1987.

[3] Braeunig, Robert A. "Orbital Mechanics", Rocket and Space Technology. http://www.braeunig.us/space/orbmech.htm, December, 2005.

[4] Chamberlain, Sally, et al. "Project Longshot: A Mission to Alpha Centauri", NASW-4435.

[5] Diedrich, Benjamin, Charles Garner, and Manfred Leipold. "A Summary of Solar Sail Technology Developments and Proposed Demonstration Missions", JPC-99-2697, 1999.

[6] Forward, Robert L. "Antiproton Annihilation Propulsion", AFAL TR-87-070, 1997.

[7] Frisbee, Robert H. "Solar Sails For Mars Cargo Mission", AIAA-01-2076, 2001.

[8] Frisbee, Robert H. "Advanced Propulsion for the XXIst Century", AIAA-2003-2589, July 2003.

[9] Frisbee, Robert H. "Systems-Level Modeling of a Beam-Core Matter-Antimatter Annihilation Propulsion System", Jet Propulsion Laboratory.

[10] Frisbee, Robert H. and Stephanie D. Leifer. "Evaluation of Propulsion Options for Interstellar Missions", AIAA-98-3403, July 15, 1998.

[11] Halliday, David and Robert Resnick. *Fundamentals of Physics*. USA: John Wiley & Sons, Inc., 1974.

[12] Hollerman, William Andrew. “The Physics of Solar Sails”, For NASA Faculty Fellowship Program, 2002.

[13] Howe, Stephen D. and Gerald P. Jackson. “Antimatter Driven Sail for Deep Space Exploration”, For Hbar Technologies, LLC, January 2004.

[14] Kim, Mischa. “Continuous Low-Thrust Trajectory Optimization: Techniques and Applications”, Dissertation to the Faculty of Virginia Polytechnic Institute, 2005.

[15] Komerath, Narayanan. “AE6450 Lecture #13: Electric Propulsion”, For Lecture, 2004.

[16] LaPointe, Michael R. “Antiproton Powered Propulsion with Magnetically Confined Plasma Engines”, AIAA-89-2334, August 1989.

[17] McMahon, Patrick B. “Antimatter Initiated Microfission/fusion (AIM) Space Propulsion”, For NEEP 602 Nuclear Power in Space, May 2000.

[18] Moche, Dinah L. *Astronomy*. USA: John Wiley & Sons, Inc., 2000.

[19] Montgomery, Edward E., Gregory P. Garbe and Andrew Heaton. “Places Only Sails Can Go”, NASA Marshall Space Flight Center, 2002.

[20] Munem, M. A. and J. P. Yizze. *Precalculus Functions and Graphs*. New York: Worth Publishers, 1990.

[21] Redheffer, R. M. and I. S. Sokolnikoff. Mathematics of Physics and Modern Engineering. USA: McGraw-Hill, Inc, 1966.

[22] Schmidt, G.P., H.P. Gerrish, J.J. Martin, G.A. Smith, and K.J. Meyer. "Antimatter Production for Near-term Propulsion Applications", NASA Marshall Space Flight Center, 1998.

[23] Smith, Bryan K. "Definition, Expansion and Screening of Architecture for Planetary Exploration Class Nuclear Electric Propulsion and Power Systems", For Master of Science in Engineering and Management, February, 2003.

[24] Tajmar, Martin. "Advanced Space Propulsion Systems", For Vienna University of Technology, 2003.

[25] Templeman, Julian and Andy Olsen. *Microsoft Visual C++ .NET*. Washington: Microsoft Press, 2003.

[26] Thomas, George B. and Ross L. Finney. *Calculus and Analytic Geometry*. USA: Addison Wesley, 2003.

[27] Vasile, Massimiliano, Robin Biesbroek, Leopold Summerer, Andres Galvex, and Gerhard Kminek. "Options For a Mission to Pluto and Beyond", AAS 03-210, February, 2003.

[28] Whitmore, Stephen A. "MAE 6530—Propulsion Systems", Utah State University College of Engineering. http://www.engineering.usu.edu/classes/mae/6530/propulsion_systems/prop.html, January, 2006.

[29] Williams, Craig H., Leonard A. Dudzinsky, Stanley K. Borowski and Albert J. Juhasz. "Realizing '2001: A Space Odyssey': Piloted Spherical Torus Nuclear Fusion Propulsion", AIAA-2001-3805, March 2005.

X. ***Appendix B***: C++ Code

1. *antimatter.cpp*

```
//////////////////////////////////////////////////////////////
//Filename:
//antimatter.cpp
//////////////////////////////////////////////////////////////


#include "stdafx.h"
#include "antimatter.h"
#using <mscorlib.dll>
using namespace System;

//This file provides the background support for antimatter.h.
//As an early file, there remain numerous production notes and a few blotted
equations.

//antimatter cost       from apfntpa
double antimatter::eta_anti (double t)    //efficiency   max=0.5
{
      double eta_anti = (4/Math::Pow(10, 8)) + ((Math::Pow(3,
(t*t*t*t*t/(t*10000000)))-1)/10);
      //double eta_anti = Eout_anti / Ein_anti;
      return eta_anti;
}

double antimatter::Ein_anti (double c, double eta_anti, double m_anti)
{
      double Ein_anti = c * m_anti / eta_anti;
      return Ein_anti;
}

double antimatter::E_cost_anti (double cost_grid, double m_anti, double c,
double eta_anti)
{
      double E_cost_anti = cost_grid * m_anti * c*c / eta_anti;
      return E_cost_anti;
}

//antimatter requirements            "
double antimatter::m_anti (double beta_anti, double gamma_anti, double
eta_exhaust_anti, double relative_m_anti, double lamda_anti, double m_pay)
{
	double m_anti = ((gamma_anti-1)*(relative_m_anti-1)*m_pay) /
(2*(1+beta_anti)*(gamma_anti+eta_exhaust_anti-1)*(1+lamda_anti-
relative_m_anti*lamda_anti));
	//double m_anti = (1 / (2*(1+beta_anti))) * ((gamma_anti-1) /
(gamma_anti+(eta_exhaust_anti-1))) * ((relative_m_anti-1) / (1+lamda_anti-
relative_m_anti*lamda_anti)) * m_pay;
	return m_anti;
}

double antimatter::relative_m_anti (double delta_v, double c, double
v_exhaust)
{
	c = 299792458;
	double relative_m_anti = Math::Pow((1+(delta_v/c)) / (1-(delta_v/c)),
(c/(2*v_exhaust)));
	return relative_m_anti;
}

double antimatter::lamda_anti (double m_struct, double m_prop)
{
	double lamda_anti = m_struct / m_prop;
	return lamda_anti;
}

double antimatter::gamma_anti (double v_exhaust, double c)
{
	c = 299792458;
	double gamma_anti = 1 / (Math::Sqrt((1-
((v_exhaust/c)*(v_exhaust/c)))));
	return gamma_anti;
}

double antimatter::m_prop (double beta_anti, double gamma_anti, double
eta_exhaust_anti)
{
	double m_prop =
	(2*1.672621718*Math::Pow(10,27)*(1+beta_anti)*(eta_exhaust_anti+gamma_a
nti-1))/(gamma_anti-1);
	return m_prop;
}

double antimatter::m_dry (double relative_m_anti, double m_prop)
{
	double m_dry = m_prop / (relative_m_anti-1);
	return m_dry;
}

double antimatter::m_struct (double m_dry)
{
	double m_struct = 0.9 * m_dry;
	return m_struct;
}

//antimatter physics for pulsed p-H rocket (system concept)	from
appwmce
double antimatter::B_min (double n_dense_anti, double T_anti, double
n_dense_H, double T_H)
```

```
{
	double B_min = Math::Sqrt(8*Math::PI * (n_dense_anti*T_anti +
n_dense_H*T_H));
	return B_min;
}

double antimatter::Prob_remain (double R_mirror)
{
	double Prob_remain = Math::Sqrt(((R_mirror-1) / R_mirror));
	return Prob_remain;
}

double antimatter::R_mirror (double B_min, double B_max)
{
	double R_mirror = B_max / B_min;
	return R_mirror;
}

double antimatter::eta_E (double E_ion, double E_electron, double n_dense_H,
double n_dense_anti)
{
	double eta_E = (E_ion+E_electron)*n_dense_H / (1877*n_dense_anti);
	return eta_E;
}

// double T_exhaust (double E_ion, double amu_anti);
double antimatter::flow_prop (double amu_H, double n_dense_H, double
V_chamber, double t_pulse)
{
	double flow_prop = amu_H * n_dense_H * V_chamber / t_pulse;
	return flow_prop;
}

double antimatter::v_exhaust (double E_ion)
{
	E_ion = 938.27231;	//MeV
	double v_exhaust = 1.4*Math::Pow(10, 6) * Math::Sqrt(E_ion);
	return v_exhaust;
}

double antimatter::Isp_anti (double v_exhaust)
{
	double Isp_anti = v_exhaust / 980;
	return Isp_anti;
}

double antimatter::Thrust (double flow_prop, double v_exhaust)
{
	double Thrust = flow_prop * v_exhaust;
	return Thrust;
}


//End of file.


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

2. *AssemblyInfo.cpp*

```
//////////////////////////////////////////////////////////////
//Filename:
//AssemblyInfo.cpp
//////////////////////////////////////////////////////////////


#include "stdafx.h"

#using <mscorlib.dll>

using namespace System::Reflection;
using namespace System::Runtime::CompilerServices;

//
// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
//
[assembly:AssemblyTitleAttribute("")];
[assembly:AssemblyDescriptionAttribute("")];
[assembly:AssemblyConfigurationAttribute("")];
[assembly:AssemblyCompanyAttribute("")];
[assembly:AssemblyProductAttribute("")];
[assembly:AssemblyCopyrightAttribute("")];
[assembly:AssemblyTrademarkAttribute("")];
[assembly:AssemblyCultureAttribute("")];

//
// Version information for an assembly consists of the following four values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
// You can specify all the value or you can default the Revision and Build
Numbers
// by using the '*' as shown below:

[assembly:AssemblyVersionAttribute("1.0.*")];

//
// In order to sign your assembly you must specify a key to use. Refer to the
// Microsoft .NET Framework documentation for more information on assembly
signing.
//
// Use the attributes below to control which key is used for signing.
//
// Notes:
//   (*) If no key is specified, the assembly is not signed.
//   (*) KeyName refers to a key that has been installed in the Crypto
Service
```

```
//       Provider (CSP) on your machine. KeyFile refers to a file which
contains
//       a key.
//   (*) If the KeyFile and the KeyName values are both specified, the
//       following processing occurs:
//       (1) If the KeyName can be found in the CSP, that key is used.
//       (2) If the KeyName does not exist and the KeyFile does exist, the
key
//           in the KeyFile is installed into the CSP and used.
//   (*) In order to create a KeyFile, you can use the sn.exe (Strong Name)
utility.
//        When specifying the KeyFile, the location of the KeyFile should be
//        relative to the project directory.
//   (*) Delay Signing is an advanced option - see the Microsoft .NET
Framework
//       documentation for more information on this.
//
[assembly:AssemblyDelaySignAttribute(false)];
[assembly:AssemblyKeyFileAttribute("")];
[assembly:AssemblyKeyNameAttribute("")];


//End of file.


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

3. *chemical.cpp*

```
//////////////////////////////////////////////////////////////
//Filename:
//chemical.cpp
//////////////////////////////////////////////////////////////


#include "stdafx.h"
#include "chemical.h"
#using <mscorlib.dll>
using namespace System;

//This *.cpp file provides the body of the chemical methods.

double chemical::delta_T (double Ti, double Tf)
{
	double delta_T = Tf-Ti;
	return delta_T;
}

double chemical::velocity_prop (double Cp, double delta_T)
{
	double velocity_prop = Math::Sqrt(2*Cp*delta_T);
	return velocity_prop;
}

double chemical::eta_prop()
{
	double eta_prop = .8;
    return eta_prop;
}

double chemical::F_chem(double P_exit, double velocity_prop, double m_dot,
double A_noz)
{
	double F_chem = velocity_prop * m_dot + P_exit * A_noz;
	return F_chem;
}

double chemical::delta_v_chem(double g_zero, double Isp_chem, double P_ref)
{
	double delta_v_chem = g_zero * Isp_chem * Math::Log((1+P_ref),
Math::E);
	return delta_v_chem;
}

/////////////////////////////////////////////////////////////

double chemical::P_ref(double delta_v_chem, double g_zero, double Isp_chem)
{
	double P_ref = Math::Pow(Math::E, (delta_v_chem/(g_zero*Isp_chem)))-1;
	return P_ref;
}
```

```
double chemical::m_prop(double P_ref, double m_sys, double m_pay)
{
	double m_prop = P_ref*(m_sys+m_pay);
	return m_prop;
}

double chemical::cost_prop(double m_prop)
{
	double cost_prop = .44*m_prop;
	return cost_prop;
}

double chemical::cost_tot_chem(double cost_prop, double cost_sys, double
cost_mission)
{
	double cost_tot_chem = cost_prop + cost_sys + cost_mission;
	return cost_tot_chem;
}

double chemical::cost_mission(double time)
{
	double cost_mission = 300000000+500000*time;
	return cost_mission;
}


//End of file.


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

4. *cost.cpp*

```
//////////////////////////////////////////////////////////////
//Filename:
//cost.cpp
//////////////////////////////////////////////////////////////


#include "stdafx.h"
#include "orbit.h"
#include "cost.h"
#using <mscorlib.dll>
using namespace System;

//This file gives definitions for the generic cost functions.

double cost::cost_pre (double cost_mech, double cost_research, double
cost_materials, double cost_place)
{
      double cost_pre = cost_mech + cost_research + cost_materials +
cost_place;
      return cost_pre;
}

double cost::cost_post (double cost_engineer, double cost_place)
{
      double cost_post = cost_engineer + cost_place;
      return cost_post;
}

double cost::cost_mech (double t, double n_mech, double price_mech)
{
      double cost_mech = t * n_mech * price_mech;
      return cost_mech;
}

double E_cost (double cost_grid, double m_anti, double c, double eta_anti)
{
      return 0;
}

double cost::cost_tot(double cost_prop, double cost_sys, double cost_mission)
{
      double cost_tot = cost_prop + cost_sys + cost_mission;
      return cost_tot;
}

double cost::cost_mission(double time, double cost_coef)
{
      double cost_mission = 300000000*cost_coef+500000*time;
      return cost_mission;
}
```

```
//End of file.

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

5. *hohmann.cpp*

```
//////////////////////////////////////////////////////////////
//Filename:
//hohmann.cpp
//////////////////////////////////////////////////////////////


#include "stdafx.h"
#include "hohmann.h"
#include "orbit.h"
#using <mscorlib.dll>
using namespace System;

//This file defines functions for the transfer orbits.

double hohmann::vi_transfer_hoh()
{
      orbit * alpha;
      alpha = new orbit;

      double omega = alpha->sun_g(6.6726*Math::Pow(10, -11),
1.9891*Math::Pow(10, 30));
      Console::Write(S"Number density = ");
      Console::WriteLine(omega);

      double MG = omega;
      double ri = 149597870691;
      double at = (149597870691 * (1+39.53)) / 2;

      Console::Write(S"\nInitial Radius = ");
      Console::WriteLine(ri);
      Console::Write(S"\nSemi-major Axis = ");
      Console::WriteLine(at);

      double vith = Math::Sqrt (MG * ((2 / ri) - (1 / at)));
      Console::Write(S"\nInitial Velocity of Transfer Orbit = ");
      Console::WriteLine(vith);

      return vith;
}

double hohmann::vf_transfer_hoh(double MG, double rf, double at)
{
      double vfth = Math::Sqrt (MG * ((2 / rf) - (1 / at)));
      return vfth;
}

double hohmann::delta_vi_h(double ri)
{
      double vith = hohmann::vi_transfer_hoh();

      orbit * beta;
      beta = new orbit;
      double MG = beta->sun_g(6.6726*Math::Pow(10, -11), 1.9891*Math::Pow(10, 
30));
      Console::Write(S"\nMG = ");
      Console::WriteLine(MG);

      double psi = beta->vi_earth(MG, ri);
      Console::Write(S"\nInitial Velocity at Earth = ");
      Console::WriteLine(psi);
      double vie = psi;

      double dvih = vith - vie;
      Console::Write(S"\nInitial Delta Velocity = ");
      Console::WriteLine(dvih);

      return dvih;
}

double hohmann::delta_vf_h(double vfth, double vfp)
{
      double dvfh = vfp - vfth;
      return dvfh;
}


//End of file.


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

6. *nuclear.cpp*

```
//////////////////////////////////////////////////////////////
//Filename:
//nuclear.cpp
//////////////////////////////////////////////////////////////


#include "stdafx.h"
#include "nuclear.h"
#using <mscorlib.dll>
using namespace System;

//This file gives the definitions for the functions of the nuclear class.
//The first three, along with others, provide constants.
//Several production notes remain; this is one of the earlier of the final
files.

double E_U_fission = 186.5;
double E_Rb_decay = 7.86;
double E_Cs_decay = 1.89;

double nuclear::total_E_fission ()
{
      double total_E_fission = 200;
      return total_E_fission;
}

//  from opt low thrust...
double nuclear::mass_flow_func (double T_sp, double m, double g_zero,double
I_sp)
{
      double mass_flow_func = T_sp*m / (g_zero*I_sp);
      return mass_flow_func;
}

double nuclear::T_sp (double T, double m)
{
      double T_sp = T/m;
      return T_sp;
}

//  from architecture
double nuclear::alpha_sys (double P_RD)
{
      double alpha_sys = (81/19)*(1-.5*(P_RD));
      return alpha_sys;
}

//                from pg.89- architecture
double nuclear::m_init_total (double alpha_sys, double m_fuel, double m_pay)
{
      double m_init_total = 100*alpha_sys+m_fuel+m_pay;
      return m_init_total;
}

double nuclear::m_fin_total (double m_init_total, double m_fuel)
{
      double m_fin_total = m_init_total-m_fuel;
      return m_fin_total;
}

//  for cost
double nuclear::mu_dyn_nuc (double P_RD)                     //RD = research
and development
{
      double mu_dyn_nuc = .2 + .1*P_RD;
      return mu_dyn_nuc;
}

double nuclear::P_RD (double RAN, double t)
{
      double P_RD =Math::Pow(.99, (t+100)/t);                //insert constant
K
      return P_RD;
}

double nuclear::m_fuel (double delta_v, double g_zero, double I_sp)
{
      double m_fuel = (1-Math::Pow(Math::E, (0-delta_v)/(g_zero*I_sp)))/(((0-
delta_v)/(g_zero*I_sp))*Math::Pow(Math::E, (delta_v)/(g_zero*I_sp))-
(delta_v)/(g_zero*I_sp)-1);
      return m_fuel;
}

double nuclear::m_sys (double delta_v, double g_zero, double I_sp, double
m_pay)
{
      double m_sys = Math::Pow(Math::E, (0-delta_v)/(g_zero*I_sp))/(((0-
delta_v)/(g_zero*I_sp))*Math::Pow(Math::E, ((delta_v)/(g_zero*I_sp))-
(delta_v)/(g_zero*I_sp))-1);
      return m_sys;
}

double nuclear::Isp_nuc (double t)
{
      double Isp_nuc = 3100 + 700*Math::Pow(10, (t*t/(t*100)));
      return Isp_nuc;
}

double nuclear::cost_fuel (double m_fuel)
{
      double cost_fuel = 75000*m_fuel;
      return cost_fuel;
}

//  use r2001aso
//    http://www.tpub.com/content/doe/h1019v1/css/h1019v1_84.htm
//    n + U-235 ->Cs-140 + Rb-93 + 3n + 200MeV
```

```
//End of file.

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

7. *orbit.cpp*

```
//////////////////////////////////////////////////////////////
//Filename:
//orbit.cpp
//////////////////////////////////////////////////////////////


#include "stdafx.h"
#include "orbit.h"
#using <mscorlib.dll>
using namespace System;

//This file defines the functions of the orbit class.
//Notes below were primitive equations that have been modified or replaced.

double orbit::sun_g(double G, double M)
{
      double MG = G * M;
      return MG;
}

double orbit::vi_earth(double MG, double ri)
{
      //double Rpe = 0.9833 * 149,597,870.691;
      //double Rae = 1.0167 * 149,597,870.691;
      double vie = Math::Sqrt(MG / ri);
      //double vie = (Rpe * Math::Sqrt ((2 * MG * Rae) / (Rpe * (Rae +
Rpe)))) / (ri * Math::Sin (zi));
      Console::Write(S"\nInitial Velocity on Earth Orbit = ");
      Console::WriteLine(vie);

      return vie;
}

double orbit::vf_pluto(double Rpp, double MG, double Rap, double rf, double
zf)
{
      double vfp = (Rpp * Math::Sqrt ((2 * MG * Rap) / (Rpp * (Rap + Rpp))))
/ (rf * Math::Sin (zf));
      return vfp;
}


//End of file.


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

8. *physics.cpp*

```
//////////////////////////////////////////////////////////////
//Filename:
//physics.cpp
//////////////////////////////////////////////////////////////


#include "stdafx.h"
#include "orbit.h"
#include "physics.h"
#using <mscorlib.dll>
using namespace System;

//This file defines the fuctions of physics.h header file.
//The notes below refers to the sources.

// from sepfp
double physics::I_tot (double Thrust, double time)
{
      double I_tot = Thrust * time;
      return I_tot;
}

double physics::I_sp (double I_tot, double g, double m_init)
{
      double I_sp = I_tot / (m_init * g);
      return I_sp;
}

double physics::v_jet (double g, double I_sp)
{
      double v_jet = I_sp/g;
      return v_jet;
}

// from r2001aso...
double physics::P_spec (double P_out, double m_system)
{
      double P_spec = P_out / m_system;
      return P_spec;
}

double physics::nu_jet (double P_out, double P_jet)         // ~0.8 for
nuclear estimates
{
      double nu_jet = P_jet / P_out;
      return nu_jet;
}

double physics::P_jet (double g, double c, double m_dot)
{
      double P_jet = m_dot*c*c / (2*g);
      return P_jet;
}

//End of file.

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

9. *pion4.cpp*

```
//////////////////////////////////////////////////////////////
//Filename:
//pion4.cpp
//////////////////////////////////////////////////////////////


//This is the primary file in the program and its namesake.
//Herein are defined the operative functions.

#include "stdafx.h"
#include "antimatter.h"
#include "hohmann.h"
#include "solarsailX.h"
#include "physics.h"
#include "chemical.h"
#include "nuclear.h"
#include "cost.h"

#using <mscorlib.dll>
using namespace System;
using namespace System::IO;

//Each of the primary functions is defined immediately below.

double time(StreamWriter* sw);
double final_cost_anti (double t, StreamWriter* sw);
double final_cost_solar (double t, StreamWriter* sw);
double final_cost_chem (double hohmann_dvi, double t, StreamWriter* sw);
double final_cost_nuclear (double time, double dvi, StreamWriter* sw);
double final_cost_anti_X (double t);
double final_cost_solar_X (double t);
double final_cost_chem_X (double hohmann_dvi, double t);
double final_cost_nuclear_X (double time, double dvi);
int postA();
int postB();
double hohmann_dvi(StreamWriter* sw);
double hohmann_dvi_X();
__int16 M ();
int Method_One ();
int Method_Two ();
int Ask ();

//Below is the operative function.

int _tmain()
{
      Ask ();
      return 0;
}

//////////////////////////////////////////////////////////
//                                  GENERIC FUNCTIONS
```

```
/////////////////////////////////////////////////////////
//These functions operate with each of the Methods described later in the
file.

//This function queries the user on the choice of two methods.
//The user must respond "1" or "2", or the function will be repeated.

int Ask ()
{
      __int16 Method = M ();

      switch (Method)
      {
      case 1:     Method_One ();    break;
      case 2:     Method_Two ();    break;
      default:    Console::WriteLine(S"\nInvalid Value\nOptions: 1, 2 ");
      Ask();      break;
      }

      return 0;
}

__int16 M ()
{
      Console::WriteLine(S"\nSpace Propulsion Methods Cost Estimator");
      Console::WriteLine(S"\nChoose A Method (1, 2) ");

      String __gc * input = Console::ReadLine();
      __int16 M = input->ToInt16(0);
      Console::Write(S"\nMethod ");
      Console::WriteLine(M);

      return M;
}

int postA()
{
      Console::WriteLine(S"Press Enter to Continue");
      String __gc * input_a = Console::ReadLine();
      return 0;
}

int postB()
{
      Console::WriteLine(S"Press Enter to Exit");
      String __gc * input_b = Console::ReadLine();
      return 0;
}

/////////////////////////////////////////////////////////
//                                 FOR METHOD TWO
/////////////////////////////////////////////////////////
//This method provides all of the data for one year.
//Output is sent to Pion_Test_Method_Two.doc in the project folder.

double time(StreamWriter* sw)
{
      //The user is permitted to define the year to be processed through this
function.

      Console::Write(S"How much pre-launch time?  ");
      sw->Write(S"Your pre-launch time =  ");

      String __gc * input = Console::ReadLine();
      double T = input->ToDouble(0);
      sw->WriteLine(T);

      return T;
}

//Each of the functions from this point to Method One has a compliment within
that method.

double hohmann_dvi(StreamWriter* sw)
{
      hohmann * alpha;
      alpha = new hohmann;

      double c = 299792458;
      double m_pay = 10;
      double ri = 149597870691;

      double  gamma = alpha->delta_vi_h(ri);
      Console::Write(S"\nDelta Velocity = ");
      Console::WriteLine(gamma);
      sw->Write(S"\nDelta Velocity = ");
      sw->WriteLine(gamma);

      return gamma;
}

//The following four functions defines total cost of a mission using each of
the propellants.

double final_cost_anti (double t, StreamWriter* sw)
{
      Console::Write(S"----------------------------------");
      Console::Write(S"\nAntimatter Results\n");
      Console::Write(S"----------------------------------");
      Console::WriteLine();

      sw->Write(S"----------------------------------");
      sw->Write(S"\nAntimatter Results\n");
      sw->Write(S"----------------------------------");
      sw->WriteLine();

      antimatter * alpha;
      alpha = new antimatter;

      hohmann * beta;
      beta = new hohmann;

      double v_exhaust = 598000;
      double c = 299792458;
```

```
	double beta_anti = 100000;
	double m_pay = 10;
	double eta_exhaust = 0.84;
	double cost_grid = 0.2;
	double ri = 149597870691;

	//This is the only function that runs output to the console and outside
file simultaneously.
	//The others run data first to the console and later to the outside
file.

	double  gamma = beta->delta_vi_h(ri);
	Console::Write(S"\nDelta Velocity = ");
	Console::WriteLine(gamma);
	sw->Write(S"\nDelta Velocity = ");
	sw->WriteLine(gamma);

	double delta = alpha->gamma_anti(v_exhaust, c);
	Console::Write(S"\nGamma Parameter = ");
	Console::WriteLine(delta);
	sw->Write(S"\nGamma Parameter = ");
	sw->WriteLine(delta);

	double epsilon = alpha->relative_m_anti(gamma, c, v_exhaust);
	Console::Write(S"\nRelative Mass = ");
	Console::WriteLine(epsilon);
	sw->Write(S"\nRelative Mass = ");
	sw->WriteLine(epsilon);

	double zeta = alpha->m_prop(beta_anti, delta, eta_exhaust);
	Console::Write(S"\nPropellant Mass = ");
	Console::WriteLine(zeta);
	sw->Write(S"\nPropellant Mass = ");
	sw->WriteLine(zeta);

	double theta = alpha->m_dry(epsilon, zeta);
	Console::Write(S"\nSystem Dry Mass = ");
	Console::WriteLine(theta);
	sw->Write(S"\nSystem Dry Mass = ");
	sw->WriteLine(theta);

	double mu = alpha->m_struct(theta);
	Console::Write(S"\nStructure Mass = ");
	Console::WriteLine(mu);
	sw->Write(S"\nStructure Mass = ");
	sw->WriteLine(mu);

	double eta = alpha->lamda_anti(mu, zeta);
	Console::Write(S"\nLamda parameter = ");
	Console::WriteLine(eta);
	sw->Write(S"\nLamda parameter = ");
	sw->WriteLine(eta);

	double iota = alpha->m_anti(beta_anti, delta, eta_exhaust, epsilon,
eta, m_pay);
	Console::Write(S"\nMass Antimatter = ");
	Console::WriteLine(iota);
```

```
        sw->Write(S"\nMass Antimatter = ");
        sw->WriteLine(iota);

        double kappa = alpha->eta_anti(t);
        Console::Write(S"\nAntimatter Production Efficiency = ");
        Console::WriteLine(kappa);
        sw->Write(S"\nAntimatter Production Efficiency = ");
        sw->WriteLine(kappa);

        double lamda = alpha->E_cost_anti(cost_grid, iota, c, kappa);

        Console::Write(S"\nTotal Antimatter Cost = ");
        Console::WriteLine(lamda);
        sw->Write(S"\nTotal Antimatter Cost = ");
        sw->WriteLine(lamda);

        cost * e;
        e = new cost;

        double f = e->cost_mission(t, 2);
        Console::Write(S"\nMission Cost = ");
        Console::WriteLine(f);
        sw->Write(S"\nMission Cost = ");
        sw->WriteLine(f);

        double h = e->cost_tot(lamda, 100000000, f);
        Console::Write(S"\nTotal Cost = ");
        Console::WriteLine(h);
        sw->Write(S"\nTotal Cost = ");
        sw->WriteLine(h);

        Console::WriteLine(S"\n ________");

        return 0;
}

double final_cost_solar (double t, StreamWriter* sw)

{
        Console::Write(S"----------------------------------");
        Console::Write(S"\nSolar Sail Results\n");
        Console::Write(S"----------------------------------");
        Console::WriteLine();

        sw->Write(S"----------------------------------");
        sw->Write(S"\nSolar Sail Results\n");
        sw->Write(S"----------------------------------");
        sw->WriteLine();

        solarsailX * z;
        z = new solarsailX;

        double y = z->A_sail(t);
        Console::Write(S"\nSail Area = ");
        Console::WriteLine(y);

        double x = z->alpha_sail();
```

```
Console::Write(S"\nSail Angle = ");
Console::WriteLine(x);

double w = z->g_const_solar();
Console::Write(S"\nHeliocentric Gravitational Constant = ");
Console::WriteLine(w);

double v = z->m_sail(y, t);
Console::Write(S"\nSail Mass = ");
Console::WriteLine(v);

double u = z->m_tot(v, 10);
Console::Write(S"\nSystem Mass = ");
Console::WriteLine(u);

double s = z->R(t);
Console::Write(S"\nSail Reflectivity = ");
Console::WriteLine(s);

double r = z->t_day();
Console::Write(S"\nSeconds per Timestep = ");
Console::WriteLine(r);

double q = 1.49597870691*Math::Pow(10, 11);
double p = 0;
double n = 29800/q;

double m = z->r_AU(q);
double l = z->P_solar(m, s);
double k = z->F_sail(l, y);
double j = z->a_char(k, m, u);
double i = z->a_rad(q, n, w, j, x);
double h = z->a_theta(n, q, p, j, x);
double f = z->v_r_m_init(p);
double e = z->v_theta_init(n);
double d = z->r_m_init(q);

Console::WriteLine(S"\nAU Distance = ");
Console::Write(m);

Console::Write(S"\nSolar Pressure = ");
Console::WriteLine(l);

Console::Write(S"\nSolar Force = ");
Console::WriteLine(k);

Console::Write(S"\nCharacteristic Acceleration = ");
Console::WriteLine(j);

Console::Write(S"\nRadial Acceleration = ");
Console::WriteLine(i);

Console::Write(S"\nAngular Acceleration = ");
Console::WriteLine(h);

Console::Write(S"\nRadial Velocity = ");
Console::WriteLine(p);
```

```
Console::Write(S"\nAngular Velocity = ");
Console::WriteLine(n);

Console::Write(S"\nRadius = ");
Console::WriteLine(q);

Console::Write(S"\nCalculating  .  .  .  \n");

sw->WriteLine(S"\nAU Distance = ");
sw->Write(m);

sw->Write(S"\nSolar Pressure = ");
sw->WriteLine(l);

sw->Write(S"\nSolar Force = ");
sw->WriteLine(k);

sw->Write(S"\nCharacteristic Acceleration = ");
sw->WriteLine(j);

sw->Write(S"\nRadial Acceleration = ");
sw->WriteLine(i);

sw->Write(S"\nAngular Acceleration = ");
sw->WriteLine(h);

sw->Write(S"\nRadial Velocity = ");
sw->WriteLine(p);

sw->Write(S"\nAngular Velocity = ");
sw->WriteLine(n);

sw->Write(S"\nRadius = ");
sw->WriteLine(q);

sw->Write(S"\nCalculating  .  .  .  \n");

for (int day = 1; m <= 39.53; day++)
{
      m = z->r_AU(q);
      l = z->P_solar(m, s);
      k = z->F_sail(l, y);
      j = z->a_char(k, m, u);
      i = z->a_rad(q, n, w, j, x);
      h = z->a_theta(n, q, p, j, x);
      f = z->v_r_m_init(p);

      p += i*r;
      q += f*r + .5*p*r;
      n += h*r;
}

double year = day/12;
Console::Write(S"\nYears = ");
Console::WriteLine(year);
```

```
Console::Write(S"\nAU Distance = ");
Console::WriteLine(m);

Console::Write(S"\nSolar Pressure = ");
Console::WriteLine(l);

Console::Write(S"\nSolar Force = ");
Console::WriteLine(k);

Console::Write(S"\nCharacteristic Acceleration = ");
Console::WriteLine(j);

Console::Write(S"\nRadial Acceleration = ");
Console::WriteLine(i);

Console::Write(S"\nAngular Acceleration = ");
Console::WriteLine(h);

Console::Write(S"\nRadial Velocity = ");
Console::WriteLine(p);

Console::Write(S"\nAngular Velocity = ");
Console::WriteLine(n);

Console::Write(S"\nRadius = ");
Console::WriteLine(q);

double aa = z->unit_cost(t);
Console::Write(S"\nUnit Cost = ");
Console::WriteLine(aa);

double bb = z->cost_sail(y, aa);
Console::Write(S"\nSail Cost = ");
Console::WriteLine(bb);

cost * cc;
cc = new cost;

double dd = cc->cost_mission(t, 1.8);
double cost_mission_sail = dd + 10000000*year;
Console::Write(S"\nMission Cost = ");
Console::WriteLine(dd);

double ee = z->cost_tot(bb, dd);
Console::Write(S"\nTotal Cost = ");
Console::WriteLine(ee);

Console::WriteLine(S"\n ________");

sw->Write(S"\nYears = ");
sw->WriteLine(year);

sw->Write(S"\nAU Distance = ");
sw->WriteLine(m);

sw->Write(S"\nSolar Pressure = ");
sw->WriteLine(l);
```

```
        sw->Write(S"\nSolar Force = ");
        sw->WriteLine(k);

        sw->Write(S"\nCharacteristic Acceleration = ");
        sw->WriteLine(j);

        sw->Write(S"\nRadial Acceleration = ");
        sw->WriteLine(i);

        sw->Write(S"\nAngular Acceleration = ");
        sw->WriteLine(h);

        sw->Write(S"\nRadial Velocity = ");
        sw->WriteLine(p);

        sw->Write(S"\nAngular Velocity = ");
        sw->WriteLine(n);

        sw->Write(S"\nRadius = ");
        sw->WriteLine(q);

        sw->Write(S"\nUnit Cost = ");
        sw->WriteLine(aa);

        sw->Write(S"\nSail Cost = ");
        sw->WriteLine(bb);

        sw->Write(S"\nMission Cost = ");
        sw->WriteLine(dd);

        sw->Write(S"\nTotal Cost = ");
        sw->WriteLine(ee);

        return 0;
}

double final_cost_nuclear (double time, double dvi, StreamWriter* sw)
{
        Console::Write(S"----------------------------------");
        Console::Write(S"\nNuclear-Electric Results\n");
        Console::Write(S"----------------------------------");
        Console::WriteLine();

        sw->Write(S"----------------------------------");
        sw->Write(S"\nNuclear-Electric Results\n");
        sw->Write(S"----------------------------------");
        sw->WriteLine();

        nuclear * a;
        a = new nuclear;

        double t = time;

        double b = a->Isp_nuc(t);
        Console::Write(S"\nIsp = ");
        Console::WriteLine(b);
        double c = a->m_fuel(dvi, 8.87*Math::Pow(10, -3), b);
        Console::Write(S"\nFuel Mass = ");
        Console::WriteLine(c);

        double d = a->m_sys(dvi, 8.87*Math::Pow(10, -3), b, 10);
        Console::Write(S"\nSystem Mass = ");
        Console::WriteLine(d);

        cost * e;
        e = new cost;

        double f = e->cost_mission(t, 1.2);
        Console::Write(S"\nMission Cost = ");
        Console::WriteLine(f);

        double g = a->cost_fuel(c);
        Console::Write(S"\nFuel Cost = ");
        Console::WriteLine(g);

        double h = e->cost_tot(g, 100000000, f);
        Console::Write(S"\nTotal Cost = ");
        Console::WriteLine(h);

        Console::WriteLine(S"\n ________");

        sw->Write(S"\nIsp = ");
        sw->WriteLine(b);

        sw->Write(S"\nFuel Mass = ");
        sw->WriteLine(c);

        sw->Write(S"\nSystem Mass = ");
        sw->WriteLine(d);

        sw->Write(S"\nMission Cost = ");
        sw->WriteLine(f);

        sw->Write(S"\nFuel Cost = ");
        sw->WriteLine(g);

        sw->Write(S"\nTotal Cost = ");
        sw->WriteLine(h);

        return h;
}

double final_cost_chem (double hohmann_dvi, double t, StreamWriter* sw)
{
        Console::Write(S"----------------------------------");
        Console::Write(S"\nChemical Results\n");
        Console::Write(S"----------------------------------");
        Console::WriteLine();

        sw->Write(S"----------------------------------");
        sw->Write(S"\nChemical Results\n");
        sw->Write(S"----------------------------------");
```

```
      sw->WriteLine();

      chemical * alpha;
    alpha = new chemical;

      double  delta = alpha->P_ref(hohmann_dvi, 9.8, 292.1);
      Console::Write(S"\nPropellant Mass Ratio = ");
      Console::WriteLine(delta);

      double  epsilon = alpha->m_prop(delta, 150, 10);
      Console::Write(S"\nPropellant Mass = ");
      Console::WriteLine(epsilon);

      double  eta = alpha->cost_prop(epsilon);
      Console::Write(S"\nPropellant Cost = ");
      Console::WriteLine(eta);

      double  zeta = alpha->cost_mission(t);
      Console::Write(S"\nMission Cost = ");
      Console::WriteLine(zeta);

      double  theta = alpha->cost_tot_chem(eta, 90000000, zeta);
      Console::Write(S"\nTotal Cost = ");
      Console::WriteLine(theta);

      Console::WriteLine(S"\n ________");

      sw->Write(S"\nPropellant Mass Ratio = ");
      sw->WriteLine(delta);

      sw->Write(S"\nPropellant Mass = ");
      sw->WriteLine(epsilon);

      sw->Write(S"\nPropellant Cost = ");
      sw->WriteLine(eta);

      sw->Write(S"\nMission Cost = ");
      sw->WriteLine(zeta);

      sw->Write(S"\nTotal Cost = ");
      sw->WriteLine(theta);

      return theta;
}

//This function is potentially called by the Ask function.

int Method_Two()
{
      try
      {

            //The following statement forms the output to a *.doc file.

          FileStream* fs = new FileStream(S"Pion_Test_Method_Two.doc",
FileMode::Create);
            StreamWriter* sw = new StreamWriter(fs);
          Console::WriteLine(S"Test");
            sw->WriteLine(S"Test");
            sw->WriteLine(S"-----------------------------------");
            sw->WriteLine();

            double t = time(sw);
            Console::WriteLine(S"\n ________");
            Console::WriteLine();
            double dvi = hohmann_dvi(sw);
            Console::WriteLine(S"\n ________");
            Console::WriteLine(S"\n");
            sw->WriteLine();
            postA();

            final_cost_chem (dvi, t, sw);
            postA();

            final_cost_solar (t, sw);
            postA();

            final_cost_nuclear (t, dvi, sw);
            postA();

            final_cost_anti (t, sw);
            postA();

            sw->Flush();
            sw->Close();
      }
      catch(System::Exception* pe)
      {
            Console::WriteLine(pe->ToString());
      }

      postB();

      return 0;
}

////////////////////////////////////////////////////////////
//                                        FOR METHOD ONE
////////////////////////////////////////////////////////////
//This method provides the final estimated cost for each propulsion mechanism
for the following 100 years.
//Output is sent to Pion_Test_Method_One.xls in the project folder.

//These functions correspond to the final six functions of the Method Two
section.

double hohmann_dvi_X()
{
      hohmann * alpha;
      alpha = new hohmann;

      double c = 299792458;
      double m_pay = 10;
```

```
	double ri = 149597870691;

	double  gamma = alpha->delta_vi_h(ri);

	return gamma;
}

double final_cost_anti_X (double t)
{
	antimatter * alpha;
	alpha = new antimatter;

	hohmann * beta;
	beta = new hohmann;

	double v_exhaust = 598000;
	double c = 299792458;
	double beta_anti = 100000;
	double m_pay = 10;
	double eta_exhaust = 0.84;
	double cost_grid = 0.2;
	double ri = 149597870691;

	double  gamma = beta->delta_vi_h(ri);
	double delta = alpha->gamma_anti(v_exhaust, c);
	double epsilon = alpha->relative_m_anti(gamma, c, v_exhaust);
	double zeta = alpha->m_prop(beta_anti, delta, eta_exhaust);
	double theta = alpha->m_dry(epsilon, zeta);
	double mu = alpha->m_struct(theta);
	double eta = alpha->lamda_anti(mu, zeta);
	double iota = alpha->m_anti(beta_anti, delta, eta_exhaust, epsilon,
eta, m_pay);
	double kappa = alpha->eta_anti(t);
	double lamda = alpha->E_cost_anti(cost_grid, iota, c, kappa);


	cost * e;
	e = new cost;

	double f = e->cost_mission(t, 2);
	double h = e->cost_tot(lamda, 100000000, f);

	return h;
}

double final_cost_solar_X (double t)

{
	solarsailX * z;
	z = new solarsailX;

	double y = z->A_sail(t);
	double x = z->alpha_sail();
	double w = z->g_const_solar();
	double v = z->m_sail(y, t);
	double u = z->m_tot(v, 10);
	double s = z->R(t);
```

```
      double r = z->t_day();

      double q = 1.49597870691*Math::Pow(10, 11);
      double p = 0;
      double n = 29800/q;

      double m = z->r_AU(q);
      double l = z->P_solar(m, s);
      double k = z->F_sail(l, y);
      double j = z->a_char(k, m, u);
      double i = z->a_rad(q, n, w, j, x);
      double h = z->a_theta(n, q, p, j, x);
      double f = z->v_r_m_init(p);
      double e = z->v_theta_init(n);
      double d = z->r_m_init(q);

      for (int day = 1; m <= 39.53; day++)
      {
            m = z->r_AU(q);
            l = z->P_solar(m, s);
            k = z->F_sail(l, y);
            j = z->a_char(k, m, u);
            i = z->a_rad(q, n, w, j, x);
            h = z->a_theta(n, q, p, j, x);
            f = z->v_r_m_init(p);

            p += i*r;
            q += f*r + .5*p*r;
            n += h*r;
      }

      double year = day/12;
      double aa = z->unit_cost(t);
      double bb = z->cost_sail(y, aa);

      cost * cc;
      cc = new cost;

      double dd = cc->cost_mission(t, 1.8);
      double cost_mission_sail = dd + 10000000*year;
      double ee = z->cost_tot(bb, dd);

      return ee;
}

double final_cost_nuclear_X (double time, double dvi)
{
      nuclear * a;
      a = new nuclear;

      double t = time;
      double b = a->Isp_nuc(t);
      double c = a->m_fuel(dvi, 8.87*Math::Pow(10, -3), b);
      double d = a->m_sys(dvi, 8.87*Math::Pow(10, -3), b, 10);

      cost * e;
      e = new cost;
```

```
        double f = e->cost_mission(t, 1.2);
        double g = a->cost_fuel(c);
        double h = e->cost_tot(g, 100000000, f);

        return h;
}

double final_cost_chem_X (double hohmann_dvi, double t)
{
        chemical * alpha;
    alpha = new chemical;

        double  delta = alpha->P_ref(hohmann_dvi, 9.8, 292.1);
        double  epsilon = alpha->m_prop(delta, 150, 10);
        double  eta = alpha->cost_prop(epsilon);
        double  zeta = alpha->cost_mission(t);
        double  theta = alpha->cost_tot_chem(eta, 90000000, zeta);

        return theta;
}

//This function is potentially called by the Ask function.

int Method_One ()
{
    try
      {

            //The following statement forms the output to a *.xls file.

          FileStream* fs = new FileStream(S"Pion_Test_Method_One.xls",
FileMode::Create);
            StreamWriter* sw = new StreamWriter(fs);

            sw->WriteLine(S"");
            sw->WriteLine(S"  Test");
            sw->WriteLine(S"Years   Chemical    Nuclear     Solar
      Antimatter");

            Console::WriteLine(S"\n");
            Console::WriteLine(S"   Test: Method One");
            Console::WriteLine(S"   Years Chemical    Nuclear     Solar
      Antimatter");

            double dvi = hohmann_dvi_X();
            double t = 0;

            for (int ToLaunch = 1; ToLaunch <= 100; ToLaunch++)
            {
                  t++;

                  double one = final_cost_chem_X (dvi, t);
                  double two = final_cost_nuclear_X (t, dvi);
                  double three = final_cost_solar_X (t);
                  double four = final_cost_anti_X (t);
```

```
                        Console::WriteLine(S"\n ");
                        Console::WriteLine(ToLaunch);
                        Console::WriteLine(S"\n ");
                        Console::WriteLine(one);
                        Console::WriteLine(S"\n ");
                        Console::WriteLine(two);
                        Console::WriteLine(S"\n ");
                        Console::WriteLine(three);
                        Console::WriteLine(S"\n ");
                        Console::WriteLine(four);

                        sw->WriteLine(S"  ");
                        sw->Write(ToLaunch);
                        sw->Write(S"      ");
                        sw->Write(one);
                        sw->Write(S"      ");
                        sw->Write(two);
                        sw->Write(S"      ");
                        sw->Write(three);
                        sw->Write(S"      ");
                        sw->Write(four);
                }

                sw->WriteLine(S"");
                sw->WriteLine(S"  End");

                sw->Flush();
                sw->Close();
        }
        catch(System::Exception* pe)
        {
                Console::WriteLine(pe->ToString());
        }

        postB();

        return 0;
}


//End of file.


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

10. *solarsail.cpp*

```
/////////////////////////////////////////////////////////////
//Filename:
//solarsail.cpp
/////////////////////////////////////////////////////////////


//This .cpp file corresponds to the abandoned solarsail.h file.


#include "stdafx.h"
#include "solarsail.h"


#using <mscorlib.dll>
using namespace System;


double solarsail::P_solar (double R_sail, double r_au)
{
      double P_solar = 4.563*Math::Pow(10, -6) * (1+R_sail) / r_au*r_au;
      return P_solar;
}


double solarsail::A_sail (double r_sail)
{
      double A_sail = r_sail * Math::PI;
      return A_sail;
}


double solarsail::f_solar (double P_solar, double A_sail)
{
      double f_solar = P_solar * A_sail;
      return f_solar;
}


double solarsail::cost_sail (double u_cost, double A_sail)
{
      double cost_sail = u_cost * A_sail;
      return cost_sail;
}


      //  end redefinition sequence
                                                              //40


double solarsail::kappa (double beta)
{
      double kappa = Math::PI/2-beta;
      return kappa;
}

double solarsail::delta (double theta)
{
      double delta = Math::PI/2-theta;
      return delta;
}

double solarsail::theta (double rad, double r, double beta)
{
      double theta = Math::Asin(r*Math::Sin(beta)/rad);
      return theta;
}

double solarsail::beta (double s_x, double s_y)
{
      double beta = Math::PI/2-Math::Atan(s_y/s_x);
      return beta;
}

      //  above here redefine for next iteration

double solarsail::vf_y (double vi_y, double delta_t, double a_y)
{
      double vf_y = vi_y + a_y*delta_t;
      return vf_y;
}

double solarsail::vf_x (double vi_x, double delta_t, double a_x)
{
      double vf_x = vi_x + a_x*delta_t;
      return vf_x;
}
                                                                        //80
double solarsail::rad (double s_net, double r_zero, double beta)
{
      double rad = Math::Sqrt(s_net*s_net + r_zero*r_zero -
2*s_net*r_zero*Math::Cos(beta));
      return rad;
}

double solarsail::s_net (double s_x, double s_y)
{
      double s_net = Math::Sqrt(s_x*s_x + s_y*s_y);
      return s_net;
}

double solarsail::s_x (double vi_x, double delta_t, double a_x)
{
      double s_x = vi_x*delta_t + (a_x*delta_t*delta_t)/(2);
      return s_x;
}

double solarsail::s_y (double vi_y, double delta_t, double a_y)
{
      double s_y = vi_y*delta_t + (a_y*delta_t*delta_t)/(2);
      return s_y;
}


double solarsail::a_x (double r, double R, double alpha, double m_unit)
{
      double a_x = ((4.56/Math::Pow(10, 6)*(1+R)*(Math::Pow(Math::Cos(alpha),
2)))/(m_unit*r*r));
      return a_x;
}

double solarsail::a_y (double r, double R, double alpha, double m_unit,
double g_sol, double ri)
{
      double a_y =
(4.56*(1+R)*Math::Sin(alpha)*Math::Sin(alpha)/(Math::Pow(10, 6)*m_unit*r*r)-
g_sol/(r*r*ri*ri));
      return a_y;
}

double solarsail::vi_x (double kappa, double delta, double beta, double vf_y,
double vf_x)
{
      double vi_x = Math::Abs(Math::Cos(kappa+delta))*vf_x + Math::Cos(beta-
delta)*vf_y;
      return vi_x;
}

double solarsail::vi_y (double kappa, double delta, double beta, double vf_y,
double vf_x)
{
      double vi_y = Math::Abs(Math::Sin(kappa+delta))*vf_x + Math::Sin(beta-
delta)*vf_y;
      return vi_y;
}




//End of file.


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

11. *solarsailX.cpp*

```
///////////////////////////////////////////////////////////////
//Filename:
//solarsailX.cpp
///////////////////////////////////////////////////////////////


#include "Stdafx.h"
#include "solarsailX.h"
#using <mscorlib.dll>
using namespace System;

//This file provides the basis of the solarsail functions used.
//As a replacement and one of the last files, it has the most developed
structure.

double solarsailX::P_solar (double r_AU, double R)
{
      double P_solar = ((4.56/Math::Pow(10, 6))*(1 + R))/(r_AU*r_AU);
      return P_solar;
}

double solarsailX::A_sail (double time)
{
      double A_sail = 10000*(1 + .1*Math::Pow(10, (time*time/(time*100))));
      return A_sail;
}

double solarsailX::F_sail (double P_solar, double A_sail)
{
      double F_sail = P_solar*A_sail;
      return F_sail;
}

double solarsailX::a_char (double F_sail, double r_AU, double m_tot)
{
      double a_char = 2*F_sail*r_AU/m_tot;
      return a_char;
}

double solarsailX::m_tot (double m_sail, double m_pay)
{
      double m_tot = m_sail + m_pay;
      return m_tot;
}

double solarsailX::m_sail (double A_sail, double time)
{
      double m_sail = A_sail * .024 - .00225*(Math::Pow(10,
(time*time*time/(time*10000))));
      return m_sail;
}
```

```
double solarsailX::a_rad (double r_m, double v_theta, double g_const_solar,
double a_char, double alpha_sail)
{
      double a_rad = r_m*v_theta*v_theta - (g_const_solar/(r_m*r_m)) +
((a_char*Math::Pow((Math::Cos(alpha_sail)), 3))/(r_m*r_m));
      return a_rad;
}

double solarsailX::a_theta (double v_theta, double r_m, double v_r_m, double
a_char, double alpha_sail)
{
      double a_theta = ((-2*v_r_m*v_theta)/r_m) +
((a_char*Math::Pow((Math::Cos(alpha_sail)),
2)*Math::Sin(alpha_sail))/(Math::Pow(r_m, 3)));
      return a_theta;
}

double solarsailX::g_const_solar ()
{
      double g_const_solar = 1.32713430118*Math::Pow(10, 20);
      return g_const_solar;
}

double solarsailX::alpha_sail ()
{
      double alpha_sail = Math::PI/6;
      return alpha_sail;
}

double solarsailX::r_AU (double r_m)
{
      double r_AU = r_m/(1.49597870691*Math::Pow(10, 11));
      return r_AU;
}

double solarsailX::r_m (double r_m_init, double v_r_m, double t_day)
{
      double r_m = r_m_init + .5*v_r_m*t_day;
      return r_m;
}

double solarsailX::v_r_m (double v_r_m_init, double a_rad, double t_day)
{
      double v_r_m = v_r_m_init + a_rad*t_day;
      return v_r_m;
}

double solarsailX::v_theta (double v_theta_init, double a_theta, double
t_day)
{
      double v_theta = v_theta_init + a_theta*t_day;
      return v_theta;
}

double solarsailX::t_day ()
{
      double t_day = 2629728;
```

```
	return t_day;
}

double solarsailX::v_theta_init (double v_theta)
{
	double v_theta_init = v_theta;
	return v_theta_init;
}

double solarsailX::v_r_m_init (double v_r_m)
{
	double v_r_m_init = v_r_m;
	return v_r_m_init;
}

double solarsailX::r_m_init (double r_m)
{
	double r_m_init = r_m;
	return r_m_init;
}

double solarsailX::R (double time)
{
	double R = .9 + .01*Math::Pow(10, (time*time/(time*1000)));
	return R;
}

double solarsailX::cost_sail (double A_sail, double unit_cost)
{
	double cost_sail = A_sail*unit_cost + 75000000;
	return cost_sail;
}

double solarsailX::unit_cost (double time)
{
	double unit_cost = 10 - .5*Math::Pow(10, (time*time/(time*100)));
	return unit_cost;
}

double solarsailX::cost_tot (double cost_sail, double cost_mission)
{
	double cost_tot = cost_sail + cost_mission;
	return cost_tot;
}


//End of file.


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

12. *stdafx.cpp*

```
//////////////////////////////////////////////////////////////
//Filename:
//stdafx.cpp
//////////////////////////////////////////////////////////////


// stdafx.cpp : source file that includes just the standard includes
// pion3.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"



//End of file.


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

13. *antimatter.h*

```
/////////////////////////////////////////////////////////////////
//Filename:
//antimatter.h
/////////////////////////////////////////////////////////////////


#pragma once
#include "stdafx.h"

__gc class antimatter          //This class gives the functions related to
Antimatter Propulsion.
{
public:

      //The notes at the top of each section denote the source as they were
drawn from several papers.

      //antimatter cost       from apfntpa
      double eta_anti (double t);   //efficiency   max=0.5
      double Ein_anti (double c, double eta_anti, double m_anti);
      double E_cost_anti (double cost_grid, double m_anti, double c, double
eta_anti);

      //antimatter requirements           "
      double m_anti (double beta_anti, double gamma_anti, double
eta_exhaust_anti, double relative_m_anti, double lamda_anti, double m_pay);
      double relative_m_anti (double delta_v, double c, double v_exhaust);
      double lamda_anti (double m_struct, double m_prop);
      double gamma_anti (double v_exhaust, double c);
      double m_prop (double beta_anti, double gamma_anti, double
eta_exhaust_anti);
      double m_dry (double relative_m_anti, double m_prop);
      double m_struct (double m_dry);

      //antimatter physics          from appwmce
      double B_min (double n_dense_anti, double T_spec, double n_dense_H,
double T_H);
      double Prob_remain (double R_mirror);
      double R_mirror (double B_min, double B_max);
      double eta_E (double E_ion, double E_electron, double n_dense_H, double
n_dense_anti);
      double flow_prop (double amu_H, double n_dense_H, double V_chamber,
double t_pulse);
      double v_exhaust (double E_ion);    //E_ion use E=mc2
      double Isp_anti (double v_exhaust);
      double Thrust (double flow_prop, double v_exhaust);
};


//End of file.
```

//

14. *chemical.h*

```
///////////////////////////////////////////////////////////////
//Filename:
//chemical.h
///////////////////////////////////////////////////////////////


#include "stdafx.h"
#include "physics.h"

__gc class chemical           //This class gives definitions for Chemical 
Propulsion.
{
public:

      //The final section was heavily used.

      double delta_T (double Ti, double Tf);
      double velocity_prop (double Cp, double delta_T);

      double eta_prop();
      double F_chem(double P_exit, double velocity_prop, double m_dot, double 
A_noz);
      double delta_v_chem(double g_zero, double Isp_chem, double P_ref);

      double P_ref(double delta_v_chem, double g_zero, double Isp_chem);
      double m_prop(double P_ref, double m_sys, double m_pay);
      double cost_prop(double m_prop);
    double cost_tot_chem(double cost_prop, double cost_sys, double 
cost_mission);
      double cost_mission(double time);
};


//End of file.


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

15. *cost.h*

```
//////////////////////////////////////////////////////////////
//Filename:
//cost.h
//////////////////////////////////////////////////////////////


#pragma once
#include "stdafx.h"

class cost        //This class gives functions related to generic mission
costs.
{
public:

      //The final two equations were favored.

      double cost_pre (double cost_mech, double cost_research, double
cost_materials, double cost_place);
      double cost_post (double cost_engineer, double cost_place);
      double cost_mech (double t, double n_mech, double price_mech);
      double E_cost (double cost_grid, double m_anti, double c, double
eta_anti);

      double cost::cost_tot(double cost_prop, double cost_sys, double
cost_mission);
    double cost::cost_mission(double time, double cost_coef);
};


//End of file.


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

16. *hohmann.h*

```
//////////////////////////////////////////////////////////////
//Filename:
//hohmann.h
//////////////////////////////////////////////////////////////


#include "stdafx.h"
#include "orbit.h"

__gc class hohmann : public orbit         //This hierarchical class defines
the parameters used for most of the transfer orbits.
{
public:
      double vi_transfer_hoh();
      double vf_transfer_hoh(double MG, double rf, double at);
      double delta_vi_h(double ri);
      double delta_vf_h(double vfth, double vfp);
};


//End of file.


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

17. *nuclear.h*

```
//////////////////////////////////////////////////////////////
//Filename:
//nuclear.h
//////////////////////////////////////////////////////////////


#include "stdafx.h"
#include "physics.h"

__gc class nuclear              //This class gives the functions for Nuclear-
Electric Propulsion.
{
public:

      //Most of the first section was not used as the lower four functions
were found as more efficient replacements.

      double total_E_fission ();
      double mass_flow_func (double T_sp, double m, double g_zero, double
I_sp);
      double T_sp (double T, double m);
      double alpha_sys (double P_RD);
      double m_init_total (double m_sys, double m_fuel, double m_pay);
      double m_fin_total (double m_sys, double m_pay);
      double mu_dyn_nuc (double P_RD);
      double P_RD (double RAN, double t);

      double m_fuel (double delta_v, double g_zero, double I_sp);
      double m_sys (double delta_v, double g_zero, double I_sp, double
m_pay);
      double Isp_nuc (double time);
      double cost_fuel (double m_fuel);
};


//End of file.


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

18. *orbit.h*

```
/////////////////////////////////////////////////////////////////
//Filename:
//orbit.h
/////////////////////////////////////////////////////////////////


#pragma once
#include "stdafx.h"
#include "physics.h"

__gc class orbit        //This class defines rudimentary orbital parameters.
{
public:
      double sun_g(double G, double M);
      double vi_earth(double MG, double ri);
      double vf_pluto(double Rpp, double MG, double Rap, double rf, double 
zf);
};


//End of file.


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

19. *physics.h*

```
///////////////////////////////////////////////////////////////
//Filename:
//physics.h
///////////////////////////////////////////////////////////////


#pragma once
#include "stdafx.h"

class physics           //This class gives some simple generic equations
pertaining to the physics of the project.
{
public:
      double I_tot (double Thrust, double time);
      double I_sp (double I_tot, double g, double m_init);
      double v_jet (double g, double I_sp);
      double P_spec (double P_out, double m_system);
      double nu_jet (double P_out, double P_jet);
      double P_jet (double g, double c, double m_dot);
};


//End of file.


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

20. *resource.h*

```
///////////////////////////////////////////////////////////////
//Filename:
//resource.h
///////////////////////////////////////////////////////////////


//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by app.rc

//End of file.


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

21. *solarsail.h*

```
//////////////////////////////////////////////////////////////
//Filename:
//solarsail.h
//////////////////////////////////////////////////////////////


//Original Solarsail Class

#include "stdafx.h"
//#include "physics.h"



__gc class solarsail //: public physics
{
public:
      double P_solar (double R_sail, double r_au);
      double A_sail (double r_sail);
      double f_solar (double P_solar, double A_sail);
      double cost_sail (double u_cost, double A_sail);

      double vi_x (double kappa, double delta, double beta, double vf_y,
double vf_x);
      double vi_y (double kappa, double delta, double beta, double vf_y,
double vf_x);
      double kappa (double beta);
      double delta (double theta);
      double theta (double rad, double r, double beta);
      double beta (double s_x, double s_y);
            //  above here redefine for next iteration
      double vf_y (double vi_y, double delta_t, double a_y);
      double vf_x (double vi_x, double delta_t, double a_x);
      double rad (double s_net, double r_zero, double beta);
      double s_net (double s_x, double s_y);
      double s_x (double vi_x, double delta_t, double a_x);
      double s_y (double vi_y, double delta_t, double a_y);
      double a_x (double r, double R, double alpha, double m_unit);
      double a_y (double r, double R, double alpha, double m_unit, double
g_sol, double ri);



      //add beamer parameters
};


//End of file.


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

22. *solarsailX.h*

```
//////////////////////////////////////////////////////////////
//Filename:
//solarsailX.h
//////////////////////////////////////////////////////////////


#pragma once
#include "stdafx.h"
#using <mscorlib.dll>
using namespace System;

__gc class solarsailX         //The "X" marks a replacement class.
                                          //The original class failed to give
the desired results and was dropped.
{
public:

      //This System of equations solves for the trajectory of the solarsail.
      //Concerns primarily with the time of travel to the destination.

      double P_solar (double r_AU, double R);
      double A_sail (double time);
      double F_sail (double P_solar, double A_sail);
      double a_char (double F_sail, double r_AU, double m_tot);
      double m_tot (double m_sail, double m_pay);
      double m_sail (double A_sail, double time);
      double a_rad (double r_m, double v_theta, double g_const_solar, double
a_char, double alpha_sail);
      double a_theta (double v_theta, double r_m, double v_r_m, double
a_char, double alpha_sail);
      double g_const_solar ();
      double alpha_sail ();
      double r_AU (double r_m);
      double r_m (double r_m_init, double v_r_m, double t_day);
      double v_r_m (double v_r_m_init, double a_rad, double t_day);
      double v_theta (double v_theta_init, double a_theta, double t_day);
      double t_day ();
      double v_theta_init (double v_theta);
      double v_r_m_init (double v_r_m);
      double r_m_init (double r_m);
      double R (double time);

      //The final three functions pertain solely to the cost of the system.

      double cost_sail (double A_sail, double unit_cost);
      double unit_cost (double time);
      double cost_tot (double cost_sail, double cost_mission);
};


//End of file.
```

//

23. *stdafx.h*

```
//////////////////////////////////////////////////////////////
//Filename:
//stdafx.h
//////////////////////////////////////////////////////////////


// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#include <iostream>
#include <tchar.h>

// TODO: reference additional headers your program requires here

//End of file.


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

XI. *Appendix C*: Charts

1.

Chart 1 **Sample Result of Method One**

Time	Chemical	Nuclear	Solar	Antimatter
Years	US$	US$	US$	US$
1	390504295.4	460500000	615604592.9	8.33E+12
2	391004295.4	461000000	616104687.4	1.97E+12
3	391504295.4	461500000	616604783.5	4.57E+11
4	392004295.4	462000000	617104881.3	1.50E+11
5	392504295.4	462500000	617604980.6	62199320529
6	393004295.4	463000000	618105081.6	30448134605
7	393504295.4	463500000	618605184.3	16778934720
8	394004295.4	464000000	619105288.6	10132154798
9	394504295.4	464500000	619605394.6	6591617973
10	395004295.4	465000000	620105502.2	4567545136
11	395504295.4	465500000	620605611.5	3343302004
12	396004295.4	466000000	621105722.4	2567976687
13	396504295.4	466500000	621605835	2057825416
14	397004295.4	467000000	622105949.2	1711153640
15	397504295.4	467500000	622606065.1	1468992628
16	398004295.4	468000000	623106182.6	1295762425
17	398504295.4	468500000	623606301.7	1169246052
18	399004295.4	469000000	624106422.4	1075149497
19	399504295.4	469500000	624606544.7	1004031028
20	400004295.4	470000000	625106668.5	949506783.9
21	400504295.4	470500000	625606793.9	907169550.2
22	401004295.4	471000000	626106920.8	873919234.7
23	401504295.4	471500000	626607049.2	847538001.5
24	402004295.4	472000000	627107179	826414641.1
25	402504295.4	472500000	627607310.3	809362140.6
26	403004295.4	473000000	628107442.8	795494717
27	403504295.4	473500000	628607576.8	784143537.9
28	404004295.4	474000000	629107711.9	774798066
29	404504295.4	474500000	629607848.3	767064656
30	405004295.4	475000000	630107985.8	760636946.3
31	405504295.4	475500000	630608124.3	755274426.2
32	406004295.4	476000000	631108263.9	750786746.8
33	406504295.4	476500000	631608404.4	747022114.9
34	407004295.4	477000000	632108545.7	743858621.6
35	407504295.4	477500000	632608687.7	741197705.6
36	408004295.4	478000000	633108830.3	738959180.9
37	408504295.4	478500000	633608973.4	737077425.5
38	409004295.4	479000000	634109117	735498436.9
39	409504295.4	479500000	634609260.7	734177539.6

40	410004295.4	480000000	635109404.6	733077590.8
41	410504295.4	480500000	635609548.5	732167563.6
42	411004295.4	481000000	636109692.2	731421424.2
43	411504295.4	481500000	636609835.5	730817235.2
44	412004295.4	482000000	637109978.3	730336436.1
45	412504295.4	482500000	637610120.3	729963262.6
46	413004295.4	483000000	638110261.3	729684276.2
47	413504295.4	483500000	638610401.2	729487980.9
48	414004295.4	484000000	639110539.7	729364508.6
49	414504295.4	484500000	639610676.5	729305361.5
50	415004295.4	485000000	640110811.4	729303198.8
51	415504295.4	485500000	640610944	729351659.8
52	416004295.4	486000000	641111074.2	729445216.9
53	416504295.4	486500000	641611201.4	729579052.4
54	417004295.4	487000000	642111325.5	729748955.8
55	417504295.4	487500000	642611446	729951236
56	418004295.4	488000000	643111562.6	730182648.8
57	418504295.4	488500000	643611674.8	730440333.8
58	419004295.4	489000000	644111782.3	730721762.2
59	419504295.4	489500000	644611884.5	731024690.7
60	420004295.4	490000000	645111980.9	731347123.5
61	420504295.4	490500000	645612071.1	731687278.6
62	421004295.4	491000000	646112154.5	732043559.3
63	421504295.4	491500000	646612230.5	732414529.6
64	422004295.4	492000000	647112298.5	732798893
65	422504295.4	492500000	647612357.9	733195474
66	423004295.4	493000000	648112407.9	733603202.6
67	423504295.4	493500000	648612447.9	734021100.4
68	424004295.4	494000000	649112477.2	734448269.1
69	424504295.4	494500000	649612494.8	734883881.1
70	425004295.4	495000000	650112499.9	735327170.6
71	425504295.4	495500000	650612491.7	735777426.9
72	426004295.4	496000000	651112469.2	736233989.2
73	426504295.4	496500000	651612431.4	736696241.3
74	427004295.4	497000000	652112377.3	737163608.7
75	427504295.4	497500000	652612305.7	737635555.6
76	428004295.4	498000000	653112215.4	738111582.6
77	428504295.4	498500000	653612105.3	738591225.8
78	429004295.4	499000000	654111974.1	739074055
79	429504295.4	499500000	654611820.3	739559673.7
80	430004295.4	500000000	655111642.5	740047717.5
81	430504295.4	500500000	655611439.2	740537854.5
82	431004295.4	501000000	656111208.9	741029783.8
83	431504295.4	501500000	656610949.7	741523235.2
84	432004295.4	502000000	657110660	742017967.9
85	432504295.4	502500000	657610337.9	742513769.5
86	433004295.4	503000000	658109981.4	743010454.2
87	433504295.4	503500000	658609588.5	743507861.4

88	434004295.4	504000000	659109156.9	744005853.8
89	434504295.4	504500000	659608684.4	744504315.1
90	435004295.4	505000000	660108168.5	745003148.1
91	435504295.4	505500000	660607606.9	745502272.4
92	436004295.4	506000000	661106996.6	746001622.6
93	436504295.4	506500000	661606335.1	746501145.8
94	437004295.4	507000000	662105619.3	747000800
95	437504295.4	507500000	662604846.1	747500552.1
96	438004295.4	508000000	663104012.4	748000376.5
97	438504295.4	508500000	663603114.5	748500253.6
98	439004295.4	509000000	664102149.1	749000168.8
99	439504295.4	509500000	664601112.2	749500110.9
100	440004295.4	510000000	665100000	750000071.9

2.

Chart 2 **Sample Result of Method Two**

```
Test
----------------------------------

Your pre-launch time = 10

Delta Velocity = 11814.8842429824

----------------------------------
Chemical Results
----------------------------------

Propellant Mass Ratio = 61.0136875884275

Propellant Mass = 9762.1900141484

Propellant Cost = 4295.3636062253

Mission Cost = 305000000

Total Cost = 395004295.363606

----------------------------------
Solar Sail Results
----------------------------------

AU Distance = 1

Solar Pressure = 8.710662160448E-06

Solar Force = 0.0980726955518219

Characteristic Acceleration = 0.000699989493208687

Radial Acceleration = 6.05298740944706E-06

Angular Acceleration = 7.84055043871015E-38

Radial Velocity = 0

Angular Velocity = 1.99200696255584E-07

Radius = 149597870691

Calculating  .  .  .

Years = 9

AU Distance = 39.5841645543753

Solar Pressure = 5.55914757901101E-09

Solar Force = 6.2590027945241E-05
```

```
Characteristic Acceleration = 1.76835737494759E-05

Radial Acceleration = 3.7805163473838E-06

Angular Acceleration = -6.3119362444022E-18

Radial Velocity = 16544.5865517251

Angular Velocity = 1.11367765470042E-09

Radius = 5986942230127.01

Unit Cost = 9.37053729410292

Sail Cost = 75105502.1804627

Mission Cost = 545000000

Total Cost = 620105502.180463

---------------------------------
Nuclear-Electric Results
---------------------------------

Isp = 3981.24778825592

Fuel Mass = 1.49185583941122E-148

System Mass = 1.4874100991766E-148

Mission Cost = 365000000

Fuel Cost = 1.11889187955841E-143

Total Cost = 465000000

---------------------------------
Antimatter Results
---------------------------------

Delta Velocity = 11814.8842429824

Gamma Parameter = 1.00000198944649

Relative Mass = 1.01995379935489

Propellant Mass = 1.41247289417973E+38

System Dry Mass = 7.07871653442095E+39

Structure Mass = 6.37084488097885E+39

Lambda parameter = 45.1041921387003

Mass Antimatter = 2.36288839914226E-11
```

```
Antimatter Production Efficiency = 0.000109961598420404

Total Antimatter Cost = 3862545135.77933

Mission Cost = 605000000

Total Cost = 4567545135.77933
```

THIS PAGE INTENTIONALLY LEFT BLANK