To the Graduate Council:

I am submitting herewith a thesis written by Herbert Daniel Thomas entitled "Comparison of Impulsive and Low Thrust Propulsion Options for an Earth-Orpheus Rendezvous Mission." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in aerospace engineering.

Dr. Gary A. Flandro, Major Professor

We have read this thesis
and recommend its acceptance:

Roy J. Schulz

Robert S Roach

Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

## STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Herbert D. Thomas

Date 11/30/93

# COMPARISON OF IMPULSIVE AND LOW THRUST PROPULSION OPTIONS FOR AN EARTH-ORPHEUS RENDEZVOUS MISSION

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Herbert Daniel Thomas

December 1993

## DEDICATION

This thesis is dedicated to my parents:

Herbert Thomas

and

Bobbie J. Thomas

## ACKNOWLEDGMENTS

I would like to thank may advisor, Dr. Gary A. Flandro for his guidance and support during the course of my graduate education. I also appreciate the assistance of the other professors on my thesis committee: Dr. Robert Roach and Dr. Roy Schulz. I thank Brian K. Funk and A. Allen McCarley for their help in developing the impulsive thrust and solar sail trajectory optimization programs. Steve Flanagan, Robert Maddock, and Allen McCarley were also very helpful in writing the thesis. Carl Sauer of the Jet Propulsion Laboratory provided invaluable assistance in developing the low thrust trajectory optimization procedures. I am also extremely grateful to the staff of the University of Tennessee Space Institute library, headed by Mary Lo, for helping me with the vast literary search required for this work. I would like to express my sincere appreciation to my family and friends in Knoxville for their support and encouragement, without which none of this work would have been possible. A very special thanks goes to Jessica L. Gass for her love, patience, and craziness. Without her, I truly would have gone "nuts".

## ABSTRACT

With decreasing natural resources here on Earth, new avenues must be explored which satisfy our consuming needs and provide a reasonable economic return. One such option is asteroid mining. An asteroid rendezvous mission is the first step in achieving this goal and may provide scientists with a deeper understanding of the formation of the solar system. The asteroid rendezvous mission must be technically and economically sound, and this will require that advanced propulsion systems be developed to reduce the cost of interplanetary travel and promote commercial involvement. The primary purpose of this thesis is to design and compare optimal rendezvous missions to Orpheus between the years 2000 - 2020 using different modes of propulsion. The options considered are chemical, nuclear thermal, nuclear electric, solar electric, and solar sail propulsion systems.

Three ANSI C programs were written on Silicon Graphics workstations to model the three-dimensional interplanetary flight of spacecraft using these propulsion schemes. An impulsive thrust program was written which solves Lambert's problem using a modified Gauss algorithm developed by Battin. The selection of optimal impulsive missions is based on the trajectory with the lowest total energy (C3) requirement. The low thrust trajectory programs utilize Pontryagin's Maximum Principle to optimize the power-limited and solar sail rendezvous trajectories. Fuel is minimized in the power-limited missions, while minimum flight time is the optimizing criterion in the solar sail missions. The resulting two point boundary value problems are solved using two iterative methods: the method of Steepest Descent, and Newton's Method.

The basis for comparison of the different propulsion schemes is the initial mass of the spacecraft injected into the transfer trajectory. All missions were designed to carry a net spacecraft mass of 250 kg. The minimum energy chemical mission is a 350 day

opportunity with a launch date of January 31, 2002. The total C3 and required rendezvous $\Delta V$ are 16.9824 $(km/s)^2$ and 3.0084 km/s, respectively. The initial spacecraft mass for this mission is 1008.47 kg. The nuclear thermal propulsion (NTP) mission launches on the next optimal impulsive opportunity: January 31, 2006. The corresponding energy requirement is 17.5308 $(km/s)^2$ and the rendezvous $\Delta V$ is 3.1498 km/s. This mission also has a 350 day time of flight. The injected NTP spacecraft mass is 1588.37 kg, which is 57.50% greater than the chemical spacecraft mass.

All low thrust missions analyzed in this thesis were found to have significant mass savings over the chemical mission. The lowest initial spacecraft mass was calculated for the NEP case with a flight time equal to the impulsive missions. The launch date for this minimum fuel opportunity is on July 22, 2005, and the resulting initial mass of 406.63 kg is 69.68% less than the chemical mass. However, this mass is considered to be extremely optimistic and based on far-term assumptions. The other power-limited scheme considered in this study is solar electric propulsion. This option launches from Earth on June 30, 2017 and arrives at Orpheus after 350 days on June 15, 2018. The corresponding injected spacecraft mass is 736.84 kg, which is 26.93% less than that for the chemical mission. Solar sail propulsion provides the greatest mass benefit of the near-term schemes considered. The optimal sail mission has a launch date of March 27, 2018 and a flight time of 378 days, which is comparable to the impulsive transfer times. These results are given for a sail with a characteristic acceleration of 1 $mm/s^2$ and translate into an initial spacecraft mass of 481.52 kg. This represents a mass savings of 52.25% over the optimal chemical spacecraft.

Analysis of the various mission scenarios suggests that solar sail propulsion represents a viable near-term technology which offers large cost advantages over existing technologies. Also, due to the reusability of these sails, this propulsion scheme is ideally suited for asteroid mining. These combined factors may provide economically feasible

interplanetary missions with commercial cooperation along with promoting further space exploration. The author therefore suggests that serious development programs for solar sail propulsion systems be initiated immediately.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

## LIST OF SYMBOLS AND ACRONYMS

| | |
|---|---|
| $a$ | Semi-major axis |
| $\bar{a}$ | Acceleration vector |
| $a_o$ | Solar sail characteristic acceleration |
| AU | Astronomical unit |
| $a_r$ | Radial acceleration |
| $a_\phi$ | Tangential acceleration |
| $a_\psi$ | Normal acceleration |
| $a_t$ | Magnitude of thrust acceleration |
| c | Exhaust velocity |
| cl | Chord length between initial and final radial positions |
| C3 | Square of the hyperbolic excess velocity |
| e | Eccentricity |
| $F_t$ | Thrust force |
| g | Acceleration due to gravity at the Earth's surface (9.81 $m/s^2$) |
| GM | Universal gravitational constant times the mass of the Sun |
| h | Angular momentum |
| H | Hamiltonian |
| HCA | Heliocentric angle |
| i | inclination |
| $\hat{i}$ | Unit vector in the direction of incident sunlight |
| Isp | Specific Impulse |
| J | Functional to be minimized |
| JPL | Jet Propulsion Laboratory |
| K | Battin defined continued fraction |

| | |
|---|---|
| $k_s$ | Solar sail structural factor |
| L | Battin defined variable |
| m | Mass |
| $\dot{m}$ | Rate at which spacecraft mass changes |
| M | Spacecraft mass |
| M1 | Initial spacecraft mass |
| M2 | Final spacecraft mass |
| Meng | Propulsion engine mass |
| Mfrig | Propellant refrigeration system mass |
| Mnet | Net spacecraft mass |
| Mp | Propellant mass |
| $\dot{m}_p$ | Propellant mass flow rate |
| Mpow | Solar electric power system mass |
| Mprop | Propulsion system mass |
| Ms | Solar sail mass |
| Mt | Solar electric thruster system mass |
| Mtnk | Propellant tank mass |
| Mw | Nuclear electric power and thruster system mass |
| n | Mean motion |
| $\hat{n}$ | Unit vector normal to the solar sail's surface |
| NASA | National Aeronautics and Space Administration |
| NERVA | Nuclear Engine for Rocket Vehicle Application |
| NEP | Nuclear electric propulsion |
| NTP | Nuclear thermal propulsion |
| OMS | Orbital maneuvering system |
| p | Orbital parameter |

| | |
|---|---|
| P | Exhaust jet power |
| PBR | Particle bed reactor |
| Pmax | Power rating of nuclear electric power supply |
| $P_o$ | Maximum power produced by solar arrays at Earth's radial position |
| $P_r$ | Auxiliary variable, costate of radius |
| $P_{rad}$ | Solar radiation pressure |
| $P_\phi$ | Auxiliary variable, costate of celestial longitude |
| $P_\psi$ | Auxiliary variable, costate of celestial latitude |
| $P_{V_r}$ | Auxiliary variable, costate of radial velocity |
| $P_{V_\phi}$ | Auxiliary variable, costate of tangential velocity |
| $P_{V_\psi}$ | Auxiliary variable, costate of normal velocity |
| $Q_i$ | Generalized force or moment |
| $q_i$ | Generalized coordinate |
| r | Radial distance |
| $r_e$ | Average radial distance from the Sun to the Earth (1 AU) |
| $R_f$ | Reflectivity of sail material |
| $r_o$ | Reference distance of 1 AU from the sun |
| $r_{op}$ | Mean radius between initial and final positions |
| RTG | Radioisotope thermoelectric generator |
| s | Semi-perimeter |
| S | Solar sail surface area |
| SEP | Solar electric propulsion |
| SNAP | Space nuclear auxiliary power |
| SNRE | Small nuclear rocket engine |
| t | time |
| T | Kinetic energy |

| | |
|---|---|
| $T_p$ | Battin defined variable (parabolic case) |
| TOF | Time of flight |
| $V_r$ | Radial velocity |
| $V_\phi$ | Tangential velocity |
| $V_\psi$ | Normal velocity |
| x | Battin defined variable |
| $x_i$ | Rectilinear x component of position |
| y | Battin defined variable |
| $y_i$ | Rectilinear y component of position |
| $z_i$ | Rectilinear z component of position |
| $\alpha$ | Specific mass |
| $\beta$ | Sail clock angle |
| $\Delta V$ | Required velocity change in propulsive maneuver |
| $\delta$ | Sail material thickness |
| $\rho$ | Sail material density |
| $\lambda$ | Battin defined parameter |
| $\eta$ | System efficiency |
| $\eta_s$ | Relative solar array efficiency |
| $\sigma$ | Ratio of spacecraft mass after/before propulsive maneuver |
| $\theta$ | Sail cone angle |
| $\phi$ | Celestial longitude |
| $\Pi$ | Sum of the longitude of ascending node and the argument of the periapsis |
| $\psi$ | Celestial Latitude |
| $\Lambda$ | Sail areal density (including support structure) |
| $\omega$ | Argument of the periapsis |
| $\Omega$ | Longitude of the ascending node |

| | |
|---|---|
| $\xi$ | Battin defined continued fraction |

# 1. INTRODUCTION

## 1.1 Statement of Problem

In view of current budget cuts and project cancellations, America's space program must explore avenues which provide foreseeable economic returns as well as scientific benefits for mankind. One class of missions which satisfies these requirements is an asteroid rendezvous mission. This mission can act as a precursor to more profitable ventures such as mining raw materials from the planetoids and may thus stimulate commercial activity in space. In order to achieve this goal, the cost of the mission must be kept to a minimum. Therefore, the objective of this thesis is to investigate and compare propulsion options which may provide cost advantages over existing chemical rocket technology. Promising propulsion schemes appear to be nuclear thermal, nuclear electric, solar electric, and solar sail systems.

NASA is planning a new program to promote low-cost planetary missions. This venture is called the "Discovery" program and is scheduled to begin in 1994 (1). In October 1991, the Discovery Science Working Group recommended a detailed reconnaissance of a near-Earth asteroid as the first mission in this program. Rendezvous with the target is required to observe the asteroid at close range for extended periods of time. The objectives of the mission are to investigate surface morphology through comprehensive imaging, measure the asteroid's surface composition, and determine gross physical properties such as size, shape, mass, density, and spin state.

Asteroids are believed to consist of materials left over from the formation of the solar system that were not swept up by the planets (2). Therefore, investigating these ancient bodies may provide clues as to the origin of our solar system. Also, asteroids consist of metals and other minerals used extensively on Earth. While some asteroids are made of rock, others consist of surprisingly pure nickel-iron alloys, water, and carbon

compounds. These carbonaceous planetoids may well be the most useful sources in the solar system for oil, synthetics, and even food. However, possibly the most desirable resource provided by asteroids is water.

To design the minimum cost mission, computational procedures have been developed to model the three dimensional interplanetary flight from Earth to the asteroid. The procedures calculate the optimal launch dates and flight times for the spacecraft using the various propulsion schemes listed above. The basis for comparison of the different systems is the total mass of the spacecraft injected into the interplanetary trajectory. For a given payload, this initial wet mass dictates the size of the launch vehicle and the amount of fuel required to complete the mission and is, thus, directly related to the cost of the mission. The propulsion option with the lowest initial spacecraft mass is selected to minimize the total mission cost.

The object of this investigation is the near-Earth asteroid Orpheus, which has an orbital inclination of only 2.69°, a radius of approximately 0.4 km, and a semi-major axis of 1.21 A.U. (Epoch = November 5, 1990). Due to its proximity to Earth, Orpheus is a potential candidate for a low cost rendezvous mission and provides the lowest energy requirements of the search conducted by Farquhar [1]. Launch dates between the years 2000 - 2020 are analyzed to obtain the minimum cost opportunities for each propulsion scheme.

### 1.2 Overview of Chemical Propulsion

The most developed area of rocket propulsion is in chemical rocket technology. Around 1232 A.D., the Chinese developed the first rockets which were solid fueled and were used by the military to terrify such enemies as the Mongol invaders [2]. Typical fuel in these archaic "fire arrows" consisted of charcoal, saltpeter, and sulfur (gun

powder). This technology eventually made its way to the Western world through the Arab and Persian trade networks.

In the 20th century, two famous pioneers laid the groundwork for the development of rocket motors: Robert H. Goddard in the U.S. and Herman Oberth in Germany. Both experimented with liquid fueled rockets. The first modern liquid fueled rockets were developed by the Germans under the leadership of Dr. Wehner Von Braun and Major General Dr. Walter Dornberger. These "A-series" rockets received the designation of V-2 once they began to be used for military puposes during World War II (3). Representing an enormous step in rocket technology, the V-2 burned ethyl alcohol-water mixtures with cryogenic liquid oxygen and produced a thrust level of 246,400 N at sea-level.

After World War II, several German rocket scientists traveled to America and the Soviet Union, sparking competition between the space programs of both countries. This led to the launching of the first artificial Earth satellite (Sputnik, Soviet Union), the orbiting of the first man in space (Yuri Gregarin, Soviet Union), and eventually to the first manned mission to another celestial body, the moon (Apollo 8, U.S.A.). Since these landmark events, chemical propulsion has been the most widely used propulsion scheme in a variety of manned and unmanned missions.

## 1.3 Overview of Nuclear Propulsion

The idea of utilizing nuclear power as a form of propulsion can be traced back to the writings of Dr. Robert H. Goddard and others before World War II (4). Generally, there are two types of nuclear (fission) propulsion, both of which are considered in this thesis: nuclear thermal propulsion (NTP) and nuclear electric propulsion (NEP).

The basic features of a nuclear thermal rocket are similar to those of a chemical propulsion system. However, in an NTP device the propellant is heated by a nuclear

reaction before expanding through a nozzle. Depending on the reactor design, these systems are classified as solid core rockets, liquid core rockets, or gas core rockets. Liquid core and gas core engines have the potential of achieving much higher temperatures (and specific impulses) than the solid core system. However, these technologies are still in the conceptual phase and are considered to be far-term propulsion schemes.

Solid core engines have received the most attention and development. In 1955, the U.S. Air Force and Atomic Energy Commission started the Rover program to develop nuclear propulsion for use on intercontinental ballistic missiles. Engineering spinoffs from this program eventually led to the development of the solid core NERVA (Nuclear Engine for Rocket Vehicle Application) engine in the early 70's. This engine operated at a power level of 1500 MW and produced 333 kN of thrust [5]. A smaller device was designed by the Los Alamos National Laboratory for unmanned missions. This Small Nuclear Rocket Engine (SNRE) was based on Pewee technology and produced 72.6 kN of thrust at 370 MW.

Since the creation of NERVA and SNRE, several nuclear propulsion concepts have originated varying in power and size. One such design uses a particle bed reactor (PBR) incorporated into an orbital transfer vehicle [6]. Even though the smaller SNRE is suited for an unmanned venture such as an rendezvousing with an asteroid, the mass of this system is much larger than that of today's chemical boosters. However, the PBR engine has the potential of providing the low mass and high propulsive efficiency (Isp $\approx$ 1000 s) suitable for a low cost asteroid rendezvous mission.

Another propulsion scheme which uses nuclear reactor technology is nuclear electric propulsion. The NEP system utilizes the nuclear reactions in a much different fashion from NTP. In nuclear electric propulsion, the power produced by the reactor is used to drive an electric propulsion device, such as an ion, magnetoplasmadynamic, or

arcjet thruster. These systems provide a much higher Isp than the NTP rockets but at a greatly reduced thrust level.

Reactors for nuclear power applications in space were extensively developed in the U.S. from the early 1950's until 1973 (7). The Space Nuclear Auxiliary Power (SNAP) programs spawned three reactors which were ground tested for more than 10,000 hours each using zirconium hydride fuel in the 800 - 1000 K range: the SNAP 2, 10A, and 8 reactors. The SNAP-10A was flight tested in 1965 and has remained the only reactor to be launched by the U.S. to this date. On the other hand, NASA has flown several Radioisotope Thermoelectric Generators (RTG's) on planetary spacecraft.

Today, the SP-100 reactor is the focus of the U.S. space nuclear reactor program. This device is a heat pipe cooled, fast spectrum reactor with an electrical power output of 100 kW. The maximum core heat pipe temperature is 1560 K. The reactor used for the NEP missions in this thesis is assumed to be based off the SP-100 design and contains a potassium Rankine dynamic power conversion system. The specific mass assumed for this propulsion/power system is 10 kg/kW (8).

### 1.4 Overview of SEP and Solar Sail Propulsion

"Mission studies at JPL and elsewhere have demonstrated the potential benefits of using a spacecraft powered by Solar Electric Propulsion for many planetary missions, in particular small body rendezvous missions" (9). Like the NEP devices, these SEP systems utilize electric thrusters as a means of propelling the spacecraft to the desired destination. However, the power is supplied to the thrusters by an array of solar cells and , therefore, depends on the distance of the spacecraft from the sun. Since SEP systems derive their power from the energy of the sun, heavy nuclear reactors or battery cells are not required. Therefore, solar electric propulsion is ideally suited for the design of low mass rendezvous missions to targets with semi-major axes of 2.5 A.U. or less.

The SEP system used for the Orpheus rendezvous mission analyzed in this thesis is assumed to provide a power of 5 kW at a distance of 1 A.U. from the sun and has a system specific mass of 30 kg/kW. Sauer [9] states that an array power level of 5 kW is suitable for delivering a small (net mass ≈ 300 kg), low cost spacecraft to inner mainbelt asteroids.

Another concept producing thrust which is dependent upon the spacecraft's position relative to the sun is the solar sail. The principal behind solar sailing is very simple. Photons emitted from the sun reflect off the sails, which are essentially very large mirrors, thereby imparting momentum to the spacecraft. The concept of using light to propel ships with "photonic engines" was first mentioned by Konstantine Tsiolkovsky, the Russian founder of the modern science of astronautics [10]. Another Russian, Frederickh Tsander, was the first to propose the idea of solar sailing, which uses sunlight as a means of thrusting a spacecraft through space. The first person to address this concept in a technical publication was an American named Carl Wiley (writing as Russell Saunders). His nonfiction article "Clipper Ships of Space" appeared in the May 1951 issue of *Astounding Science Fiction* [11]. The first professional publication appeared in 1958 by Richard Garwin.

In 1976 while working for the Battelle Columbus Laboratories, Jerome Wright discovered a trajectory which could rendezvous a solar sail spacecraft with Halley's comet with a launch date in late 1981 or early 1982. This motivated JPL to initiate a program to develop a Haley's comet mission using solar sail technology. Another concept being proposed for the Haley mission was solar electric propulsion by NASA Lewis. Competition escalated between these two programs until 1977 when NASA selected the SEP mission option due to the lack of the technological maturity of solar sails.

Until recently, solar sail technology has received little attention. In 1993, the Russian space program launched a large reflecting mirror into orbit and demonstrated the feasibility of using solar reflectors for thrust. The World Space Foundation has constructed and deployed a prototype sail in a successful ground test but is presently attempting to reach an agreement with NASA and other organizations to flight test their ideas.

The driving factor behind the use of solar sails is that they require no propellant; the thrust is produced by sunlight. Therefore, these propulsive devices have an infinite specific impulse. However, the thrust acceleration is very low, resulting in long trip times. Also, the sails tend to be quite large (on the order of 1 $km^2$). For unmanned missions in which the flight time is not critical, solar sails still provide an attractive means of interplanetary travel and, once constructed, may be returned to earth and reused for other missions. This capability makes the solar sail a very desirable option for asteroid mining; the sail may be used to transport supplies and mined material between the asteroid mining station and Earth without the need for propellant.

### 1.5 Assumptions Common to all Concepts

A patched conic approximation is assumed for all trajectory analyses in this investigation. The trajectory analysis is broken into three phases which separately consider the influence on the spacecraft due to the gravity fields of the departure planet, sun, and target asteroid. While in the parking orbit about the Earth, the spacecraft's motion is dictated by the inverse square gravity field of the planet. A chemical boost is supplied which carries the payload outside the sphere of influence of Earth's gravity and places it on the interplanetary trajectory between Earth and Orpheus. While on this heliocentric conic, the only gravitational force on the spacecraft is produced by the sun. If a parking orbit is desired about the target, another chemical boost is supplied to place

the spacecraft in an orbit governed by the target's gravitational field. However, since the size and mass of Orpheus is very small compared to the sun and surrounding planets, no parking orbit is necessary. The propulsion system merely matches the spacecraft's position and velocity (rendezvous) with that of the asteroid. The gravity of the asteroid is neglected, and the spacecraft follows Orpheus in its orbit, affected only by the gravity field of the sun. All of the gravity fields considered in this study are assumed to be inverse square fields.

In the missions analyzed in this report, the launch vehicle is assumed to inject the spacecraft directly into the interplanetary transfer trajectory. At the other end of the trajectory, a boost from the propulsion system is required to match the position and velocity of Orpheus. In the impulsive thrust cases, additional propellant is allocated for a 100 m/s midcourse correction and 100 m/s for orbital operations at the target. For the low thrust missions, the launch vehicle gives the spacecraft just enough energy for a parabolic escape from the Earth. Outside Earth's sphere of influence, the low thrust device "kicks in" and deposits the spacecraft in a spiraling trajectory to the rendezvous point. No second impulsive maneuver is required.

The net mass of a spacecraft is defined as the total mass minus the propulsion and propellant masses and incorporates the spacecraft bus and payload. In this study, all missions are assumed to carry the same net mass of 250 kg to Orpheus. This mass includes 50 kg of science payload required to accomplish the mission objectives stated in section 1.1 [1].

# 2. IMPULSIVE THRUST MISSIONS

## 2.1 General Theory

For preliminary mission analysis, high thrust propulsion systems may be modeled as impulsive devices. The thrust generated by such systems is assumed to produce the desired velocity changes instantaneously instead of accelerating the spacecraft over a specified firing period. Two propulsion schemes are investigated in this study which fall into this category: chemical propulsion and nuclear thermal propulsion.

With the patched conic assumption, a thrust pulse is applied while in a parking orbit about the departure planet. This maneuver places the spacecraft on an escape hyperbola which caries it beyond the planet's sphere of influence and then inserts the vehicle into the heliocentric coast trajectory. At the arrival planetoid, another impulsive thrust maneuver places the spacecraft into a desired parking orbit or simply matches the velocity of the target. The interplanetary coast trajectory is determined by solving the well-known Lambert's problem.

### 2.1.1 Definition of Lambert's Problem

According to Lambert's theory, the orbital transfer time between two points depends solely upon the semimajor axis of the connecting conic, the sum of the initial and final radial distances from the gravity center, and the chord length between the points. Therefore, specifying two position vectors and the corresponding orbital transfer time completely determines the ballistic trajectory between the endpoints. The task of determining this trajectory is called Lambert's problem.

The first substantial progress toward the solution of Lambert's problem was made by Carl Fredrich Gauss in the first year of the 19th century [12]. On New Year's day in 1801 while searching for the existence of a planet between the orbits of Mars and Jupiter,

Giuseppe Piazzi of Palermo observed the motion of an apparent comet approaching the sun. This comet actually turned out to be Ceres, the first of many asteroids discovered orbiting between Mars and Jupiter. The asteroid was only visible for one month before the sun's glare blocked out its view. The challenge of reacquiring the position of Ceres sparked Gauss into developing a method for determining the precise orbit and position of the target based on observations during a short period of time. After a year had elapsed since the disappearance of the asteroid, Ceres was found at the exact location predicted by Gauss.

Due to this monumental step in orbit determination, Lambert's problem is also referred to as the Gauss problem. The data used by Gauss consisted of right ascension and declination measurements taken at three observation times. His method may be simplified by using, instead, two position vectors and the time of flight between them. The determining of orbits given two positions and the transfer time is of paramount interest in modern astrodynamics. Today, the method is fundamental in the targeting of ballistic missiles and spacecraft and in determining redezvous orbits.

### 2.1.2 Modified Algorithm

Since the days of Gauss, several methods for the solution of Lambert's problem have been developed. The procedure used in the this study for ballistic orbit determination is identical to that formulated by Battin [13]. This method is a modified version of the original Gauss' Method. Continued fractions, rather than power series, are used to represent the hypergeometric functions to assure a more rapid convergence to the solution of the desired trajectory conic. Also, Battin removed the singularity encountered in Gauss' Method for orbits with 180 degree transfer angles.

The trajectory analysis begins by specifying the launch date and the desired time of flight to the target. This defines the initial and final positions ($r_1$ and $r_2$, respectively)

on the heliocentric interplanetary trajectory. Given these positions and the flight time, the following algorithm is used to determine the elements of the transfer conic.

Step 1: Calculate heliocentric angle (HCA) from the dot product of the initial and final position vectors:

$$\vec{r}_1 \bullet \vec{r}_2 = r_1 r_2 \cos(\text{HCA}) = x_1 x_2 + y_1 y_2 + z_1 z_2$$

Solving for HCA gives:

$$\cos(\text{HCA}) = \left( \frac{x_1}{r_1}\frac{x_2}{r_2} + \frac{y_1}{r_1}\frac{y_2}{r_2} + \frac{z_1}{r_1}\frac{z_2}{r_2} \right) \tag{2.1.1}$$

where:

$$\frac{x_i}{r_i} = \cos(\psi_i)\cos(\phi_i)$$

$$\frac{y_i}{r_i} = \cos(\psi_i)\sin(\phi_i)$$

$$\frac{z_i}{r_i} = \sin(\psi_i)$$

Step 2: Calculate chord length (cl) between $r_1$ and $r_2$ according to:

$$\text{cl} = \sqrt{r_1^2 + r_2^2 - 2 r_1 r_2 \cos(\text{HCA})} \tag{2.1.2}$$

Step 3: Calculate semi-perimeter (s) of the triangle $r_1 r_2 \text{cl}$:

$$s = \frac{r_1 + r_2 + cl}{2} \tag{2.1.3}$$

Step 4: Introduce new parameter, $\lambda$, defined by:

$$\lambda = \frac{\sqrt{r_1 r_2}\cos\left(\frac{HCA}{2}\right)}{s} \tag{2.1.4}$$

Step 5: Calculate radius ($r_{o_p}$) to the mean point between $r_1$ and $r_2$:

$$r_{o_p} = \frac{s(1+\lambda)^2}{4} \tag{2.1.5}$$

Step 6: Define dimensionless parameters T and m:

$$T = \sqrt{\frac{8GM}{s^3}}(TOF) \tag{2.1.6}$$

$$m = \frac{T^2}{(1+\lambda)^6} \tag{2.1.7}$$

Step 7: Define L as follows:

for $0 < HCA < \pi$:

$$L = \frac{\sin^2\left(\frac{HCA}{4}\right) + \tan^2(2\omega)}{\sin^2\left(\frac{HCA}{4}\right) + \tan^2(2\omega) + \cos\left(\frac{HCA}{2}\right)} \tag{2.1.8}$$

for $\pi < HCA < 2\pi$:

$$L = \frac{\cos^2\left(\frac{HCA}{4}\right) + \tan^2(2\omega) - \cos\left(\frac{HCA}{2}\right)}{\cos^2\left(\frac{HCA}{4}\right) + \tan^2(2\omega)} \qquad (2.1.9)$$

The expressions containing the defined quantity $\omega$ may be calculated using:

$$\tan^2(2\omega) = \frac{\frac{1}{4}\left(\frac{r_2}{r_1} - 1\right)^2}{\sqrt{\frac{r_2}{r_1}} + \frac{r_2}{r_1}\left(2 + \sqrt{\frac{r_2}{r_1}}\right)}$$

Step 8: Calculate T for the parabolic case according to:

$$T_p = \frac{4}{3}\left(1 - \lambda^3\right) \qquad (2.1.10)$$

Battin defines two new variables x and y, each having values dependent on the transfer conic. The value of x is computed with an iterative procedure which guesses an initial x and calculates the corresponding value for y. This y is then used to calculate a new value for x. The procedure is repeated until x converges to a solution of desired accuracy.

Step 9: Guess an initial value for x:

If $(T > T_p)$

$x = L$

else

$x = 0$

Step 10: Given x, solve the following cubic equation for the largest possible real root:

$$y^3 - y^2 - h_1 y^2 - h_2 = 0 \tag{2.1.11}$$

The coefficients h1 and h2 are calculated using the following relations:

$$h_1 = \frac{(L+x)^2(1+3x+\xi)}{(1+2x+L)[4x+\xi(3+x)]}$$

$$h_2 = \frac{m(x-L+\xi)}{(1+2x+L)[4x+\xi(3+x)]}$$

$\xi(x)$ in the above expressions is calculated using the continued fraction approach defined by Battin:

$$\xi(x) = \cfrac{8\left(\sqrt{1+x}+1\right)}{3+\cfrac{1}{5+\eta+\cfrac{\frac{9}{7}\eta}{1+\cfrac{\frac{16}{63}\eta}{1+\cfrac{\frac{25}{99}\eta}{1+\ddots}}}}} \tag{2.1.12}$$

where $\eta$ is defined as follows:

$$\eta = \frac{x}{\left(\sqrt{1+x}+1\right)^2} \tag{2.1.13}$$

The largest real root is then given by:

$$y = \frac{1+h_1}{3}\left(2 + \frac{\sqrt{1+B}}{1+2uK^2(u)}\right) \tag{2.1.14}$$

where:

$$u = \frac{B}{2\left(\sqrt{1+B}+1\right)} \quad \text{and} \quad B = \frac{27h_2}{4(1+h_1)^3}$$

The function K(u) appearing in (2.1.14) is evaluated from the continued fraction:

$$K(u) = \cfrac{\frac{1}{3}}{1+\cfrac{\frac{4}{27}u}{1+\cfrac{\frac{8}{27}u}{1+\cfrac{\frac{2}{9}u}{1+\cfrac{\frac{22}{81}u}{1+\ddots}}}}} \tag{2.1.15}$$

Step 11: Solve for new x:

$$x = \sqrt{\left(\frac{1-L}{2}\right)^2 + \frac{m}{y^2}} - \frac{1+L}{2} \tag{2.1.16}$$

If this new x differs from the previous value by more than the desired amount, repeat steps steps 10 and 11 until convergence to an acceptable accuracy is obtained.

Step 12: Given x, the semimajor axis (a), parameter (p), and the eccentricity (e) of the

orbit may be determined from the following expressions:

$$a = \frac{ms(1+\lambda)^2}{8xy^2} \tag{2.1.17}$$

$$p = \frac{2r_1r_2y^2(1+x)^2\sin^2\left(\frac{HCA}{2}\right)}{ms(1+\lambda)^2} \tag{2.1.18}$$

$$e = \sqrt{1-\frac{p}{a}} \tag{2.1.19}$$

Once a, e, and p are known, the other elements of the transfer conic may be obtained along with the velocities at the endpoints of the trajectory. The algorithm defined above is incorporated into the computer code given in Appendix A. This code is used to generate several trajectories for specified launch dates and flight times.

## 2.2 Chemical Propulsion

Chemical rockets fall into the category of energy-limited devices. In these rockets, energy is supplied to the propellants via an exothermic combustion reaction. The reactants consist of an oxidizer and the fuel. The combustion products are then expanded through a converging-diverging nozzle, thereby producing thrust. Therefore, the kinetic energy that the exhaust stream may attain is limited by the energy released by combustion.

The chemical orbital maneuvering system (OMS) used in this study is a single stage device consisting of the XLR-132 engine, which is being developed by Rocketdyne [14]. This 51.26 kg liquid propulsion system uses a $N_2O_4$/MMH bipropellant and produces a nominal thrust of 16.7 kN at a specific impulse of 340 s. The XLR-132 is

chosen due to its moderately high thrust level and modest weight, requiring shorter burn times (≈ 3 min.) than lighter, lower thrust engines (≈ 1 - 2 hours). Therefore, the impulsive thrust assumption still may be applied to this system with reasonable accuracy.

### 2.2.1 Trajectory Optimization

The optimizing criteria for impulsive trajectories is the minimization of the energy required for rendezvous. The parameter of importance in designing minimum energy missions is the square of the hyperbolic excess speed at the departure or target body. This quantity is referred to as C3 and is a measure of the energy costs of the impulsive maneuvers. Total C3, the sum of the departure and arrival values, is minimized to obtain the rendezvous trajectory with the lowest total energy requirement.

The method of obtaining the optimal launch dates and flight times is very simple. The program listed in Appendix A can be easily modified to generate several constant time of flight trajectories over the desired period of interest. The corresponding variation of C3's is examined to give the launch periods containing local minimum energy opportunities. Using an iterative search method, constant C3 curves are then generated over these periods for diminishing values of C3. From these energy contours, the optimal launch date and time of flight may be selected. Also, typical NASA 30 day launch windows may be superimposed on these graphs to illustrate the penalties involved in missing the optimal launch dates.

### 2.2.2 Spacecraft Mass Calculation

The total mass of the impulsive spacecraft consists of three parts: Mnet, Mprop, and Mp. Mnet is the net mass of the spacecraft and includes the spacecraft bus, scientific payload, structure, and all other subsystems required for power, telecommunications, etc. Mprop is the mass of the propulsion system, including the engine and dry propellant

storage tanks. Mp is the mass of the fuel required to complete the mission. Assuming that 1% of the fuel is unusable, and that the OMS/payload adapter mass is 2.5% of the net mass, gives the following relation for the initial wet mass of the spacecraft after injection into the transfer trajectory:

$$M_1 = 1.025Mnet + Mprop + 1.01Mp \tag{2.2.1}$$

Given a tankage factor of 0.115 [5], the mass of the propulsion system is:

$$Mprop = Meng + Mtnk \tag{2.2.2}$$

where Meng represents the engine mass, and the propellant tank mass is given by:

$$Mtnk = 0.115Mp$$

Substituting into (2.2.1) gives an initial mass of:

$$M_1 = 1.025Mnet + Meng + 1.125Mp \tag{2.2.3}$$

The spacecraft mass after rendezvous with the target is simply the initial mass minus the propellant consumed in the rendezvous burn and all corrective maneuvers. Therefore, the final spacecraft mass is given by:

$$M_2 = 1.025Mnet + Meng + 0.125Mp \tag{2.2.4}$$

Equation (2.2.5) is used to calculate the propellant mass required for a given propulsive maneuver $\Delta V$:

$$\sigma = \frac{M_1}{M_2} = e^{(\Delta V / gIsp)} \tag{2.2.5}$$

where $\sigma$ is the mass fraction. $M_1$ and $M_2$ are the masses of vehicle before and after the burn, respectively. Substituting (2.2.3) and (2.2.4) into (2.2.5) gives:

$$\frac{1.025Mnet + Meng + 1.125Mp}{1.025Mnet + Meng + 0.125Mp} = e^{(\Delta V / gIsp)} \tag{2.2.6}$$

Assuming that no mass other than propellant is dropped from the spacecraft during the mission, the $\Delta V$ in (2.2.6) represents the sum of the propulsion maneuvers required for the midcourse correction, rendezvous, and orbital operations at Orpheus. Solving for Mp in (2.2.6) gives the following result:

$$Mp = \frac{1.025\left(e^{\Delta V / gIsp} - 1\right)Mnet + \left(e^{\Delta V / gIsp} - 1\right)Meng}{1.125 - 0.125e^{\Delta V / gIsp}} \tag{2.2.7}$$

Therefore, given the net mass of the spacecraft along with the engine mass and desired velocity increments, the total propellant mass required to complete the chemical mission is calculated from equation (2.2.7).

### 2.2.3 Results

Prospective launch windows for a minimum energy Earth-Orpheus mission are identified by modifying the program in Appendix A to generate constant time of flight trajectories over a range of launch dates. The total C3 variation for several flight times over the 2000 - 2020 launch period is displayed in Figure 2.2.1. As seen in the graph,

desirable windows occur approximately every four years when the phase angle between Earth and Orpheus is the same. This interval is defined as the synodic period. If the optimal launch date is missed, the next launch window occurs four years later.

Examining the local minima in Figure 2.2.1 in more detail, three desirable opportunities appear over the period of interest. The minimum energy trajectories in 2002 and 2006 have a similar geometry with a flight time on the order of 400 days. The 2017 mission represents an interesting opportunity. The time of flight for this mission is very short, about 100 days. However, this shorter flight time is paid for by a larger $\Delta V$ requirement (discussed later).

In order to select the launch period containing the mission with the lowest energy cost, the three windows given above are analyzed in more detail. Constant C3 curves are generated over each period using the program described in section 2.2.1 and are shown in Figures 2.2.2 - 2.2.4. Note that typical NASA 30 day launch windows are superimposed over the energy contours to show the penalties for missing the optimal launch dates. However, only the optimal launch dates are considered in this study. The mission with the lowest energy requirement is the 2002 opportunity (see Figure 2.2.5), which launches from Earth on January 31, 2002 and rendezvous with Orpheus after 350 days. The propulsion requirements and trajectory conic data for this opportunity, along with information for the 2006 and 2017 missions, are listed in Table 2.2.1. The total C3 and required rendezous $\Delta V$ for the 2002 mission are 16.9824 $(km/s)^2$ and 3.0084 km/s, respectively. The 2017 mission has a flight time of only 110 days, but the velocity change required from the propulsion system is significantly greater at 3.5130 km/s.

The total $\Delta V$ produced by the propulsion system during the mission is the sum of the rendezvous maneuver, midcourse correction, and orbital $\Delta V$ correction at Orpheus. Fuel is allocated for 100 m/s velocity increments in both correction maneuvers. This results in a total $\Delta V$ of 3.2084 km/s required for the optimal rendezvous with Orpheus

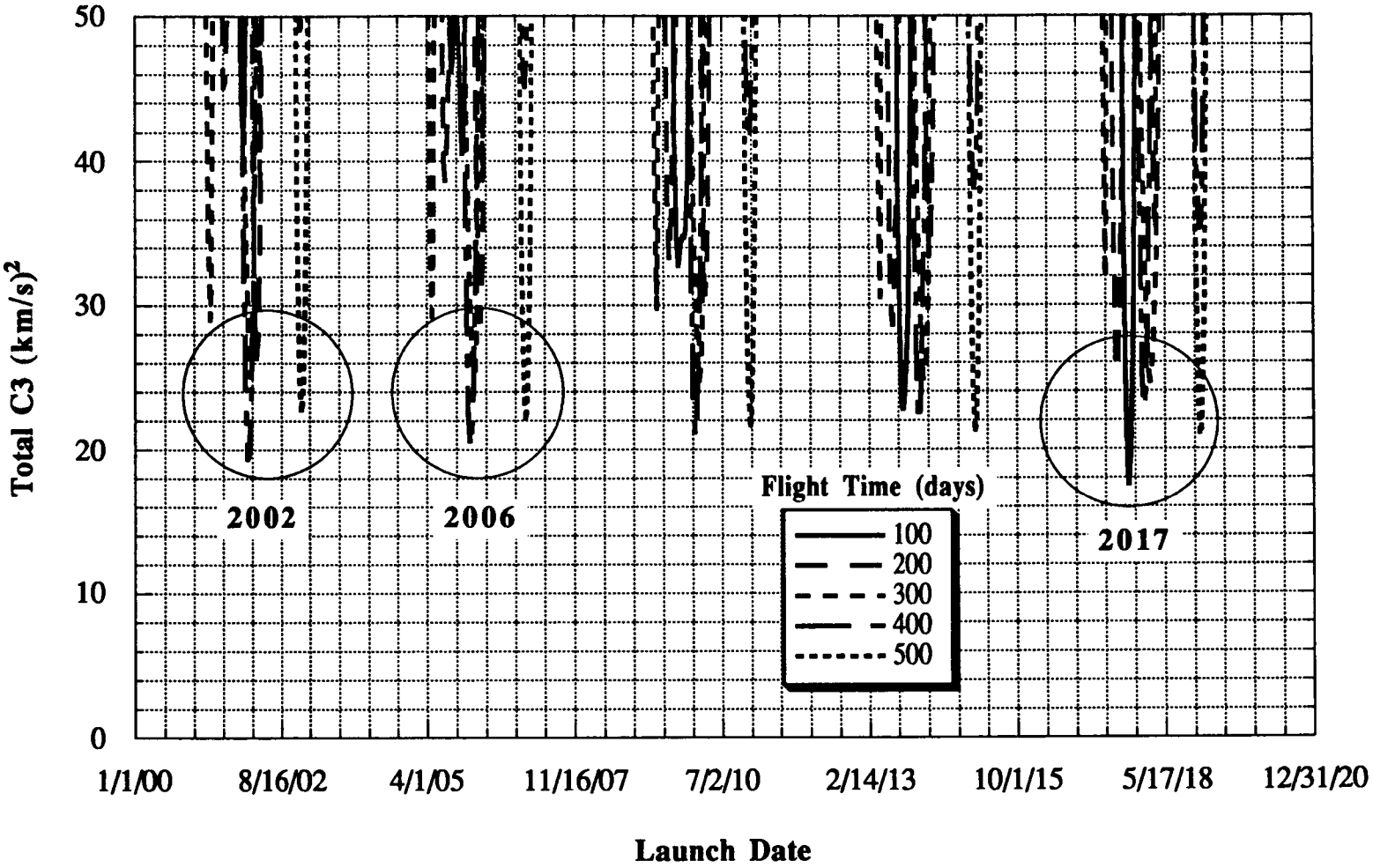


Figure 2.2.1: Variation of C3 with Launch Date for Various Flight Times (1/1/00 - 12/31/20)

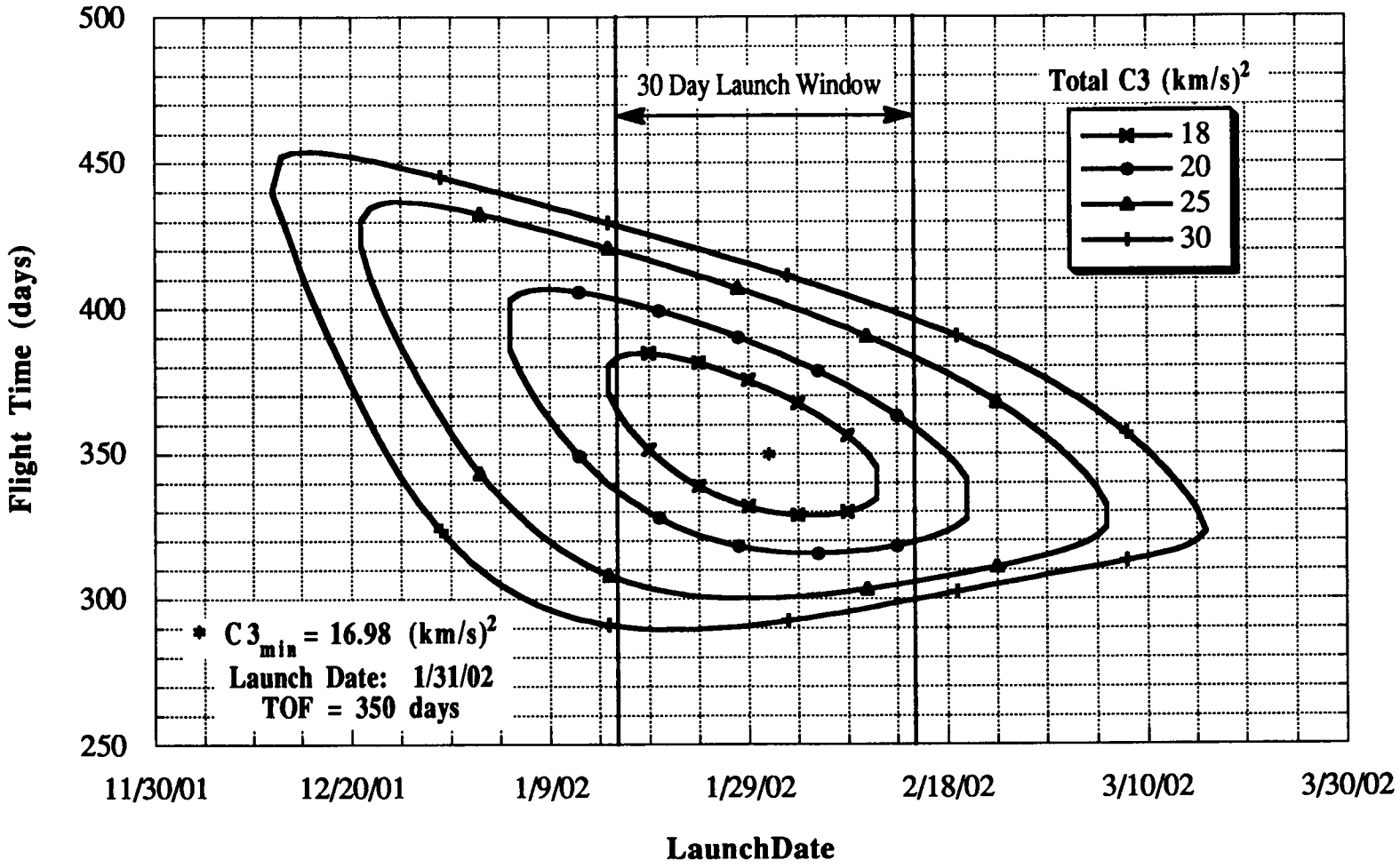


Figure 2.2.2: Total C3 Contours for 2002 Orpheus Mission

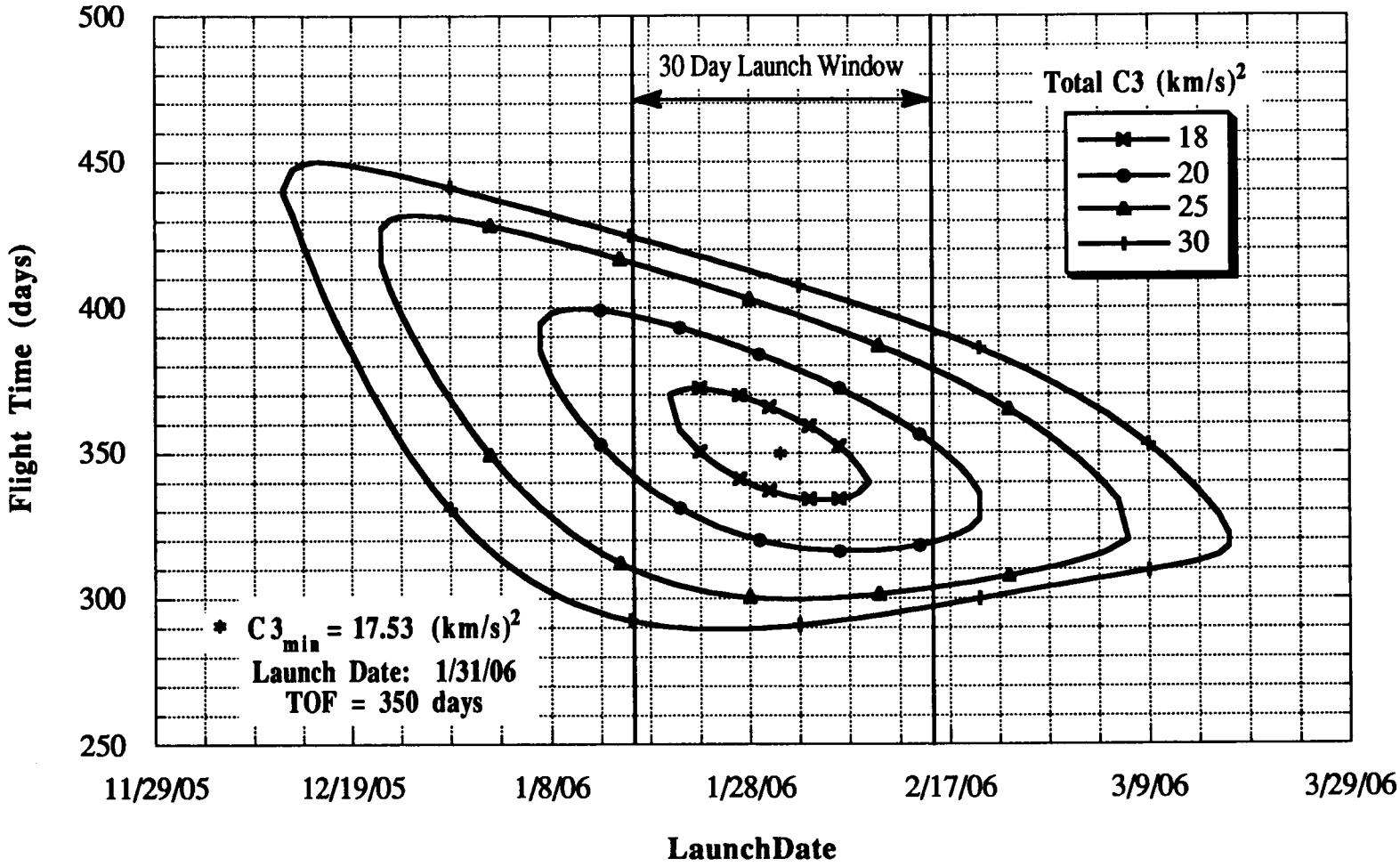


Figure 2.2.3: Total C3 Contours for 2006 Orpheus Mission

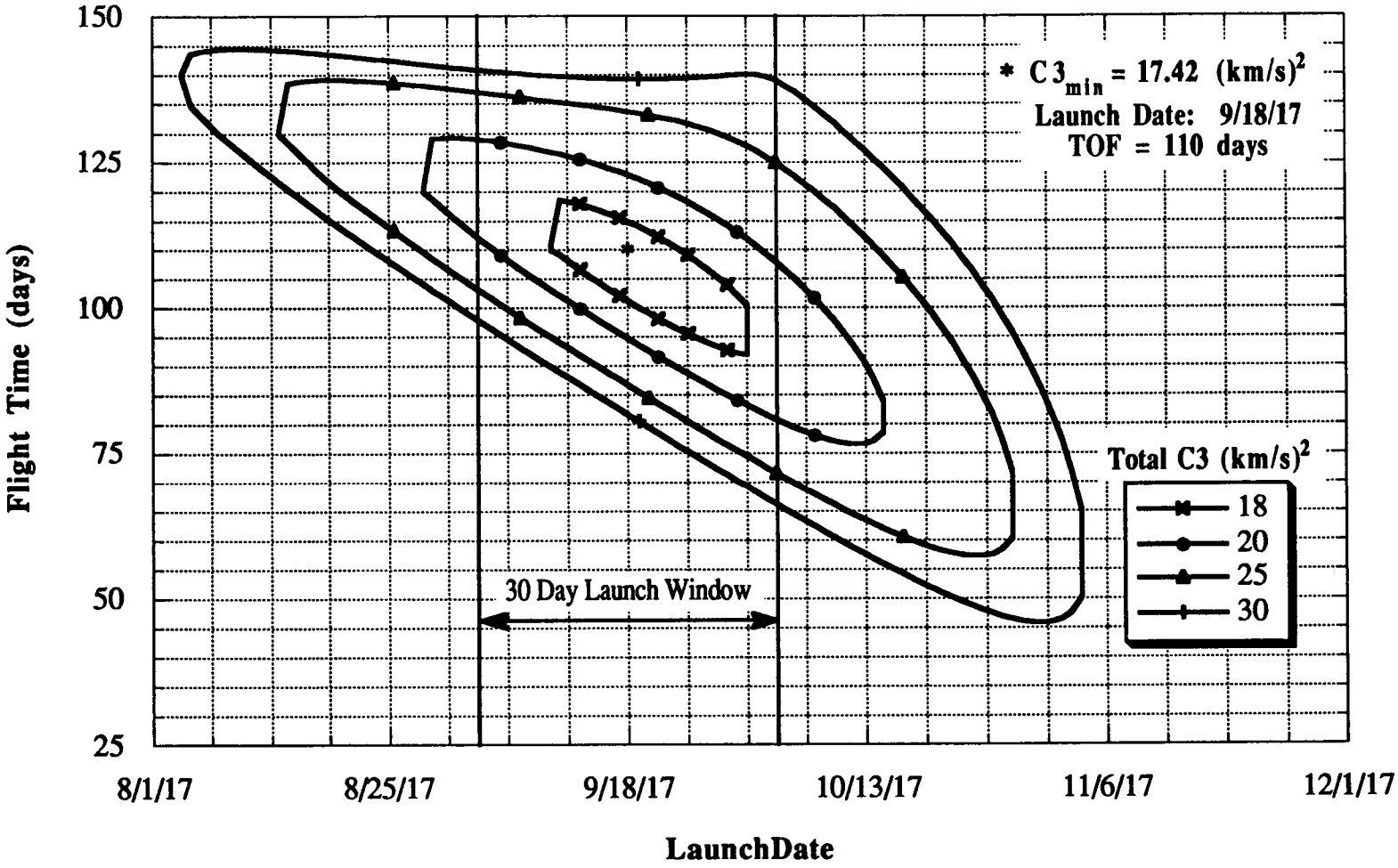


Figure 2.2.4: Total C3 Contours for 2017 Orpheus Mission

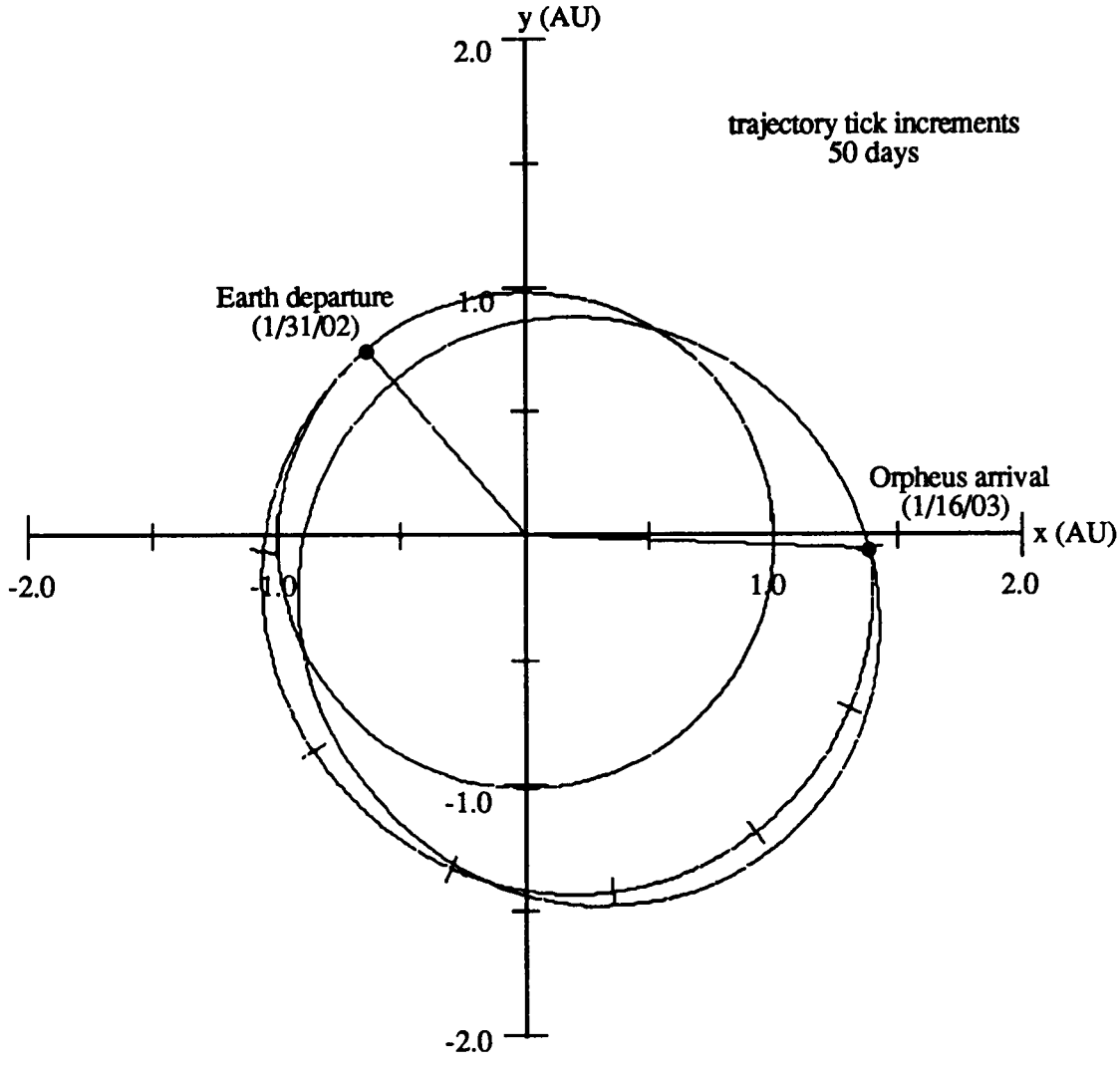


Figure 2.2.5: Ecliptic Projection of 350 Day Earth-Orpheus Impulsive Trajectory (1/31/02 - 1/16/03)

Table 2.2.1: Optimal Impulsive Earth-Orpheus Rendezvous Missions

| **Launch Date** | 1/31/02 | 1/31/06 | 9/18/17 |
|---|---|---|---|
| TOF (days) | 350 | 350 | 110 |
| Arrival Date | 1/16/03 | 1/16/07 | 1/6/18 |
| **INITIAL POSITION** | | | |
| Radius (AU) | 0.9851 | 0.9851 | 1.0050 |
| Celestial Longitude (deg) | 130.5068 | 130.5363 | 354.8689 |
| Celestial Latitude (deg) | 0.0000 | 0.0000 | 0.0000 |
| Velocity (km/s) | 33.0141 | 32.9624 | 28.0015 |
| Flight Path Angle (Deg.) | 0.1035 | 0.0180 | -1.6780 |
| C3 Required at Launch $(km/s)^2$ | 7.9321 | 7.6096 | 5.0804 |
| **ORBITAL ELEMENTS OF TRANSFER CONIC** | | | |
| Heliocentric Angle (Deg.) | 227.4271 | 229.4096 | 120.3134 |
| a (A.U.) | 1.24753 | 1.2416 | 0.9040 |
| e | 0.2103 | 0.2065 | 0.1155 |
| p (A.U.) | 1.1923 | 1.1886 | 0.8920 |
| h $(km^2/s)$ | 4.8655e+9 | 4.8578e+9 | 4.2082e+9 |
| n (1/s) | 1.4289e-7 | 1.4392e-7 | 0.0000 |
| i (Deg.) | 0.7099 | 0.5660 | 2.9922 |
| $\Omega$ (Deg.) | 310.5068 | 310.5363 | 174.8689 |
| $\omega$ (Deg.) | 179.4046 | 179.8948 | 346.9858 |
| $\Pi$ (Deg.) | 129.9114 | 130.4311 | 161.8547 |
| **FINAL POSITION** | | | |
| Radius (A.U.) | 1.3875 | 1.3727 | 0.8265 |
| Celestial Longitude (Deg.) | 357.9316 | 356.9446 | 115.2164 |
| Celestial Latitude (Deg.) | 0.5228 | 0.4298 | -2.5828 |
| Velocity (km/s) | 23.8244 | 24.0425 | 34.1383 |
| Flight Path Angle (Deg.) | -10.3131 | -10.2824 | -4.4498 |
| C3 Required at Rendezvous $(km/s)^2$ | 9.0503 | 9.9212 | 12.3410 |
| $\Delta V$ Required at Rendezvous (km/s) | 3.0084 | 3.1498 | 3.5130 |

using chemical propulsion. Given the net spacecraft mass of 250 kg along with the 51.26 kg engine mass, the propellant required to produce this total velocity change is calculated using the results of section 2.2.2 and is equal to 623.08 kg. The other calculated mass components are listed in Table 2.2.2. The initial wet mass of the spacecraft is 1008.47 kg, which can be injected into the desired transfer trajectory by a Delta II-7925 class launch vehicle (15).

Table 2.2.2: Mass Summary for Optimal Chemical Propulsion Mission

| Component | Mass (kg) |
|---|---|
| Spacecraft Bus and Payload | 250.00 |
| Propulsion System | 122.91 |
| Used Propellant | 623.08 |
| Unused Propellant | 6.23 |
| OMS/Payload Adapter | 6.25 |
| Total Injected Spacecraft Mass | 1008.47 |

## 2.3 Nuclear Thermal Propulsion

The other high thrust scheme considered in this study is nuclear thermal propulsion. In nuclear thermal rockets, themal energy from nuclear reactions (fission, fusion, etc.) is used to heat a working fluid with a low molecular weight such as hydrogen. These engines may fall into three classifications: solid-core, liquid-core, and gas-core rockets. Nuclear fission reactors containing solid fuel elements has received the most attention in the past and are at a higher level of development (NERVA). Therefore, solid-core nuclear propulsion is chosen for this investigation since it represents a near-term technology.

The nuclear orbital transfer vehicle (NOTV) analyzed in this study contains a particle bed reactor through which the working fluid is pumped. The NOTV is assumed to have a mass of 600 kg with an specific impulse of 1000 s [6]. This system is chosen due to its relatively low weight compared to the larger NERVA type engines intended for manned interplanetary flight.

### 2.3.1 Trajectory Optimization

As in chemical propulsion, the thrust produced by a nuclear thermal device is modeled as impulsive in preliminary mission design. Therefore, the optimal Earth-Orpheus rendezvous opportunities are the same as those obtained in section 2.2.1 for the missions using the chemical rocket technology.

### 2.3.2 Spacecraft Mass Calculation

The mass calculations for the NTP spacecraft are very similar to those for the chemical propulsion system (see section 2.2.2), except that a refrigerator must now be included to prevent the boiloff of the cryogenic liquid hydrogen fuel. The mass (in kg) of this sorption compressor device is given by the following relation taken from Frisbee [5]:

$$Mfrig = 45.9 + 1.74Mp^{2/3} \tag{2.3.1}$$

This mass is considered to be part of the nuclear thermal propulsion system. The tankage factor in the storage of the liquid hydrogen is 0.258. Substituting (2.3.1) and the new tankage factor into equation (2.2.2) gives:

$$Mprop = Meng + 0.258Mp + 1.74Mp^{2/3} + 45.9 \tag{2.3.2}$$

Again, the initial wet mass of the spacecraft is given by equation (2.2.1), which assumes that 1% of the fuel is unusable and that the OMS/payload adapter mass is equal to 2.5% of the net mass. Including the propulsion system mass given above into (2.2.1) results in the formula for the injected wet mass of the NTP spacecraft:

$$M_1 = 1.025Mnet + Meng + 1.268Mp + 1.74Mp^{2/3} + 45.9 \qquad (2.3.3)$$

Subtracting the propellant consumed in the rendezvous burn and correction maneuvers gives the final spacecraft mass after rendezvous with Orpheus:

$$M_2 = 1.025Mnet + Meng + 0.268Mp + 1.74Mp^{2/3} + 45.9 \qquad (2.3.4)$$

Equations (2.3.3) and (2.3.4) may be substituted into (2.2.5), which relates the spacecraft masses before and after the propulsive maneuvers:

$$\frac{1.025Mnet + Meng + 1.268Mp + 1.74Mp^{2/3} + 45.9}{1.025Mnet + Meng + 0.268Mp + 1.74Mp^{2/3} + 45.9} = e^{(\Delta V/gIsp)} \qquad (2.3.5)$$

Given the net spacecraft and engine masses, equation (2.3.5) is iteratively solved for the propellant mass required to produce the desired total $\Delta V$. Again, the spacecraft is assumed have a single stage nuclear propulsion system.

### 2.3.3 Results

As previously stated, the trajectory designs for the chemical propulsion and nuclear thermal propulsion options are identical. As a reminder, the optimum rendezvous mission to Orpheus launches from Earth on January 31, 2002 and reaches the destination on January 16, 2003 after a 350 day flight time. However, the earliest that an operable

nuclear engine may be available is in the year 2004 (16). Therefore, the 2006 opportunity is chosen for the NTP mission (see Table 2.2.1). The launch and arrival dates for this case is January 31, 2006 and January 16, 2007, respectively. Again, the optimal time of flight between Earth and Orpheus is 350 days. The total C3 for the NTP mission is 17.5308 $(km/s)^2$, while the total $\Delta V$ is 3.1498 km/s. The ecliptic projection of the optimal 2006 NTP mission is shown in Figure 2.3.1.

Even though the results of the dynamical optimization are nearly the same for both impulsive systems, the mass of the spacecraft injected into the transfer trajectory may vary by a large amount. This difference is largely affected by the much higher Isp of the NOTV (1000 s) along with the distinct weight of the particle bed reactor. Using the relations given in section 2.3.2, the spacecraft component masses are calculated for the 600 kg engine and 250 kg net mass. The resulting mass summary of the NTP vehicle for the 2006 mission is listed in Table 2.3.1. Powell (6) states that with future modifications the engine mass may be reduced to 480 kg and possibly to 200 kg. The mass results for these engine weights are also given in Table 2.3.1. The only NTP system mass that is comparable to the chemical spacecraft mass and that can be launched aboard a Delta II-7925 class launcher is the NTP spacecraft with the 200 kg engine. However, this greatly reduced mass is extremely optimistic for the 2006 Earth-Orpheus rendezvous mission.

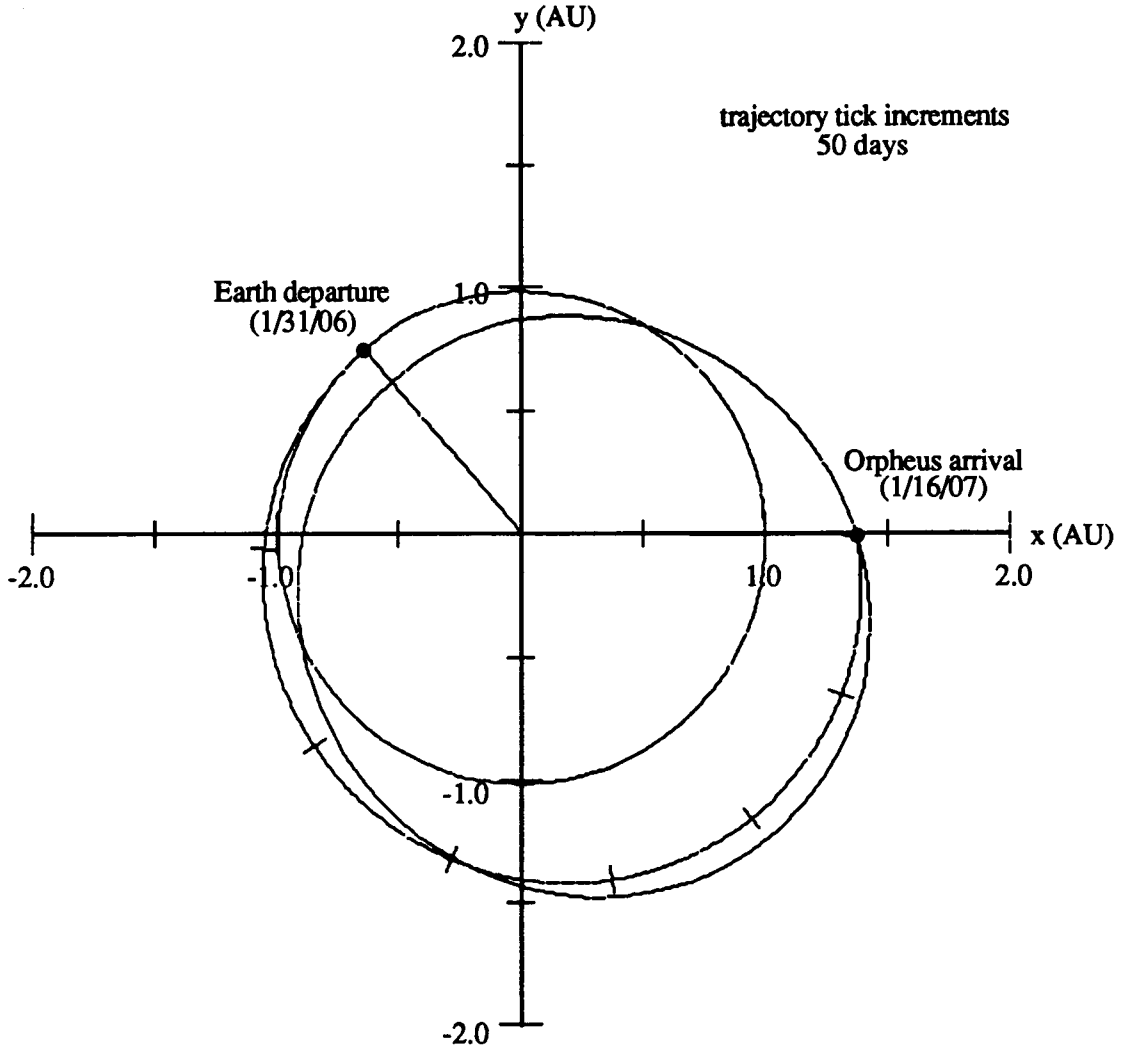


Figure 2.3.1: Ecliptic Projection of 350 Day Earth-Orpheus Impulsive Trajectory (1/31/06 - 1/16/07)

Table 2.3.1: Mass Summary for Optimal NTP Missions

| **Engine Mass (kg)** | 200 | 480 | 600 |
|---|---|---|---|
| **Component Mass (kg)** | | | |
| Spacecraft Bus and Payload | 250.00 | 250.00 | 250.00 |
| Propulsion System | 384.72 | 723.77 | 868.05 |
| Used Propellant | 261.95 | 400.51 | 459.47 |
| Unused Propellant | 2.62 | 4.01 | 4.60 |
| OMS/Payload Adapter | 6.25 | 6.25 | 6.25 |
| Total Injected Spacecraft Mass | 905.54 | 1384.54 | 1588.37 |

# 3. LOW THRUST MISSIONS

## 3.1 General Theory

The state equations which govern the motion of a low thrust spacecraft in an inverse square gravity field are derived using Lagrange's equation. To minimize the cost of the rendezvous mission, it is desired to minimize fuel expenditure, flight time, or some other characteristic functional. This is accomplished using the calculus of variations. Specifically, Pontryagin's Maximum Principle is utilized to obtain the co-state equations and desired control laws governing the optimal motion of the spacecraft. The problem of attaining the minimum fuel or minimum time rendezvous trajectory represents a two point boundary value problem which is solved with the aid of the numerical techniques described later in this section.

### 3.1.1 General Equations of Motion

The motion of a spacecraft in an inverse square gravity field is easily described utilizing the spherical coordinate system shown in Figure 3.1.1. By convention, the X axis lies along the line to Aries. The XY plane represents the ecliptic plane with Z perpendicular to the ecliptic in the galactic north direction. The angles $\phi$ and $\psi$ are commonly known as the celestial longitude and latitude, respectively.

The equations of motion for the spacecraft affected by non-conservative forces may be derived using Lagrange's equation:

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{q}_i}\right) - \frac{\partial T}{\partial q_i} = Q_i \qquad (3.1.1)$$

where T is the total kinetic energy, $q_i$ and $Q_i$ (i = 1, 2, ..., n) are the generalized coordinate and force acting on the spacecraft, and n represents the number of degrees of freedom.

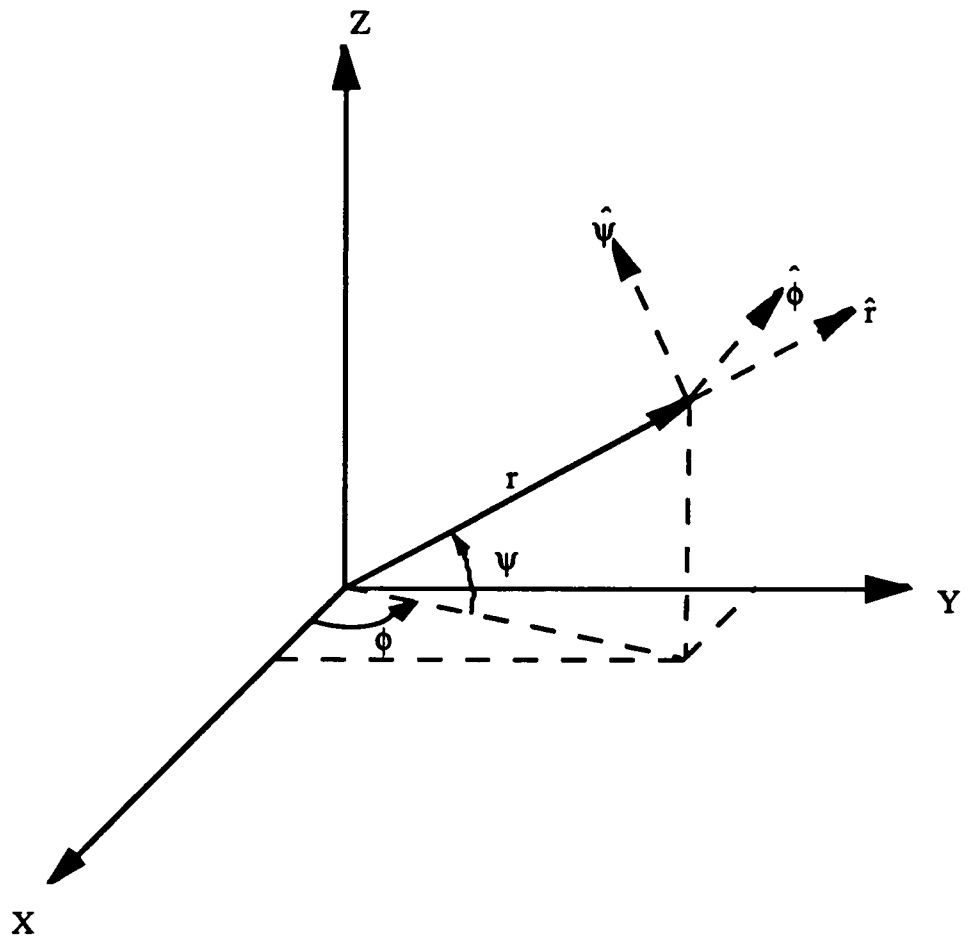


Figure 3.1.1: Spherical Coordinate System

The dynamical system considered in this analysis contains three degrees of freedom: r, $\phi$, and $\psi$. Therefore, r, $\phi$, and $\psi$ are suitable generalized coordinates which specify the spacecraft's position.

The kinetic energy is defined as:

$$T = \frac{1}{2} m\left(V_r^2 + V_\phi^2 + V_\psi^2\right) \tag{3.1.2}$$

where the velocity components in the spherical coordinate system are given by:

$$V_r = \dot{r} \tag{3.1.3}$$

$$V_\phi = r\cos(\psi)\dot{\phi} \tag{3.1.4}$$

$$V_\psi = r\dot{\psi} \tag{3.1.5}$$

Substituting (3.1.3) - (3.1.5) into (3.1.2) gives:

$$T = \frac{1}{2}m\left(\dot{r}^2 + r^2\cos^2(\psi)\dot{\phi}^2 + r^2\dot{\psi}^2\right) \tag{3.1.6}$$

This expression along with those for $Q_r$, $Q_\phi$, and $Q_\psi$ (given below) are used with (3.1.1) to obtain the governing equations for the system.

Lagrange's equation for $q_1 = r$ is:

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{r}}\right) - \frac{\partial T}{\partial r} = Q_r \tag{3.1.7}$$

For the interplanetary portion of the mission, only the motion about the sun is considered, neglecting the perturbations due to the other celestial bodies. Therefore, the only forces acting on the spacecraft are the force due to the sun's gravity and the thrust produced by the propulsion system. The generalized force in the r-direction is:

$$Q_r = -\frac{GMm}{r^2} + ma_r \tag{3.1.8}$$

where GM is the gravitational constant of the sun and $a_r$ is the thrust acceleration in the r-direction. Substituting (3.1.8) into (3.1.7) and dividing through by the spacecraft mass (m) results in the following second-order equation for r:

$$\ddot{r} - r\cos^2(\psi)\dot{\phi}^2 - r\dot{\psi}^2 = -\frac{GM}{r^2} + a_r \tag{3.1.9}$$

Lagrange's equations for the other generalized coordinates, $q_2 = \phi$ and $q_3 = \psi$, are listed below:

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{\phi}}\right) - \frac{\partial T}{\partial \phi} = Q_\phi \tag{3.1.10}$$

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{\psi}}\right) - \frac{\partial T}{\partial \psi} = Q_\psi \tag{3.1.11}$$

Since $\phi$ and $\psi$ represent angular displacements, the corresponding generalized force terms are torques. These moments are independent of gravity, which acts only in the radial direction, and depend solely on the thrust produced by the spacecraft. Therefore, the generalized forces in the $\phi$ and $\psi$ directions are:

$$Q_\phi = mr\cos(\psi)a_\phi \tag{3.1.12}$$

and

$$Q_\psi = mra_\psi \tag{3.1.13}$$

Substituting into (3.1.10) - (3.1.11) gives the equations describing the angular motion:

$$r\cos(\psi)\ddot{\phi} + 2\dot{r}\dot{\phi}\cos(\psi) - 2r\dot{\phi}\dot{\psi}\sin(\psi) = a_\phi \tag{3.1.14}$$

$$r\ddot{\psi} + 2\dot{r}\dot{\psi} + r\dot{\phi}^2\cos(\psi)\sin(\psi) = a_\psi \tag{3.1.15}$$

These three second-order differential equations, (3.1.9), (3.1.14), and (3.1.15), may be transformed to the system of six first-order equations given below. The resulting state equations describe the motion of a spacecraft utilizing any propulsion system within an inverse square gravity field.

$$\frac{dr}{dt} = V_r \tag{3.1.16}$$

$$\frac{d\phi}{dt} = \frac{V_\phi}{r\cos(\psi)} \tag{3.1.17}$$

$$\frac{d\psi}{dt} = \frac{V_\psi}{r} \tag{3.1.18}$$

$$\frac{dV_r}{dt} = \frac{V_\phi^2 + V_\psi^2}{r} - \frac{GM}{r^2} + a_r \tag{3.1.19}$$

$$\frac{dV_\phi}{dt} = -\frac{V_r V_\phi}{r} + \frac{V_\phi V_\psi \tan(\psi)}{r} + a_\phi \tag{3.1.20}$$

$$\frac{dV_\psi}{dt} = -\frac{V_r V_\psi}{r} - \frac{V_\phi^2 \tan(\psi)}{r} + a_\psi \tag{3.1.21}$$

### 3.1.2 Pontryagin's Maximum Principle

Of primary importance in low thrust trajectory optimization is the minimization of some functional (fuel expenditure, flight time, etc.). This class of problems belongs to those solved by the calculus of variations. In this study, Pontryagin's Maximum Principle is utilized to obtain the optimal control law which minimizes the desired quantity [17].

The integral functional to be minimized is denoted as J and is given by:

$$J = \int_{t_1}^{t_2} f^0[x(t), u(t)]dt \tag{3.1.22}$$

where x corresponds to the n state variables ($x^1$, $x^2$, ..., $x^n$), u denotes the r control variables , and $f^0$ is a given function. According to the maximum principle, an additional coordinate, $x^0$, must be added to the phase coordinates $x^1$, $x^2$, ..., $x^n$. This extra coordinate represents the law of variation of which has the following form:

$$\frac{dx^0}{dt} = f^0\left(x^1, x^2, ..., x^n, u\right) \tag{3.1.23}$$

Therefore, the complete system of n+1 state equations with r control variables is given by:

$$\frac{dx^i}{dt} = f^i\left(x^1, x^2, ..., x^n, u^1, ..., u^r\right) = f^i(x, u) \tag{3.1.24}$$

where i = 0, 1, ..., n. Note that the right hand side of (3.1.24) does not depend on $x^0$.

Along with the above system of equations, Pontryagin also considers an adjoint system of auxiliary variables $P_0$, $P_1$, ..., $P_n$:

$$\frac{dP_i}{dt} = -\sum_{\alpha=0}^{n} \frac{\partial f^\alpha(x,u)}{\partial x^i} P_\alpha \tag{3.1.25}$$

This system may be combined with (3.1.24) by introducing a function known as the Hamiltonian of the combined system. The Hamiltonian, denoted by H, is a function of the state and auxiliary variables along with the control functions.

$$H(P,x,u) = \sum_{\alpha=0}^{n} P_\alpha f^\alpha(x,u) \tag{3.1.26}$$

With the aid of H, the systems of equations given in (3.1.24) and (3.1.25) may be combined into the following Hamiltonian system:

$$\frac{dx^i}{dt} = \frac{\partial H}{\partial P_i}, \qquad i = 0, 1, ..., n \tag{3.1.27}$$

$$\frac{dP_i}{dt} = -\frac{\partial H}{\partial x^i}, \qquad i = 0, 1, ..., n \tag{3.1.28}$$

In order to obtain an optimal trajectory (minimize J), the Hamiltonian of the system must attain a maximum value which is constant and greater than or equal to zero. The relations for the optimal control laws, u(t), that satisfy this condition are obtained by maximizing the Hamiltonian with respect to each control variable.

$$\frac{\partial H}{\partial u^j} = 0, \quad j = 1, 2, ..., r \tag{3.1.29}$$

A special case of the theory considered above that is of interest in trajectory analysis is the minimization of the transfer time between two points. In this instance, the functional J takes the form:

$$J = \int_{t_1}^{t_2} f^0[x(t),u(t)]dt = t_2 - t_1 \tag{3.1.30}$$

where $f^0(x, u) = 1$. The corresponding Hamiltonian resulting from (3.1.26) is:

$$H(P,x,u) = P_0 + \sum_{\alpha=1}^{n} P_\alpha f^\alpha(x,u) = C1 \tag{3.1.31}$$

Equation (3.1.28) for i = 0 gives the adjoint equation corresponding to $P_0$:

$$\frac{dP_0}{dt} = -\frac{\partial H}{\partial x^0} = 0 \tag{3.1.32}$$

Therefore, $P_0$ is a constant and subtracting $P_0$ from both sides in (3.1.31) results in a new constant value for H. A new Hamiltonian system may thus be defined ignoring i = 0, which is now unnecessary.

$$H(P,x,u) = \sum_{\alpha=1}^{n} P_\alpha f^\alpha(x,u) \tag{3.1.33}$$

$$\frac{dx^i}{dt} = \frac{\partial H}{\partial P_i}, \qquad i = 1, 2, ..., n \tag{3.1.34}$$

$$\frac{dP_i}{dt} = -\frac{\partial H}{\partial x^i}, \qquad i = 1, 2, ..., n \tag{3.1.35}$$

The new Hamiltonian is maximized as before with the optimal control functions given by (3.1.29).

### 3.1.3 Optimal Trajectories (Two Point Boundary Value Problem)

Obtaining the optimal trajectory for the Hamiltonian system described above represents a two point boundary value problem. In this analysis, the hyperbolic excess speeds at the initial and final positions are assumed to be zero (parabolic departure from origin and rendezvous with the target). Therefore, the initial position and velocity

components are set equal to those of the earth at the time of departure. For rendezvous, the final spacecraft position and velocity must match Orpheus' position and velocity.

In order to solve the two point boundary value problem, the number of initial and final conditions must be equal. The final conditions are specified by the rendezvous requirement. The corresponding three position and three velocity requirements require six unknown initial conditions. These unknown quantities are the initial values for $P_r$, $P_{V_r}$, $P_{V_\phi}$, $P_\psi$, $P_{V_\psi}$ (discussed later in section 3.2.2). The additional guess may be the launch date or time of flight. For very low thrust systems which require additional flexibility in converging to rendezvous solutions (such as solar sails), both the launch date and time of flight may be used as initial guesses. This requires an extra final condition. Since the value for the maximized Hamiltonian is a constant, the additional condition may be satisfied by specifying this value of the Hamiltonian. The only constrain on this specification is that the maximized Hamiltonian must be greater than or equal to zero.

The initial conditions that satisfy the rendezvous requirement are calculated using two iterative methods: the method of Steepest Descent and Newton's method [18]. The method of Steepest Descent is used due to its global nature. However, this method converges only linearly to the solution. Newton's method provides a quadratic convergence but requires initial values that are very close to the solution. Therefore, the method of Steepest Descent is used first to obtain initial guesses that are close enough to the solution for Newton's method to converge to the optimal trajectory.

## 3.2 Nuclear Electric Propulsion

A nuclear electric propulsion (NEP) system belongs to the category of power-limited (LP) engines. With a chemical propulsion system, the exhaust speed depends on the energy per unit mass of the combustion products. For a power-limited device, energy

is supplied by the powerplant carried by the spacecraft. In this case, the powerplant is a nuclear reactor, which provides constant power to an electric propulsion device such as an arcjet, magnetoplasmadynamic (MPD), or ion thruster. Only one constraint is imposed upon this system: the power that goes into the kinetic energy of the expellant beam must be less than or equal to the total useful power available from the power supply, hence the name power-limited.

The NEP system assumed for this study uses a nuclear power source based in SP-100 reactor technology with a potassium Rankine dynamic power conversion system [8] and an efficiency of 70% [7]. A representative specific mass for this type of power/propulsion system is 10 kg/kW, which includes the reactor, shield, power conversion, heat rejection, power processing, and thruster subsystems. The propulsion system is assumed to be completely throttlable. In a throttled engine, the exhaust velocity and mass flow may be varied at will, as long as the power constraint is satisfied. The direction of the thrust is, thus, unconstrained. Also, no limit is imposed on the values of specific impulse produced by the propulsion system during the mission.

### 3.2.1 Equations of Motion

The equations of motion for a power-limited spacecraft are given by (3.1.16) - (3.1.21). For convenience in the numerical analysis, the equations are desired in nondimensional form. The following expressions, taken from Lebedev [19], relate the dimensional variables to their corresponding dimensionless values:

$$r = \frac{r^*}{r_o} \tag{3.2.1}$$

$$V = \frac{V^*}{\sqrt{\frac{GM}{r_o}}} \tag{3.2.2}$$

$$a = \frac{a^*}{\left(\frac{GM}{r_o^2}\right)} \tag{3.2.3}$$

$$t = \frac{t^*}{\sqrt{\frac{r_o^3}{GM}}} \tag{3.2.4}$$

where the asterisk denotes dimensional values. The radial scaling factor, $r_o$, corresponds to the radius of a circular reference orbit. This distance is taken to be the mean radius of the earth about the sun and has a value of 1.0 AU. The denominator in (3.2.2) represents the magnitude of the circular orbital velocity along the reference orbit. The scaling parameter in (3.2.3) is the acceleration produced by the gravitational center (Sun) at a radial distance of $r_o$. The scaling parameter for time corresponds to the period of revolution along the circular reference orbit divided by $2\pi$. Applying (3.2.1) - (3.2.4) to (3.1.16) - (3.1.21) gives the nondimensional equations of motion for a spacecraft using a power-limited propulsion system (or any other system) in an inverse square gravity field.

$$\frac{dr}{dt} = V_r \tag{3.2.5}$$

$$\frac{d\phi}{dt} = \frac{V_\phi}{r\cos(\psi)} \tag{3.2.6}$$

$$\frac{d\psi}{dt} = \frac{V_\psi}{r} \tag{3.2.7}$$

$$\frac{dV_r}{dt} = \frac{V_\phi^2 + V_\psi^2}{r} - \frac{1}{r^2} + a_r \tag{3.2.8}$$

$$\frac{dV_\phi}{dt} = -\frac{V_r V_\phi}{r} + \frac{V_\phi V_\psi \tan(\psi)}{r} + a_\phi \tag{3.2.9}$$

$$\frac{dV_\psi}{dt} = -\frac{V_r V_\psi}{r} - \frac{V_\phi^2 \tan(\psi)}{r} + a_\psi \tag{3.2.10}$$

### 3.2.2 Trajectory Optimization

In order to minimize the mission cost of a power-limited mission, it is desired to minimize the fuel expenditure of the propulsion system. This is synonymous to maximizing the payload or terminal mass for any given power supply mass. A functional must be derived which is characteristic of the fuel required along the interplanetary trajectory. Once given, this functional may be minimized using the optimization techniques outlined in section 3.1.2.

From Newton's second law, the thrust produced by the spacecraft is:

$$F_t = \dot{m}_p c \tag{3.2.11}$$

where $\dot{m}_p$ is the propellant mass flow rate and c is the exhaust speed. The power supplied to the exhaust beam is given by:

$$P = \frac{1}{2} F_t c \tag{3.2.12}$$

A useful relationship between the spacecraft masses at the initial and final positions on the transfer trajectory is obtained by eliminating c between (3.2.11) and (3.2.12). Also, note that the rate at which the spacecraft mass changes, $\dot{m}$, results solely from the propellant loss. Therefore, $\dot{m} = -\dot{m}_p$ where:

$$\dot{m}_p = \frac{F_t}{c} = \frac{F_t^2}{2P} = \frac{m^2 a_t^2}{2P} = -\dot{m}$$

or

$$\frac{a_t}{2P} = -\frac{\dot{m}}{m} = -\frac{d}{dt}\left(\frac{1}{m}\right) \tag{3.2.13}$$

where $a_t$ is the magnitude of the thrust accleration. Integrating (3.2.13) over the total trip time gives:

$$\frac{1}{M_2} - \frac{1}{M_1} = \int_0^T \frac{a_t^2}{2P} dt \tag{3.2.14}$$

where $M_1$ and $M_2$ are the initial and final masses, respectively. To maximize the terminal mass (i.e. payload), the beam power must be kept equal to the largest value available over the mission duration:

$$P = \eta P_{max} \tag{3.2.15}$$

where $P_{max}$ is the power rating and $\eta$ is the power to thrust conversion efficiency of the propulsion system in converting power from the supply to kinetic power in the beam [20]. Thus, equation (3.2.14) is rewritten as:

$$\frac{1}{M_2} - \frac{1}{M_1} = \frac{1}{2\eta P_{max}} \int_0^T a_t^2 dt \tag{3.2.16}$$

where $P_{max}$ and $\eta$ are assumed to be constant over the mission duration. Therefore, the problem of minimizing the fuel expelled by the rocket is reduced to minimizing the integral:

$$J = \frac{1}{2}\int_0^T a_t{}^2 dt \tag{3.2.17}$$

The functional given above is an important parameter in power-limited flight and represents the payload capability of the spacecraft.

The optimal thrust variation over the entire mission duration may be obtained by minimizing the functional J, given above. This is accomplished using the optimization techniques outlined in section 3.1.2. Substituting the equations of motion, (3.2.5) - (3.2.10), along with the additional state equation :

$$\frac{dJ}{dt} = \frac{1}{2}a_t{}^2 = \frac{1}{2}\left(a_r{}^2 + a_\phi{}^2 + a_\psi{}^2\right) \tag{3.2.18}$$

into (3.1.26) gives the Hamiltonian of the dynamic power-limited system:

$$\begin{aligned} H = P_J\left[\frac{1}{2}\left(a_r{}^2 + a_\phi{}^2 + a_\psi{}^2\right)\right] + P_r\left[V_r\right] + P_\phi\left[\frac{V_\phi}{r\cos(\psi)}\right] + P_\psi\left[\frac{V_\psi}{r}\right] \\ + P_{V_r}\left[\frac{V_\phi{}^2 + V_\psi{}^2}{r} - \frac{1}{r^2} + a_r\right] + P_{V_\phi}\left[-\frac{V_r V_\phi}{r} + \frac{V_\phi V_\psi \tan(\psi)}{r} + a_\phi\right] \\ + P_{V_\psi}\left[-\frac{V_r V_\psi}{r} - \frac{V_\phi{}^2 \tan(\psi)}{r} + a_\psi\right] \end{aligned} \tag{3.2.19}$$

Applying (3.1.28) to the Hamiltonian results in the following differential equations for the auxiliary variables:

$$\frac{dP_J}{dt} = -\frac{\partial H}{\partial J} = 0 \tag{3.2.20}$$

$$\frac{dP_r}{dt} = -\frac{\partial H}{\partial r} = \frac{P_\phi V_\phi}{r^2\cos(\psi)} + \frac{P_\psi V_\psi}{r^2} + \frac{P_{V_r}\left(V_\phi^2 + V_\psi^2\right)}{r^2} - \frac{2P_{V_r}}{r^3} - \frac{P_{V_\phi} V_r V_\phi}{r^2} + \frac{P_{V_\phi} V_\phi V_\psi \tan(\psi)}{r^2} - \frac{P_{V_\psi} V_r V_\psi}{r^2} - \frac{P_{V_\psi} V_\phi^2 \tan(\psi)}{r^2} \quad (3.2.21)$$

$$\frac{dP_\phi}{dt} = -\frac{\partial H}{\partial \phi} = 0 \quad (3.2.22)$$

$$\frac{dP_\psi}{dt} = -\frac{\partial H}{\partial \psi} = -\frac{P_\phi V_\phi \sin(\psi)}{r\cos^2(\psi)} - \frac{P_{V_\phi} V_\phi V_\psi}{r\cos^2(\psi)} + \frac{P_{V_\psi} V_\phi^2}{r\cos^2(\psi)} \quad (3.2.23)$$

$$\frac{dP_{V_r}}{dt} = -\frac{\partial H}{\partial V_r} = -P_r + \frac{P_{V_\phi} V_\phi}{r} + \frac{P_{V_\psi} V_\psi}{r} \quad (3.2.24)$$

$$\frac{dP_{V_\phi}}{dt} = -\frac{\partial H}{\partial V_\phi} = -\frac{P_\phi}{r\cos(\psi)} - \frac{2P_{V_r} V_\phi}{r} + \frac{P_{V_\phi} V_r}{r} - \frac{P_{V_\phi} V_\psi \tan(\psi)}{r} + \frac{2P_{V_\psi} V_\phi \tan(\psi)}{r} \quad (3.2.25)$$

$$\frac{dP_{V_\psi}}{dt} = -\frac{\partial H}{\partial V_\psi} = -\frac{P_\psi}{r} - \frac{2P_{V_r} V_\psi}{r} - \frac{P_{V_\phi} V_\phi \tan(\psi)}{r} + \frac{P_{V_\psi} V_r}{r} \quad (3.2.26)$$

As seen in (3.2.20), the Hamiltonian does not explicitly depend on J. Therefore, $P_J$ is constant over the entire trajectory. This constant is arbitrarily assigned the value of -1 (21). The negative sign results from the fact that -J is maximized. According to (3.2.22), $P_\phi$ is also constant. If the initial or final celestial longitudes ($\phi$) are free to vary, $P_\phi$ is unimportant and is set equal to zero (Marec, p.38).

The controls for the power-limited spacecraft are the thrust accelerations $a_r$, $a_\phi$, and $a_\psi$. The relations for the optimal thrust program are generated by applying (3.1.29)

to the Hamiltonian for each of the controls. Taking the partial derivative of the Hamiltonian with respect to the radial control gives:

$$\frac{\partial H}{\partial a_r} = P_{V_r} + P_J a_r = 0$$

which is then solved for $a_r$:

$$a_r = -\frac{P_{V_r}}{P_J} = P_{V_r} \tag{3.2.27}$$

The optimal thrust accelerations in the $\phi$ and $\psi$ directions are obtained in a similar fashion:

$$a_\phi = P_{V_\phi} \tag{3.2.28}$$

$$a_\psi = P_{V_\psi} \tag{3.2.29}$$

As seen in equations (3.2.27) - (3.2.29), the optimal thrust acceleration for the power-limited system is equal to the primer vector $\mathbf{P_V} = \{ P_{V_r}, P_{V_\phi}, P_{V_\psi} \}$. Substituting the optimal thrust accelerations into the state equations, (3.2.5) - (3.2.10) and (3.2.18), and the co-state equations given by (3.2.20) - (3.2.26), results in a system of eleven differential equations (J, $P_J$, and $P_\phi$ are ignorable due to the independence of H on J and $\phi$).

Obtaining the minimum fuel trajectory for the Hamiltonian system described above represents the two point boundary value problem described in section 3.1.3. The rendezvous requirement completely specifies the final position and velocity of the

spacecraft (r, $\phi$, $\psi$, $V_r$, $V_\phi$, and $V_\psi$). For the power-limited system, the six unknown quantities are the initial values for $P_r$, $P_{V_r}$, $P_{V_\phi}$, $P_\psi$, $P_{V_\psi}$, and the launch date. The initial conditions that satisfy the rendezvous requirement are calculated using the numerical method in 3.1.3. By specifying a desired time of flight, the above procedure calculates the initial auxiliary variables and launch date which give the minimum fuel rendezvous trajectory for that flight time.

The maximum Hamiltonian which results from the above boundary value problem is equal to a constant. Marec (p. 69) states that this value must be equal to the initial value of $\frac{1}{2}\left(a_r^2 + a_\phi^2 + a_\psi^2\right)$. Another requirement for an optimal solution is that the initial and final magnitudes of the thrust accelerations be equal.

### 3.2.3 Spacecraft Mass Calculation

Along with obtaining the minimum fuel trajectory, it is also desired to optimize the propulsion system mass. This may be accomplished separately from the dynamic problem described above. For the NEP spacecraft considered in this study, the initial mass consists of the net spacecraft and OMS/payload adapter, propulsion system, and the consumed and unused propellant (see equation (2.2.1)). The mass of the propulsion system is given by the following relation:

$$Mprop = Mw + Mtnk + Mfrig \tag{3.2.30}$$

where Mw represents the mass of the power and thruster systems. Using the same refrigeration system mass (in kg) as for the NTP vehicle and assuming a tankage factor of 0.15 [9] gives:

$$Mprop = Mw + 0.15Mp + 1.74Mp^{2/3} + 45.9 \tag{3.2.31}$$

The power-thruster system mass is assumed to be proportional to Pmax:

$$Mw = \alpha P_{max} \tag{3.2.32}$$

where the constant of proportionality, $\alpha$, is defined as the specific mass of the propulsion system and is assumed to include the nuclear reactor, shield, power conversion, heat rejection, power processing, and thruster subsystems. Substituting (3.2.32) into (3.2.16) and multiplying by $M_1$ gives:

$$\frac{M_1}{M_2} = 1 + \frac{M_1}{Mw}\frac{\alpha J}{2\eta} \tag{3.2.33}$$

where J is defined by equation (3.2.17).

Substituting (3.2.31) into equation (2.2.1) results in the relation for the wet mass for the NEP vehicle after launch:

$$M_1 = 1.025Mnet + Mw + 1.16Mp + 1.74Mp^{2/3} + 45.9 \tag{3.2.34}$$

The spacecraft mass after rendezvous with the target is equal to the initial mass minus the propellant consumed by the propulsion system. Therefore, the final spacecraft mass is given by:

$$M_2 = 1.025Mnet + Mw + 0.16Mp + 1.74Mp^{2/3} + 45.9 \tag{3.2.34}$$

Dividing by the initial mass and rearranging yields:

$$\frac{Miw}{M_1} = \frac{M_2}{M_1} - \frac{Mw}{M_1} \tag{3.2.35}$$

where all components independent of the power-thruster systems have been grouped into Miw:

$$Miw = 1.025Mnet + 0.16Mp + 1.74Mp^{2/3} + 45.9 \tag{3.2.36}$$

The mass ratio, (3.2.33), is substituted into (3.2.35) and then simplified to give:

$$\frac{Miw}{M_1} = \frac{M_w}{M_1}\left[\frac{1}{\frac{M_w}{M_1} + \gamma^2} - 1\right] \tag{3.2.37}$$

where the dimensionless parameter $\gamma$ is given by:

$$\gamma = \sqrt{\frac{\alpha J}{2\eta}} \tag{3.2.38}$$

The optimal power-thruster system mass is that which maximizes the payload capability. Since Miw consists of the payload and components independent of power and thrust, the relation for the best value of $Mw/M_1$ is obtained by taking the derivative of (3.2.37) with respect to $Mw/M_1$ and setting the result equal to zero. This gives:

$$\frac{Mw}{M_1} = \gamma - \gamma^2 \tag{3.2.39}$$

Substituting this result back into (3.2.37) yields the optimal $Miw/M_1$ ratio

$$\frac{Miw}{M_1} = (1-\gamma)^2 \tag{3.2.40}$$

To obtain the propellant required to deliver the optimized spacecraft to the target, equation (3.2.34) is divided by $M_1$ and substituting the optimal mass ratios:

$$\begin{aligned}\frac{Mp}{M_1} &= 1 - \frac{Miw}{M_1} - \frac{Mw}{M_1} \\ &= 1 - (1-\gamma)^2 - (\gamma - \gamma^2) \\ &= \gamma\end{aligned} \tag{3.2.41}$$

This result demonstrates that the problem of optimizing payload capability of the power-limited vehicle reduces to obtaining the minimum value of $\gamma$ (i.e. J) for the mission. The problem of calculating the intial spacecraft mass for a given mission (i.e. given $\gamma$) is now reduced to calculating the amount of fuel required to complete the mission. Multiplying (3.2.40) and (3.2.41) by $M_1$ and dividing gives:

$$\frac{1.025Mnet + 0.16Mp + 1.74Mp^{2/3} + 45.9}{Mp} = \frac{(1-\gamma)^2}{\gamma} \tag{3.2.42}$$

where (3.2.36) has been substituted back into Miw. Given the net spacecraft mass and minimized value of $\gamma$, equation (3.2.42) is iteratively solved for the propellant mass required for the minimum fuel rendezvous mission. The corresponding $M_1$, Mw, and Mprop are then calculated using equations (3.2.41), (3.2.39), and (3.2.31), respectively. The maximum power required to accomplish the mission is obtained from (3.2.32).

### 3.2.4 Results

The computer code listed in Appendix B is used to generate minimum fuel trajectories using constant power electric propulsion (typical of NEP missions). Once a desired transfer time is chosen, the program iterates to the optimum launch date. The launch dates which provide minimum fuel trajectories over the period of interest (2000 - 2020) along with the corresponding values for the minimized functional J are given in Table 3.2.1 for several flight times. As in the chemical propulsion trajectory analysis, optimal solutions occur every 4 years. Also note that as the time of flight increases, the value for J decreases. This diminishing fuel requirement is expected since lower thrust acceleration magnitudes occur over the mission duration for longer flight times. The increase due to integrating over a longer transfer time is offset by the reduction due to squaring these lower accelerations in J (see equation 3.2.17).

In order to compare an Orpheus rendezvous mission using NEP to a chemical propulsion mission, a flight time of 350 days is selected. This TOF corresponds to the minimum energy time of flight in the chemical case. For this transfer time, the opportunity with the lowest fuel requirement launches from LEO on August 3, 2001 and arrives at Orpheus on July 7, 2002; the resulting minimum value of J is 0.6969 $m^2/s^3$. However, as stated in section 2.3.3, suitable nuclear engine technology is not expected to be available until the year 2004. Therefore, the 2005 opportunity is chosen as the optimal NEP mission with J equal to 0.7020 $m^2/s^3$. The launch date is July 22, 2005, and the rendezvous date is July 7, 2006. The ecliptic projection of the interplanetary transfer for this mission is illustrated in Figure 3.2.1. The optimal thrust program and corresponding auxiliary variable histories which produce this trajectory are shown in Figures 3.2.2 and 3.2.3, respectively. Note the very low values for the thrust acceleration (in the range of $mm/s^2$).

Table 3.2.1: Minimum Fuel NEP Earth-Orpheus Rendezvous Missions

| Time of Flight (Days) | Launch Date | Payoff Function, J ($m^2/s^3$) |
|---|---|---|
| 150 | 11/16/01 | 2.6942 |
| | 11/4/05 | 2.6164 |
| | 10/24/09 | 2.4487 |
| | 10/11/13 | 2.1949 |
| | 9/26/17 | 1.8542 |
| 200 | 10/22/01 | 1.5081 |
| | 10/11/05 | 1.4742 |
| | 9/30/09 | 1.4151 |
| | 9/19/13 | 1.3340 |
| | 9/7/17 | 1.2336 |
| 250 | 9/27/01 | 1.0424 |
| | 9/17/05 | 1.0262 |
| | 9/7/09 | 1.0026 |
| | 8/27/13 | 0.9732 |
| | 8/16/17 | 0.9399 |
| 300 | 9/1/01 | 0.8141 |
| | 8/22/05 | 0.8093 |
| | 8/12/09 | 0.8030 |
| | 8/1/13 | 0.7964 |
| | 7/20/17 | 0.7906 |
| 350 | 8/3/01 | 0.6969 |
| | 7/22/05 | 0.7020 |
| | 7/10/09 | 0.7087 |
| | 6/26/13 | 0.7177 |
| | 6/9/17 | 0.7295 |
| 400 | 3/9/01 | 0.6140 |
| | 3/16/05 | 0.6286 |
| | 3/27/09 | 0.6490 |

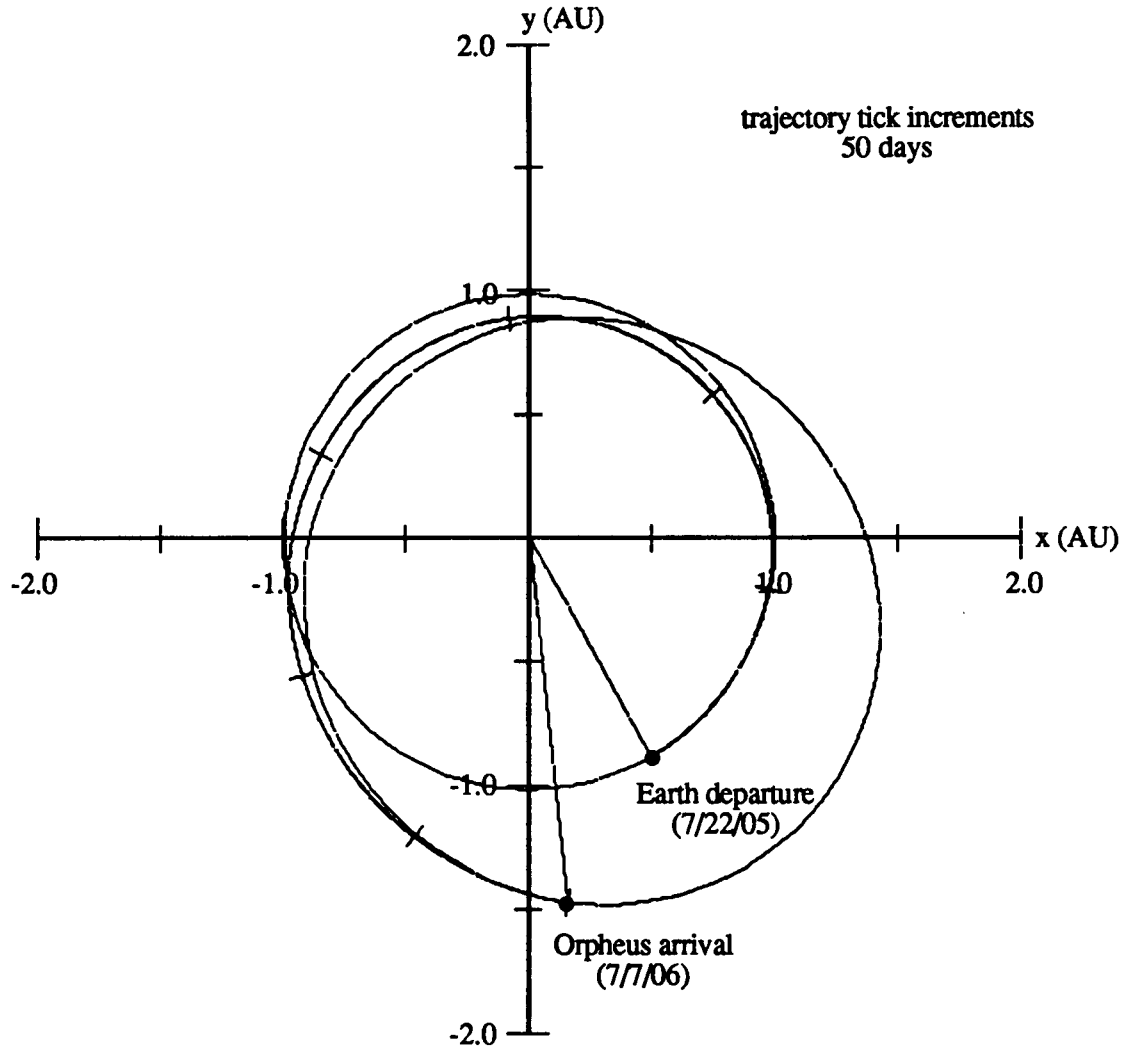


Figure 3.2.1: Ecliptic Projection of Optimal 350-day NEP Earth-Orpheus Rendezvous Trajectory

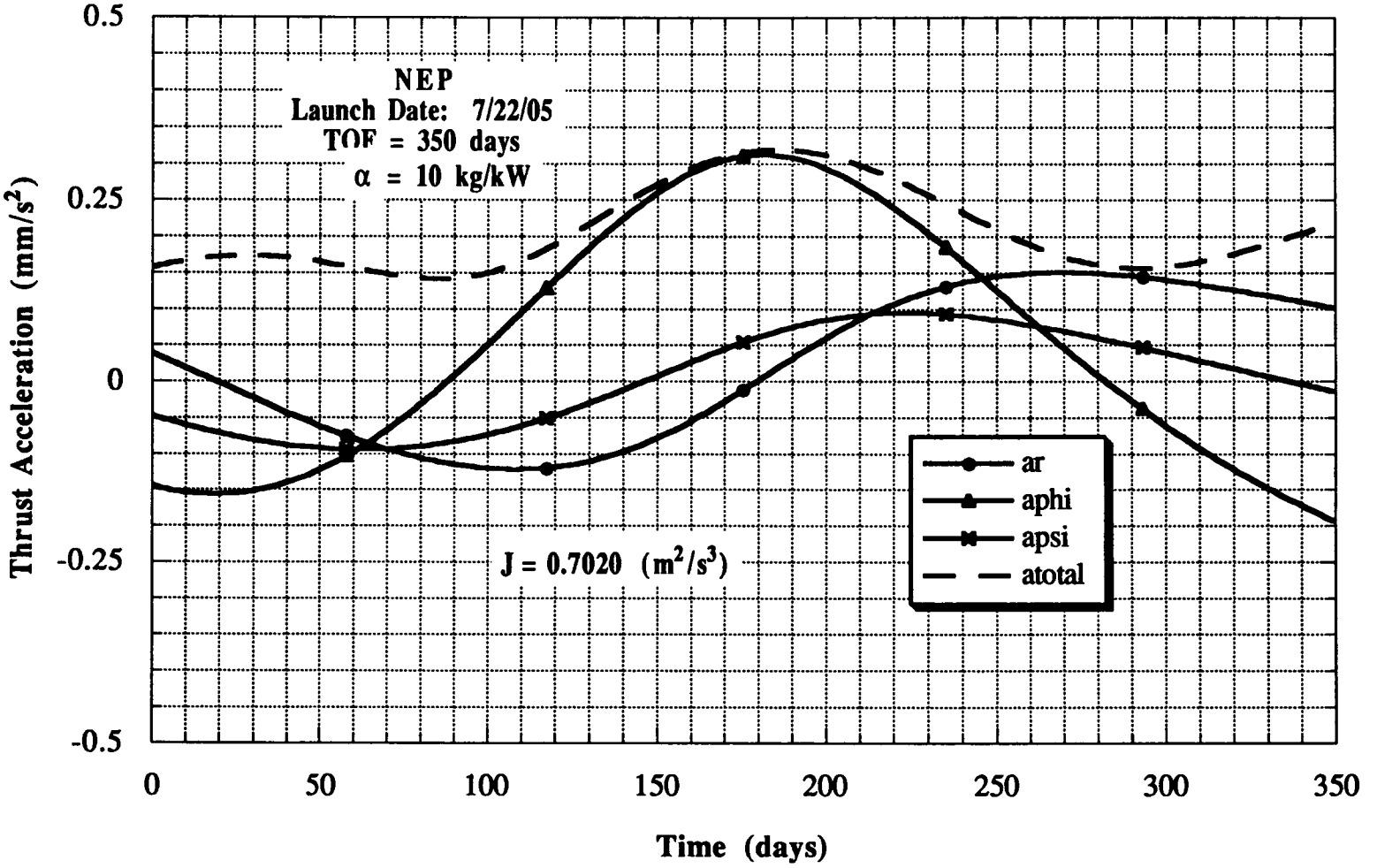


Figure 3.2.2: NEP Thrust Acceleration Program for Optimal 350-day Earth-Orpheus Mission

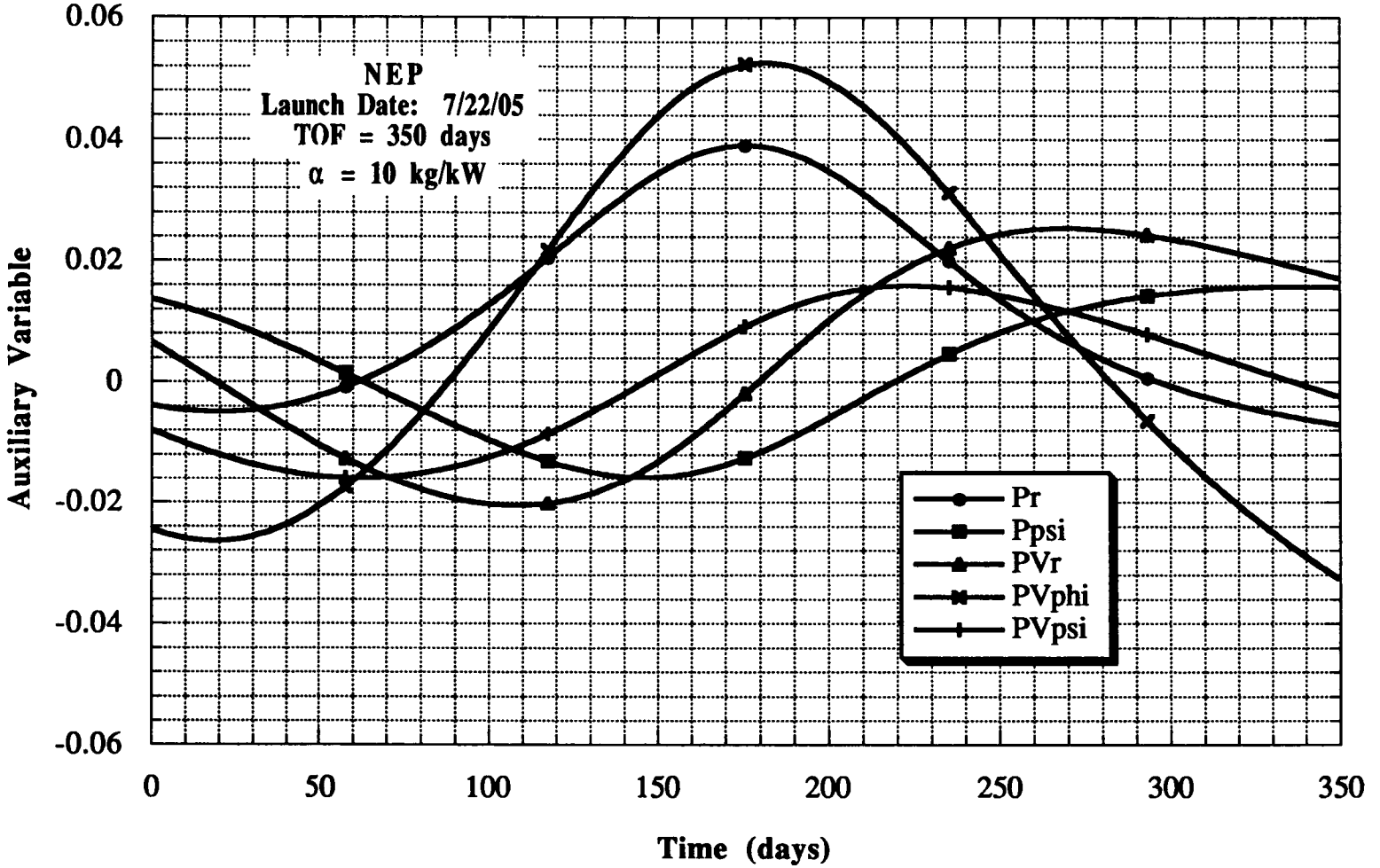


Figure 3.2.3: NEP Auxiliary Variable Histories for Optimal 350-day Earth-Orpheus Mission

The specific impulse of the propulsion system is defined as the exhaust velocity, c, divided by the acceleration due to gravity, g, on the Earth's surface. Using equations (3.2.11) and (3.2.12), the Isp may be written as:

$$Isp = \frac{2P}{M(t)a_t}$$

M(t) and P may be eliminated using equations (3.2.15), (3.2.16), and (3.2.32) to give:

$$Isp = \frac{1}{ga_t(t)}\left[\frac{2\eta}{\alpha}\frac{Mw}{M_1} + \int_0^t a_t^2 dt\right]$$

Assuming that the ratio $Mw/M_1$ takes on its optimum value of $\gamma - \gamma^2$ (see equation 3.2.39) gives the following expression for the specific impulse for the propulsion system as a function of time:

$$Isp(t) = \frac{1}{ga_t(t)}\left[\frac{2\eta}{\alpha}\left(\gamma - \gamma^2\right) + \int_0^t a_t^2 dt\right] \qquad (3.2.43)$$

This variation for the NEP mission is given in Figure 3.2.4. Again, no limit is placed on the maximum value which the Isp may attain. In a more precise analysis, the maximum Isp is restricted to a value characteristic of a specific thruster. As the Isp approaches this value, the thruster switches to a constant exhaust velocity (i.e. constant Isp) mode with the specific impulse equal to the limit. At some other prescribed point on the trajectory, the thrust would then switch back to the variable Isp mode. The constant exhaust velocity mode could also be analyzed for coasting periods [22]. However, for the preliminary mission design in this study, only an ideal throttlable engine is considered.

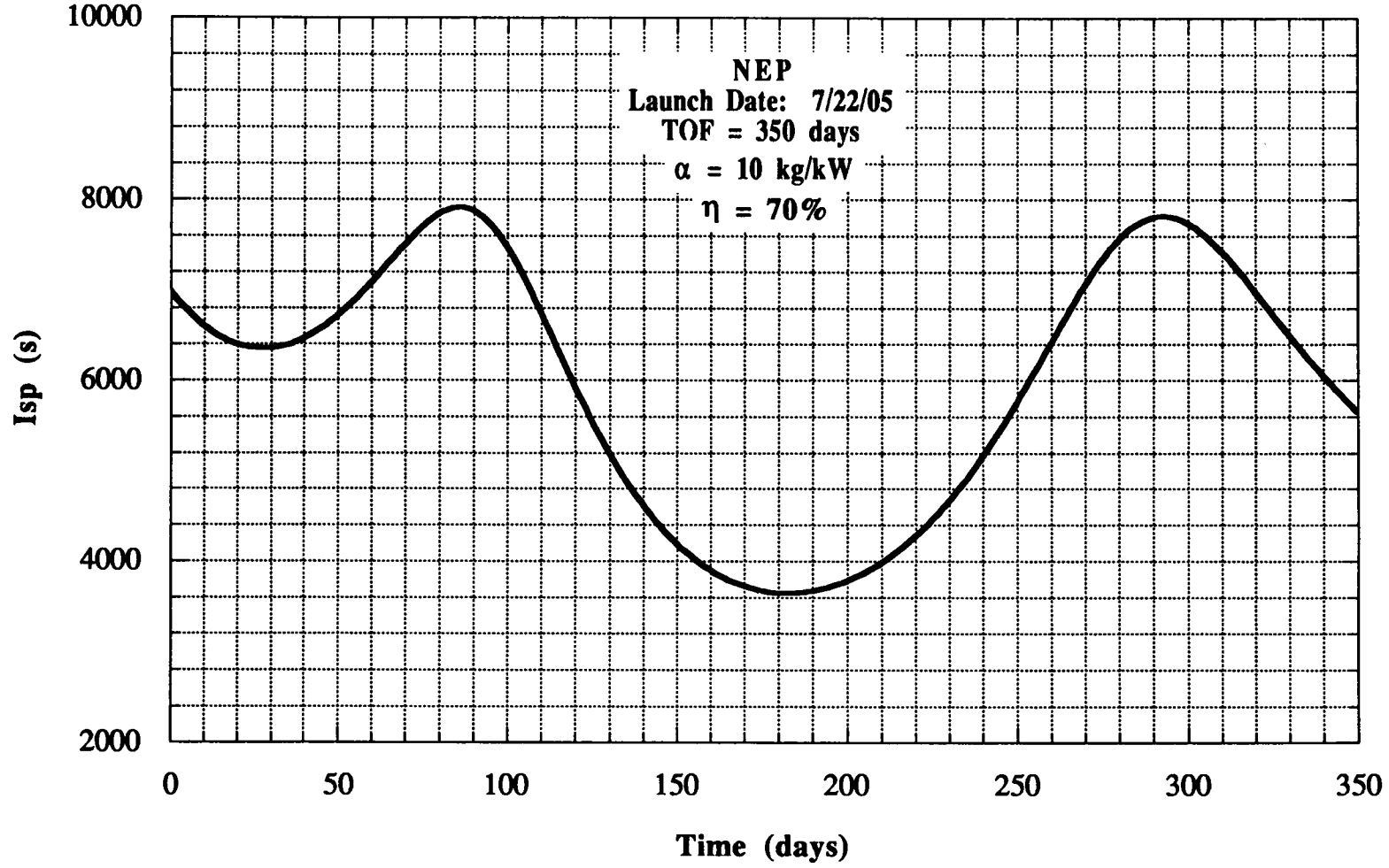


Figure 3.2.4: NEP Specific Impulse History for Optimal 350-day Earth-Orpheus Mission

The histories of the state variables for the 350 day minimum fuel NEP mission are illustrated in Figure 3.2.5 through Figure 3.2.8.

Using the relations given in section 3.2.3, the spacecraft component masses are calculated for the NEP spacecraft with a net mass of 250 kg. The resulting mass summary for the 350 day, 2005 mission is listed below in Table 3.2.2. Note the very low initial spacecraft mass of 406.63 kg. This is an extremely optimistic estimate based on the assumption that the reactor size is dictated by the 10 kg/kW specific mass for any level of power output. For the 2005 NEP mission, the optimal power required to rendezvous with Orpheus is only 3.66 kW. This reduced power requirement results in a very low total propulsion mass of 109.25 kg. For comparison, a mass summary for a standard SP-100 power output of 100 kW is also given in the table. At this power rating, the mass of the propulsion system increases to 1060.33 kg, driving the initial spacecraft mass to 1334.38 kg.

Table 3.2.2: Mass Summary for Optimal NEP Missions

| **Power Rating (kW)** | Optimal Power<br>3.66 | SP-100<br>100.00 |
|---|---|---|
| **Component Mass (kg)** | | |
| Spacecraft Bus and Payload | 250.00 | 250.00 |
| Propulsion System | 109.25 | 1060.33 |
| Used Propellant | 40.72 | 17.62 |
| Unused Propellant | 0.41 | 0.18 |
| OMS/Payload Adapter | 6.25 | 6.25 |
| Total Injected Spacecraft Mass | 406.63 | 1334.38 |

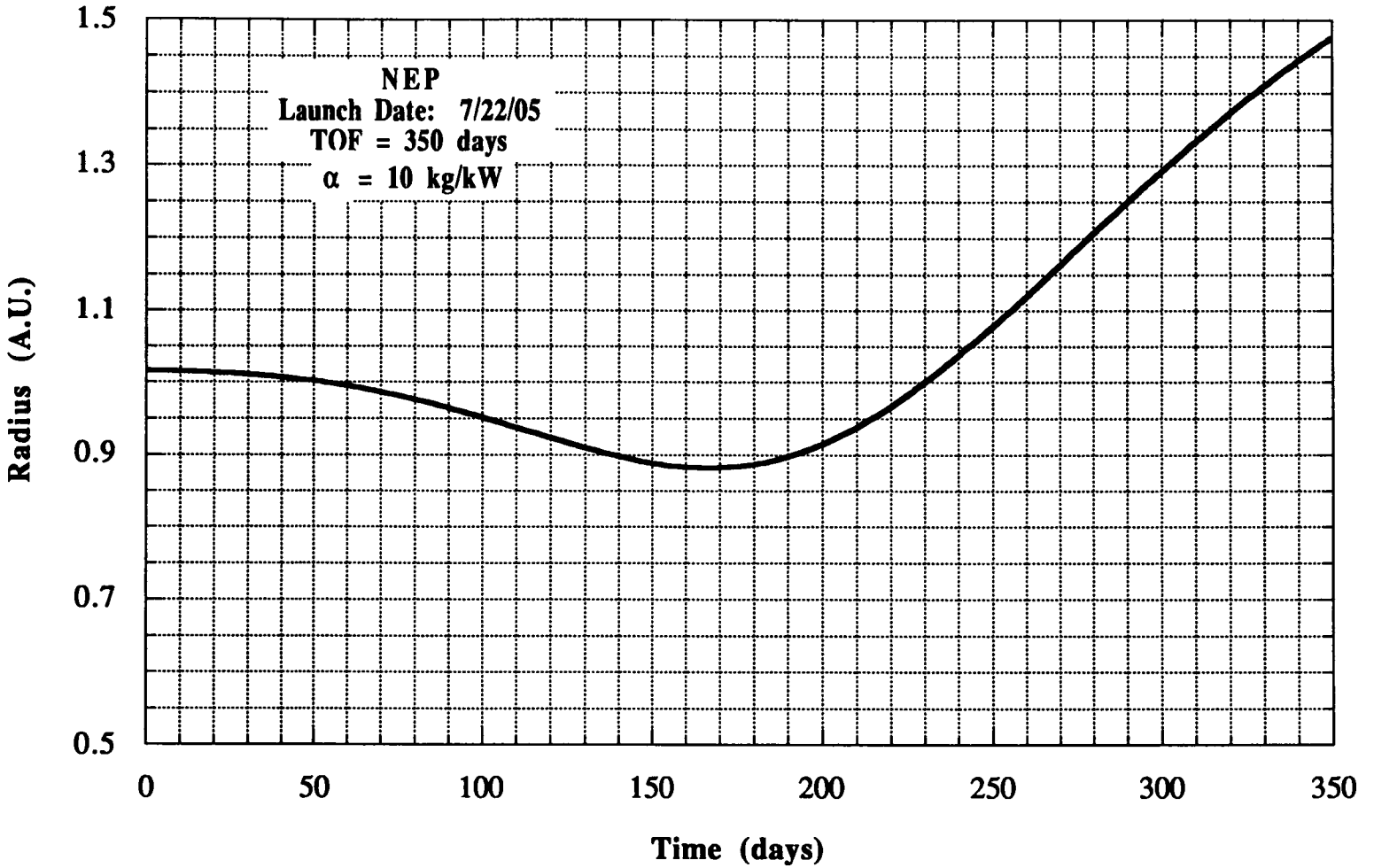


Figure 3.2.5: NEP Radial Position History for Optimal 350-day Earth-Orpheus Mission

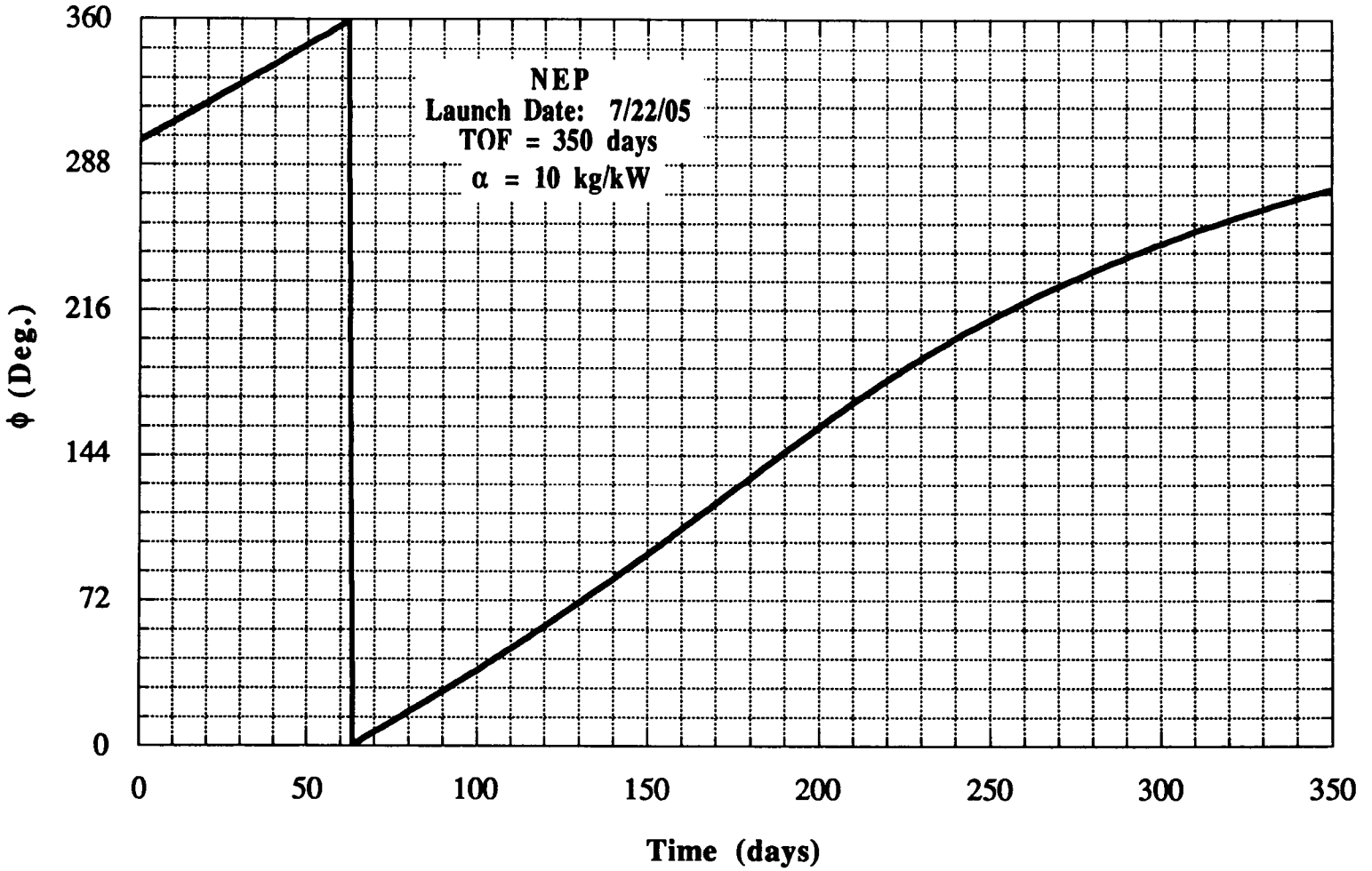


Figure 3.2.6: NEP Celestial Longitude History for Optimal 350-day Earth-Orpheus Mission

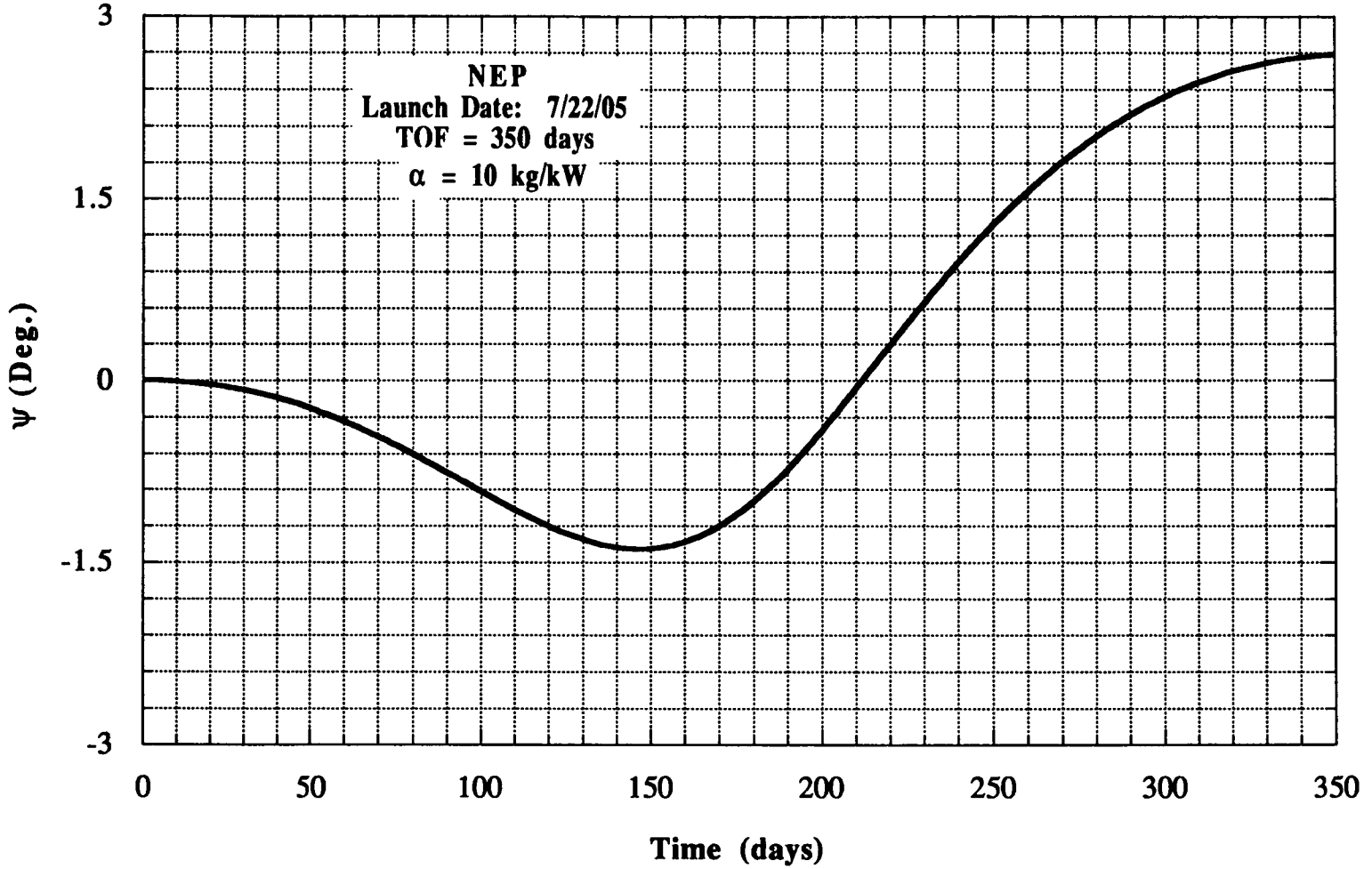


Figure 3.2.7: NEP Celestial Latitude History for Optimal 350-day Earth-Orpheus Mission

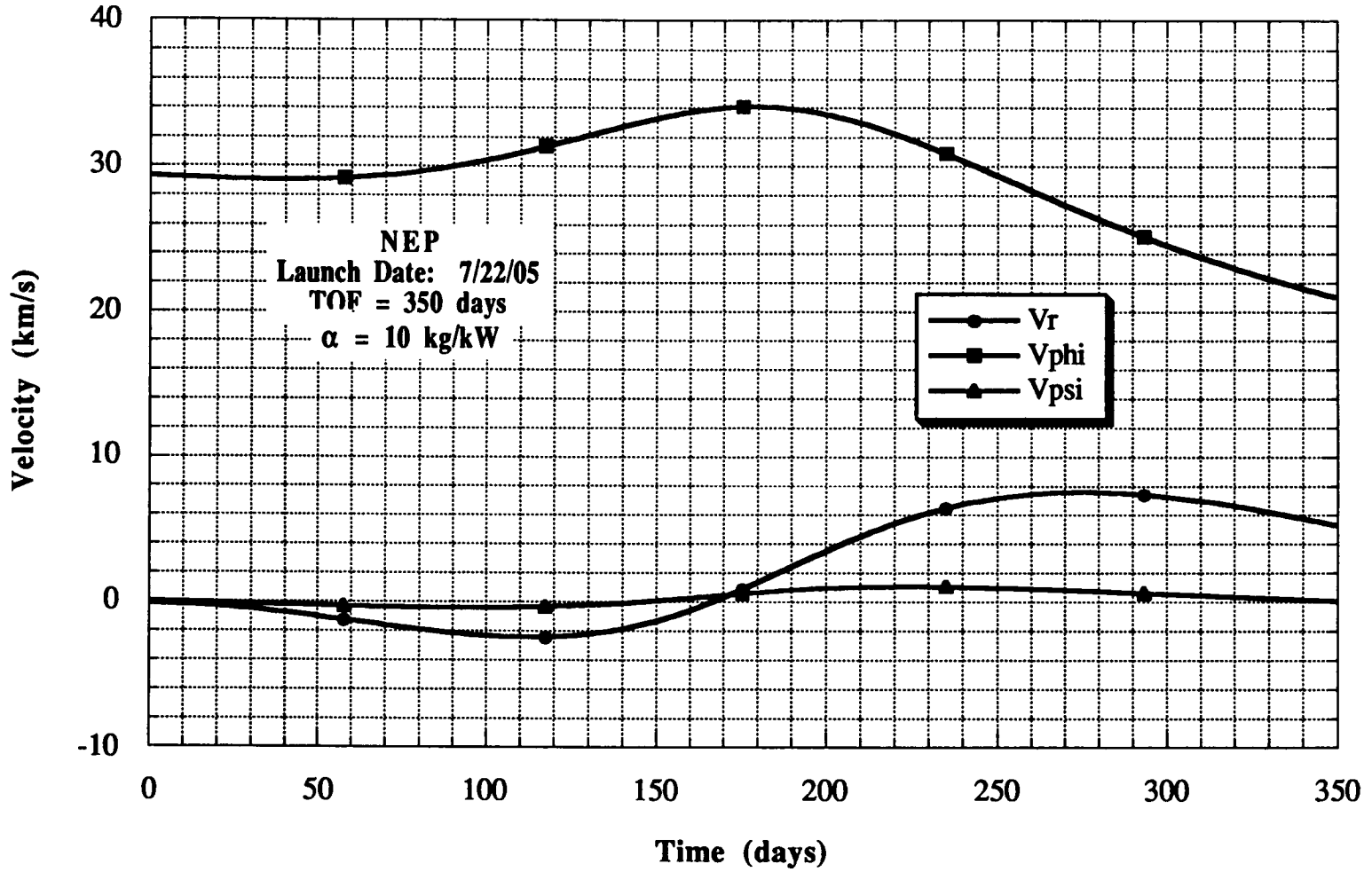


Figure 3.2.8: NEP Velocity Component Histories for Optimal 350-day Earth-Orpheus Mission

### 3.3 Solar Electric Propulsion

Like the nuclear electric system considered above, solar electric propulsion (SEP) also belongs to the power-limited category. However, whereas the nuclear electric system provides constant power to the propulsion device, the maximum power available from the solar electric powerplant depends on the position of the spacecraft, specifically the distance from the sun. The array of solar cells on the spacecraft convert the energy of the photons emitted by the sun into electrical energy which is then used to drive the propulsion system. The solar array characteristics assumed for this analysis are identical to those used in the JPL low thrust trajectory analysis [23]. The solar electric propulsion system is assumed to have a specific mass of 30 kg/kW and an efficiency of 60 % [9]. As with the NEP system, throttlable engines are considered in this study with no upper limit placed on the specific impulse.

#### 3.3.1 Equations of Motion

Since the only difference between the NEP and SEP systems is the method by which power is produced, the power-limited equations that governed the motion of the NEP spacecraft also dictate the motion of the SEP system. Therefore, the state equations for the motion of the SEP vessel in an inverse square gravity field are given by (3.2.5) - (3.2.10).

#### 3.3.2 Trajectory Optimization

In order to minimize the mission cost of a power-limited mission, it is desired to minimize the fuel expenditure of the propulsion system. This is synonymous to maximizing the payload or terminal mass for any given power supply mass. A functional must be derived which is characteristic of the fuel required along the interplanetary

trajectory. Once given, this functional may be minimized using the optimization techniques outlined in section 3.1.2.

As before, the relation between the initial and final masses of the spacecraft is given by:

$$\frac{1}{M_2} - \frac{1}{M_1} = \int_0^T \frac{a_t^2}{2P} dt \tag{3.3.1}$$

The power available to the propulsion system is:

$$P = \eta \eta_s \left( \frac{r_e}{r} \right)^2 P_o \tag{3.3.2}$$

where $\eta$ is the total system efficiency, $\eta_s$ is the relative array efficiency, and $P_o$ is the maximum power available at the radial position of the Earth, $r = r_e$. The power given in (3.3.2) includes a $1/r^2$ term, which is the expected variation for a system dependent upon the solar energy flux.

The solar array characteristics used in this thesis are adopted from JPL's low-thrust trajectory analysis [23]. The corresponding solar array efficiency is given by the following relation:

$$\eta_s = \frac{a_1 + a_2 r^{-1} + a_3 r^{-2}}{1 + a_4 r + a_5 r^2} \tag{3.3.3}$$

Substituting (3.3.2) into (3.3.1) gives the equation relating the initial and final spacecraft masses:

$$\frac{1}{M_2}-\frac{1}{M_1}=\frac{1}{2\eta P_o}\int_0^T\left(\frac{r}{r_e}\right)^2\left(\frac{1}{\eta_s}\right)a_t{}^2dt \tag{3.3.4}$$

where the first ratio in the integral is simply the non-dimensional r. The problem of minimizing the fuel consumption of the solar electric propulsion system is now reduced to minimizing the integral:

$$J=\frac{1}{2}\int_0^T\frac{r^2}{\eta_s}a_t{}^2dt \tag{3.3.5}$$

This functional is similar to that given by (3.2.17) with the inclusion of the power dependence on the radial distance from the sun.

The method for obtaining the optimal thrust variation is the same as in section 3.2.1. However, the additional state equation is given by:

$$\frac{dJ}{dt}=\frac{1}{2}\frac{r^2}{\eta_s}a_t{}^2=\frac{1}{2}\frac{r^2}{\eta_s}\left(a_r{}^2+a_\phi{}^2+a_\psi{}^2\right) \tag{3.3.6}$$

The resulting Hamiltonian for the dynamic solar electric system is:

$$\begin{aligned}H=P_J\left[\frac{1}{2}\frac{r^2}{\eta_s}\left(a_r{}^2+a_\phi{}^2+a_\psi{}^2\right)\right]+P_r\left[V_r\right]+P_\phi\left[\frac{V_\phi}{r\cos(\psi)}\right]+P_\psi\left[\frac{V_\psi}{r}\right]\\ +P_{V_r}\left[\frac{V_\phi{}^2+V_\psi{}^2}{r}-\frac{1}{r^2}+a_r\right]+P_{V_\phi}\left[-\frac{V_rV_\phi}{r}+\frac{V_\phi V_\psi\tan(\psi)}{r}+a_\phi\right]\\ +P_{V_\psi}\left[-\frac{V_rV_\psi}{r}-\frac{V_\phi{}^2\tan(\psi)}{r}+a_\psi\right]\end{aligned} \tag{3.3.7}$$

Applying (3.1.28) to the Hamiltonian gives the following adjoint equation for the radial state variable:

$$\frac{dP_r}{dt} = -\frac{\partial H}{\partial r} = -\frac{1}{2}P_J\left(a_r{}^2 + a_\phi{}^2 + a_\psi{}^2\right)\left(\frac{2r\eta_s - r^2\frac{d\eta_s}{dr}}{\eta_s{}^2}\right) + \frac{P_\phi V_\phi}{r^2\cos(\psi)}$$
$$+\frac{P_\psi V_\psi}{r^2} + \frac{P_{V_r}\left(V_\phi{}^2 + V_\psi{}^2\right)}{r^2} - \frac{2P_{V_r}}{r^3} - \frac{P_{V_\phi}V_r V_\phi}{r^2}$$
$$+\frac{P_{V_\phi}V_\phi V_\psi\tan(\psi)}{r^2} - \frac{P_{V_\psi}V_r V_\psi}{r^2} - \frac{P_{V_\psi}V_\phi{}^2\tan(\psi)}{r^2} \qquad (3.3.8)$$

where:

$$\frac{d\eta_s}{dr} = -\frac{\left(a_2r^{-2} + 2a_3r^{-3}\right)\left(1 + a_4r + a_5r^2\right) + \left(a_1 + a_2r^{-1} + a_3r^{-2}\right)\left(a_4 + 2a_5r\right)}{\left(1 + a_4r + a_5r^2\right)^2} \qquad (3.3.9)$$

Again the value of $P_J$ is assigned the value of -1. The other auxiliary differential equations for $P_\phi$, $P_\psi$, $P_{V_r}$, $P_{V_\phi}$, and $P_{V_\psi}$ are the same as in (3.2.22) - (3.2.26).

The optimal thrust relations for $a_r$, $a_\phi$, and $a_\psi$ are generated by applying (3.1.29) to the Hamiltonian given above. Taking the partial derivative of H with respect to r gives:

$$\frac{\partial H}{\partial r} = P_J\frac{r^2}{\eta_s}a_r + P_{V_r} = 0$$

Solving for $a_r$:

$$a_r = -\frac{P_{V_r}\eta_s}{P_J r^2} = \frac{P_{V_r}\eta_s}{r^2} \qquad (3.3.10)$$

Similarly, the optimal thrust accelerations in the $\phi$ and $\psi$ directions are:

$$a_\phi = \frac{P_{V_\phi}\eta_s}{r^2} \tag{3.3.11}$$

and

$$a_\psi = \frac{P_{V_\psi}\eta_s}{r^2} \tag{3.3.12}$$

Substituting the optimal thrust accelerations into the state equations, (3.2.5) - (3.2.10) and (3.3.6), and the co-state equations given by (3.3.8) and (3.2.22) - (3.2.26), results in a system of eleven differential equations.

### 3.3.3 Spacecraft Mass Calculation

The mass calculations for the SEP spacecraft are similar to those for the NEP system (see section 3.2.3). The major difference is in the propulsion system mass; the mass of the power-thruster system, Mw, is now given by the following expression [9]:

$$Mw = Mpow + Mt \tag{3.3.13}$$

where:

$$Mpow = \alpha P_o$$

and

$$Mt = 120kg$$

Equation (3.3.13) consists of the solar power array and thruster subsystems, which includes the addition of redundant thrusters required to meet lifetime and reliablility

constraints. Substituting Po = Mpow/$\alpha$ into equation (3.3.4) and multiplying by M1 yields:

$$\frac{M_1}{M_2} = 1 + \frac{M_1}{Mpow}\frac{\alpha J}{2\eta} \tag{3.3.14}$$

where J is given by (3.3.5).

Using the same refrigeration system and tankage factor as those in the NEP vehicle, the total mass of the propulsion system is given by:

$$Mprop = Mpow + 0.15Mp + 1.74Mp^{2/3} + 165.9 \tag{3.3.15}$$

This relation is now substituted into equation (2.2.1) to give the initial mass of the SEP vehicle after launch:

$$M_1 = 1.025Mnet + Mpow + 1.16Mp + 1.74Mp^{2/3} + 165.9 \tag{3.3.16}$$

Subracting the propellant consumed by the propulsion system results the relation for the final spacecraft mass:

$$M_2 = 1.025Mnet + Mpow + 0.16Mp + 1.74Mp^{2/3} + 165.9 \tag{3.3.17}$$

Dividing by the initial mass and rearranging yields equation (3.2.35) with Mw replaced by Mpow:

$$\frac{Miw}{M_1} = \frac{M_2}{M_1} - \frac{Mpow}{M_1} \tag{3.3.18}$$

where:

$$Miw = 1.025Mnet + 0.16Mp + 1.74Mp^{2/3} + 165.9 \tag{3.3.19}$$

The payload capability of the SEP spacecraft is optimized in the same manner as with the NEP vehicle. The resulting optimal relations are identical to equations (3.2.39) - (3.2.41) with Mw replaced by Mpow:

$$\frac{Mpow}{M_1} = \gamma - \gamma^2 \tag{3.3.20}$$

$$\frac{Miw}{M_1} = (1-\gamma)^2 \tag{3.3.21}$$

$$\frac{Mp}{M_1} = \gamma \tag{3.3.22}$$

The dimensionless parameter, $\gamma$, is still defined by equation (3.2.38) but with the characteristic J for the SEP mission. Multiplying (3.3.21) and (3.3.22) by $M_1$ and dividing gives:

$$\frac{1.025Mnet + 0.16Mp + 1.74Mp^{2/3} + 165.9}{Mp} = \frac{(1-\gamma)^2}{\gamma} \tag{3.3.23}$$

where (3.3.19) has been substituted back into Miw. Given the net spacecraft mass and $\gamma$, the above relation is solved for Mp. The corresponding $M_1$, Mpow, and Mprop are then calculated using equations (3.3.22), (3.3.20), and (3.3.15), respectively. The desired $P_o$ required to rendezvous with the asteroid is obtained by dividing Mpow by the SEP specific mass.

### 3.3.4 Results

With slight modifications, the computer code given in Appendix B is used to generate minimum fuel SEP trajectories. Previous expressions for H, Pr, $a_r$, $a_\phi$, and $a_\psi$ in the NEP code are replaced by the associated SEP relations given in section 3.3.2. As before, the program iterates to the optimal launch date which minimizes the fuel expenditure for a given time of flight.

The optimal SEP Earth-Orpheus rendezvous missions between 2000 - 2020 for various flight times are listed in Table 3.3.1. The corresponding opportunities have similar launch dates and minimum J values as those in the NEP case. This is a result of the NEP and SEP systems using the same general scheme of propulsion: power-limited.

As with the NEP mission, a desired flight time of 350 days is selected for comparison with the chemical case. A difference in the variation of J values can be seen between the NEP and SEP cases for this time of flight. Instead of increasing over the examined period, the minimum values of J continue to decrease to an absolute minimum of 0.6814 $m^2/s^3$ in 2017. This opportunity has a launch date of June 30, 2017 and an arrival date of June 15, 2018. The ecliptic projection of the trajectory is illustrated in Figure 3.3.1. The optimal controls and corresponding auxiliary variable histories are given in Figures 3.3.2 and 3.3.3, respectively; the curves show the same characteristics as the NEP variations. Histories of the specific impulse (equation 3.2.43) and state variables for the 350 day minimum fuel SEP mission are graphed in Figure 3.3.4 through Figure 3.3.8. Note that the Isp values are lower than those in Figure 3.2.4. This is a result of the higher specific mass of the SEP system (30 kg/kW) compared to the NEP system (10 kg/kW).

Table 3.3.1: Minimum Fuel SEP Earth-Orpheus Rendezvous Missions

| Time of Flight (Days) | Launch Date | Payoff Function, J ($m^2/s^3$) |
|---|---|---|
| 150 | 11/17/01 | 2.6121 |
| | 11/5/05 | 2.4901 |
| | 10/24/09 | 2.2910 |
| | 10/11/13 | 2.0213 |
| | 9/25/17 | 1.6814 |
| 200 | 10/25/01 | 1.5038 |
| | 10/13/05 | 1.4423 |
| | 10/2/09 | 1.3583 |
| | 9/20/13 | 1.2560 |
| | 9/6/17 | 1.1388 |
| 250 | 10/3/01 | 1.0641 |
| | 9/23/05 | 1.0286 |
| | 9/12/09 | 0.9858 |
| | 8/31/13 | 0.9378 |
| | 8/19/17 | 0.8870 |
| 300 | 9/13/01 | 0.8426 |
| | 9/2/05 | 0.8219 |
| | 8/22/09 | 0.7993 |
| | 8/11/13 | 0.7759 |
| | 7/28/17 | 0.7529 |
| 350 | 8/23/01 | 0.7168 |
| | 8/11/05 | 0.7060 |
| | 7/30/09 | 0.6956 |
| | 7/17/13 | 0.6869 |
| | 6/30/17 | 0.6814 |
| 400 | 3/1/01 | 0.5701 |
| | 3/7/05 | 0.5790 |
| | 3/16/09 | 0.5922 |

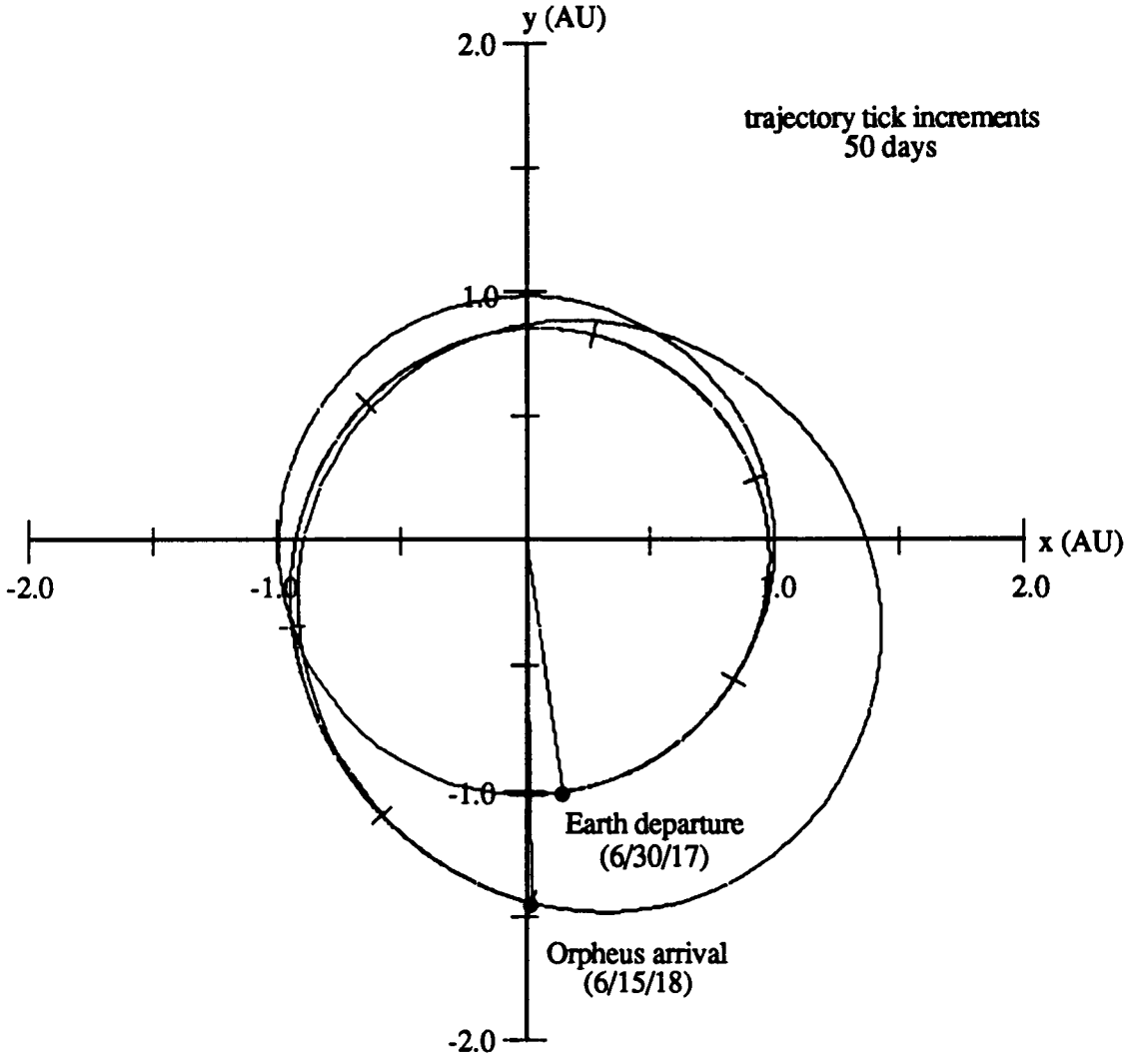


Figure 3.3.1: Ecliptic Projection of Optimal 350-day SEP Earth-Orpheus Rendezvous Trajectory

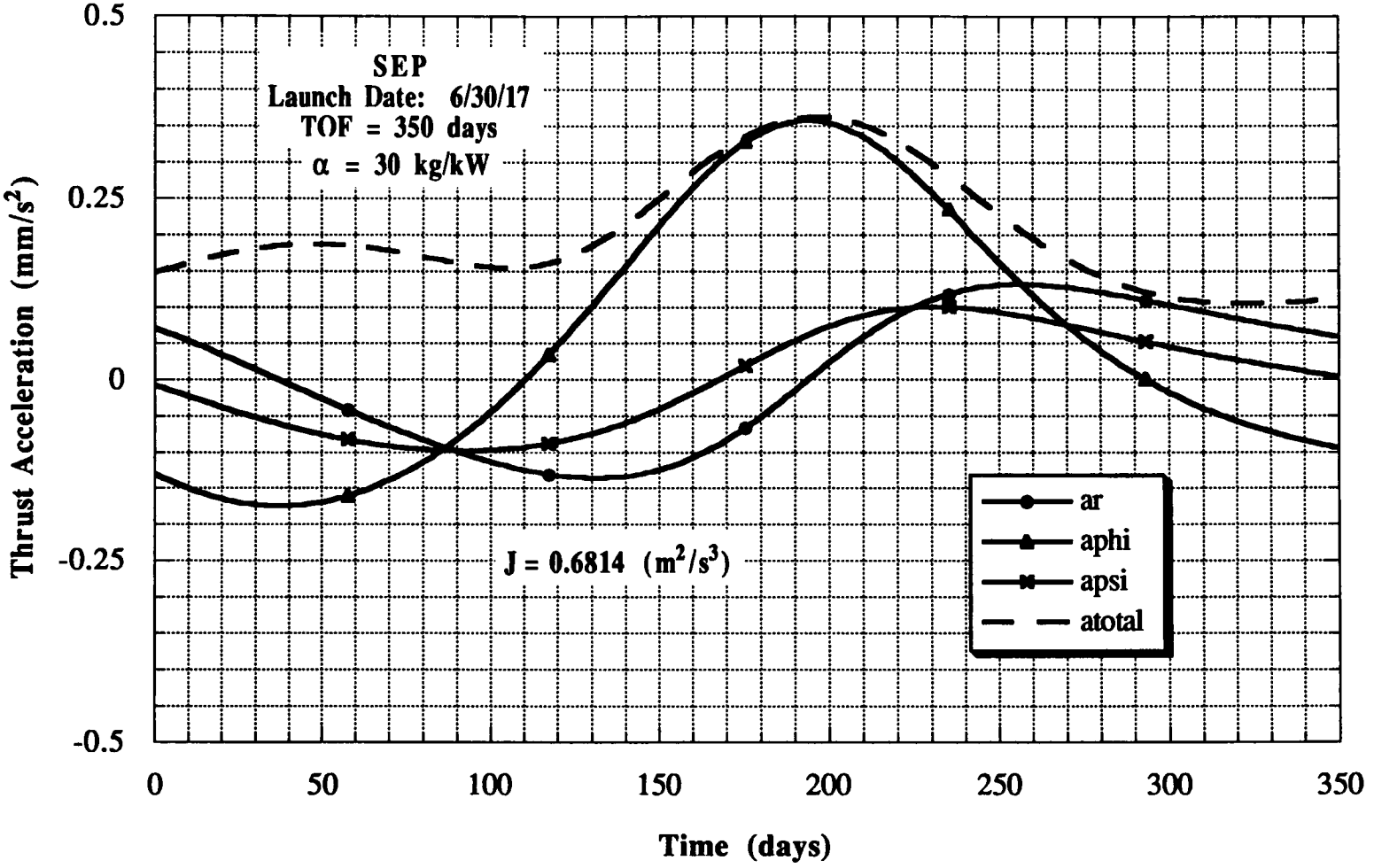


Figure 3.3.2: SEP Thrust Acceleration Program for Optimal 350-day Earth-Orpheus Mission

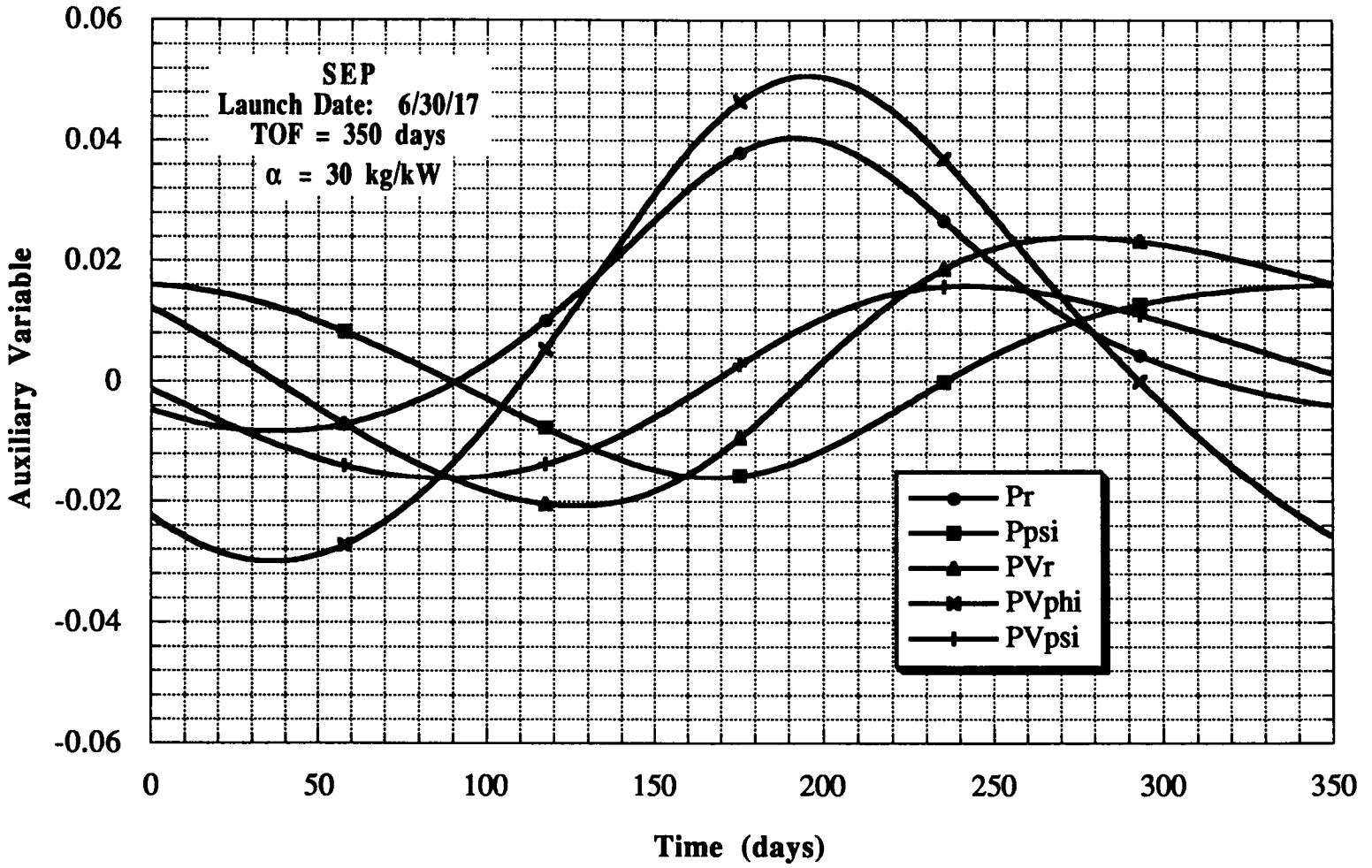


Figure 3.3.3: SEP Auxiliary Variable Histories for Optimal 350-day Earth-Orpheus Mission

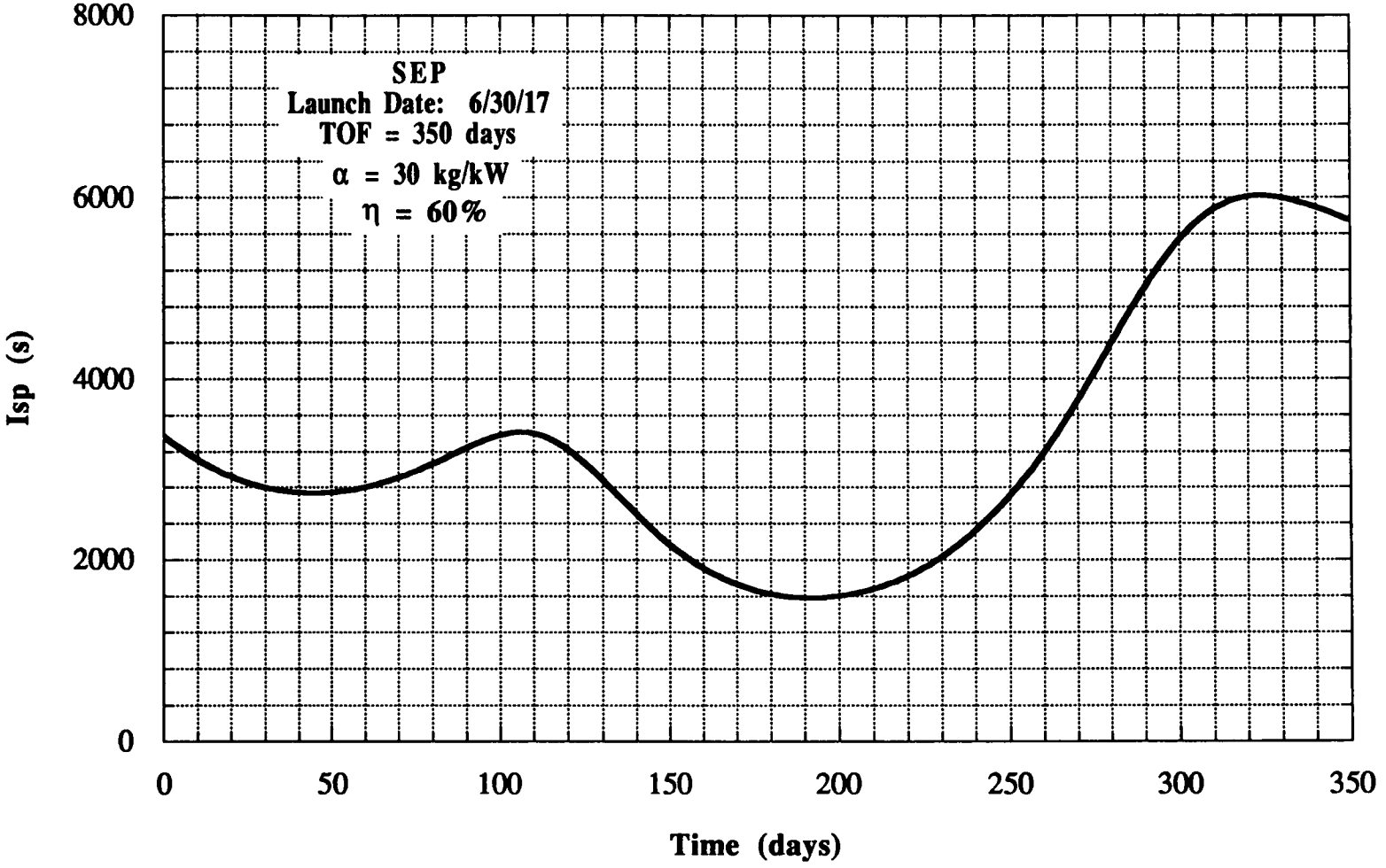


Figure 3.3.4: SEP Specific Impulse History for Optimal 350-day Earth-Orpheus Mission

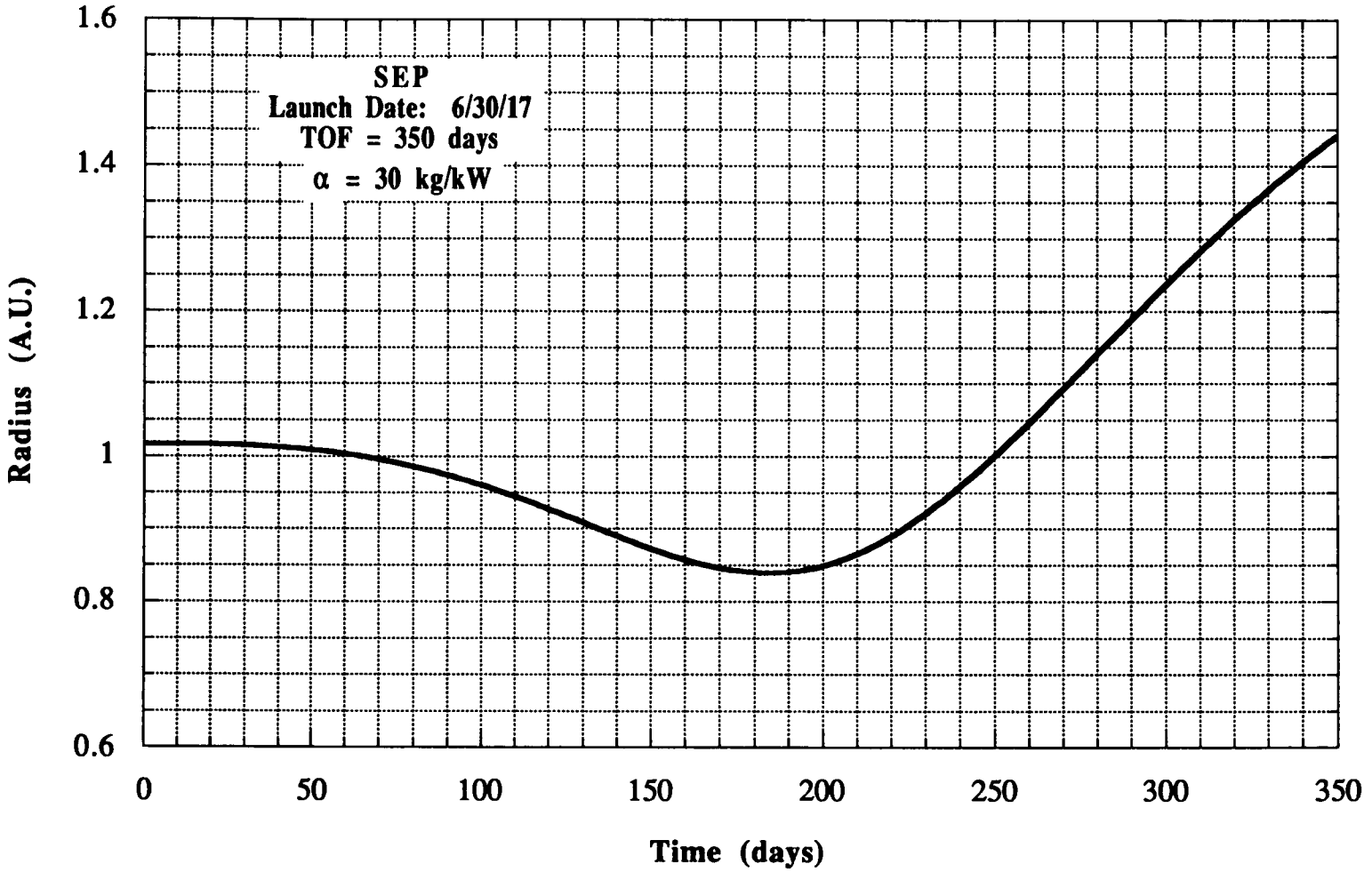


Figure 3.3.5: SEP Radial Position History for Optimal 350-day Earth-Orpheus Mission

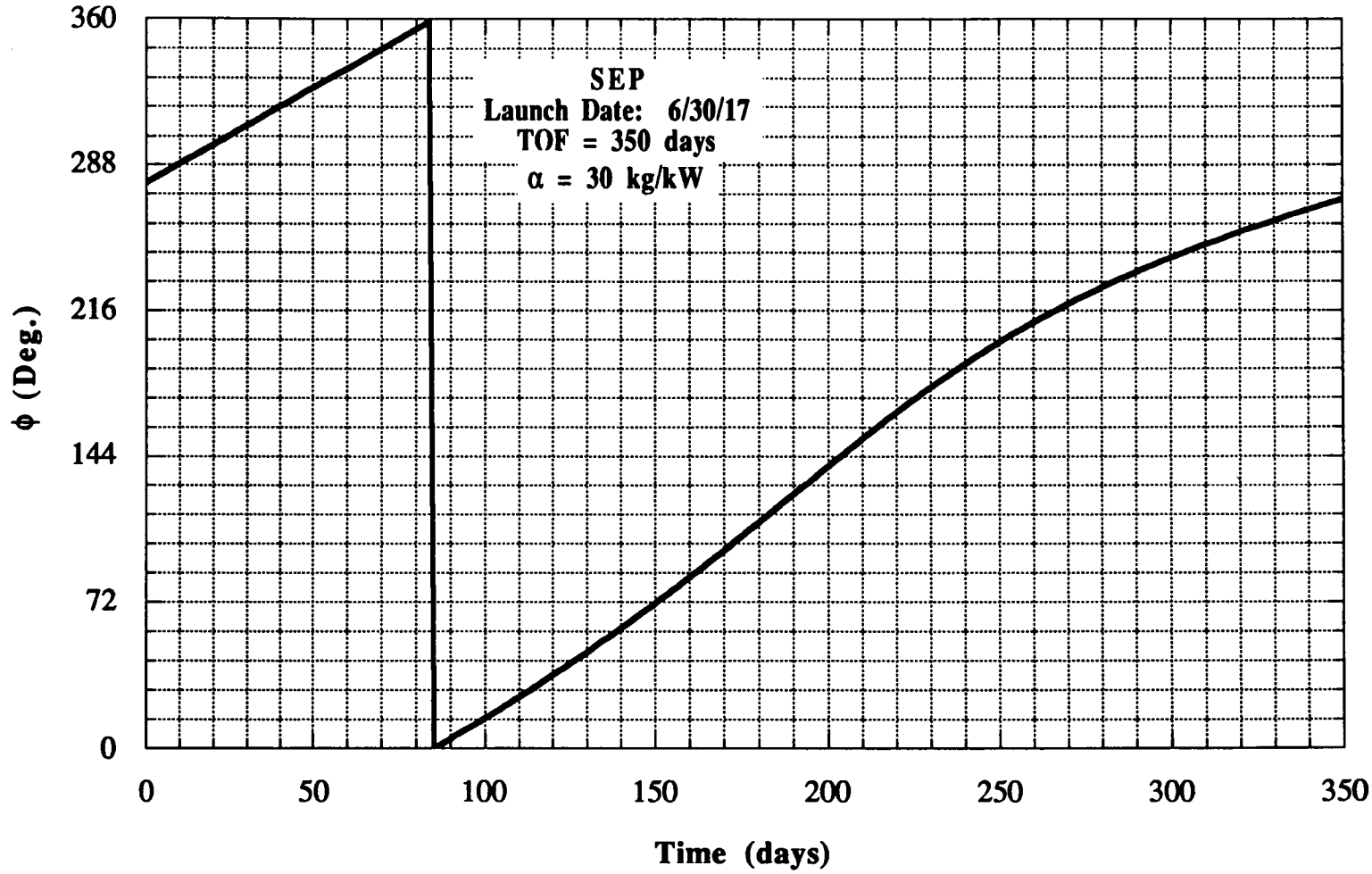


Figure 3.3.6: SEP Celestial Longitude History for Optimal 350-day Earth-Orpheus Mission

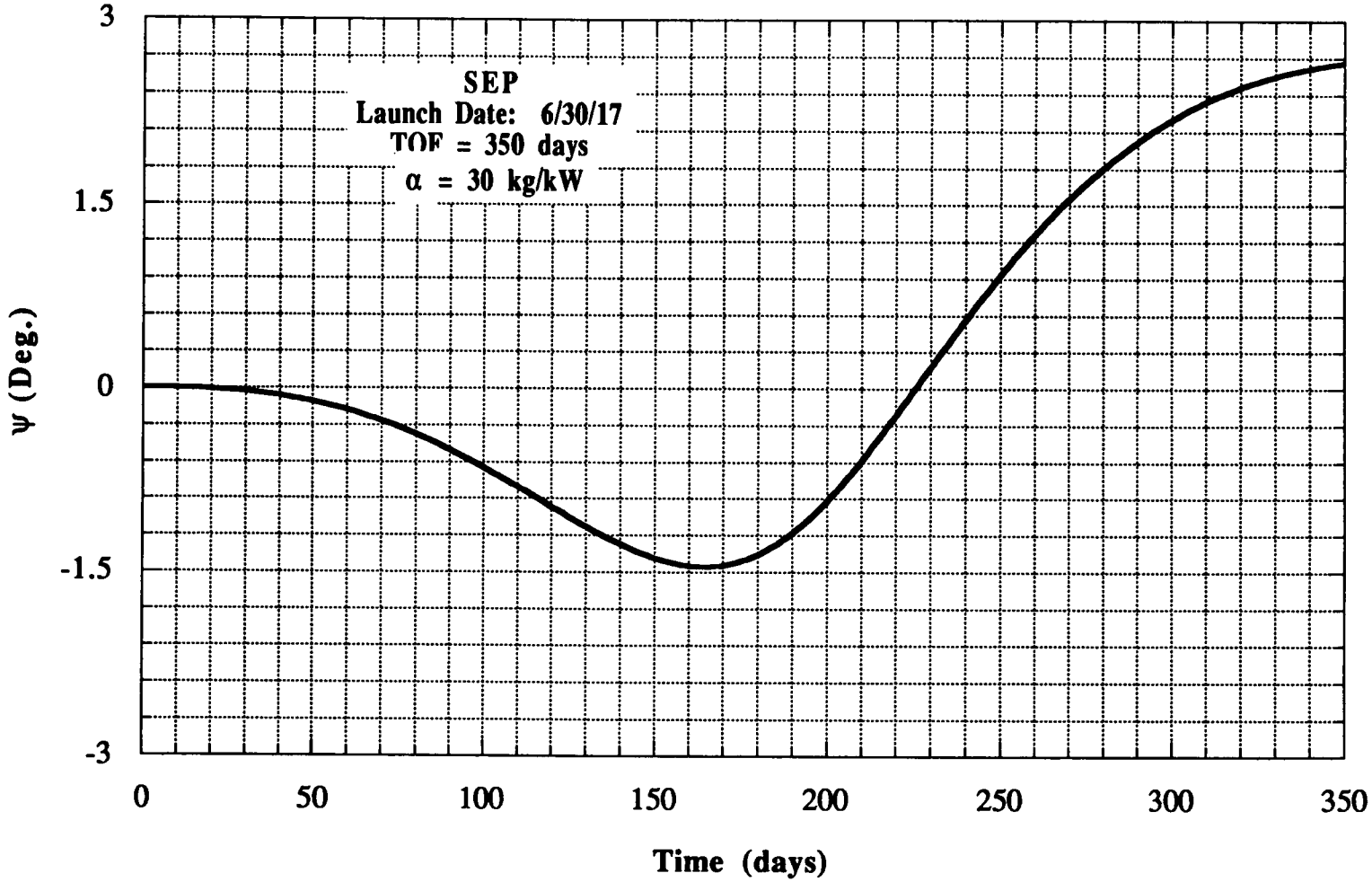


Figure 3.3.7: SEP Celestial Latitude History for Optimal 350-day Earth-Orpheus Mission

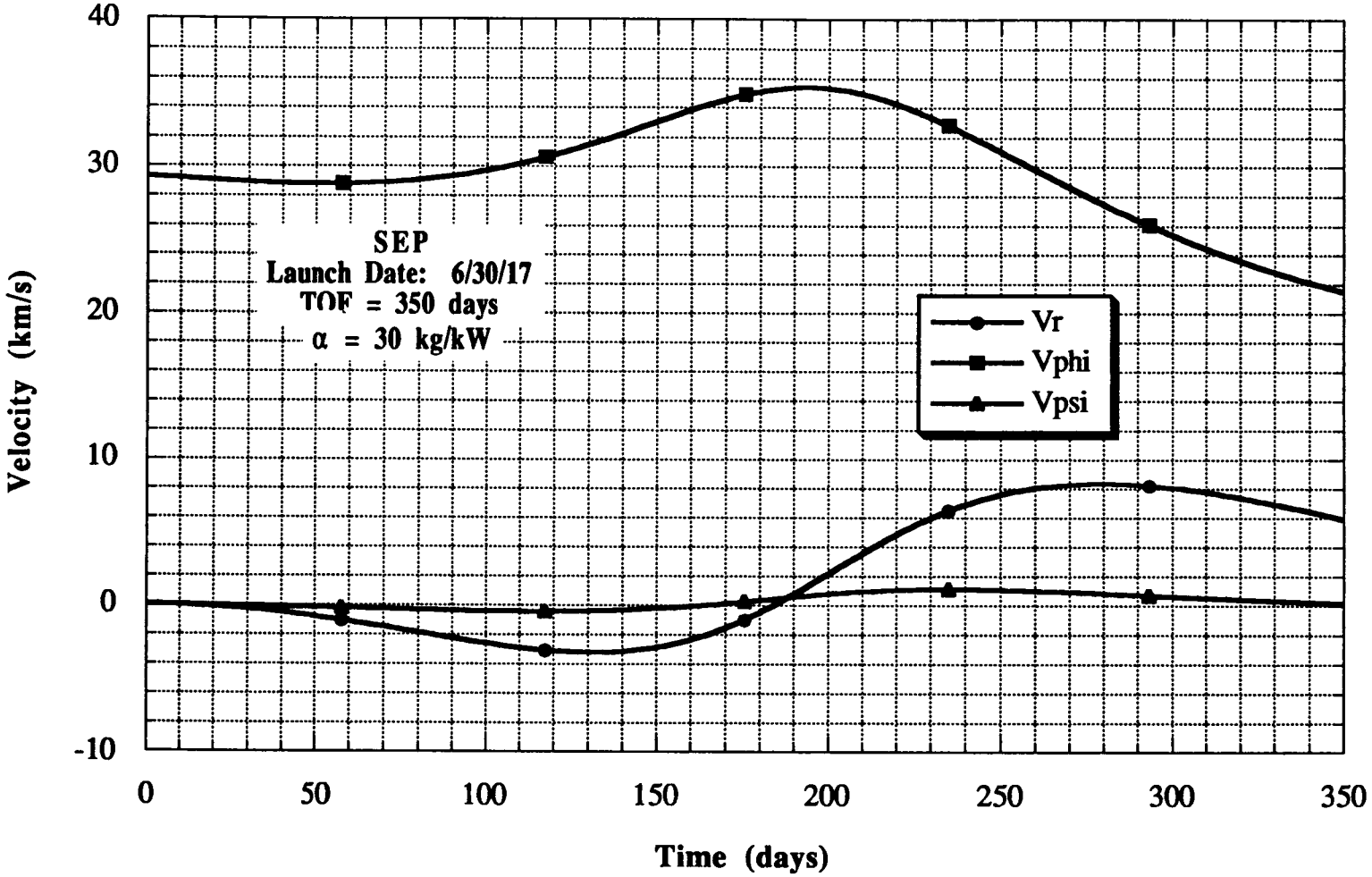


Figure 3.3.8: SEP Velocity Component Histories for Optimal 350-day Earth-Orpheus Mission

The spacecraft mass summary for the optimal, 2017 SEP mission is listed in Table 3.2.2. The initial mass after launch for the optimal power rating 3.70 kW at Earth's radius is 736.84 kg. This result is not as low as the NEP mass but is more realistic. The total propulsion system mass and the amount of required propellant are 343.22 kg and 136.01 kg, respectively. For comparison with the optimal case, a mass summary for a 10 kW SEP vehicle given in the table. At this power rating, the initial wet mass of the spacecraft is 836.65 kg. This shows a significant increase of approximately 100 kg for a 6.30 kW increase in power. Comparing the optimal SEP vehicle to the best chemical propulsion mission displays a mass savings of 271.63 kg.

Table 3.3.2: Mass Summary for Optimal SEP Missions

| **Power Rating (kW)** | 3.70 (Optimal) | 10 |
|---|---|---|
| **Component Mass (kg)** | | |
| Spacecraft Bus and Payload | 250.00 | 250.00 |
| Propulsion System | 343.22 | 507.07 |
| Used Propellant | 136.01 | 72.60 |
| Unused Propellant | 1.36 | 0.73 |
| OMS/Payload Adapter | 6.25 | 6.25 |
| Total Injected Spacecraft Mass | 736.84 | 836.65 |

### 3.4 Solar Sail Propulsion

Solar sails belong to the category of thrust-limited devices. The sail thrust is produced by the photon pressure of the incident sunlight acting on its surface. Thus, not only is thrust dependent upon the spacecraft's radial distance from the sun, but it is also limited by the intensity of the sun rays striking the sail. In this study, the sail is assumed

to be flat and completely rigid. Also, no restraint is placed on the rate at which the orientation of the sail can change.

### 3.4.1 Equations of Motion

Equations (3.1.16) through (3.1.21) describe the motion for any spacecraft. To describe the motion of a solar sail, relations must be obtained for the thrust accelerations produced by the sail and then substituted into (3.1.16) - (3.1.21). The resulting state equations may then be nondimensionalized using the dimensionless variables given in (3.2.1) - (3.2.4).

In this study, the force model is developed for a plane square sail. Therefore, the thrust is always normal to the sail and away from the light source. The sail geometry with respect to the spherical coordinate system is shown in Figure 3.4.1 and is identical to that used by Sauer [24]. The normal to the sail is denoted by $\hat{n}$, which is in the direction of thrust. The control variables for the sail are the thrust vector cone and clock angles, $\alpha$ and $\beta$, respectively. The cone angle is defined as the angle between the thrust vector, directed normal to the sail, and the direction of the incident sun rays. The clock angle is the angle between the $\hat{\psi}$ vector and the projection of the thrust direction onto the plane normal to the sunline ($\phi\psi$-plane).

The acceleration due to the change in momentum of the photons reflecting off the sail surface is given by:

$$\bar{a} = \frac{P_{rad} R_f S}{Mnet + Ms} \left( \frac{r_e}{r} \right)^2 \left( \hat{n} \cdot \hat{i} \right)^2 \hat{n} \qquad (3.4.1)$$

This relation is nearly identical to that given in Grodzovskii [25] but is modified to include the reflectivity, $R_f$, of the sail material. $P_{rad}$ is the solar radiation pressure imparted to the sail at a reference distance $r_e$. In this study, $r_e$ is assigned the value of 1

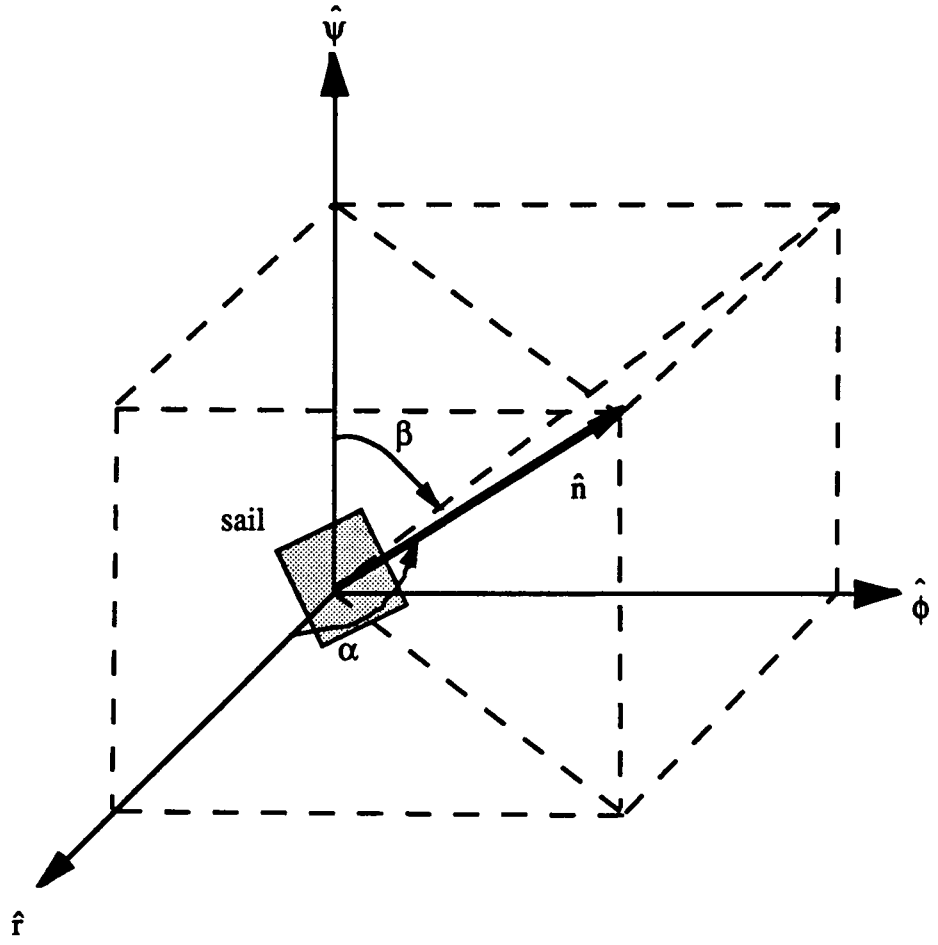


Figure 3.4.1: Sail Orientation

AU, at which $P_{rad}$ is $9 \cdot 10^{-6}$ N/m$^2$. S is the area of the square sail, Mnet is the net spacecraft mass, Ms is the mass of the sail, and $\hat{i}$ is a unit vector in the direction of the solar rays. Here, the effect of the solar wind is assumed to be much less than that of the radiation pressure and is neglected.

Given the sail thickness, δ, the mass of the solar sail is simply the product of the material density, ρ, and volume, given by δS. Sauer [24] introduces a structural factor, $k_s$, which is assumed proportional to the sail area. Therefore, the mass of the sail may be written as:

$$Ms = \rho\delta S(1 + k_s) \tag{3.4.2}$$

Substituting into (3.4.1) results in the following relation for the sail thrust acceleration:

$$\vec{a} = \frac{P_{rad} R_f S}{Mnet + \rho\delta S(1 + k_s)} \left(\frac{r_e}{r}\right)^2 (\hat{n} \cdot \hat{i})^2 \hat{n} \tag{3.4.3}$$

The first term in (3.4.3) corresponds to the acceleration produced by the solar radiation pressure on the sail with its surface normal to the sun-line and located at 1 AU from the sun. This is defined as the characteristic thrust acceleration, $a_o$, of the sail and is an important parameter relating the size of the sail to the amount of payload which may be carried by the spacecraft:

$$a_o = \frac{P_{rad} R_f S}{Mnet + S\rho\delta(1 + k_s)} \tag{3.4.4}$$

Substituting (3.4.3) into (3.4.4) gives a more useful relation for the thrust acceleration of the sail:

$$\vec{a} = a_o \left(\frac{r_e}{r}\right)^2 (\hat{n} \cdot \hat{i})^2 \hat{n} \tag{3.4.5}$$

The acceleration produced by the sail in the r, $\phi$, and $\psi$ directions may now be derived given (3.4.5) and the control angles defined previously. Considering the geometry depicted in Figure 3.4.1, the unit vector normal to the sail's surface is:

$$\hat{n} = \cos(\alpha)\hat{r} + \sin(\alpha)\sin(\beta)\hat{\phi} + \sin(\alpha)\cos(\beta)\hat{\psi}$$

and the unit vector in the direction of incident photons is $\hat{i} = \hat{r}$. The dot product of $\hat{n}$ and $\hat{i}$ is simply $\cos(\alpha)$. Substituting into (3.4.5) and separating into the individual components yield the formulae for $a_r$, $a_\phi$, and $a_\psi$.

$$a_r = \frac{a_o r_e^2}{r^2}\cos^3(\alpha) \tag{3.4.6}$$

$$a_\phi = \frac{a_o r_e^2}{r^2}\cos^2(\alpha)\sin(\alpha)\sin(\beta) \tag{3.4.7}$$

$$a_\psi = \frac{a_o r_e^2}{r^2}\cos^2(\alpha)\sin(\alpha)\cos(\beta) \tag{3.4.8}$$

Substituting these acceleration components into (3.1.16) - (3.1.21) and applying (3.2.1) - (3.2.4) give the dimensionless state equations governing the motion of a solar sail in an inverse square gravity field.

$$\frac{dr}{dt} = V_r \tag{3.4.9}$$

$$\frac{d\phi}{dt} = \frac{V_\phi}{r\cos(\psi)} \tag{3.4.10}$$

$$\frac{d\psi}{dt} = \frac{V_\psi}{r} \tag{3.4.11}$$

$$\frac{dV_r}{dt} = \frac{V_\phi^2 + V_\psi^2}{r} - \frac{1}{r^2} + \frac{a_o\cos^3(\alpha)}{r^2} \tag{3.4.12}$$

$$\frac{dV_\phi}{dt} = -\frac{V_r V_\phi}{r} + \frac{V_\phi V_\psi \tan(\psi)}{r} + \frac{a_o\cos^2(\alpha)\sin(\alpha)\sin(\beta)}{r^2} \tag{3.4.13}$$

$$\frac{dV_\psi}{dt} = -\frac{V_r V_\psi}{r} - \frac{V_\phi^2 \tan(\psi)}{r} + \frac{a_o \cos^2(\alpha)\sin(\alpha)\cos(\beta)}{r^2} \qquad (3.4.14)$$

### 3.4.2 Trajectory Optimization

For a solar sail rendezvous mission, no fuel is consumed by the propulsion system since the thrust is produced by the photon pressure acting on the sail's surface. Therefore, optimal sail trajectories are generated by minimizing the time of flight. Minimizing the flight time lowers the total mission cost along with reducing the potential wear on the spacecraft systems due to radiation or interplanetary debris.

The optimal sail controls which assure a minimum time transfer are obtained using the time minimal optimization techniques described in section 3.1.2. Substituting the sail equations of motion, (3.4.9) - (3.4.14), into (3.1.33) gives the corresponding Hamiltonian for this system:

$$\begin{aligned} H = {} & P_r[V_r] + P_\phi\left[\frac{V_\phi}{r\cos(\psi)}\right] + P_\psi\left[\frac{V_\psi}{r}\right] + P_{V_r}\left[\frac{V_\phi^2 + V_\psi^2}{r} - \frac{1}{r^2} + \frac{a_o \cos^3(\alpha)}{r^2}\right] \\ & + P_{V_\phi}\left[-\frac{V_r V_\phi}{r} + \frac{V_\phi V_\psi \tan(\psi)}{r} + \frac{a_o \cos^2(\alpha)\sin(\alpha)\sin(\beta)}{r^2}\right] \\ & + P_{V_\psi}\left[-\frac{V_r V_\psi}{r} - \frac{V_\phi^2 \tan(\psi)}{r} + \frac{a_o \cos^2(\alpha)\sin(\alpha)\cos(\beta)}{r^2}\right] \end{aligned} \qquad (3.4.15)$$

Applying (3.1.35) to the above relation produces the following auxiliary system of differential equations:

$$\dot{P}_r = -\frac{\partial H}{\partial r} = \frac{P_\phi V_\phi}{r^2\cos(\psi)} + \frac{P_\psi V_\psi}{r^2} + \frac{P_{V_r}\left(V_\phi{}^2 + V_\psi{}^2\right)}{r^2} - \frac{2P_{V_r}}{r^3} + \frac{2P_{V_r} a_o \cos^3(\alpha)}{r^3}$$
$$- \frac{P_{V_\phi} V_r V_\phi}{r^2} + \frac{P_{V_\phi} V_\phi V_\psi \tan(\psi)}{r^2} + \frac{2P_{V_\phi} a_o \cos^2(\alpha)\sin(\alpha)\sin(\beta)}{r^3}$$
$$- \frac{P_{V_\psi} V_r V_\psi}{r^2} - \frac{P_{V_\psi} V_\phi{}^2 \tan(\psi)}{r^2} + \frac{2P_{V_\psi} a_o \cos^2(\alpha)\sin(\alpha)\cos(\beta)}{r^3} \qquad (3.4.16)$$

$$\dot{P}_\phi = -\frac{\partial H}{\partial \phi} = 0 \qquad (3.4.17)$$

$$\dot{P}_\psi = -\frac{\partial H}{\partial \psi} = -\frac{P_\phi V_\phi \sin(\psi)}{r\cos^2(\psi)} - \frac{P_{V_\phi} V_\phi V_\psi}{r\cos^2(\psi)} + \frac{P_{V_\psi} V_\phi{}^2}{r\cos^2(\psi)} \qquad (3.4.18)$$

$$\dot{P}_{V_r} = -\frac{\partial H}{\partial V_r} = -P_r + \frac{P_{V_\phi} V_\phi}{r} + \frac{P_{V_\psi} V_\psi}{r} \qquad (3.4.19)$$

$$\dot{P}_{V_\phi} = -\frac{\partial H}{\partial V_\phi} = -\frac{P_\phi}{r\cos(\psi)} - \frac{2P_{V_r} V_\phi}{r} + \frac{P_{V_\phi} V_r}{r} - \frac{P_{V_\phi} V_\psi \tan(\psi)}{r}$$
$$+ \frac{2P_{V_\psi} V_\phi \tan(\psi)}{r} \qquad (3.4.20)$$

$$\dot{P}_{V_\psi} = -\frac{\partial H}{\partial V_\psi} = -\frac{P_\psi}{r} - \frac{2P_{V_r} V_\psi}{r} - \frac{P_{V_\phi} V_\phi \tan(\psi)}{r} + \frac{P_{V_\psi} V_r}{r} \qquad (3.4.21)$$

According to (3.4.17), $P_\phi$ is constant. Using the same reasoning as in the power-limited case (section 3.2.2), this adjoint variable is set equal to zero.

The solar sail controls, $\alpha$ and $\beta$, are optimized by applying (3.1.29) to the Hamiltonian given in (3.4.15):

$$\frac{\partial H}{\partial \beta} = 0 \qquad (3.4.22)$$

$$\frac{\partial H}{\partial \alpha} = 0 \tag{3.4.23}$$

Maximizing H with respect to the clock angle yields:

$$\tan(\beta) = \frac{P_{V_\phi}}{P_{V_\psi}} \tag{3.4.24}$$

Equally valid relations for $\beta$ can be derived from geometrical interpretation of (3.4.24):

$$\sin(\beta) = \frac{P_{V_\phi}}{\sqrt{P_{V_\phi}{}^2 + P_{V_\psi}{}^2}} \tag{3.4.25}$$

$$\cos(\beta) = \frac{P_{V_\psi}}{\sqrt{P_{V_\phi}{}^2 + P_{V_\psi}{}^2}} \tag{3.4.26}$$

Maximizing H with respect to the cone angle results in a second-order polynomial in $\tan(\alpha)$. Solving the quadratic and rewriting with the aid of (3.4.25) and (3.4.26) gives:

$$\tan(\alpha) = \frac{-3P_{V_r} + \sqrt{9P_{V_r}{}^2 + 8\left(P_{V_\phi}{}^2 + P_{V_\psi}{}^2\right)}}{4\sqrt{P_{V_\phi}{}^2 + P_{V_\psi}{}^2}} \tag{3.4.27}$$

The positive root is chosen in (3.4.27) to restrict $\alpha$ to lie between 0 and 90 degrees. This constraint basically states that the thrust vector must always be directed away from the sun.

Substituting the optimal $\alpha$ and $\beta$ into the state equations, (3.4.9) - (3.4.14), and the co-state equations given by (3.4.16) - (3.4.21), results in a Hamiltonian system of eleven

differential equations ($P_\phi$ is ignorable due to the independence of H on $\phi$). Maximizing the Hamiltonian to obtain the minimum time trajectory is a two point boundary value problem, which is solved according to the procedure described in section 3.1.3. Again, only the rendezvous case is considered. The unknown quantities used in the iterative procedures are the initial values for $P_r$, $P_{V_r}$, $P_{V_\phi}$, $P_\psi$, $P_{V_\psi}$, the launch date, and the time of flight. Since the sail thrust is very small, the launch date is included in the initial guesses to provide the added flexibility required for convergence to a solution. However, another final condition is needed in addition to the six rendezvous requirements. This condition is fulfilled by satisfying the constraint that the maximized Hamiltonian be equal to a constant which is greater than or equal to zero. By examining (3.4.15), the value of the constant is seen to be insubstantial; scaling the initial auxiliary variables in the boundary value problem results in an identical trajectory with a different value for H. The value chosen for the maximized Hamiltonian is taken from Pontryagin who states that minimum time trajectories can always be accomplished in such a way that H = 1.

### 3.4.3 Spacecraft Mass Calculation

Given a characteristic acceleration, $a_o$, the solar sail must be sized to transport the desired net spacecraft mass to the rendezvous destination. This required area of the sail is obtained from equation (3.4.4). Here, it is convenient introduce the sail areal density, $\Lambda$, which is defined as the volumetric material density multiplied by the sail thickness. Also included in the definition is the allowance for the sail support structure. Therefore, areal density is given by:

$$\Lambda = \rho\delta(1 + k_s) \tag{3.4.28}$$

Substituting $\Lambda$ into equation (3.4.4) and solving for the surface area gives a useful formula for sizing the sail for a particular mission and sail material:

$$S = \frac{a_o Mnet}{P_{rad} R_f - a_o \Lambda} \tag{3.4.29}$$

After calculating the surface area, the mass of the solar sail is easily obtained by multiplying the area by $\Lambda$, also known as the sail loading:

$$Ms = \Lambda S \tag{3.4.30}$$

Another important parameter that can be obtained from the areal density of the sail is the maximum characteristic thrust acceleration that can be produced for a given sail loading. This upper limit on $a_o$ is obtained by setting the net spacecraft mass equal to zero in equation (3.4.4). Thus, the maximum $a_o$ for a particular $\Lambda$ is given by:

$$[a_o]_{max} = \frac{P_{rad} R_f}{\Lambda} \tag{3.4.31}$$

### 3.4.4 Results

Two sail sizes are considered in this study. One sail is sized to give a characteristic acceleration of 1 mm/s$^2$ while the larger sail produces an $a_o$ of 2 mm/s$^2$. The code given in Appendix B is used to generate minimum time Earth-Orpheus rendezvous trajectories over the 2000 - 2020 period for these sails.

First consider the 1 mm/s$^2$ sail. The corresponding optimal solutions are listed in Table 3.4.1. As seen in the table, five solutions exist with launch dates ranging from 3/16/02 to 3/27/18. All the trajectories basically have the same geometry with a time of flight equal to approximately 380 days. However, the flight times slightly decrease for

the later launch dates as the rendezvous point is pushed toward the perigee of Orpheus' orbit. This gives an optimum launch opportunity in 2018 with a 378 day flight time. As with the other propulsion options, similar mission geometries occur every four years, the time period at which the orientation between Earth and Orpheus repeats itself.

Table 3.4.1: Minimum Time 1 mm/s$^2$ Solar Sail Rendezvous Missions to Orpheus (2000 - 2020)

| Calendar Launch Date | Calendar Arrival Date | Time of Flight (Days) |
|---|---|---|
| 3/16/02 | 4/4/03 | 384 |
| 3/19/06 | 4/6/07 | 383 |
| 3/22/10 | 4/8/11 | 382 |
| 3/25/14 | 4/9/15 | 380 |
| 3/27/18 | 4/9/19 | 378 |

The ecliptic projection of the interplanetary transfer is illustrated in Figure 3.4.2. This graph displays a characteristic common to most solar sail rendezvous trajectories. Before rendezvous, the sail passes outside the orbit of the target and then decreases its radius while increasing speed to meet the target's position and velocity. The control angles which generate the 2018 trajectory are shown in Figure 3.4.3, while the auxiliary variables that produce these optimal controls are given in Figure 3.4.4. The corresponding histories of the state variables are displayed in Figures 3.4.5 - 8.

Figure 3.4.5 shows the characteristic mentioned earlier. The spacecraft spirals away from the Earth at a radius of 1 A.U. and zero $\psi$, reaches a maximum radius, and then falls back in toward the sun to rendezvous with Orpheus. The rendezvous point is at a radius of 0.91 A.U. and celestial longitude and latitude equal to 77.1 deg. and -2.5 deg., respectively. Due to the small inclination of Orpheus' orbit (only 2.7 deg.), the spacecraft

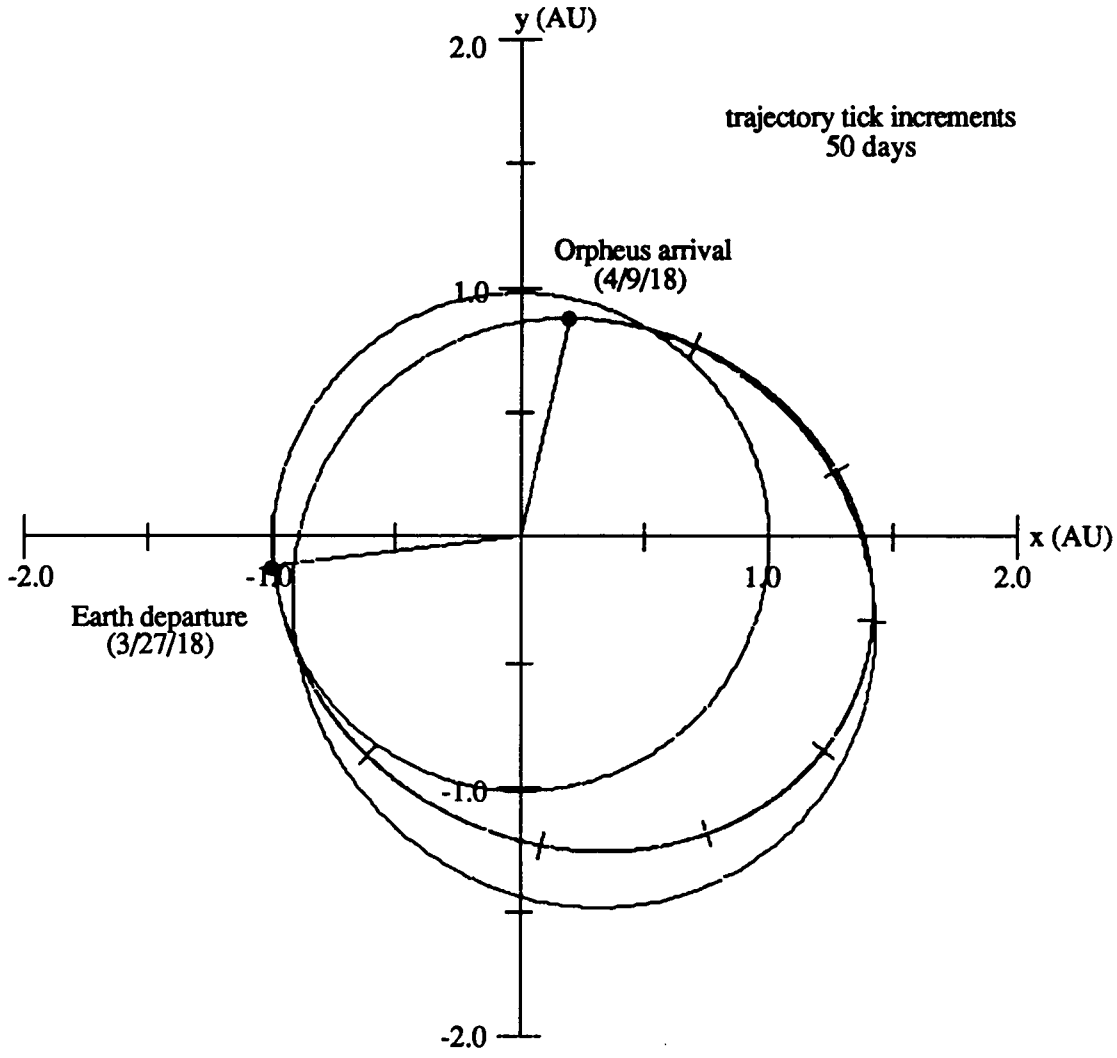


Figure 3.4.2: Ecliptic Projection of Optimal 1 mm/s$^2$ Solar Sail Earth-Orpheus Rendezvous Trajectory
(Time of Flight = 378 Days)

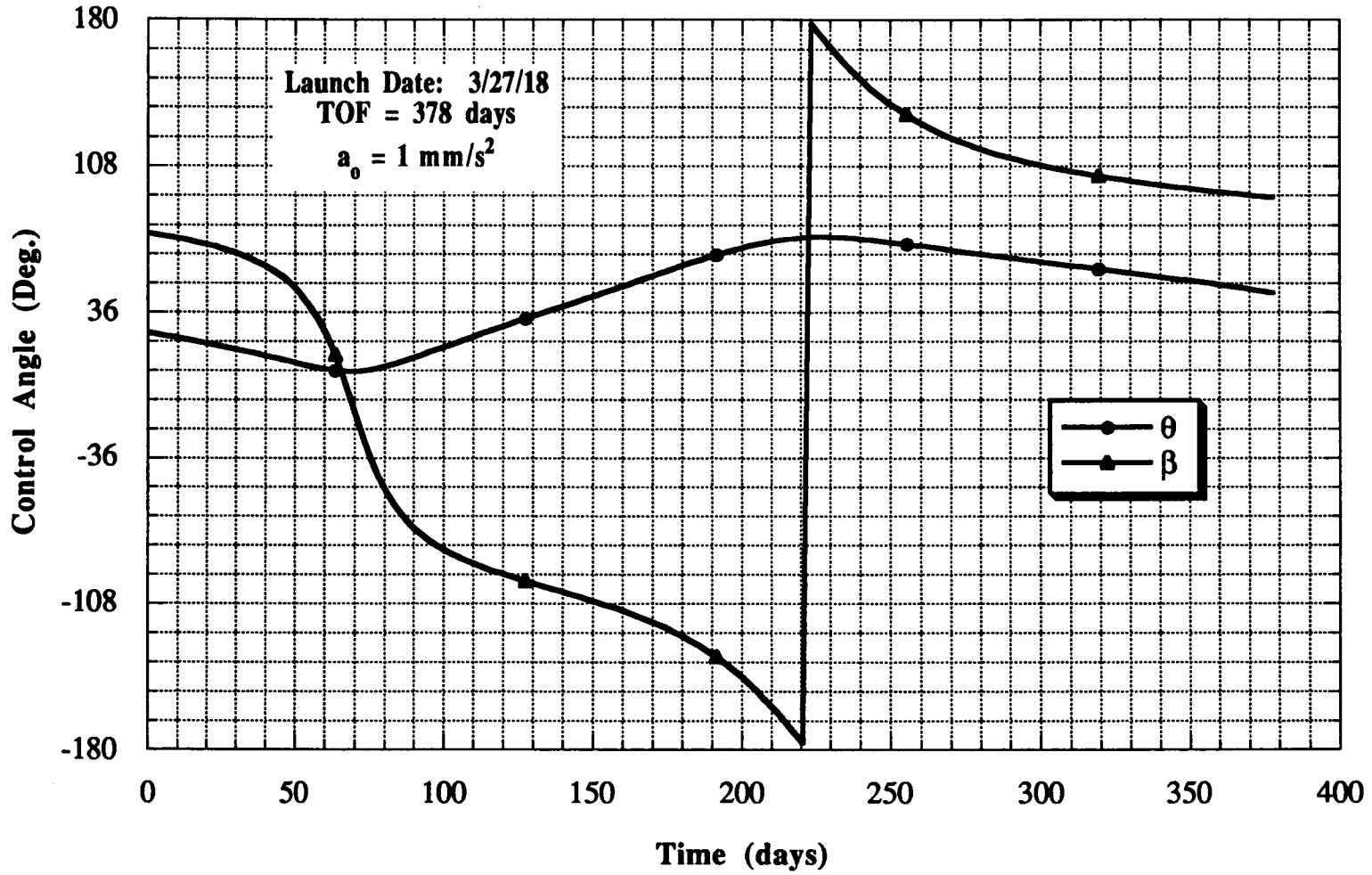


Figure 3.4.3: Solar Sail Control Angle Histories for Optimal 2018 Earth-Orpheus Mission ($a_o$ = 1 mm/s$^2$)

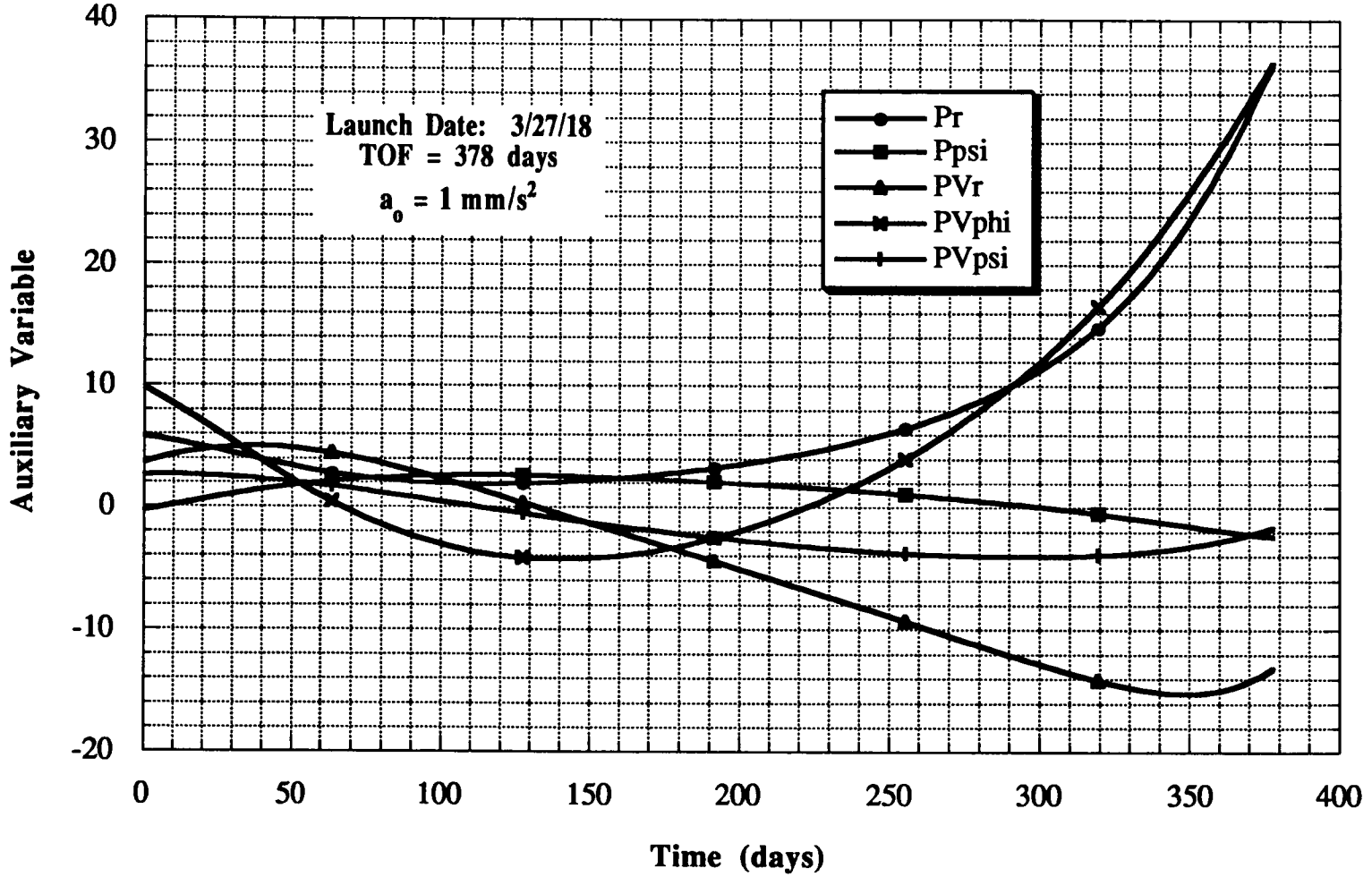


Figure 3.4.4: Solar Sail Auxiliary Variable Histories for Optimal 2018 Earth-Orpheus Mission ($a_o$ = 1 mm/s$^2$)

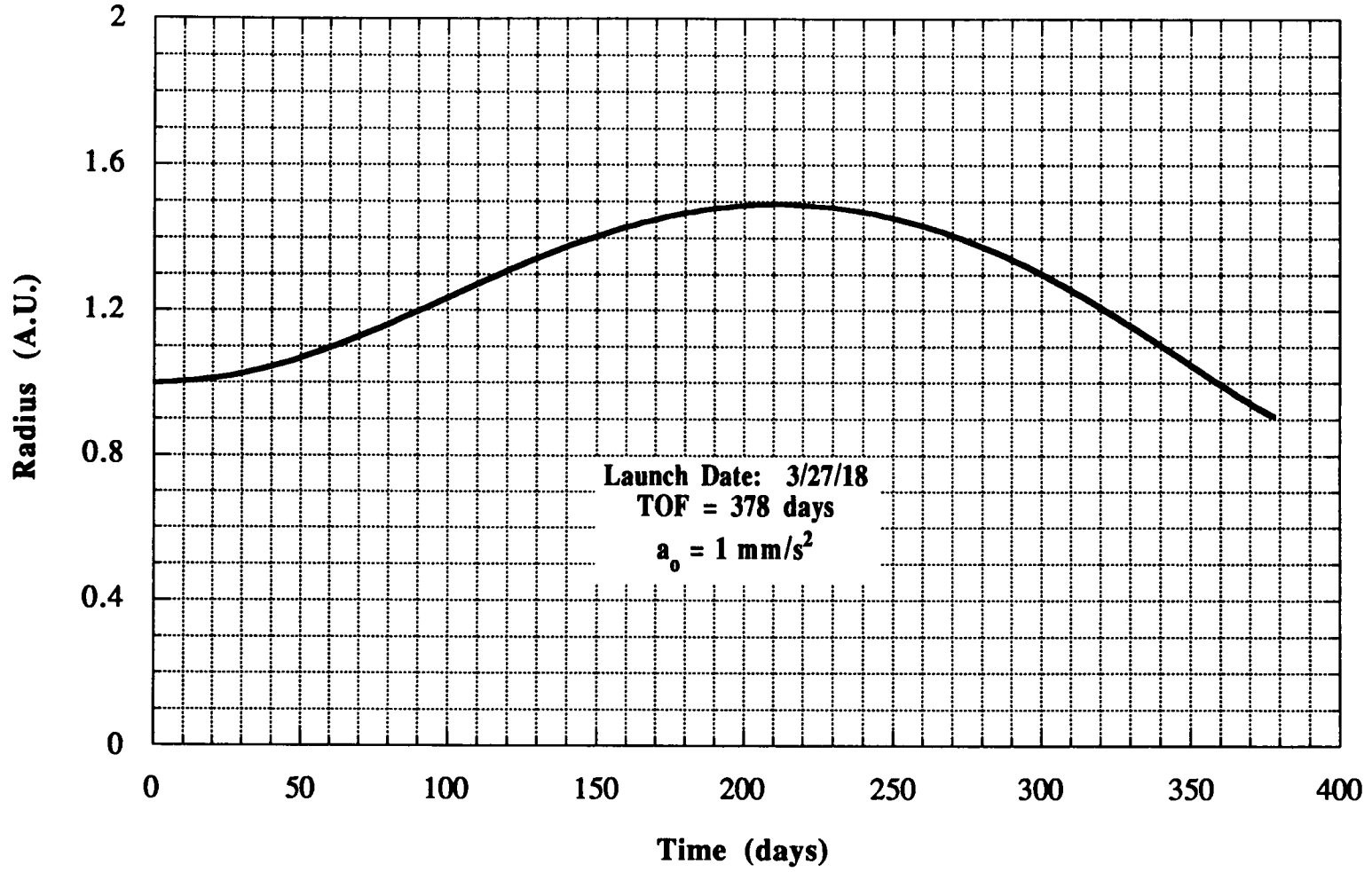


Figure 3.4.5: Solar Sail Radial Position History for Optimal 2018 Earth-Orpheus Mission ($a_o$ = 1 mm/s$^2$)

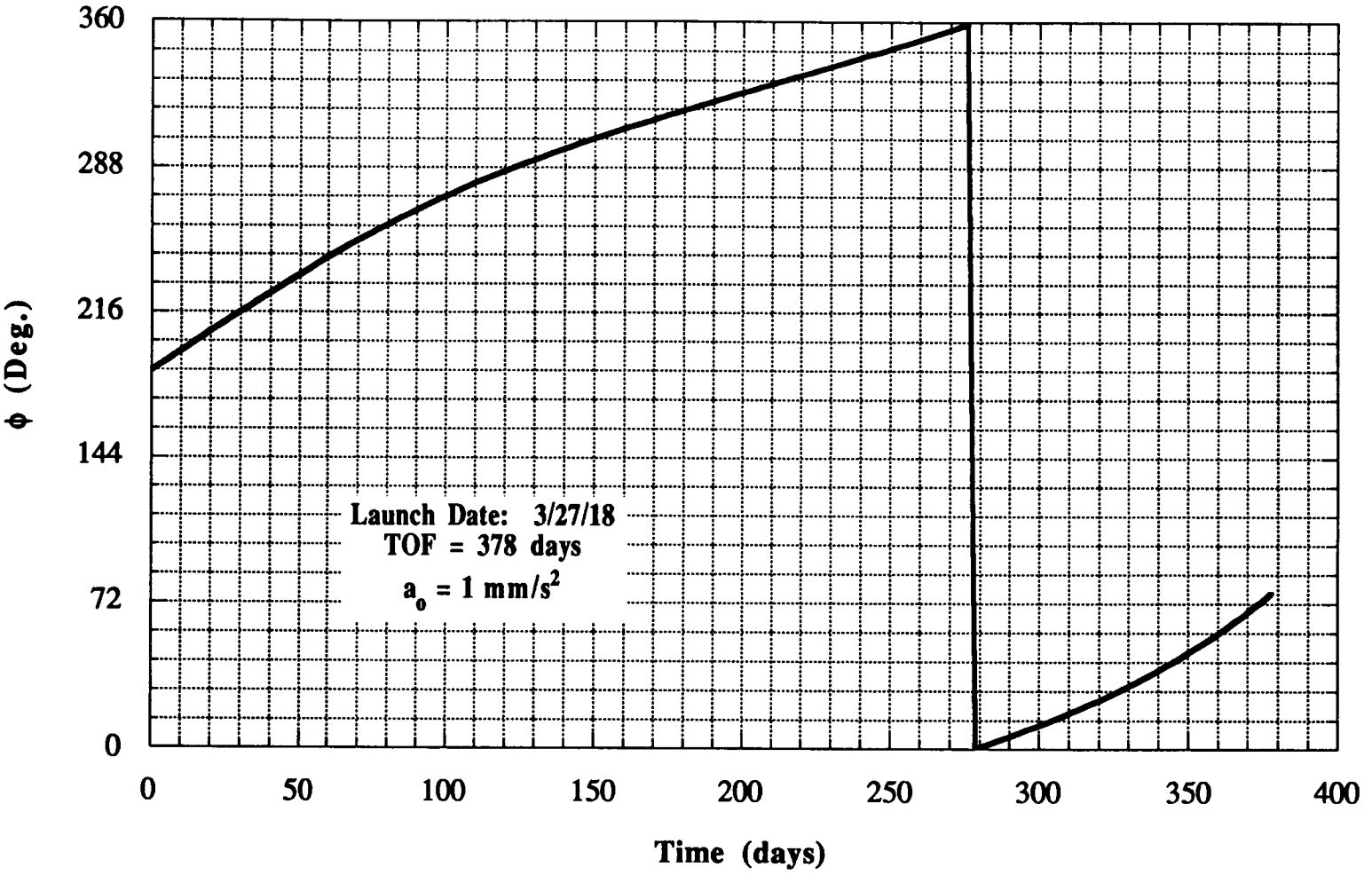


Figure 3.4.6: Solar Sail Celestial Longitude History for Optimal 2018 Earth-Orpheus Mission ($a_o$ = 1 mm/s$^2$)

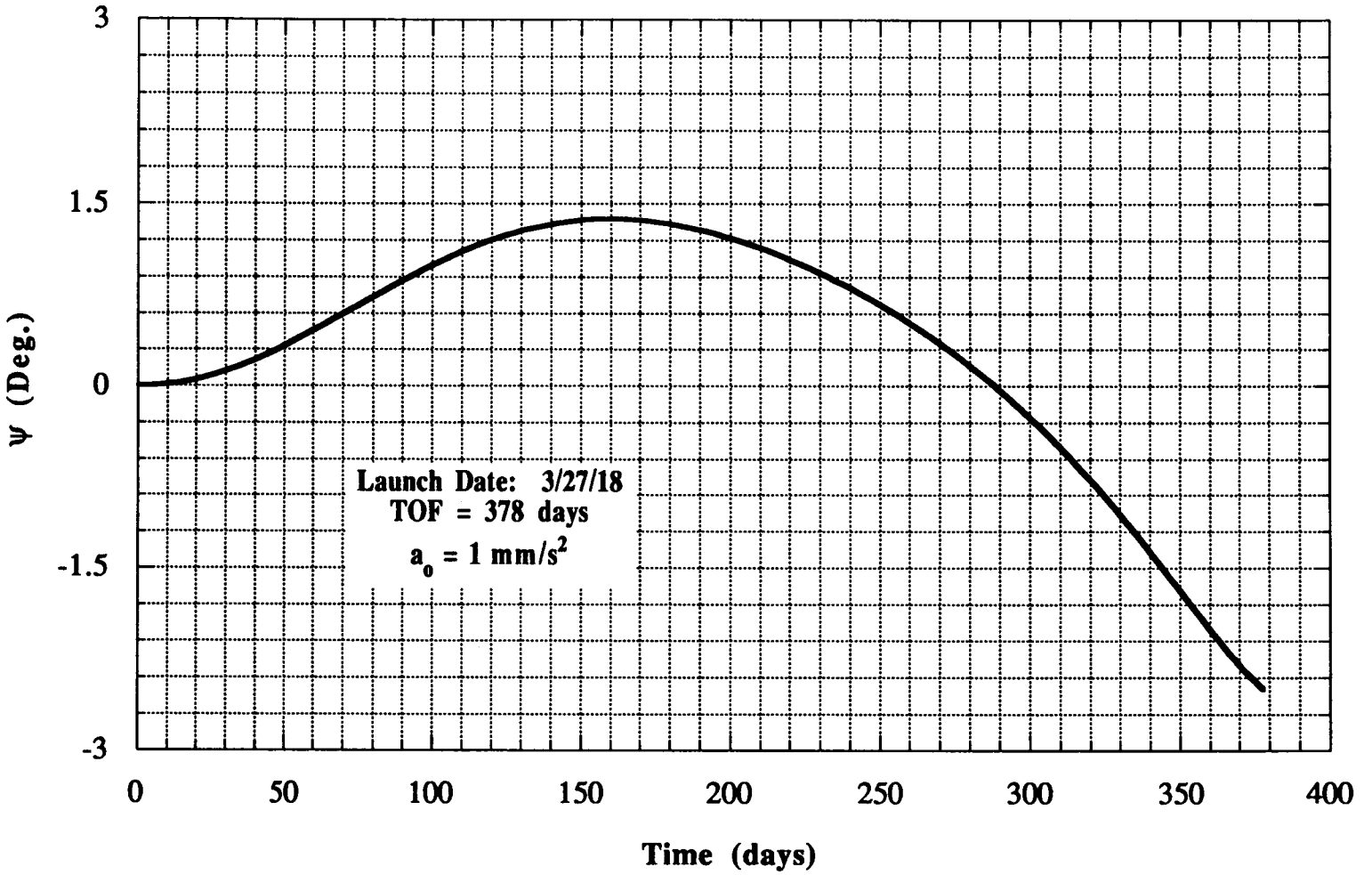


Figure 3.4.7: Solar Sail Celestial Latitude History for Optimal 2018 Earth-Orpheus Mission ($a_o$ = 1 mm/s$^2$)

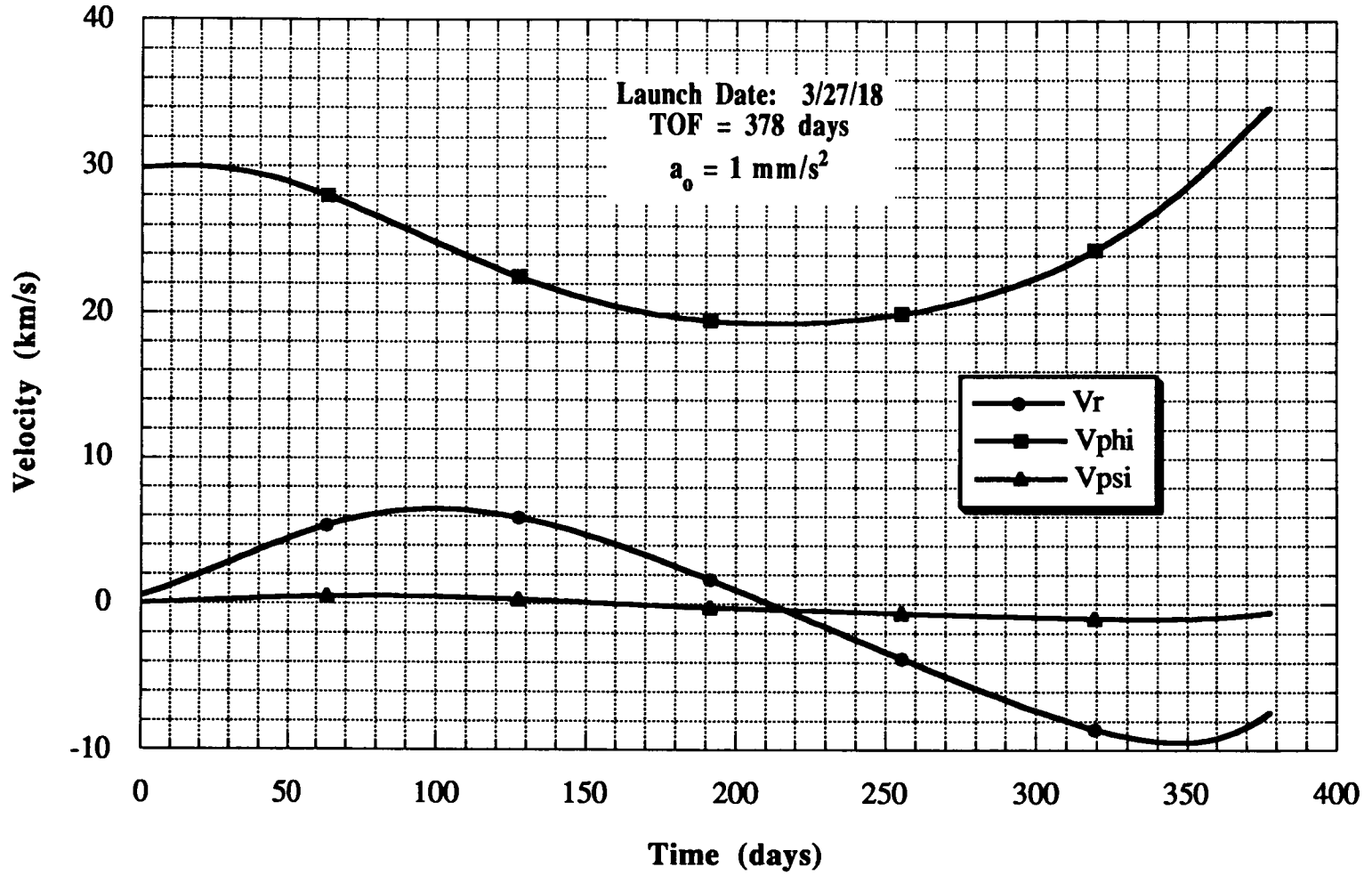


Figure 3.4.8: Solar Sail Velocity Component Histories for Optimal 2018 Earth-Orpheus Mission ($a_o$ = 1 mm/s$^2$)

velocity in the $\psi$ direction remains at a very low and almost constant value around 0 km/s.

Now consider the larger 2 mm/s$^2$ sail. The minimum time solutions obtained for this spacecraft are listed in Table 3.4.2. Unlike the 1 mm/s$^2$ case, two different mission geometries are found using the larger sail, and, again, each geometry appears four years apart. The first geometry is a 168 day opportunity launching from Earth on 8/31/01 and reaching Orpheus just before the asteroid travels through perigee. No other trajectories of this type exist over the period examined in this study. This may be due to the rendezvous point being pushed pass the perigee of Orpheus' orbit. With rendezvous before perigee, the sail's radial distance from the sun is decreasing while its speed increases. Since the sail is traveling closer to the sun, more thrust can be produced by the sail. This allows the spacecraft to meet Orpheus' position and velocity even though the orbital speed is at its greatest. However, when attempting to rendezvous past the perigee of Orpheus, the sail is moving away from the sun. Therefore, the thrust capability of the sail is constantly decreasing. This reduction in flexibility along with the asteroid's increased orbital

Table 3.4.2: Minimum Time 2 mm/s$^2$ Solar Sail Rendezvous Missions to Orpheus (2000 - 2020)

| Calendar Launch Date | Calendar Arrival Date | Time of Flight (Days) |
|---|---|---|
| 8/31/01 | 2/15/02 | 168 |
| 4/24/02 | 3/9/03 | 319 |
| 4/27/06 | 3/13/07 | 320 |
| 4/30/10 | 3/16/11 | 320 |
| 5/3/14 | 3/17/15 | 318 |
| 5/6/18 | 3/18/19 | 316 |

velocity prevents a minimum time rendezvous just after perigee for the 2 mm/s$^2$ sail. Larger sails with greater characteristic accelerations may be able to overcome this problem.

The second mission geometry possesses an interplanetary transfer time of approximately 320 days. The best opportunity among these trajectories launches from the departure planet and requires 316 days to reach the destination. This time of flight is much greater than that in the first geometry . Therefore, the optimum 2 mm/s$^2$ mission is the 168 day opportunity departing from Earth on 8/31/01.

The transfer schematic for the minimum time 2 mm/s$^2$ solar sail mission is shown in Figure 3.4.9. As mentioned earlier, the spacecraft rendezvous with Orpheus is just before perigee. The optimal controls and auxiliary variables that produce this trajectory are shown in Figures 3.4.10 and 3.4.11, respectively. The variations of the state variables are over the mission time are displayed in Figures 3.4.12 - 15. The rendezvous point for this case is at a radius of 0.84 A.U. and a celestial longitude and latitude of 153.9 deg. and -1.6 deg., respectively.

The masses for the 1mm/s$^2$ and 2 mm/s$^2$ solar sail spacecraft are calculated using the method outlined in section 3.4.3. In this analysis, both sails are assumed to be deployable type systems, which are constructed on Earth and then folded into the launch vehicles; after launch , the spacecraft travels outside the sphere of influence of Earth's gravity where the sail is then unfolded to produce thrust. The deployable type sails must be relatively thick to survive folding on the ground, packing into the launch vehicle, and then unfolding in space. Staehle has considered a 2.5 micron thick sail composed of coated Kapton [TM of Du Pot] (5). This configuration has a total areal density of 5 g/m$^2$. Friedman states that a thickness of 2.0 microns is reasonable for the first operational solar sail (11). Therefore, the sails analyzed in this study are assumed to be constructed of 2.0

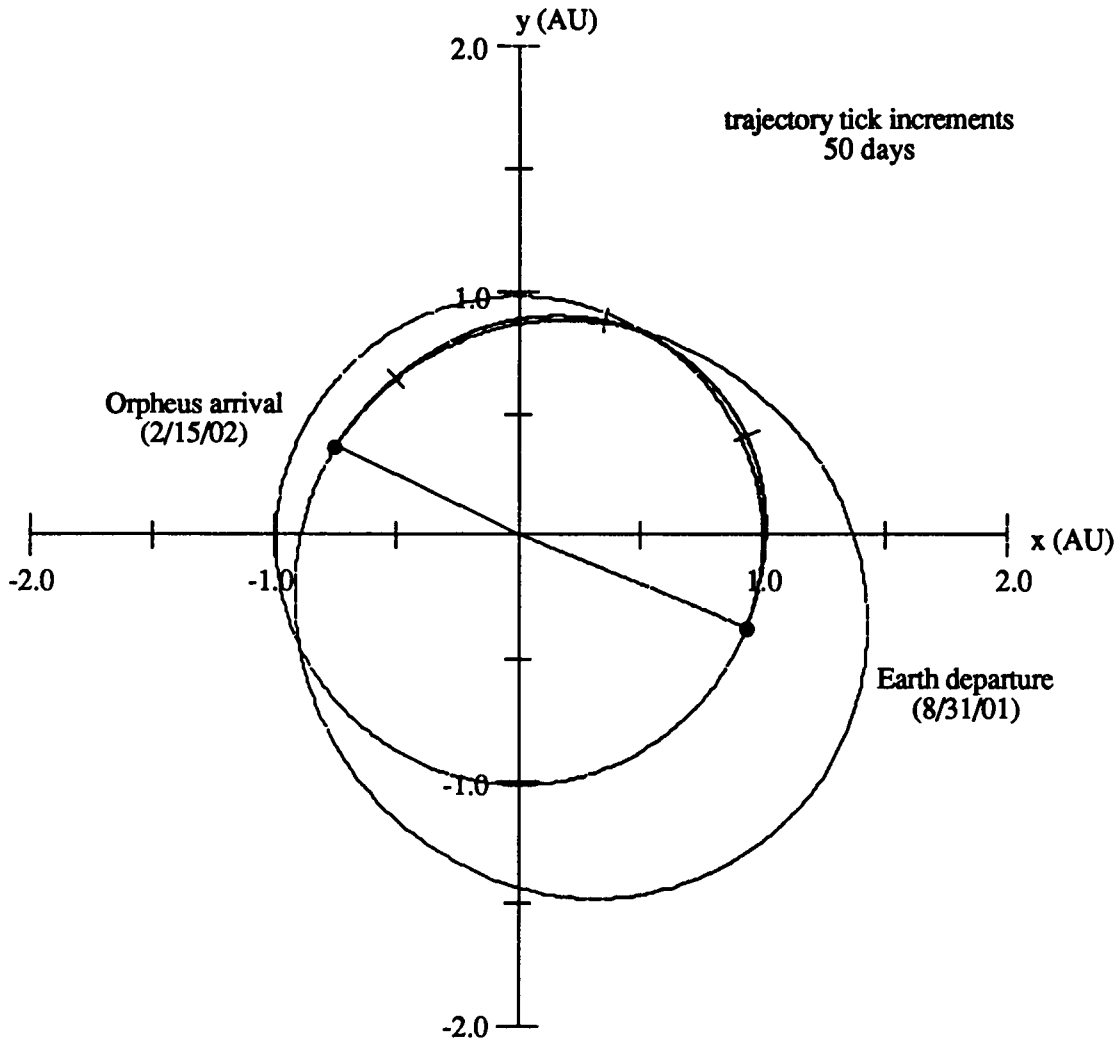


Figure 3.4.9: Ecliptic Projection of Optimal 2 mm/s$^2$ Solar Sail Earth-Orpheus Rendezvous Trajectory
(Time of Flight = 168 Days)

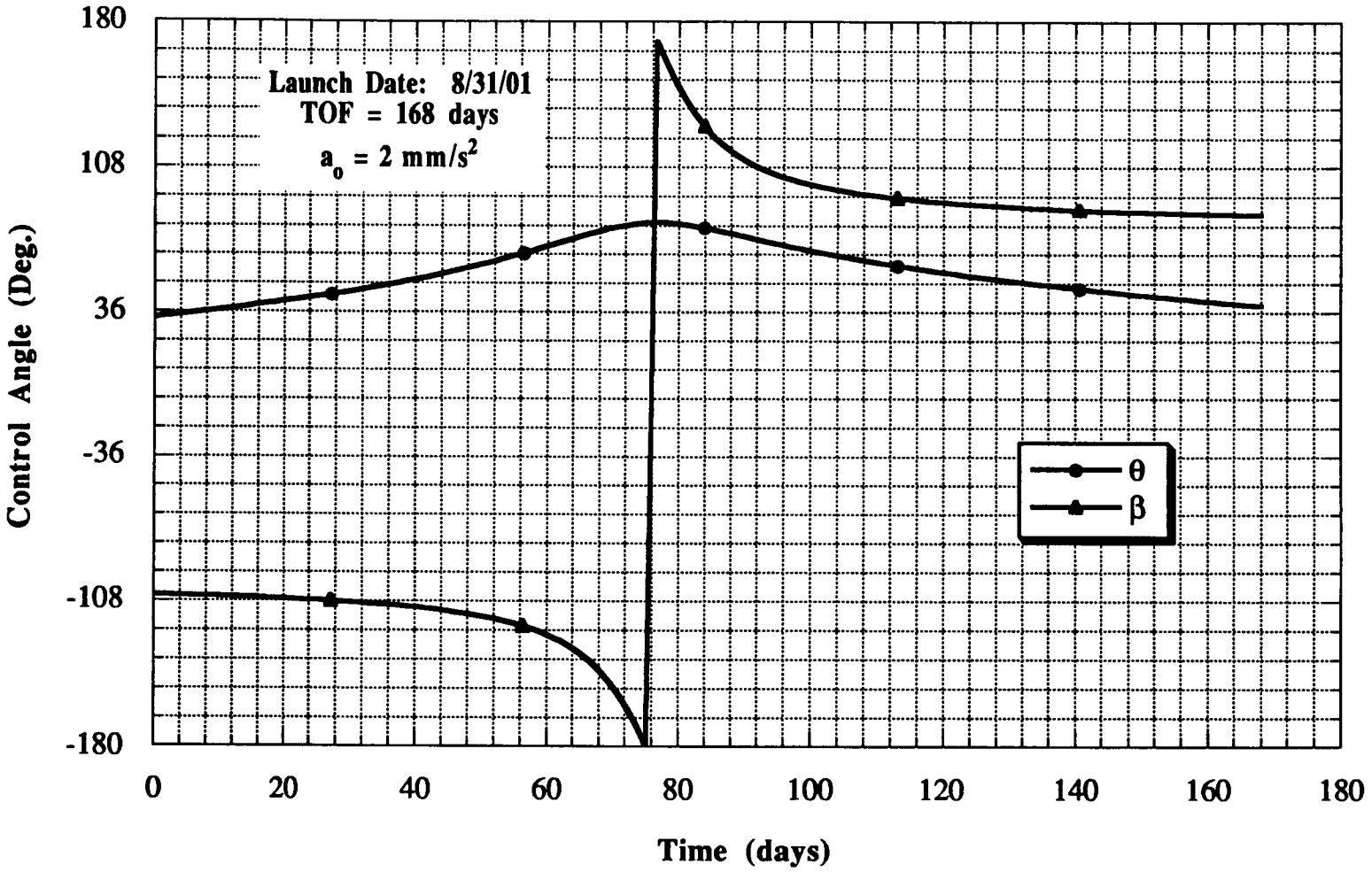


Figure 3.4.10: Solar Sail Control Angle Histories for Optimal 2001 Earth-Orpheus Mission ($a_o$ = 2 mm/s$^2$)

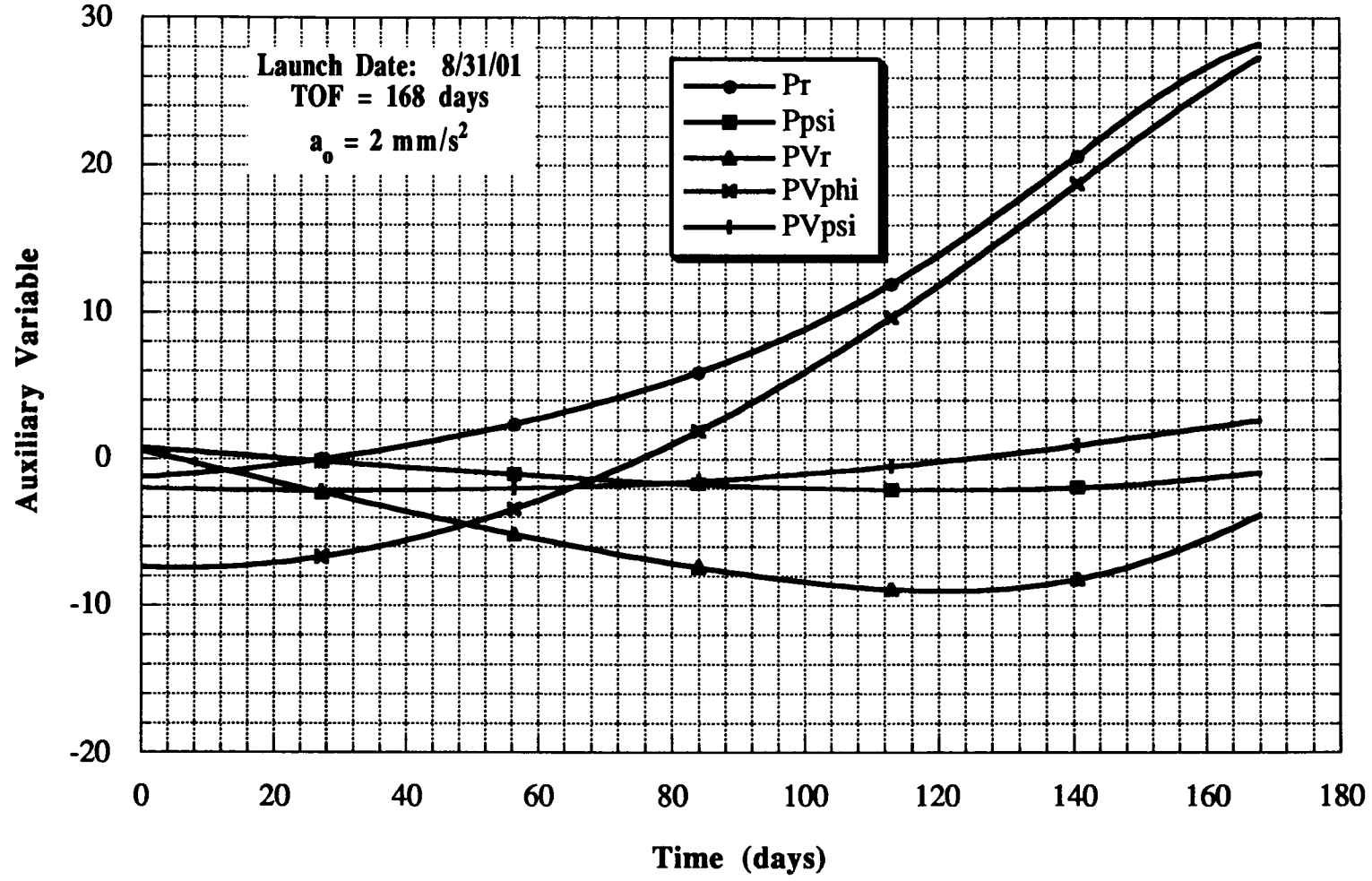


Figure 3.4.11: Solar Sail Auxiliary Variable Histories for Optimal 2001 Earth-Orpheus Mission ($a_o$ = 2 mm/s$^2$)

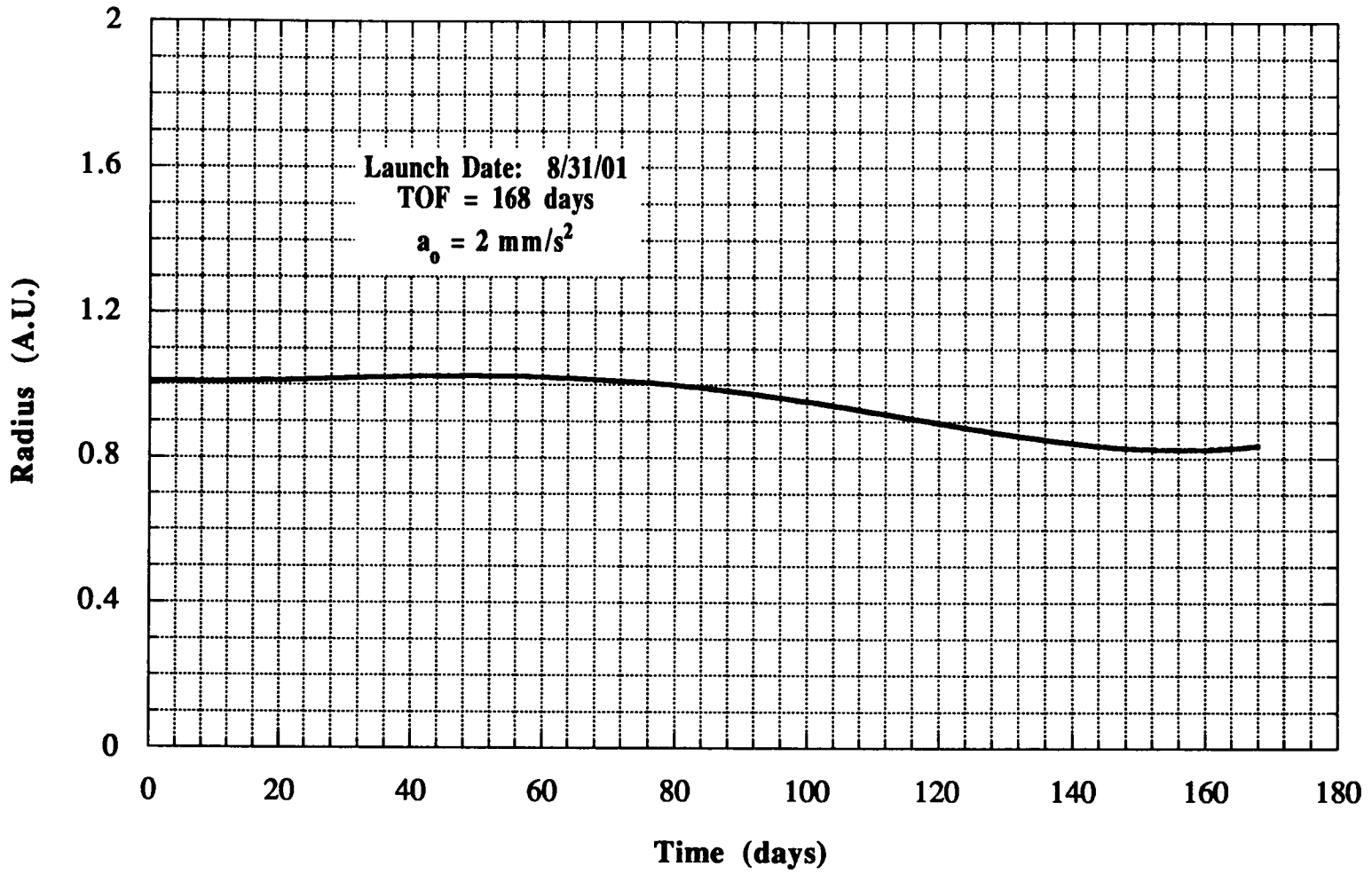


Figure 3.4.12: Solar Sail Radial Position History for Optimal 2001 Earth-Orpheus Mission ($a_o$ = 2 mm/s$^2$)

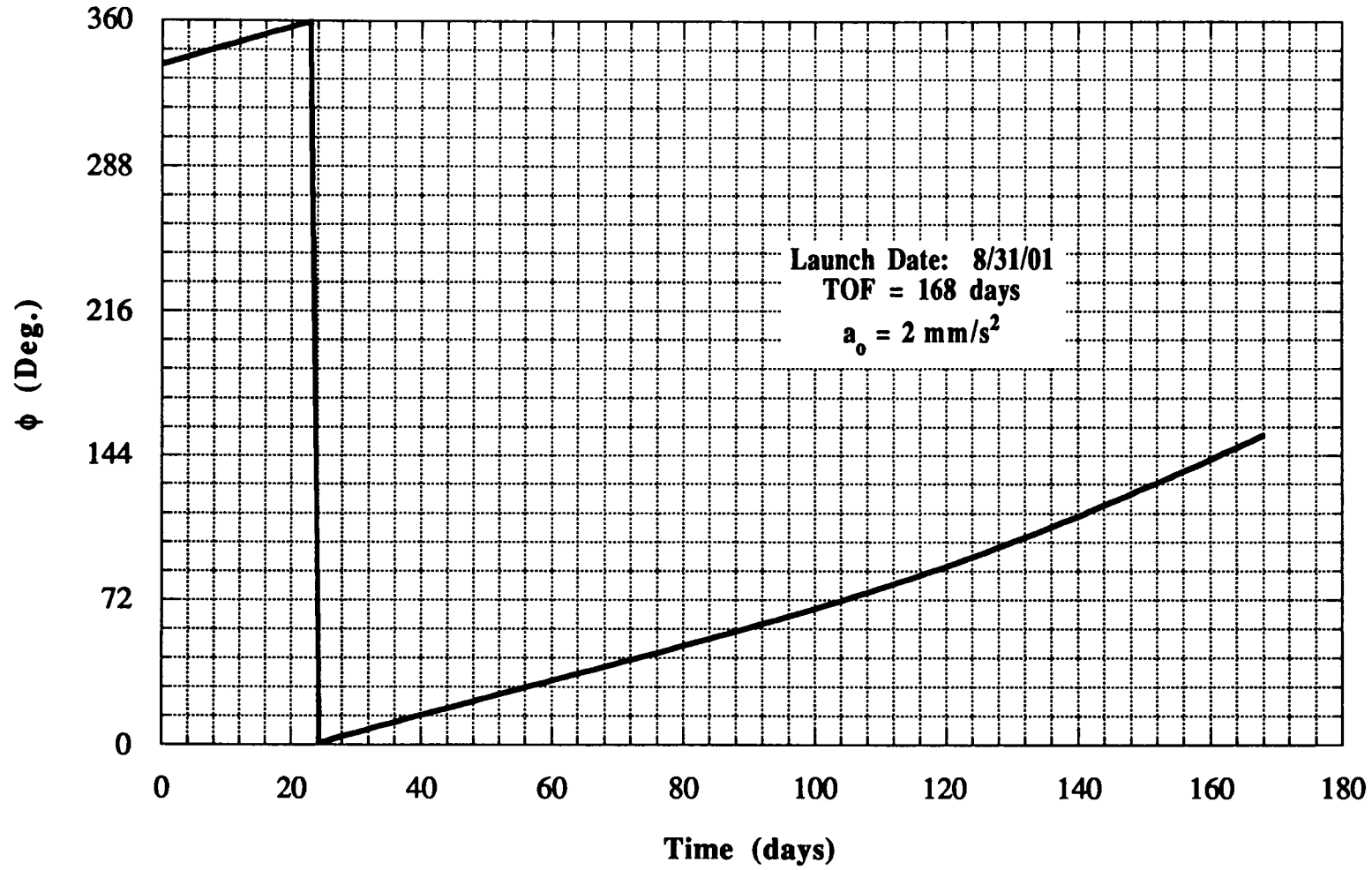


Figure 3.4.13: Solar Sail Celestial Longitude History for Optimal 2001 Earth-Orpheus Mission ($a_o$ = 2 mm/s$^2$)

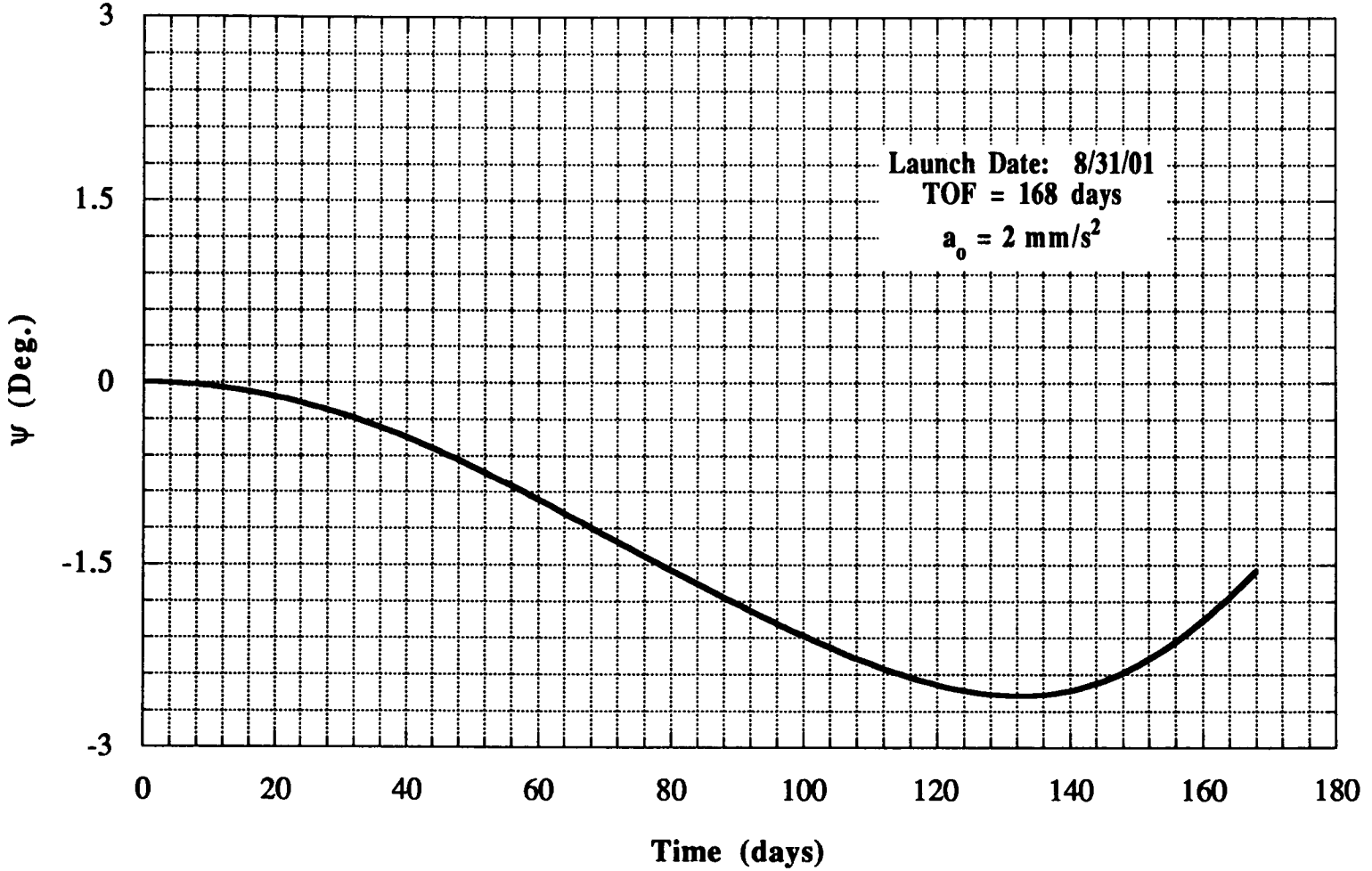


Figure 3.4.14: Solar Sail Celestial Latgitude History for Optimal 2001 Earth-Orpheus Mission ($a_o$ = 2 mm/s$^2$)

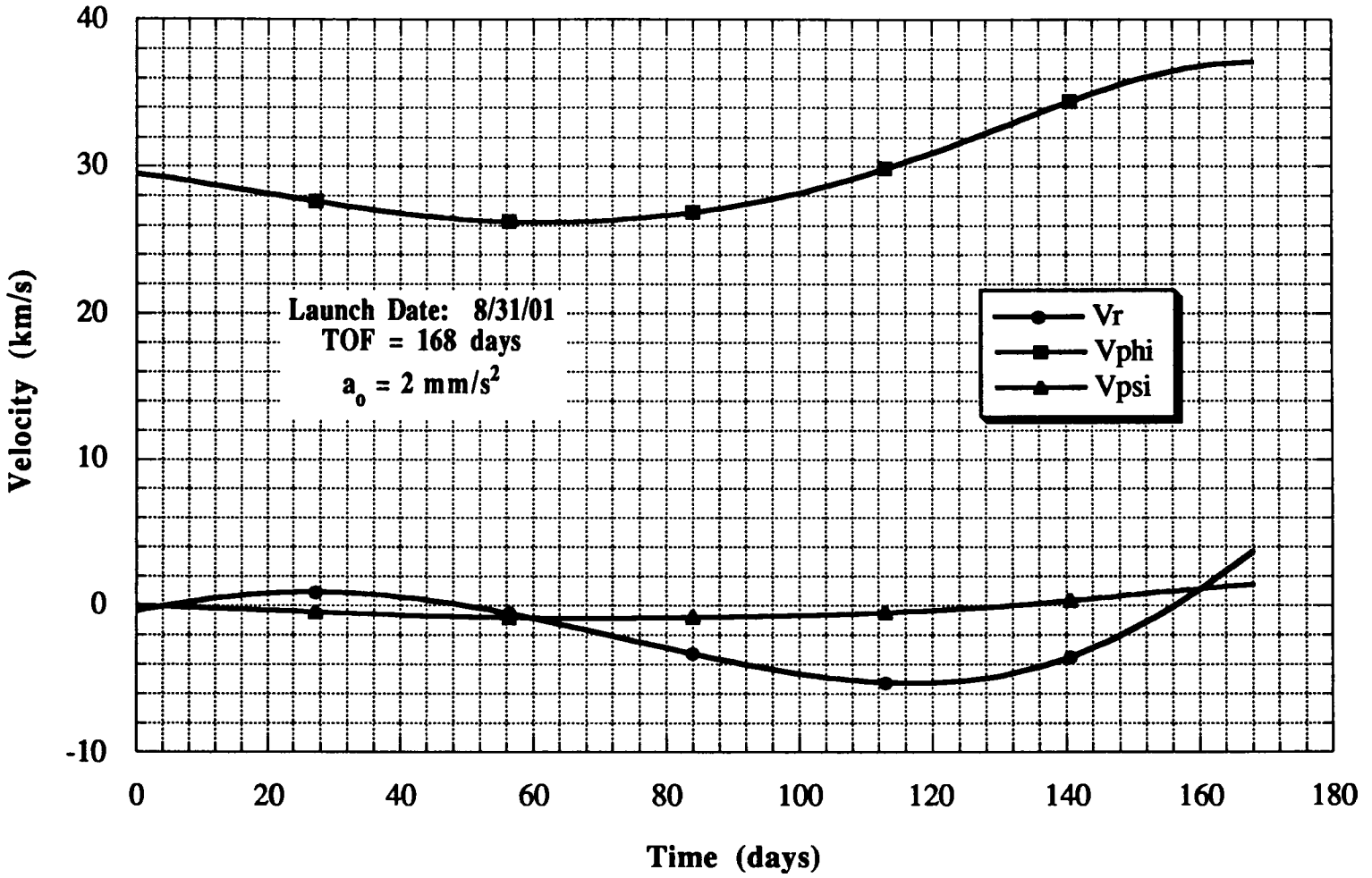


Figure 3.4.15: Solar Sail Velocity Component Histories for Optimal 2001 Earth-Orpheus Mission ($a_o$ = 2 mm/s$^2$)

micron thick, coated Kapton with a reflectivity of 0.95.

The areal density of 5 g/m$^2$ given above is for a sail thickness of 2.5 microns. The loading for the 2.0 micron sail can be calculated from equation (3.4.28) by assuming the structural correction is the same for both cases. With this in mind, the new areal density is:

$$\Lambda_{2.0} = \frac{\delta_{2.0}}{\delta_{2.5}}\Lambda_{2.5} = \left(\frac{2.0}{2.5}\right)(5) = 4.0\frac{g}{m^2}$$

The maximum characteristic acceleration that can be achieved by a sail with this loading and a reflectivity of 0.95 is 2.14 mm/s$^2$. Both sails chosen for the mission analysis have characteristic values below this upper limit. Using the 4.0 g/m$^2$ sail loading in the relations given in section 3.4.3, the initial spacecraft masses and required sail sizes for the 1 mm/s$^2$ and 2 mm/s$^2$ vehicles are calculated and listed in Table 3.4.3. For the $a_o$ = 1 mm/s$^2$ mission, the sail area required to transport a mass of 256.25 kg to Orpheus in 378 days is 0.0563 km$^2$. This corresponds to a sail mass of 225.27 kg resulting in an injected spacecraft mass of only 481.52 kg. Assuming the same areal density, the sail surface area required to achieve the 2 mm/s$^2$ characteristic acceleration is 0.9318 km$^2$, which is almost 17 times larger than the 1 mm/s$^2$ sail. This translates into a sail mass of 3727.27 kg and a total injected spacecraft mass of 3983.52 kg. These results show the large mass penalty paid for the shorter flight time of 168 days. Since the Earth-Orpheus rendezvous mission is unmanned, the decrease in time of flight does not justify the large increase in mass. Also, the 3983.52 kg spacecraft requires a larger launch vehicle than the 481.52 kg vehicle. Therefore, the 1 mm/s$^2$ solar sail mission represents a more attractive option with a reasonable flight time of 378 days, which is comparable to 350 day transfer time in the previously analyzed missions.

Table 3.4.3: Mass Summary for Optimal Solar Sail Missions

| **Characteristic Acceleration ($mm/s^2$)** | 1.0 | 2.0 |
|---|---|---|
| Sail Area ($km^2$) | 0.0563 | 0.9318 |
| Areal Density ($g/m^2$) | 4.0 | 4.0 |
| Sail Thickness (microns) | 2.0 | 2.0 |
| Reflectivity | 0.95 | 0.95 |
| Spacecraft Bus and Payload Mass (kg) | 250.00 | 250.00 |
| OMS/Payload Adapter Mass (kg) | 6.25 | 6.25 |
| Sail Mass (kg) | 225.27 | 3727.27 |
| Total Injected Spacecraft Mass (kg) | 481.52 | 3983.52 |

# 4. CONCLUSIONS

In analyzing low cost options for a Earth-Orpheus rendezvous mission, several advanced propulsion schemes were considered, including nuclear thermal, nuclear electric, solar electric, and solar sail propulsion. Computational procedures were developed to generate optimal interplanetary trajectories for the different propulsion models. The propellant mass and propulsion system mass required to complete these missions were then calculated. With these results, the initial mass to be launched from Earth was calculated for the different spacecraft and then compared to the vehicle mass for the mission using existing chemical propulsion technology. This injected spacecraft mass was used to compare the several options since it is an indicator of the mission costs.

The mass summaries calculated for the analyzed systems are given below in Table 4.1. All propulsion systems analyzed show mass advantages over the chemical mission except for the nuclear thermal option. For the near-Earth, low energy mission considered in this study, the reduced propellant requirement is outweighed by the increased engine mass. For a mission with a much larger initial spacecraft mass such as a manned Mars mission, nuclear thermal propulsion represents a more desirable alternative [26]. The opportunity with the lowest spacecraft mass is the nuclear electric mission with an injected mass of 406.63 kg. However, this mass is based off of the assumption that the power-thruster system mass is directly proportional to the power required from the reactor. The corresponding specific mass of 10 kg/kW for the SP-100 based system is assumed to be valid at the very low power rating of 3.66 kW. Most missions analyzed to this date have been for power ratings greater than 100 kW, which is the design output of the initial SP-100 reactor. Therefore, the NEP results given below are considered to be overly optimistic for a near term mission.

Table 4.1: Mass Comparison for Optimal Earth-Orheus Rendezvous Missions

| **Propulsion Option** | Chemical | NTP (600 kg Engine) | NEP (3.66 kW) | SEP (3.70 kW) | Solar Sail ($a_o$ = 1 mm/s$^2$) |
|---|---|---|---|---|---|
| Launch Date | 1/31/02 | 1/31/06 | 7/22/05 | 6/30/17 | 3/27/18 |
| Arrival Date | 1/16/03 | 1/16/07 | 7/7/06 | 6/15/18 | 4/9/18 |
| TOF (Days) | 350 | 350 | 350 | 350 | 378 |
| Propulsion System Mass (kg) | 122.91 | 868.05 | 109.25 | 343.22 | 225.27 |
| Total Propellant Mass (kg) | 629.31 | 464.07 | 41.13 | 137.37 | N.A. |
| Total Injected Spacecraft Mass (kg) | 1008.47 | 1588.37 | 406.63 | 736.84 | 481.52 |

Solar electric propulsion also provides a significant mass savings (27%) compared to the chemical mission. To rendezvous with Orpheus, this scheme requires 137.37 kg of propellant and a 343.22 kg propulsion system. This leads to an initial spacecraft mass of 736.84 kg, which may be easily launched aboard a Delta-class launch vehicle. Solar sail technology provides an even more desirable option with a 52% mass savings over the chemical system. The sail size which provides a low mass and a flight time comparable to the impulsive missions has a characteristic acceleration of 1 mm/s$^2$. For this sail, the time required to rendezvous with Orpheus is 378 days, less than a month longer than in the chemical case. The mass of this 0.0563 km$^2$ sail is 225.27 kg, resulting in a total initial spacecraft mass of only 481.52 kg.

Due to the inherent uncertainties in the NEP mass calculations, nuclear electric propulsion is ruled out for the Earth-Orpheus rendezvous mission. Also, given the current public concern with transporting nuclear materials into orbit and the lack of

technological development of nuclear propulsion, a NEP (or NTP) mission as early as the year 2005 is very optimistic. Therefore, solar electric and solar sail propulsion represent the most desirable options for a near term, near-Earth asteroid rendezvous mission. Of these two schemes, the solar sail provides the most cost effective venture.

With no propellant being consumed by the propulsion system, solar sails represent an attractive mode of propulsion in asteroid mining missions. After the initial mission, the sail may be used to continually transport mined materials and supplies between Earth and the asteroid mining station. Therefore, after construction and launch, the economic and scientific benefits gained by the initial solar sail spacecraft may continue to be exploited at little extra cost. These combined factors may stimulate commercial activity in interplanetary ventures and promote further space exploration. Based on these results, high priority should be given to the research and development of solar sail propulsion system technology.

# LIST OF REFERENCES

## LIST OF REFERENCES

(1) Farquhar, R., Jen, Shao-Chiang, McAdams, J. V., "Extended-Mission Opportunities for a Discovery-Class Asteroid Rendezvous Mission", AAS 93-127, AAS/AIAA Spaceflight Mechanics Meeting, Pasadena, CA, February 22-24, 1993.

(2) McDonough, T. R., Space: The Next Twenty-five Years, John Wiley & Sons, Inc., New York, NY, 1987.

(3) Schulz, R. J., Lecture Notes in Rocket Propulsion, Vol. I, pp. 5-10, UT Space Institute, Tullahoma, TN, 1985.

(4) Rosen, R., Reck, G. M., Bennett, G. L., "Application of Nuclear Propulsion to Mars Missions", IAF-91-182, 42nd Congress of the International Astronautical Federation, Montreal, Canada, October 5-11, 1991.

(5) Frisbee, R. H., et al., "Advanced Propulsion Options for the Mars Cargo Missions", AIAA 90-1997, AIAA/SAE/ASME/ASEE 26th Joint Propulsion Conference, Orlando, FL, July 16-18, 1990.

(6) Powell, J. R., et al., "Nuclear Propulsion Systems for Orbit Transfer based on the Particle Bed Reactor", Space Nuclear Power Systems, ed. by M. S. El-Genk and M. D. Hoover, Orbit Book Company, Malabar, FL, 1989, Chapter 28.

(7) Jones, R. M., and Sauer, C. G., "Nuclear Electric Propulsion Missions", *Journal of the British Interplanetary Society*, Vol. 37, 1984, pp. 395-400.

(8) Gilland, J. H., "Mission and System Optimization of Nuclear Electric Propulsion Vehicles for Lunar and Mars Missions", IEPC-91-038, International Electric Propulsion Conference, Viareggio, Italy, October 1991.

(9) Sauer, C. G., "Planetary Mission Performance for Small Solar Electric Propulsion Spacecraft", AAS 93-561, AAS/AIAA Astrodynamics Specialist Conference, Victoria, B.C., Canada, August 16-19, 1993.

(10) Wright, J. L., Space Sailing, Gordon and Breach Science Publishers, Philadelphia, PA, 1992.

(11) Friedman, L., Starsailing: Solar Sails and Interstellar Travel, John Wiley & Sons, Inc., New York, NY, 1991.

s(12) Bate, R. R., Mueller, D. D., and White, J. E., Fundamentals of Astrodynamics, Dover publications, Inc., New York, NY, 1971.

(13) Battin, R. H., An Introduction to the Mathematics and Methods of Astrodynamics, AIAA, Inc., New York, NY, 1987.

(14) Wertz, J. R. and Larson, W. J. (editors), Space Mission Analysis and Design, Kluwer Academic Publishers, Boston, MA, 1991.

(15) McAdams, J. V., "Mission Options for Rendezvous with the Most Accessible Near-Earth Asteroid - 1989 ML", AAS 91-397, 1991.

(16) Bennett, G. L. and Miller, T. J., "The NASA Program Plan for Nuclear Propulsion", *AIP Conference Proceedings of the Eighth Symposium on Space Nuclear Power Systems*, CONF-910116, Albuquerque, NM, January 6-10, 1991, pp. 524-530.

(17) Pontryagin, L.S., Boltyanskii, V. G., Gamkrelidze, R. V., and Mishchenko, E. F., The Mathematical Theory of Optimal Processes, The Macmillan Company, New York, NY, 1964.

(18) Burden, R. L. and Faires, J. D., Numerical Analysis, PWS-Kent Publishing Company, Boston, MA, 1989.

(19) Lebedev, V. N., "Calculation of the Motion of a Low Thrust Spacecraft", NASA TTF-586, Washington D.C., November 1969.

(20) Melbourne, W. G., "Interplanetary Trajectory and Payload Capabilities of Advanced Propulsion Vehicles", JPL Technical Report No. 32-68, Jet Propulsion Laboratory, Pasadena, CA, 1961.

(21) Marec, J. P., Optimal Space Trajectories, Elsevier Scientific Publishing Company, Amsterdam, The Netherlands, 1979.

(22) Melbourne, W. G. and Sauer, C. G., "Optimal Thrust Programs for Power-Limited Propulsion Systems", JPL Technical Report No. 32-118, Jet Propulsion Laboratory, Pasadena, CA, 1961.

(23) Sauer, C. G., "Modeling of Thruster and Solar Array Characteristics in the JPL Low-Thrust Trajectory Analysis", AIAA 78-645, AIAA/DGLR 13th International Electric Propulsion Conference, San Diego, CA, April 25-27, 1978.

(24) Sauer, C. G., "Optimum Solar-Sail Interplanetary Trajectories", AIAA 76-792, AIAA/AAS Astrodynamics Conference, San Diego, CA, August 18-20, 1976.

(25) Grodzovskii, G. L., Ivanov, Y. N., and Tokarev, V. V., Mechanics of Spaceflight Low-Thrust, Translated from Russian, Israel Press, Jerusalem, 1969.

(26) McCarley, A. A., "A Propulsive Comparison Study between Chemical Rockets, Nuclear Thermal Rockets, and Solar Sails for a Manned Mars Exploration Mission", Master of Science Thesis in Aerospace Engineering, The University of Tennessee, Knoxville, August 1993.

# APPENDICES

## APPENDIX A: Impulsive Program

```
/*************************************************************************/
/*************************************************************************/
/*      Program:        3D Impulsive Trajectory (pt. to pt.) (outbound)  */
/*                                                                       */
/*      By:             H. Dan. Thomas, Brian K. Funk, A. Allen McCarley */
/*                                                                       */
/*      Date:           December 2, 1992                                 */
/*                                                                       */
/*      Last Major Revision:  April 7, 1993                              */
/*      Last Minor Revision:  April 7, 1993                              */
/*                                                                       */
/*      Description:    This ANSI C program calculates the impulsive thrust transfer */
/*                      conic between earth and select asteroids using Battin's */
/*                      method, given the desired launch date, time of flight, and */
/*                      parking orbits.                                  */
/*                                                                       */
/*************************************************************************/
/*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/device.h>

double AU = 1.4959965E+8;               /* km/au */
double GM = 3.96396415708E-14;          /* au3/s2 */
double stepdays = 1.0;                  /* time stepsize in days */
double tol = 1.0e-10;
double RAD;                             /* Conversion from radians to degrees */

double pos1[3];
double pos2[3];
double spheresize;
long zval;
Object objinit = 11, obj1 = 1, obj2 = 2, objpath = 12, objcraft = 13;
Object objAster = 14, objfinal = 15;
int count, interval;

double r[5];            /* Radial position */
double comdist;         /* Distance between earth and spacecraft */
double nu[5];           /* True anomaly */
double inclin;          /* Inclination of the transfer ellipse */
double rInitial;        /* Initial r of departure planet */
double rfinal;          /* Final r of target planet */
double R1[4];           /* Initial radius vector of spacecraft */
double R2[4];           /* final radius vector of spacecraft */
double Long[5];         /* Celestial longitude */
double LongInitial;     /* Initial celestial longitude of departure planet */
```

```
double Longfinal;       /* Final celestial longitude of target planet */
double Lat[5];          /* Celestial latitude */
double LatInitial;      /* Initial celestial latitude of departure planet */
double Latfinal;        /* Final celestial latitude of target planet */
double HCA;             /* Heliocentric angle between endpts. */
double Vel[5];          /* Velocity */
double V1;              /* Velocity at initial heliocentric position (1) */
double V2;              /* Velocity at final heliocentric position (2) */
double Vel1[4];         /* Initial velocity vector of spacecraft */
double Vel2[4];         /* Final velocity vector of spacecraft */
double pathang;         /* Flight path angle at any time */
double gamma1;          /* Flight path angle at position (1) */
double gamma2;          /* Flight path angle at position (2) */
double a;               /* Semi-major axis of transfer conic */
double ecc;             /* Eccentricity of transfer conic */
double p;               /* Oribital parameter of transfer conic */
double AngMom;          /* Angular momentum on transfer conic */
double nuInitial;       /* Initial true anomaly on transfer conic */
double nufinal;         /* Final true anomaly on transfer conic */
double EAInitial;       /* Initial eccentric anomaly on transfer conic */
double EAfinal;         /* Final eccentric anomaly on transfer conic */
double n;               /* Mean motion of transfer conic */
double C3[5];           /* Square of the hyperbolic excess speed */
double DeltaV[5];       /* Velocity increment required to enter or leave parking orbit */
double t;               /* Time measured from perigee passage on transfer conic */
double time;            /* Time ellapsed in days */
double Jt1;             /* Julian time (in centuries) */
double Jt2;             /* Julian time (in centuries) */
double Jt3;             /* Julian time (in centuries) */
double TOF;             /* Time of flight */
double launch;          /* Entered launch date */
double ClaunchT;        /* Calendar departure date */
double CarriveT;        /* Calendar arrival date */
double CdateT;          /* Calendar date along trajectory */
double Juliandate;      /* Julian date (in days) */
double Jcentury1[5];    /* Initial Julian time (in centuries) */
double Jcentury2[5];    /* Final Julian time (in centuries) */
double stepdays;        /* Stepsize in days */
double Jdt;             /* Julian stepsize (in centuries) */
double Battin_x;        /* Battin variable */
double departure[8];    /* Oribtal elements of the departure planet */
double target[8];       /* Oribtal elements of the target planet */
double Marselem[8];     /* Oribtal elements of Mars */
double spacecraft[8];   /* Oribtal elements of the spacecraft */
double Eclipt[5][4];    /* x, y, and z position components */
double EcliptV[5][4];   /* x, y, and z velocity components (ecliptic) */
double EcliptVP[5][4];  /* x, y, and z velocity components (pericentric) */
double rdum[5], Phidum[5], Psidum[5], Vrdum[5], Vphidum[5], Vpsidum[5];
double JEpoch1 = 2451545.0;
double JEpoch2 = 11.051990;
double JEpoch3 = 6.191986;
double nplan, tau;
```

```
int pickroid;
char str[15];

/*************************************************************************/
/*************************** PROCEDURES **************************/
/*************************************************************************/

void earth(double elem[8], double argt)

        /* This procedure determines the orbital elements of        */
        /* earth at a specific time.                                */

/* Epoch = 2000 AD */

{
        elem[1] = 0.0;
        elem[2] = (100.0466449 + 36000.769823*argt + 0.000304*argt*argt)/RAD;
        elem[3] = (102.937348 + 1.719539*argt + 0.000460*argt*argt)/RAD;
        elem[4] = 0.01670862 - 0.00004204*argt;
        elem[5] = 1.00001161;
        elem[6] = 0.0;
        elem[7] = 0.0;
}

/*************************************************************************/

void Mars(double elem[8], double argt)

        /* This procedure determines the orbital elements of        */
        /* Mars at a specific time.                                 */

/* Epoch = 2000 AD */

{
        elem[1] = 0.0;
        elem[2] = (355.433275 + 19141.696475*argt + 0.000311*argt*argt)/RAD;
        elem[3] = (336.060234 + 1.841044*argt + 0.000135*argt*argt)/RAD;
        elem[4] = 0.09340062 + 0.00009048*argt - 0.00000008*argt*argt;
        elem[5] = 1.52371069;
        elem[6] = (1.849726 - 0.000601*argt + 0.000013*argt*argt)/RAD;
        elem[7] = (49.558093 + 0.772096*argt + 0.000016*argt*argt)/RAD;
}

/*************************************************************************/

void XB(double elem[8], double argt)

/* Epoch = 11.051990 */

{
        elem[1] = 16.77969/RAD;
```

```
        elem[2] = 56.89879/RAD;
        elem[3] = 91.30613/RAD;
        elem[4] = 0.4466558;
        elem[5] = 1.8355863;
        elem[6] = 3.87446/RAD;
        elem[7] = 74.52644/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch2;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void Orpheus(double elem[8], double argt)

/* Epoch = 11.051990 */

{
        elem[1] = 301.56931/RAD;
        elem[2] = 197.58723/RAD;
        elem[3] = 130.70829/RAD;
        elem[4] = 0.3226346;
        elem[5] = 1.2093243;
        elem[6] = 2.68775/RAD;
        elem[7] = 189.13898/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch2;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void McAuliffe(double elem[8], double argt)

/* Epoch = 11.051990 */

{
        elem[1] = 15.59117/RAD;
        elem[2] = 283.00658/RAD;
        elem[3] = 122.49259/RAD;
        elem[4] = 0.3694649;
        elem[5] = 1.8787307;
        elem[6] = 4.77743/RAD;
        elem[7] = 106.90142/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch2;
        elem[2] = nplan*((argt*36525.0) - tau);
```

```
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void Seleucus(double elem[8], double argt)

/* Epoch = 11.051990 */

{
        elem[1] = 349.18871/RAD;
        elem[2] = 343.98574/RAD;
        elem[3] = 207.31474/RAD;
        elem[4] = 0.4571202;
        elem[5] = 2.0325801;
        elem[6] = 5.93181/RAD;
        elem[7] = 218.12603/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch2;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void Eros(double elem[8], double argt)

/* Epoch = 6.191986 */

{
        elem[1] = 178.510/RAD;
        elem[2] = 170.451/RAD;
        elem[3] = 123.006/RAD;
        elem[4] = 0.2229;
        elem[5] = 1.4582;
        elem[6] = 10.832/RAD;
        elem[7] = 304.496/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch3;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void InvConvert(double date2, double *argdate)

                        /* This procedure converts any Julian date to the      */
                        /* standard calender date.                             */
                        /* Taken from Astronomical Formulae for Calculators,   */
```

```
                    /* Meeus, p. 26.                                    */

{
long Month;         /* The portion of the enterred date that represents the month */
double Day;         /* The portion of the enterred date that represents the day */
long Year;          /* The protion of the enterred date that represents the year */
long a, b, c, d, e;
double f;

        f = date2 + 0.5;
        a = trunc(f);
        f = f - a;
        if (a >= 2299161) {
                b = (a - 1867216.25)/36524.25;
                a += 1 + b - b/4;
        }
        b = a + 1524;
        c = (b - 122.1)/365.25;
        d = 365.25*c;
        e = (b - d)/30.6001;
        Day = b - d - trunc(30.6001*e) + f;
        if (e <= 13)
                Month = e - 1;
        else
                Month = e - 13;
        if (Month <= 2)
                Year = c - 4715;
        else
                Year = c - 4716;
        *argdate = trunc(Month) + 0.01*rint(Day) + 0.000001*trunc(Year);
}

/*************************************************************************/

void Convert(double *date1)

                    /* This procedure converts any date entered in the  */
                    /* standard calender into a Julian date.            */
                    /* Taken from Astronomical Formulae for Calculators,*/
                    /* Meeus, p. 24.                                    */

{
long Month;         /* The portion of the enterred date that represents the month */
long Day;           /* The portion of the enterred date that represents the day */
long Year;          /* The protion of the enterred date that represents the year */
long a, b;

        Month = trunc(*date1);
        Day = trunc(100*(*date1 - Month));
        Year = rint((1000000.0*(*date1)) - (Month*1000000.0) - (Day*10000.0));
        if (Month <= 2) {
                Year = Year - 1;
```

```
            Month = Month + 12;
        }
        a = Year/100;
        b = 2 - a + a/4;
        *date1 = trunc(365.25*Year) + trunc(30.6001*(Month + 1.0)) + Day +
                        1720994.5 + b;
}

/*************************************************************************/

void GetCaldate(double Juldate1, double *argdate)

{
double Juldate2, Caldate1, Caldate2;

        InvConvert(Juldate1, &Caldate1);
        Juldate2 = Caldate1;
        Convert(&Juldate2);
        *argdate = Caldate1;
        if (Juldate2 > Juldate1)
                *argdate = (*argdate) - 0.01;
}

/*************************************************************************/

void CaseEntry()

                        /* This procedure prompts for the launch date and time of   */
                        /* flight.                                                  */

{
double JlaunchT, JarriveT;

        if ((JEpoch1/1000.0) < 1)
                Convert(&JEpoch1);
        if ((JEpoch2/1000.0) < 1)
                Convert(&JEpoch2);
        if ((JEpoch3/1000.0) < 1)
                Convert(&JEpoch3);

        printf("Enter the number for the desired asteroid:\n");
        printf("        1: XB-1982\n");
        printf("        2: Orpheus\n");
        printf("        3: McAuliffe\n");
        printf("        4: Seleucus\n");
        printf("        5: Eros\n");
        scanf("%i", &pickroid);

        printf("Enter the desired launch date as mm.ddyyyy or as a Julian date:\n");
        scanf("%lf", &launch);
        JlaunchT = launch;
        if ((JlaunchT/1000.0) < 1)
```

```
            Convert(&JlaunchT);
        Juliandate = JlaunchT;
        Jcentury1[1] = (JlaunchT - JEpoch1)/(36525.0);
        Jcentury1[2] = JlaunchT/36525.0;
        Jcentury1[3] = JlaunchT/36525.0;
        printf("Enter the desired time of flight to target in days:\n");
        scanf("%lf", &TOF);

        JarriveT = JlaunchT + TOF;
        Jcentury2[1] = (JarriveT - JEpoch1)/(36525.0);
        Jcentury2[2] = JarriveT/36525.0;
        Jcentury2[3] = JarriveT/36525.0;

        GetCaldate(JlaunchT, &ClaunchT);
        GetCaldate(JarriveT, &CarriveT);

        Jdt = (((JlaunchT + stepdays) - JEpoch1)/36525.0) - Jcentury1[1];
}

/*************************************************************************/

void BuildTrans(double body[8], double Trans[4][4])

                        /* This procedure builds the matrix which serves */
                        /* as the transformation matrix between the */
                        /* ecliptic and pericentric coordinate system */

{
double SmOmega;

   SmOmega = body[3] - body[7];
   Trans[1][1] = cos(body[7])*cos(SmOmega) -
                        sin(body[7])*sin(SmOmega)*cos(body[6]);
   Trans[1][2] = -cos(body[7])*sin(SmOmega) -
                        sin(body[7])*cos(SmOmega)*cos(body[6]);
   Trans[1][3] = sin(body[7])*sin(body[6]);
   Trans[2][1] = sin(body[7])*cos(SmOmega) +
                        cos(body[7])*sin(SmOmega)*cos(body[6]);
   Trans[2][2] = -sin(body[7])*sin(SmOmega) +
                        cos(body[7])*cos(SmOmega)*cos(body[6]);
   Trans[2][3] = -cos(body[7])*sin(body[6]);
   Trans[3][1] = sin(SmOmega)*sin(body[6]);
   Trans[3][2] = cos(SmOmega)*sin(body[6]);
   Trans[3][3] = cos(body[6]);
}

/*************************************************************************/

void CoordConvert(double body[8], double TransC[4][4], double nuC[5], int i,
                double EcliptC[5][4])

{
```

```
double Peri[5][4];

   Peri[i][1] = (body[5]*(1 - body[4]*body[4])/(1 + body[4]*cos(nuC[i])))*cos(nuC[i]);
   Peri[i][2] = (body[5]*(1 - body[4]*body[4])/(1 + body[4]*cos(nuC[i])))*sin(nuC[i]);
   Peri[i][3] = 0.0;
   EcliptC[i][1] = TransC[1][1]*Peri[i][1] + TransC[1][2]*Peri[i][2] +
                        TransC[1][3]*Peri[i][3];
   EcliptC[i][2] = TransC[2][1]*Peri[i][1] + TransC[2][2]*Peri[i][2] +
                        TransC[2][3]*Peri[i][3];
   EcliptC[i][3] = TransC[3][1]*Peri[i][1] + TransC[3][2]*Peri[i][2] +
                        TransC[3][3]*Peri[i][3];
}

/*************************************************************************/

void PlanetPosition(double body[8], double rp[5], double nup[5], double CelesLongp[5],
                        double CelesLatp[5], double Velp[5], double Ecliptp[5][4],
                        double EcliptVPp[5][4], int index)

{
double Ma, Mo, Ea, D, SminAx, SmOmega, VPx, VPy;
double Transp[4][4];

        Ma = (body[2] - body[3]);
        if (Ma < 0.0)
                Ma = Ma + (2.0*M_PI);
        Ea = Ma + body[4]*(sin(Ma) + body[4]*sin(2.0*Ma)/2);
        do{
                Mo = Ea - body[4]*sin(Ea);
                D = (Ma - Mo)/(1 - body[4]*cos(Ea));
                Ea = Ea + D;
        }while (fabs(D) > tol);
        nup[index] = 2.0*atan(sqrt((1 + body[4])/(1 - body[4]))*
                        ((sin(Ea/2.0)/cos(Ea/2.0))));
        if (nup[index] < 0.0)
                nup[index] = nup[index] + 2.0*M_PI;
        SminAx = body[5]*sqrt(1 - body[4]*body[4]);
        SmOmega = body[3] - body[7];
        BuildTrans(body, Transp);
        CoordConvert(body, Transp, nup, index, Ecliptp);
        CelesLongp[index] = atan(Ecliptp[index][2]/Ecliptp[index][1]);
        if ((Ecliptp[index][1]<0) && (Ecliptp[index][2]>0) && (CelesLongp[index]<0))
                CelesLongp[index] = CelesLongp[index] + M_PI;
        if ((Ecliptp[index][1]<0) && (Ecliptp[index][2]<0) && (CelesLongp[index]>0))
                CelesLongp[index] = CelesLongp[index] + M_PI;
        if ((Ecliptp[index][1]>0) && (Ecliptp[index][2]<0) && (CelesLongp[index]<0))
                CelesLongp[index] = CelesLongp[index] + 2.0*M_PI;
        if (body[6] == 0.0)
                CelesLatp[index] = 0.0;
        else
                CelesLatp[index] = asin(sin(SmOmega + nup[index])*sin(body[6]));
        rp[index] = body[5]*(1 - body[4]*body[4])/(1 + body[4]*cos(nup[index]));
```

```
        VPx = (-body[5]*sin(Ea)*sqrt(GM)/(sqrt(body[5]*body[5]*body[5])*
                (1 - body[4]*cos(Ea))))*AU;
        VPy = (SminAx*cos(Ea)*sqrt(GM)/(sqrt(body[5]*body[5]*body[5])*
                (1 - body[4]*cos(Ea))))*AU;
        Velp[index] = sqrt(2.0/rp[index] - 1.0/body[5])*AU;
        EcliptVPp[index][1] = Transp[1][1]*VPx + Transp[1][2]*VPy;
        EcliptVPp[index][2] = Transp[2][1]*VPx + Transp[2][2]*VPy;
        EcliptVPp[index][3] = Transp[3][1]*VPx + Transp[3][2]*VPy;
}

/*************************************************************************/

void SpecifyEndpts(double argEcliptV[5][4])

{
double rplanet[5], nuplanet[5], Vplanet[5], CelesLong[5], CelesLat[5];
double departureI[8], targetI[8], argEclipt[5][4];
int i;

        earth(departureI, Jcentury1[1]);
        if (pickroid == 1)
                XB(targetI, Jcentury2[2]);
        else if (pickroid == 2)
                Orpheus(targetI, Jcentury2[2]);
        else if (pickroid == 3)
                McAuliffe(targetI, Jcentury2[2]);
        else if (pickroid == 4)
                Seleucus(targetI, Jcentury2[2]);
        else if (pickroid == 5)
                Eros(targetI, Jcentury2[3]);

        PlanetPosition (departureI, rplanet, nuplanet, CelesLong, CelesLat, Vplanet,
                                argEclipt, argEcliptV, 1);

        for (i = 1; i <= 3; i++)
                R1[i] = argEclipt[1][i];

        PlanetPosition (targetI, rplanet, nuplanet, CelesLong, CelesLat, Vplanet,
                                argEclipt, argEcliptV, 2);

        for (i = 1; i <= 3; i++)
                R2[i] = argEclipt[2][i];

        rInitial = rplanet[1];
        rfinal = rplanet[2];
        LongInitial = CelesLong[1];
        Longfinal = CelesLong[2];
        LatInitial = CelesLat[1];
        Latfinal = CelesLat[2];
}

/*************************************************************************/
```

```
void FindAngle(double argLat1, double argLat2, double argLong1, double argLong2,
                        double *deltang)

                        /* This procedure calculates the angle between 2     */
                        /* points.                                           */

{
double x1, x2, y1, y2, z1, z2, ang1;

        x1 = cos(argLat1)*cos(argLong1);
        x2 = cos(argLat2)*cos(argLong2);
        y1 = cos(argLat1)*sin(argLong1);
        y2 = cos(argLat2)*sin(argLong2);
        z1 = sin(argLat1);
        z2 = sin(argLat2);
        ang1 = acos(x1*x2 + y1*y2 + z1*z2);
        if (sin(argLong2 - argLong1) >= 0.0)
                *deltang = ang1;
        else
                *deltang = 2*M_PI - ang1;
}

/*************************************************************************/

void Parameters(double *argRop, double *argm, double *argL, double *args,
                        double *arglamb, double *argT)

{
double c, s, dt, tmp, argk[7];

        FindAngle(LatInitial, Latfinal, LongInitial, Longfinal, &HCA);
        c = sqrt(rInitial*rInitial + rfinal*rfinal - 2.0*rInitial*rfinal*cos(HCA));
        *args = (rInitial + rfinal + c)/2.0;
        *arglamb = sqrt(rInitial*rfinal)*cos(HCA/2.0)/(*args);
        *argRop = (*args)*(1 + (*arglamb))*(1 + (*arglamb))/4.0;
        dt = TOF*86400;
        *argT = dt*sqrt(8.0*GM/((*args)*(*args)*(*args)));
        tmp = (1.0 + (*arglamb));
        *argm = (*argT)*(*argT)/pow(tmp,6.0);
        argk[0] = rfinal/rInitial;
        argk[1] = sqrt(argk[0]);
        argk[2] = (argk[0] - 1.0);
        argk[3] = argk[2]*argk[2]/(4.0*(argk[1] + argk[0]*(2.0 + argk[1])));
        argk[4] = sin(HCA/4.0)*sin(HCA/4.0) + argk[3];
        argk[5] = cos(HCA/4.0)*cos(HCA/4.0) + argk[3];
        argk[6] = cos(HCA/2.0);
        if ((0.0 < HCA) && (HCA < M_PI))
                *argL = argk[4]/(argk[4] + argk[6]);
        else
                *argL = (argk[5] - argk[6])/argk[5];
}
```

```
/*************************************************************************/

void CalcXi(double argx, double *argXi)

                    /* This procedure approximates the function Xi using    */
                    /* the continued fraction scheme.                       */

{
double eta, c, c2, A, A1, A2, B, B1, B2, d, Xilast;
int i;

        eta = argx/((sqrt(1.0 + argx) + 1.0)*(sqrt(1.0 + argx) + 1.0));
        c2 = 8.0*(sqrt(1.0 + argx) + 1.0);
        A1 = 5.0 + eta;
        B1 = 1.0;
        A2 = 1.0;
        B2 = 0.0;
        *argXi = 0;
        for (i = 1; i <= 100; i++) {
                if (i == 1)
                        c = 9.0*eta/7.0;
                else
                        c = eta*((i + 2.0)*(i + 2.0))/((2.0*i + 5.0)*(2.0*i + 3.0));
                d = 1.0;
                A = d*A1 + c*A2;
                B = d*B1 + c*B2;
                A1 = A1/B;
                B1 = B1/B;
                A = A/B;
                A2 = A1;
                B2 = B1;
                A1 = A;
                B1 = 1.0;
                Xilast = *argXi;
                *argXi = A;
                if (fabs(*argXi) == fabs(Xilast)) {
                        *argXi = c2/(3.0 + (1.0/(*argXi)));
                        return;
                }
        }
}

/*************************************************************************/

void CalcK(double u1, double *argK)

                    /* This procedure approximates the function K using     */
                    /* the continued fraction scheme.                       */

{
double a1, c, d, coeff, n1, Klast, index;
```

```
int i;

        a1 = 1.0/3.0;
        d = 1.0;
        c = a1;
        *argK = c;
        index = 0.0;
        for (i = 0; i <= 99; i++) {
                index = index + 0.5;
                n1 = trunc(index);
                if (fmod(i, 2.0) == 0.0)
                        coeff = (2.0*(3.0*n1 + 2.0)*(6.0*n1 + 1.0))/(9.0*(4.0*n1+
                                        1.0)*(4.0*n1 + 3.0));
                else
                        coeff = (2.0*(3.0*n1 + 1.0) *(6.0*n1 - 1.0))/(9.0*(4.0*n1 -
                                        1.0)*(4.0*n1 + 1.0));
                a1 = coeff*u1;
                d = 1.0/(1.0 + a1*d);
                c = c*(d - 1.0);
                Klast = *argK;
                *argK = *argK + c;
                if (fabs(*argK) == fabs(Klast))
                        return;
        }
}

/*************************************************************************/

void Cubic(double argx, double argL, double argm, double *argy)

                        /* This procedure calls CalcK in order to sove the   */
                        /* cubic equation for y.                             */

{
double h1, h2, Xi, b, c0, c1, K, u;

        CalcXi(argx, &Xi);
        c0 = (1.0 + 2.0*argx + argL)*(4.0*argx + Xi*(3.0 + argx));
        h1 = ((1.0 + 3.0*argx + Xi)*(argL + argx)*(argL + argx))/c0;
        h2 = argm*(argx - argL + Xi)/c0;
        b = 27.0*h2/(4.0*(1.0 + h1)*(1.0 + h1)*(1.0 + h1));
        if ((1 + b) < 0.0) {
                printf("neg. sqrt. in Cubic\n");
                exit(0);
        }
        else {
                c1 = sqrt(1.0 + b);
                u = b/(2.0*(c1 + 1.0));
                CalcK(u, &K);
                *argy = ((1.0 + h1)/3.0)*(2.0 + c1/(1.0 + 2.0*u*K*K));
        }
}
```

```
/*****************************************************************************/

void Verify()

                /* This procedure verifies that the value for nu1 has been assigned  */
                /* to the proper quadrant in the case of a nonhyperbolic trajectory.  */
                /* The time of flight is solved for using Kepler's equations, and the */
                /* quadrant of nu1 is alterred until this time matches the input time */
                /* of flight.  nu2 is assigned, dependant upon the value of nu1       */
{
double time1;                   /* initial time since perigee passage */
double time2;                   /* final time since perigee passage */
double time;                    /* time of flight */
double Period;                  /* orbit period */
double arg1;                    /* cosine of eccentric anomoly at r1 */
double arg2;                    /* cosine of eccentric anomoly ar r2 */
double check1, check2;          /* comparison values for time of flight */
int loop;                       /* an indexing variable */

        loop = 0.0;
        nuInitial = acos((p/rInitial - 1.0)/ecc);
        do{
                loop = loop++;
                if (loop == 1)
                        nuInitial = nuInitial;
                else if (loop ==2)
                        nuInitial = 2.0*M_PI - nuInitial;
                else if (loop == 3) {
                        printf("Not!!!!!!!????\n");
                        exit(0);
                }
                nufinal = nuInitial + HCA;
                if (nufinal > 2.0*M_PI)
                        nufinal = nufinal - 2.0*M_PI;
                arg1 = (ecc + cos(nuInitial))/(1.0 + ecc*cos(nuInitial));
                arg2 = (ecc + cos(nufinal))/(1.0 + ecc*cos(nufinal));
                EAInitial = acos(arg1);
                EAfinal = acos(arg2);
                if (nuInitial > M_PI)
                        EAInitial = 2.0*M_PI - EAInitial;
                if (nufinal > M_PI)
                        EAfinal = 2.0*M_PI - EAfinal;
                time1 = (EAInitial - ecc*sin(EAInitial))/n;
                time2 = (EAfinal - ecc*sin(EAfinal))/n;
                time = time2 - time1;
                Period = 2.0*M_PI/n;
                if (EAInitial > EAfinal)
                        time = Period + time;
                check1 = time/86400.0;
                check2 = TOF;
        }while ((fabs(check1 - check2) > 0.001));
```

```
}

/*****************************************************************************/

void HyperVerify()

        /* This procedure verifies that the value for nu1 has been assigned  */
        /* to the proper quadrant in the case of a hyperbolic trajectory.    */
        /* The time of flight is solved for using Kepler's equations, and the */
        /* quadrant of nu1 is alterred until this time matches the input time */
        /* of flight.  nu2 is assigned, dependant upon the value of nu1      */
{
double time1;                   /* initial time since perigee passage */
double time2;                   /* final time since perigee passage */
double time;                    /* time of flight */
double arg1;                    /* sinh of eccentric anomoly at r1 */
double arg2;                    /* sinh of eccentric anomoly ar r2 */
double check1, check2;          /* comparison values for time of flight */
double numax;                   /* maximum possible true anomaly within hyperbola */
int loop;                       /* an indexing variable */
double tmp1, tmp2;

        loop = 0.0;
        tmp1 = (p/rInitial - 1.0)/ecc;
        tmp2 = sqrt(1.0 - ((p/rInitial - 1.0)/ecc*(p/rInitial - 1.0)/ecc));
        nuInitial = M_PI/2.0 - atan(tmp1/tmp2);
        numax = M_PI/2.0 - atan((-1.0/ecc)/(sqrt(1.0 - ((-1.0/ecc)*(-1.0/ecc)))));
        if ((nuInitial + HCA) > numax) {
                nuInitial = -nuInitial;
                nufinal = nufinal + HCA;
                arg1 = (sqrt(ecc*ecc - 1.0)*sin(nuInitial))/(1.0 + ecc*cos(nuInitial));
                arg2 = (sqrt(ecc*ecc - 1.0)*sin(nufinal))/(1.0 + ecc*cos(nufinal));
                EAInitial = asinh(arg1);
                EAfinal = asinh(arg2);
        }
        else {
          do{
                loop = loop++;
                if (loop == 1)
                        nuInitial = nuInitial;
                else if (loop ==2)
                        nuInitial = - nuInitial;
                else if (loop == 3) {
                        printf("Not!!!!!!!???\n");
                        exit(0);
                }
                nufinal = nuInitial + HCA;
                arg1 = (sqrt(ecc*ecc - 1.0)*sin(nuInitial))/(1.0 + ecc*cos(nuInitial));
                arg2 = (sqrt(ecc*ecc - 1.0)*sin(nufinal))/(1.0 + ecc*cos(nufinal));
                EAInitial = asinh(arg1);
                EAfinal = asinh(arg2);
                time1 = (ecc*sinh(EAInitial) - EAInitial)/n;
```

```
                time2 = (ecc*sinh(EAfinal) - EAfinal)/n;
                time = time2 - time1;
                check1 = time/86400.0;
                check2 = TOF;
            }while ((fabs(check1 - check2) > 0.001));
        }
}

/*************************************************************************/

void QuadrantCheck(double *Omega, double delLong, double Lat1, double Lat2,
                        double L1, double L2)

                        /* This procedure utilizes geometric arguements within    */
                        /* the celestial sphere to ensure that Omega has been     */
                        /* assigned in the correct quadrant                       */

{
        delLong = delLong;
        Lat1 = Lat1;
        Lat2 = Lat2;
        L1 = L1;
        L2 = L2;
        if (delLong <= M_PI) {
                if ((Lat1 >= 0.0) && (Lat2 >= 0.0)) {
                        if ((L2<=M_PI) && ((L2<*Omega) && (*Omega<(L2 + M_PI))))
                                *Omega = *Omega + M_PI;
                        if ((L2 > M_PI) && (((L2 <= *Omega) &&
                                (*Omega<=2*M_PI)) || ((0.0 <= *Omega) &&
                                (*Omega < (L2 + M_PI - 2.0*M_PI)))))
                                        *Omega = *Omega + M_PI;
                }
                if ((Lat1 <= 0.0) && (Lat2 <= 0.0)) {
                        if ((L2 <= M_PI) && !((L2 <= *Omega) && (*Omega < (L2 +
                                M_PI))))
                                        *Omega = *Omega + M_PI;
                        if ((L2 > M_PI) && !(((L2 <= *Omega) &&
                                (*Omega <= 2.0*M_PI)) || ((0.0 <= *Omega) &&
                                (*Omega < (L2 + M_PI - 2.0*M_PI)))))
                                        *Omega = *Omega + M_PI;
                }
                if ((Lat1 > 0.0) && (Lat2 > 0.0)) {
                        if ((L1 <= M_PI) && ((L1 <= *Omega) && (*Omega < (L1 +
                                M_PI))))
                                        *Omega = *Omega + M_PI;
                        if ((L1 > M_PI) && (((L1 <= *Omega) &&
                                (*Omega <= 2.0*M_PI)) || ((0.0 <= *Omega) &&
                                (*Omega < (L1 + M_PI - 2.0*M_PI)))))
                                        *Omega = *Omega + M_PI;
                }
                if ((Lat1 < 0.0) && (Lat2 > 0.0)) {
                        if ((L1 <= M_PI) && !((L1 <= *Omega) && (*Omega < (L1 +
```

```
                        M_PI))))
                                *Omega = *Omega + M_PI;
                        if ((L1 > M_PI) && !(((L1 < *Omega) &&
                                (*Omega <= 2.0*M_PI)) || ((0.0 <= *Omega) &&
                                (*Omega < (L1 + M_PI - 2.0*M_PI)))))
                                *Omega = *Omega + M_PI;
                }
        }
        else {
                if ((Lat2 >= 0.0) && (Lat1 >= 0.0)) {
                        if ((L1 <= M_PI) && ((L1 < *Omega) && (*Omega < (L1 +
                                M_PI))))
                                *Omega = *Omega + M_PI;
                        if ((L1 > M_PI) && (((L1 <= *Omega) && (*Omega <=
                                2.0*M_PI)) || ((0.0 <= *Omega) && (*Omega < (L1 +
                                M_PI - 2.0*M_PI)))))
                                *Omega = *Omega + M_PI;
                }
                if ((Lat2 <= 0.0) && (Lat1 <= 0.0)) {
                        if ((L1 <= M_PI) && !((L1 <= *Omega) && (*Omega < (L1 +
                                M_PI))))
                                *Omega = *Omega + M_PI;
                        if ((L1 > M_PI) && !(((L1 <= *Omega) && (*Omega <=
                                2.0*M_PI)) || ((0.0 <= *Omega) && (*Omega < (L1 +
                                M_PI - 2.0*M_PI)))))
                                *Omega = *Omega + M_PI;
                }
                if ((Lat2 > 0.0) && (Lat1 > 0.0)) {
                        if ((L2 <= M_PI) && ((L2 <= *Omega) && (*Omega<(L2 +
                                M_PI))))
                                *Omega = *Omega + M_PI;
                        if ((L2 > M_PI) && (((L2 <= *Omega) && (*Omega <=
                                2.0*M_PI)) || ((0.0 <= *Omega) && (*Omega < (L2 +
                                M_PI - 2.0*M_PI)))))
                                *Omega = *Omega + M_PI;
                }
                if ((Lat2 < 0.0) && (Lat1 > 0.0)) {
                        if ((L2 <= M_PI) && !((L2 <= *Omega) && (*Omega < (L2 +
                                M_PI))))
                                *Omega = *Omega + M_PI;
                        if ((L2 > M_PI) && !(((L2 <= *Omega) && (*Omega <=
                                2.0*M_PI)) || ((0.0 <= *Omega) && (*Omega < (L2 +
                                M_PI - 2.0*M_PI)))))
                                *Omega = *Omega + M_PI;
                }
        }
}

/*****************************************************************************/

void PieQuadrant(double *Pie, double Omega)
```

```
                /* This procedure utilizes spherical trigonometrical arguements    */
                /* to ensure that Pie is assigned to the correct quadrant          */

{
double PieSin;                  /* The sine of the sum of Omega and SmOmega */
double PieCos;                  /* The cosine of the sum of Omega and SmOmega */

        PieSin = sin(Latfinal)/sin(inclin);
        if (Latfinal == 0.0)
                PieSin = sin(LatInitial)/sin(inclin);
        PieCos = cos(Longfinal - Omega)*cos(Latfinal);
        if (Latfinal == 0.0)
                PieCos = cos(LongInitial - Omega)*cos(LatInitial);
        if ((PieSin > 0.0) && (PieCos < 0.0))
                *Pie = M_PI - (*Pie);
        if ((PieSin < 0.0) && (PieCos < 0.0))
                *Pie = M_PI - (*Pie);
}

/*************************************************************************/

void Trajelem(double craftelem[8], double EcliptVPc[5][4], double *C3Initial,
                double *C3final)

                        /* This procedure calculates the orbital elements of  */
                        /* the trajectory between the departure and target    */
                        /* planets.                                           */

{
double Vrad[4], Vtang[4];
double x, xlast, y, Rop, m, k[8], L, lambda, T, Tp, s, tmp;
double delLong, wplusnu, arg1, arg2;
double VP1x, VP1y, VP2x, VP2y, Transc[4][4], EcliptVc[5][4];
double vfactor1, vfactor2, eccen[4], totecc, argecc[4];
double omega;
int i;

        Parameters(&Rop, &m, &L, &s, &lambda, &T);
        Tp = 4.0*(1.0 - lambda*lambda*lambda)/3.0;
        if (T > Tp)
                x = L;
        else
                x = 0.0;
        do{
                Cubic(x, L, m, &y);
                xlast = x;
                x = sqrt(((1.0 - L)/2.0)*((1.0 - L)/2.0) + m/(y*y)) - (1.0 + L)/2.0;
        }while ((fabs(x - xlast)) > tol);

        Battin_x = x;

        tmp = m*s*(1.0 + lambda)*(1.0 + lambda);
```

```
a = tmp/(8.0*x*y*y);
p = 2.0*rInitial*rfinal*y*y*(1.0 + x)*(1.0 + x)*sin(HCA/2.0)*sin(HCA/2.0)/tmp;
ecc = sqrt(1.0 - p/a);
AngMom = sqrt(GM*p)*AU*AU;
n = sqrt(GM/(fabs((a)*(a)*(a))));
V1 = sqrt(GM*(2.0/rInitial - 1.0/a))*AU;
V2 = sqrt(GM*(2.0/rfinal - 1.0/a))*AU;
Vtang[1] = (AngMom)/(rInitial*AU);
Vtang[2] = (AngMom)/(rfinal*AU);
Vrad[1] = sqrt(V1*V1 - Vtang[1]*Vtang[1]);
Vrad[2] = sqrt(V2*V2 - Vtang[2]*Vtang[2]);
gamma1 = atan(Vrad[1]/Vtang[1]);
gamma2 = atan(Vrad[2]/Vtang[2]);

arg1 = sqrt(GM*p)/(rInitial*rfinal*sin(HCA));
arg2 = (1 - cos(HCA))/p;
for (i = 1; i <= 3; i++)
        Vel1[i] = arg1*(R2[i] - R1[i] + rfinal*arg2*R1[i]);
for (i = 1; i <= 3; i++)
        Vel2[i] = arg1*(R2[i] - R1[i] - rInitial*arg2*R2[i]);

if (a > 0.0)
        Verify();
else
        HyperVerify();
if (nuInitial > M_PI)
        gamma1 = -gamma1;
if (nufinal > M_PI)
        gamma2 = -gamma2;

craftelem[4] = ecc;
craftelem[5] = a;
delLong = Longfinal - LongInitial;

arg1 = R1[2]*Vel1[3] - R1[3]*Vel1[2];
arg2 = R1[1]*Vel1[3] - R1[3]*Vel1[1];
craftelem[7] = atan2(arg1,arg2);
if (craftelem[7] < 0.0)
        craftelem[7] = craftelem[7] + 2*M_PI;
QuadrantCheck(&craftelem[7], delLong, LatInitial, Latfinal, LongInitial,
                        Longfinal);
if (craftelem[7] > 2.0*M_PI)
        craftelem[7] = craftelem[7] - 2*M_PI;

arg1 = sqrt((R1[2]*Vel1[3] - R1[3]*Vel1[2])*(R1[2]*Vel1[3] - R1[3]*Vel1[2]) +
            (R1[1]*Vel1[3] - R1[3]*Vel1[1])*(R1[1]*Vel1[3] - R1[3]*Vel1[1]));
arg2 = R1[1]*Vel1[2] - R1[2]*Vel1[1];
craftelem[6] = atan2(arg1,arg2);
inclin = craftelem[6];

argecc[1] = Vel1[2]*(R1[1]*Vel1[2] - R1[2]*Vel1[1]) -
                Vel1[3]*(R1[3]*Vel1[1] - R1[1]*Vel1[3]);
```

```
        argecc[2] = Vel1[3]*(R1[2]*Vel1[3] - R1[3]*Vel1[2]) -
                        Vel1[1]*(R1[1]*Vel1[2] - R1[2]*Vel1[1]);
        argecc[3] = Vel1[1]*(R1[3]*Vel1[1] - R1[1]*Vel1[3]) -
                        Vel1[2]*(R1[2]*Vel1[3] - R1[3]*Vel1[2]);

        for (i = 1; i <= 3; i++)
                eccen[i] = argecc[i]/GM - R1[i]/rInitial;

        arg1 = eccen[3];
        arg2 = sin(craftelem[6])*(eccen[1]*cos(craftelem[7]) +
                        eccen[2]*sin(craftelem[7]));
        omega = atan2(arg1, arg2);
        if (craftelem[7] < 0.0)
                omega = omega + 2*M_PI;

        craftelem[3] = craftelem[7] + omega;

        EcliptVc[1][1] = Vel1[1]*AU;
        EcliptVc[1][2] = Vel1[2]*AU;
        EcliptVc[1][3] = Vel1[3]*AU;
        EcliptVc[2][1] = Vel2[1]*AU;
        EcliptVc[2][2] = Vel2[2]*AU;
        EcliptVc[2][3] = Vel2[3]*AU;

        *C3Initial = (EcliptVc[1][1]-EcliptVPc[1][1])*(EcliptVc[1][1]-EcliptVPc[1][1])+
                        (EcliptVc[1][2]-EcliptVPc[1][2])*(EcliptVc[1][2]-
                        EcliptVPc[1][2]) + (EcliptVc[1][3]-
                        EcliptVPc[1][3])*(EcliptVc[1][3]-EcliptVPc[1][3]);
        *C3final = (EcliptVc[2][1]-EcliptVPc[2][1])*(EcliptVc[2][1]-EcliptVPc[2][1]) +
                        (EcliptVc[2][2]-EcliptVPc[2][2])*(EcliptVc[2][2]-
                        EcliptVPc[2][2]) +(EcliptVc[2][3]-
                        EcliptVPc[2][3])*(EcliptVc[2][3]-EcliptVPc[2][3]);
}

/*************************************************************************/

void earthPark(double argEcliptVP[5][4], double argEcliptV[5][4], double argC3,
                        double *argDV)

                        /* This procedure calculates the DeltaV required to  */
                        /* enter or leave a desired parking orbit around the  */
                        /* earth.                                             */

{
double Vreq, Vcirc, Rpark, Vinfx, Vinfy, Vinfz, Vinearthx, Vinearthy, Vinearthz, Vinf;
double B, Bhold, ahyp, ehyp, rho, relinc, DeltaVinc, Omeg1, Omeg2, omeg, term;
double parki, tilt;
double GMearth = 398600;    /* km3/s2 */

        printf("Enter the desired circular parking orbit altitude around earth (in km):\n");
        scanf("%lf", &Rpark);
        Vreq = sqrt(2.0*GMearth/(Rpark + 6378.145) + argC3);
```

```
Vcirc = sqrt(GMearth/(Rpark + 6378.145));
*argDV = Vreq - Vcirc;

DeltaVinc = 0.0;
parki = 28.5/RAD;
tilt = 23.443597/RAD;
Vinfx = argEcliptV[1][1] - argEcliptVP[1][1];
Vinfy = argEcliptV[1][2] - argEcliptVP[1][2];
Vinfz = argEcliptV[1][3] - argEcliptVP[1][3];
Vinearthx = -Vinfx*sin(LongInitial) + (Vinfy*cos(tilt) -
            Vinfz*sin(tilt))*cos(LongInitial);
Vinearthy = -Vinfx*cos(LongInitial) + (-Vinfy*cos(tilt) +
            Vinfz*sin(tilt))*sin(LongInitial);
Vinearthz = Vinfy*sin(tilt) + Vinfz*cos(tilt);
Vinf = sqrt(Vinearthx*Vinearthx + Vinearthy*Vinearthy + Vinearthz*Vinearthz);
B = acos(Vinearthz/Vinf);
Bhold = B;
if (B > M_PI/2.0)
        Bhold = M_PI - Bhold;
if ((Bhold + parki) < M_PI/2.0)
        parki = 51.0/RAD;
if ((Bhold + parki) < M_PI/2.0) {
        printf("\n");
        printf("Nontangential injection. Plane change required.\n");
        printf("\n");
        ahyp = -GMearth/(Vinf*Vinf);
        ehyp = 1.0 - Rpark/ahyp;
        rho = acos(1.0/ehyp);
        if ((Bhold + parki) > (M_PI/2.0 - rho)) {
                relinc = asin(cos(Bhold + parki)/cos(M_PI/2.0 - rho));
                DeltaVinc = Vinf*sin(relinc/2.0);
                printf("Inclination change: %.5lf {deg}\n", (relinc*RAD));
                printf("Delta V for plane change: %.5lf {km/s}\n", DeltaVinc);
                *argDV = (*argDV) + DeltaVinc;
        }
        else
                printf("This is a non-horizontal injection case.\n");
}
if ((Bhold + parki) >= M_PI/2.0) {
        printf("\n");
        printf("Tangential injection. No plane change required.\n");
        printf("\n");
        parki = M_PI/2.0 - Bhold;
        if (parki < 28.5/RAD)
                parki = 28.5/RAD;
        term = (1.0/(tan(B)*tan(parki)));
        if (term > 1.0) {
                Omeg1 = M_PI + M_PI/2.0;
                Omeg2 = 2.0*M_PI - M_PI/2.0;
        }
        if (term < -1.0) {
                Omeg1 = M_PI - M_PI/2.0;
```

```
                Omeg2 = 2.0*M_PI + M_PI/2.0;
        }
        if ((term > -1.0) && (term < 1.0)) {
                Omeg1 = M_PI + asin(1.0/(tan(B)*tan(parki)));
                Omeg2 = 2.0*M_PI - asin(1.0/(tan(B)*tan(parki)));
        }
        if (Omeg1 > 2.0*M_PI)
                Omeg1 = Omeg1 - 2.0*M_PI;
        if (Omeg2 > 2.0*M_PI)
                Omeg2 = Omeg2 - 2.0*M_PI;
        printf("Possible Node of Parking Orbit: %.5lf\n", Omeg1);
        printf("Possible Node of Parking Orbit: %.5lf\n", Omeg2);
    }
    printf("Inclination of Parking Orbit: %.5lf\n", (parki*RAD));
}

/*************************************************************************/

void AsterRend(double argC3, double *argDV)

                /* This procedure calculates the DeltaV required to  */
                /* rendezvous with the desired asteroid.             */

{
    *argDV = sqrt(argC3);
}

/*************************************************************************/

void CS(double argz, double *argC, double *argS)

                /* This procedure calculates the C and S functions   */
                /* (eqtns 4.4-10 and 4.4-11 in Bate).                */

{
double sqz;

    if (fabs(argz) < 0.05) {
        *argC = (1.0 - argz/12.0*(1.0 - argz/30.0*(1.0 - argz/56.0*(1.0 -
                argz/90.0*(1.0 - argz/132.0)))))/2.0;
        *argS = (1.0 - argz/20.0*(1.0 - argz/42.0*(1.0 - argz/72.0*(1.0 -
                argz/110.0*(1.0 - argz/156.0)))))/6.0;
    }
    else if (argz >= 0.0) {
        sqz = sqrt(argz);
        *argC = (1.0 - cos(sqz))/argz;
        *argS = (sqz - sin(sqz))/(sqz*sqz*sqz);
    }
    else {
        sqz = sqrt(-argz);
        *argC = (1.0 - cosh(sqz))/argz;
        *argS = (sinh(sqz) - sqz)/(sqz*sqz*sqz);
```

```
        }
}

/*************************************************************************/

void CraftPos(double bodyc[8], double rc[5], double nuc[5], double CelesLongc[5],
              double CelesLatc[5], double Ecliptc[5][4], double EcliptVc[5][4],
              int index)
{
double MA, EA, D, term, check, SmOmegac, Transc[4][4], V, Vtang, Vrad;
double C, S, x, sqmu, P, ainv, A1, z, k1, Dt, Dtlast, xr, f, g, fdot, gdot, tc;
int i, s;

        sqmu = sqrt(GM);

        tc = time*86400.0;
        ainv = 2.0/rInitial - V1*V1/GM/AU/AU;
        A1 = (R1[1]*Vel1[1] + R1[2]*Vel1[2] + R1[3]*Vel1[3])/sqmu;
        if (tc == 0.0)
                x = 0.0;
        else {
                if (ainv > 0.0) {
                        P = 2.0*M_PI/n;
                        while (tc > P)
                                tc -= P;
                        while (tc < P)
                                tc += P;
                        x = sqmu*tc*ainv;
                }
                else {
                        if (tc > 0.0)
                                s = 1;
                        else
                                s = -1;
                        if (ainv != 0.0) {
                                k1 = s*sqrt(-1.0/ainv);
                                x = k1*log(-2.0*sqmu*ainv*tc/(A1 + k1*(1.0 -
                                        ainv*rInitial)));
                        }
                        else
                                x = 0.0;
                }
        }

        Dtlast = 9.0e20;
        Dt = 0.0;
        i = 0;

        for (i = 1; i <= 1000; i++) {
                z = ainv*x*x;
                CS(z, &C, &S);
                k1 = 1.0 - z*S;
```

```
        Dt = x*(rInitial*k1 + x*(A1*C + x*S)) - sqmu*tc;
        xr = rInitial*(1.0 - z*C) + x*(k1*A1 + x*C);
        x -= Dt/xr;
        if ((fabs(Dt) >= fabs(Dtlast)) && (fabs(Dt) < tol) && (i > 3))
                break;
        Dtlast = Dt;
}


if ( i == 1001) {
        printf("\n*** Convergence Failure in 'CraftPos' ***\n");
        gexit();
        exit(0);
}

f = 1.0 - x*x*C/rInitial;
g = tc - x*x*x*S/sqmu;
fdot = -k1*sqmu*x/(rInitial*xr);
gdot = 1.0 - x*x*C/xr;

for (i = 1; i <= 3; i++) {
        Ecliptc[index][i] = f*R1[i] + g*Vel1[i];
        EcliptVc[index][i] = fdot*R1[i] + gdot*Vel1[i];
}

rc[index] = sqrt(Ecliptc[index][1]*Ecliptc[index][1] +
                Ecliptc[index][2]*Ecliptc[index][2] +
                Ecliptc[index][3]*Ecliptc[index][3]);


EA = x/sqrt(fabs(bodyc[5])) + EAInitial;
if (EA > 2.0*M_PI)
        EA = EA - 2.0*M_PI;
if (bodyc[5] > 0.0) {
        nuc[index] = 2.0*atan(sqrt((1 + bodyc[4])/(1 - bodyc[4]))*
                        ((sin(EA/2.0)/cos(EA/2.0))));
        if (nuc[index] < 0.0)
                nuc[index] = nuc[index] + 2.0*M_PI;
}
else {
        nuc[index] = acos((ecc - cosh(EA))/(ecc*cosh(EA) - 1.0));
        if ((EA < 0.0) && (nuc[index] > 0.0))
                nuc[index] = -nuc[index];
}

CelesLongc[index] = atan(Ecliptc[index][2]/Ecliptc[index][1]);
if ((Ecliptc[index][1]<0) && (Ecliptc[index][2]>0) && (CelesLongc[index]<0))
        CelesLongc[index] = CelesLongc[index] + M_PI;
if ((Ecliptc[index][1]<0) && (Ecliptc[index][2]<0) && (CelesLongc[index]>0))
        CelesLongc[index] = CelesLongc[index] + M_PI;
if ((Ecliptc[index][1]>0) && (Ecliptc[index][2]<0) && (CelesLongc[index]<0))
        CelesLongc[index] = CelesLongc[index] + 2.0*M_PI;
```

```
        if (bodyc[6] == 0.0)
                CelesLatc[index] = 0.0;
        else
                CelesLatc[index] = atan2(Ecliptc[index][3], rc[index]);

        V = sqrt(GM*(2.0/rc[index] - 1.0/a))*AU;
        Vtang = (AngMom)/(rc[index]*AU);
        Vrad = sqrt(V*V - Vtang*Vtang);
        pathang = atan2(Vrad,Vtang);
        if (sin(nuc[index]) < 0.0)
                pathang = -pathang;
}

/*************************************************************************/

void GiveOutput(double craftelem[8])

{
        printf("\n");
        printf("\n");
        printf("\n");
        printf("DESTINATION ASTEROID:  ");
        if (pickroid == 1)
                printf("XB\n");
        else if (pickroid == 2)
                printf("Orpheus\n");
        else if (pickroid == 3)
                printf("McAuliffe\n");
        else if (pickroid == 4)
                printf("Seleucus\n");
        else if (pickroid == 5)
                printf("Eros\n");
        printf("\n");
        printf("\n");
        printf("\n");
        printf("*************************************************\n");
        printf("*              OUTBOUND                 *\n");
        printf("*************************************************\n");

        printf("\n");
        printf("Launch date = %.6lf\n", launch);
        printf("Time of flight = %.4lf {days}\n", TOF);
        printf("\n");

        printf("HCA = %.15lf {deg}\n", HCA*RAD);
        printf("x = %.15lf\n", Battin_x);
        printf("\n");

        printf("------- Orbital Elements of Transfer Conic -------\n");
        printf("\n");
        printf("a = %.15lf {au}\n", a);
        printf("e = %.15lf\n", ecc);
```

```
        printf("p = %.15lf {au}\n", p);
        printf("h = %.15lf {km2/s}\n", AngMom);
        printf("n = %.15lf {1/s}\n", n);
        printf("i = %.15lf {deg}\n", inclin*RAD);
        printf("BigOmega = %.15lf {deg}\n", craftelem[7]*RAD);
        printf("Pie = %.15lf {deg}\n", craftelem[3]*RAD);

        printf("\n");
        printf("---------- Initial and Final Positions ----------\n");
        printf("\n");
        printf("r1 = %.15lf {au}\n", rInitial);
        printf("Lat1 = %.15lf {deg}\n", LatInitial*RAD);
        printf("Long1 = %.15lf {deg}\n", LongInitial*RAD);
        printf("nu1 = %.15lf {deg}\n", nuInitial*RAD);
        printf("EA1 = %.15lf {deg}\n", EAInitial*RAD);
        printf("V1 = %.15lf {km/s}\n", V1);
        printf("gamma1 = %.15lf {deg}\n", gamma1*RAD);
        printf("C3[1] = %.15lf {km/s}^2\n", C3[1]);
        printf("DeltaV[1] = %.15lf {km/s}\n", DeltaV[1]);
        printf("\n");
        printf("r2 = %.15lf {au}\n", rfinal);
        printf("Lat2 = %.15lf {deg}\n", Latfinal*RAD);
        printf("Long2 = %.15lf {deg}\n", Longfinal*RAD);
        printf("nu2 = %.15lf {deg}\n", nufinal*RAD);
        printf("EA2 = %.15lf {deg}\n", EAfinal*RAD);
        printf("V2 = %.15lf {km/s}\n", V2);
        printf("gamma2 = %.15lf {deg}\n", gamma2*RAD);
        printf("C3[2] = %.15lf {km/s}^2\n", C3[2]);
        printf("DeltaV[2] = %.15lf {km/s}\n", DeltaV[2]);
        printf("\n");
        printf("\n");
        printf("Total C3 = %.15lf {km/s}^2\n", (C3[1] + C3[2]));
        printf("Total DeltaV = %.15lf {km/s}\n", (DeltaV[1] + DeltaV[2]));
        printf("\n");
        printf("\n");
}


/*************************************************************************/
/***************************** GRAPHICS **********************************/
/*************************************************************************/

/********** Define Lighting Model **********/

float mat[] = {
        AMBIENT, .1, .1, .1,
        DIFFUSE, 0, .369, .165,
        SPECULAR, .5, .5, .5,
        SHININESS, 10,
        LMNULL,
};
```

```
static float lm[] = {
        AMBIENT, .1, .1, .1,
        LOCALVIEWER, 1,
        LMNULL,
};

static float lt[] = {
        LCOLOR, 1, 1, 1,
        POSITION, 4, 4, 9, 0,
        LMNULL,
};


/***************** Define Axes ***************/

double axisdata1[6][3] = {
        {-2.0, 0.0, 0.0},
        {2.0, 0.0, 0.0},
        {0.0, -2.0, 0.0},
        {0.0, 2.0, 0.0},
        {0.0, 0.0, -1.3},
        {0.0, 0.0, 1.3}
};

float xaxis1 = 2.0, yaxis1 = 2.0, zaxis1 = 1.3;

double axisdata2[6][3] = {
        {-3.0, 0.0, 0.0},
        {3.0, 0.0, 0.0},
        {0.0, -3.0, 0.0},
        {0.0, 3.0, 0.0},
        {0.0, 0.0, -1.3},
        {0.0, 0.0, 1.3}
};

float xaxis2 = 3.0, yaxis2 = 3.0, zaxis2 = 1.3;


/*************** Define Colors ***************/

long colr[8][3] = {
        {255, 0, 0},        /* red */
        {0, 255, 0},        /* green */
        {0, 0, 255},        /* blue */
        {255, 0, 255},      /* magenta */
        {255, 255, 255},    /* white */
        {255, 255, 0},      /* yellow */
        {0, 255, 255},      /* cyan */
        {0, 0, 0},          /* black */
};

/*********************************************************************/
```

```
void InitGraphics()

{
long xsize, ysize;

        printf("Enter the desired interval at which the scene is drawn:\n");
        scanf("%d", &interval);

        xsize = getgdesc(GD_XPMAX) - 15;
        ysize = getgdesc(GD_YPMAX) - 40;
        prefposition(0, xsize, 0, ysize);

        if (pickroid == 1)
                winopen("Earth ---> XB (Impulsive Thrust)   [H. Dan. Thomas]");
        else if (pickroid == 2)
                winopen("Earth ---> Orpheus (Impulsive Thrust)   [H. Dan. Thomas]");
        else if (pickroid == 3)
                winopen("Earth ---> McAuliffe (Impulsive Thrust)   [H. Dan. Thomas]");
        else if (pickroid == 4)
                winopen("Earth ---> Seleucus (Impulsive Thrust)   [H. Dan. Thomas]");
        else if (pickroid == 5)
                winopen("Earth ---> Eros (Impulsive Thrust)   [H. Dan. Thomas]");


        mmode(MVIEWING);
        perspective(180, xsize/(float)ysize, 0.1, 400.0);

        if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                lookat(7.0, 7.0, 16.0, 0.0, 0.0, 0.0, 0);
        if ((pickroid == 2) || (pickroid == 5))
                lookat(4.0, 4.0, 10.0, 0.0, 0.0, 0.0, 0);

        zbuffer(TRUE);          /* hidden surface removal */

        RGBmode();
        doublebuffer();
        gconfig();

        linewidth(2);

        lmdef(DEFMATERIAL, 1, 0, mat);
        lmdef(DEFLIGHT, 1, 0, lt);
        lmdef(DEFLMODEL, 1, 0, lm);
        lmbind(MATERIAL, 1);
        lmbind(LMODEL, 1);
        lmbind(LIGHT0, 1);

        zval = getgdesc(GD_ZMAX);

        rotate(-900, 'x');
```

```
        lsetdepth(getgdesc(GD_ZMIN), getgdesc(GD_ZMAX));

        qdevice(LEFTMOUSE);
        qdevice(MIDDLEMOUSE);
        qdevice(RIGHTMOUSE);
        qdevice(ESCKEY);
}

/*****************************************************************************/

void drawsphere(double size)

{
double inc = 15;
double theta, dtheta;
double phi, dphi;
double rsin1, rsin2;
double phicos, phisin;
double x, y, z, zorig;
double x1, y1, z1;
float n[3], v[3];
int i, j;

   zorig = 0.0;
   dtheta = M_PI/inc;
   dphi = 2.0*M_PI/inc;

   for (i = 0, theta = 0;  i < inc;  i++, theta += dtheta)  {
        z = cos(theta);
        z1 = cos(theta + dtheta);
        rsin1 = sin(theta);
        rsin2 = sin(theta + dtheta);
        bgntmesh();
          for (j = 0, phi = 0;  j <= inc;  j++, phi += dphi)  {
               if (j == 20)  phi = 0;
               phicos = cos(phi);
               phisin = sin(phi);

               x = rsin1*phicos;
               y = rsin1*phisin;

               n[0] = x; n[1] = y; n[2] = z;
               n3f(n);

               v[0] = x*size; v[1] = y*size; v[2] = (z + zorig)*size;
               v3f(v);

               x1 = rsin2*phicos;
               y1 = rsin2*phisin;

               v[0] = (x1)*size;
               v[1] = (y1)*size;
```

```
                v[2] = (z1 + zorig)*size;
                v3f(v);
            }
        endtmesh();
    }
}

/*****************************************************************************/

void Drawtick(double argx, double argy, double argz, double length, Angle angrot)

{
double tick1[3], tick2[3];

        tick1[0] = -length/2.0;
        tick1[1] = 0.0;
        tick1[2] = 0.0;
        tick2[0] = length/2.0;
        tick2[1] = 0.0;
        tick2[2] = 0.0;
        pushmatrix();
                translate(argx, argy, argz);
                rot(angrot, 'z');
                bgnline();
                        v3d(tick1);
                        v3d(tick2);
                endline();
        popmatrix();
}

/*****************************************************************************/

void Drawaxes1()

{
        c3i(colr[0]);
        bgnline();
                v3d(axisdata1[0]);
                v3d(axisdata1[1]);
        endline();
        Drawtick(-2.0, 0.0, 0.0, 0.17, 90);
        Drawtick(-1.5, 0.0, 0.0, 0.10, 90);
        Drawtick(-1.0, 0.0, 0.0, 0.17, 90);
        Drawtick(-0.5, 0.0, 0.0, 0.10, 90);
        Drawtick(0.5, 0.0, 0.0, 0.10, 90);
        Drawtick(1.0, 0.0, 0.0, 0.17, 90);
        Drawtick(1.5, 0.0, 0.0, 0.10, 90);
        Drawtick(2.0, 0.0, 0.0, 0.17, 90);
        c3i(colr[1]);
        cmov(xaxis1 + 0.05, 0, 0);
        charstr("x");
```

```
        c3i(colr[0]);
        bgnline();
                v3d(axisdata1[2]);
                v3d(axisdata1[3]);
        endline();
        Drawtick(0.0, -2.0, 0.0, 0.17, 0);
        Drawtick(0.0, -1.5, 0.0, 0.10, 0);
        Drawtick(0.0, -1.0, 0.0, 0.17, 0);
        Drawtick(0.0, -0.5, 0.0, 0.10, 0);
        Drawtick(0.0, 0.5, 0.0, 0.10, 0);
        Drawtick(0.0, 1.0, 0.0, 0.17, 0);
        Drawtick(0.0, 1.5, 0.0, 0.10, 0);
        Drawtick(0.0, 2.0, 0.0, 0.17, 0);
        c3i(colr[1]);
        cmov(0, yaxis1 + 0.05, 0);
        charstr("y");

        c3i(colr[0]);
        bgnline();
                v3d(axisdata1[4]);
                v3d(axisdata1[5]);
        endline();
        Drawtick(0.0, 0.0, -1.0, 0.17, 90);
        Drawtick(0.0, 0.0, -0.5, 0.10, 90);
        Drawtick(0.0, 0.0, 0.5, 0.10, 90);
        Drawtick(0.0, 0.0, 1.0, 0.17, 90);
        Drawtick(0.0, 0.0, -1.0, 0.17, 0);
        Drawtick(0.0, 0.0, -0.5, 0.10, 0);
        Drawtick(0.0, 0.0, 0.5, 0.10, 0);
        Drawtick(0.0, 0.0, 1.0, 0.17, 0);
        c3i(colr[1]);
        cmov(0, 0, zaxis1 + 0.05);
        charstr("z");

}

/*************************************************************************/

void Drawaxes2()

{
        c3i(colr[0]);
        bgnline();
                v3d(axisdata2[0]);
                v3d(axisdata2[1]);
        endline();
        Drawtick(-3.0, 0.0, 0.0, 0.17, 90);
        Drawtick(-2.5, 0.0, 0.0, 0.10, 90);
        Drawtick(-2.0, 0.0, 0.0, 0.17, 90);
        Drawtick(-1.5, 0.0, 0.0, 0.10, 90);
        Drawtick(-1.0, 0.0, 0.0, 0.17, 90);
        Drawtick(-0.5, 0.0, 0.0, 0.10, 90);
```

```
        Drawtick(0.5, 0.0, 0.0, 0.10, 90);
        Drawtick(1.0, 0.0, 0.0, 0.17, 90);
        Drawtick(1.5, 0.0, 0.0, 0.10, 90);
        Drawtick(2.0, 0.0, 0.0, 0.17, 90);
        Drawtick(2.5, 0.0, 0.0, 0.10, 90);
        Drawtick(3.0, 0.0, 0.0, 0.17, 90);
        c3i(colr[1]);
        cmov(xaxis2 + 0.05, 0, 0);
        charstr("x");

        c3i(colr[0]);
        bgnline();
                v3d(axisdata2[2]);
                v3d(axisdata2[3]);
        endline();
        Drawtick(0.0, -3.0, 0.0, 0.17, 0);
        Drawtick(0.0, -2.5, 0.0, 0.10, 0);
        Drawtick(0.0, -2.0, 0.0, 0.17, 0);
        Drawtick(0.0, -1.5, 0.0, 0.10, 0);
        Drawtick(0.0, -1.0, 0.0, 0.17, 0);
        Drawtick(0.0, -0.5, 0.0, 0.10, 0);
        Drawtick(0.0, 0.5, 0.0, 0.10, 0);
        Drawtick(0.0, 1.0, 0.0, 0.17, 0);
        Drawtick(0.0, 1.5, 0.0, 0.10, 0);
        Drawtick(0.0, 2.0, 0.0, 0.17, 0);
        Drawtick(0.0, 2.5, 0.0, 0.10, 0);
        Drawtick(0.0, 3.0, 0.0, 0.17, 0);
        c3i(colr[1]);
        cmov(0, yaxis2 + 0.05, 0);
        charstr("y");

        c3i(colr[0]);
        bgnline();
                v3d(axisdata2[4]);
                v3d(axisdata2[5]);
        endline();
        Drawtick(0.0, 0.0, -1.0, 0.17, 90);
        Drawtick(0.0, 0.0, -0.5, 0.10, 90);
        Drawtick(0.0, 0.0, 0.5, 0.10, 90);
        Drawtick(0.0, 0.0, 1.0, 0.17, 90);
        Drawtick(0.0, 0.0, -1.0, 0.17, 0);
        Drawtick(0.0, 0.0, -0.5, 0.10, 0);
        Drawtick(0.0, 0.0, 0.5, 0.10, 0);
        Drawtick(0.0, 0.0, 1.0, 0.17, 0);
        c3i(colr[1]);
        cmov(0, 0, zaxis2 + 0.05);
        charstr("z");

}

/*************************************************************************/
```

```
void drawendpt(double argr, double argPhi, double argPsi)

{
double x1, y1, vert1[3], vert2[3];

        vert1[0] = 0.0;
        vert1[1] = 0.0;
        vert1[2] = 0.0;
        vert2[0] = argr*cos(argPsi)*cos(argPhi);
        vert2[1] = argr*cos(argPsi)*sin(argPhi);
        vert2[2] = argr*sin(argPsi);
        bgnline();
                v3d(vert1);
                v3d(vert2);
        endline();
}

/*************************************************************************/

void PlanetOrbit(double body[8], int index)

{
double M, E, Mo, D, initx, inity, initz, vert1[3], vert2[3];
double xold, yold, zold, final;
double Transp[4][4], Ecliptp[5][4], nup[5];

   M = body[2] - body[3];
   if (M < 0)
        M = M + (2*M_PI);
   E = M + body[4]*(sin(M) + body[4]*sin(2*M)/2);
   do{
        Mo = E - body[4]*sin(E);
        D = (M - Mo)/(1 - body[4]*cos(E));
        E = E + D;
   }while (fabs(D) > tol);
   nup[index] = 2*atan(sqrt((1 + body[4])/(1 - body[4]))*((sin(E/2)/cos(E/2))));
   if (nup[index] < 0)
        nup[index] = nup[index] + 2*M_PI;
   BuildTrans(body, Transp);
   CoordConvert(body, Transp, nup, index, Ecliptp);
   initx = Ecliptp[index][1];
   inity = Ecliptp[index][2];
   initz = Ecliptp[index][3];
   final = nup[index] + 2*M_PI;
   do{
        xold = Ecliptp[index][1];
        yold = Ecliptp[index][2];
        zold = Ecliptp[index][3];
        nup[index] = nup[index] + 5.0/RAD;
        CoordConvert (body, Transp, nup, index, Ecliptp);
        vert1[0] = xold;
        vert1[1] = yold;
```

```
        vert1[2] = zold;
        vert2[0] = Ecliptp[index][1];
        vert2[1] = Ecliptp[index][2];
        vert2[2] = Ecliptp[index][3];
        bgnline();
          v3d(vert1);
          v3d(vert2);
        endline();
    }while(nup[index]<final);
    Ecliptp[index][1] = initx;
    Ecliptp[index][2] = inity;
    Ecliptp[index][3] = initz;
}

/*************************************************************************/

void GraphicStart()

{
        makeobj(objinit);

        /***************** Create Sun ******************/

                lmcolor(LMC_DIFFUSE);
                RGBcolor(255, 255, 0);
                lmcolor(LMC_COLOR);
                spheresize = 0.25;
                drawsphere(spheresize);

        /***************** Create Orbits *****************/

                if ((pickroid == 2) || (pickroid == 5))
                        Drawaxes1();
                if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                        Drawaxes2();

                c3i(colr[4]);
                PlanetOrbit(departure, 1);
                PlanetOrbit(Marselem, 2);
                PlanetOrbit(target, 3);

        /******* Create Initial and Final Position ******/

                c3i(colr[1]);
                drawendpt(rInitial, LongInitial, LatInitial);
                drawendpt(rfinal, Longfinal, Latfinal);

        closeobj();


/***************** Create earth ******************/
```

```
        makeobj(obj1);
                lmcolor(LMC_DIFFUSE);
                RGBcolor(0, 150, 255);
                lmcolor(LMC_COLOR);
                spheresize = 0.1;
                drawsphere(spheresize);
        closeobj();

/****************** Create Mars ******************/

        makeobj(obj2);
                lmcolor(LMC_DIFFUSE);
                RGBcolor(255, 80, 0);
                lmcolor(LMC_COLOR);
                spheresize = 0.05;
                drawsphere(spheresize);
        closeobj();

        if (pickroid == 1) {

        /****************** Create XB ******************/

                makeobj(objAster);
                        lmcolor(LMC_DIFFUSE);
                        RGBcolor(150, 50, 0);
                        lmcolor(LMC_COLOR);
                        spheresize = 0.02;
                        drawsphere(spheresize);
                closeobj();
        }
        else if (pickroid == 2) {

        /*************** Create Orpheus ****************/

                makeobj(objAster);
                        lmcolor(LMC_DIFFUSE);
                        RGBcolor(150, 50, 0);
                        lmcolor(LMC_COLOR);
                        spheresize = 0.02;
                        drawsphere(spheresize);
                closeobj();
        }
        else if (pickroid == 3) {

        /*************** Create McAuliffe **************/

                makeobj(objAster);
                        lmcolor(LMC_DIFFUSE);
                        RGBcolor(150, 50, 0);
                        lmcolor(LMC_COLOR);
                        spheresize = 0.025;
                        drawsphere(spheresize);
```

```
            closeobj();
        }
        else if (pickroid == 4) {

        /*************** Create Seleucus ***************/

            makeobj(objAster);
                lmcolor(LMC_DIFFUSE);
                RGBcolor(150, 50, 0);
                lmcolor(LMC_COLOR);
                spheresize = 0.025;
                drawsphere(spheresize);
            closeobj();
        }
        else if (pickroid == 5) {

        /***************** Create Eros *****************/

            makeobj(objAster);
                lmcolor(LMC_DIFFUSE);
                RGBcolor(150, 50, 0);
                lmcolor(LMC_COLOR);
                spheresize = 0.03;
                drawsphere(spheresize);
            closeobj();
        }

/************ Begin Trjectory Trace *************/

        pos1[0] = (rInitial*cos(LatInitial)*cos(LongInitial));
        pos1[1] = (rInitial*cos(LatInitial)*sin(LongInitial));
        pos1[2] = (rInitial*sin(LatInitial));

        pos2[0] = pos1[0];
        pos2[1] = pos1[1];
        pos2[2] = pos1[2];

        makeobj(objpath);
            c3i(colr[1]);
            bgnline();
                v3d(pos1);
                v3d(pos2);
            endline();
        closeobj();

/*************** Create Spacecraft **************/

/*      makeobj(objcraft);*/

/*      closeobj();*/
}
```

```
/*************************************************************************/

void GoGraphics()

{
        c3i(colr[6]);

/************* Draw Sun and Orbits **************/

        callobj(objinit);

/****************** Draw earth *******************/

        pushmatrix();
                translate(Eclipt[1][1], Eclipt[1][2], Eclipt[1][3]);
                callobj(obj1);
        popmatrix();

/****************** Draw Mars *******************/

        pushmatrix();
                translate(Eclipt[2][1], Eclipt[2][2], Eclipt[2][3]);
                callobj(obj2);
        popmatrix();

        if (pickroid == 1) {

        /******************* Draw XB ********************/

                pushmatrix();
                        translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
                        callobj(objAster);
                        c3i(colr[6]);
                        cmov(0.02, 0.02, 0.02);
                        charstr("XB");
                popmatrix();
        }
        else if (pickroid == 2) {

        /***************** Draw Orpheus *****************/

                pushmatrix();
                        translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
                        callobj(objAster);
                        c3i(colr[6]);
                        cmov(0.02, 0.02, 0.02);
                        charstr("Orph");
                popmatrix();
        }
        else if (pickroid == 3) {
```

```
        /*************** Draw McAuliffe ***************/

                pushmatrix();
                        translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
                        callobj(objAster);
                        c3i(colr[6]);
                        cmov(0.02, 0.02, 0.02);
                        charstr("Mc");
                popmatrix();
        }
        else if (pickroid == 4) {

        /*************** Draw Seleucus ***************/

                pushmatrix();
                        translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
                        callobj(objAster);
                        c3i(colr[6]);
                        cmov(0.02, 0.02, 0.02);
                        charstr("Sel");
                popmatrix();
        }
        else if (pickroid == 5) {

        /***************** Draw Eros ******************/

                pushmatrix();
                        translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
                        callobj(objAster);
                        c3i(colr[6]);
                        cmov(0.02, 0.02, 0.02);
                        charstr("Eros");
                popmatrix();
        }

/*************** Draw Trajectory **************/

        callobj(objpath);

/*************** Draw Spacecraft **************/

/*      pushmatrix();*/
/*              translate(pos2[0], pos2[1], pos2[2]);*/
/*              callobj(objcraft);*/
/*      popmatrix();*/
}


/*****************************************************************************/

void PathUpdate()
```

```
{
double phi, gammapath, anglep;
double length = 0.10;

        pos1[0] = pos2[0];
        pos1[1] = pos2[1];
        pos1[2] = pos2[2];
        pos2[0] = Eclipt[4][1];
        pos2[1] = Eclipt[4][2];
        pos2[2] = Eclipt[4][3];

        editobj(objpath);
                c3i(colr[1]);
                bgnline();
                        v3d(pos1);
                        v3d(pos2);
                endline();
                if (fabs(fmod(time,50)) == 0.0) {
                        phi = atan2(pos2[1],pos2[0]);
                        gammapath = pathang;
                        anglep = (phi - gammapath)*RAD;
                        Drawtick(Eclipt[4][1], Eclipt[4][2], Eclipt[4][3], 0.10, anglep);
                }
        closeobj();
}

/*************************************************************************/

void PrintTOF()

{
char Tstr[30];

        c3i(colr[1]);
        sprintf(Tstr, "FLIGHT TIME = %4.2lf days", time);

        if ((pickroid == 2) || (pickroid == 5))
                cmov(-2.0, 1.25, 0.5);
        else if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                cmov(-3.0, 2.25, 0.5);

        charstr(Tstr);
}

/*************************************************************************/

void Printdates()

{
char datestr1[30];
char datestr2[30];
```

```
        c3i(colr[1]);
        sprintf(datestr1, "Departure Date:  %9.6lf", ClaunchT);
        sprintf(datestr2, "Arrival Date:    %9.6lf", CarriveT);
        if ((pickroid == 2) || (pickroid == 5))
                cmov(-1.954, 1.55, 0.8);
        else if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                cmov(-2.954, 2.55, 0.8);
        charstr(datestr1);
        if ((pickroid == 2) || (pickroid == 5))
                cmov(-1.97, 1.45, 0.7);
        else if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                cmov(-2.97, 2.45, 0.7);
        charstr(datestr2);
}


/*************************************************************************/
/************************** MAIN PROGRAM *************************/
/*************************************************************************/

main()
{
double tper;
int i;

        RAD = 180.0/M_PI;

        CaseEntry ();
        SpecifyEndpts(EcliptVP);
        Trajelem(spacecraft, EcliptVP, &C3[1], &C3[2]);
        earthPark(EcliptVP, EcliptV, C3[1], &DeltaV[1]);
        AsterRend(C3[2], &DeltaV[2]);
        GiveOutput(spacecraft);
        printf("\n");

        tper = (1.0/n)*(EAInitial - ecc*sin(EAInitial));
        t = tper;
        time = 0.0;
        Jt1 = Jcentury1[1];
        Jt2 = Jcentury1[2];
        Jt3 = Jcentury1[3];
        earth(departure, Jt1);
        Mars(Marselem, Jt1);

        if (pickroid == 1)
                XB(target, Jt2);
        else if (pickroid == 2)
                Orpheus(target, Jt2);
        else if (pickroid == 3)
                McAuliffe(target, Jt2);
        else if (pickroid == 4)
                Seleucus(target, Jt2);
```

```
        else if (pickroid == 5)
                Eros(target, Jt3);

        PlanetPosition(departure, r, nu, Long, Lat, Vel, Eclipt, EcliptV, 1);
        PlanetPosition(Marselem, r, nu, Long, Lat, Vel, Eclipt, EcliptV, 2);
        PlanetPosition(target, r, nu, Long, Lat, Vel, Eclipt, EcliptV, 3);

        Eclipt[4][1] = Eclipt[2][1];
        Eclipt[4][2] = Eclipt[2][2];
        Eclipt[4][3] = Eclipt[2][3];
        r[4] = rInitial;
        nu[4] = nuInitial;

        printf("Press enter to continue \n");
        getchar();
        getchar();

        InitGraphics();
        GraphicStart();

        GetCaldate(Juliandate, &CdateT);

/******************** Draw Trajectory ********************/

        while (Jt1 < Jcentury2[1]) {

          count = count + 1;

          if (count == interval) {
                czclear(0x000000, zval);        /* use with zbuffer */
                GoGraphics();
                PrintTOF();
                Printdates();
                swapbuffers();
                count = 0;
          }

          t = t + stepdays*86400.0;
          time = time + stepdays;
          Juliandate = Juliandate + stepdays;
          GetCaldate(Juliandate, &CdateT);

          Jt1 = Jt1 + Jdt;
          Jt2 = Jt2 + Jdt;
          Jt3 = Jt3 + Jdt;

          earth(departure, Jt1);
          Mars(Marselem, Jt1);
          if (pickroid == 1)
                XB(target, Jt2);
          else if (pickroid == 2)
                Orpheus(target, Jt2);
```

```
        else if (pickroid == 3)
            McAuliffe(target, Jt2);
        else if (pickroid == 4)
            Seleucus(target, Jt2);
        else if (pickroid == 5)
            Eros(target, Jt3);

        PlanetPosition(departure, r, nu, Long, Lat, Vel, Eclipt, EcliptV, 1);
        PlanetPosition(Marselem, r, nu, Long, Lat, Vel, Eclipt, EcliptV, 2);
        PlanetPosition(target, r, nu, Long, Lat, Vel, Eclipt, EcliptV, 3);

        CraftPos(spacecraft, r, nu, Long, Lat, Eclipt, EcliptV, 4);
        PathUpdate();

        if (getbutton(ESCKEY)) {
            gexit();
            exit(0);
        }
    }

/****************** Create Final Scene Object ********************/

    Jt1 = Jcentury2[1];
    Jt2 = Jcentury2[2];
    Jt3 = Jcentury2[3];
    t = tper + TOF*86400;
    time = TOF;

    earth(departure, Jt1);
    Mars(Marselem, Jt1);
    if (pickroid == 1)
        XB(target, Jt2);
    else if (pickroid == 2)
        Orpheus(target, Jt2);
    else if (pickroid == 3)
        McAuliffe(target, Jt2);
    else if (pickroid == 4)
        Seleucus(target, Jt2);
    else if (pickroid == 5)
        Eros(target, Jt3);

    PlanetPosition(departure, r, nu, Long, Lat, Vel, Eclipt, EcliptV, 1);
    PlanetPosition(Marselem, r, nu, Long, Lat, Vel, Eclipt, EcliptV, 2);
    PlanetPosition(target, r, nu, Long, Lat, Vel, Eclipt, EcliptV, 3);

    Eclipt[4][1] = Eclipt[3][1];
    Eclipt[4][2] = Eclipt[3][2];
    Eclipt[4][3] = Eclipt[3][3];
    r[4] = rfinal;
    nu[4] = nufinal;

    PathUpdate();
```

```
        makeobj(objfinal);
                GoGraphics();
                PrintTOF();
                Printdates();
        closeobj();

/************************ Draw Final Scene ************************/

        czclear(0x000000, zval);
        callobj(objfinal);
        swapbuffers();

        while (1) {
                if (getbutton(LEFTMOUSE)) {
                        rotate(10, 'x');
                        czclear(0x000000, zval);
                                callobj(objfinal);
                        swapbuffers();
                }
                if (getbutton(MIDDLEMOUSE)) {
                        rotate(10, 'y');
                        czclear(0x000000, zval);
                                callobj(objfinal);
                        swapbuffers();
                }
                if (getbutton(RIGHTMOUSE)) {
                        rotate(10, 'z');
                        czclear(0x000000, zval);
                                callobj(objfinal);
                        swapbuffers();
                }
                if (getbutton(ESCKEY)) {
                        gexit();
                        exit(0);
                }
        }
}
```

## APPENDIX B: Power-Limited Program

```
/*************************************************************************/
/*************************************************************************/
/*      Program:      3D Power-limited Trajectory (pt. to pt.)           */
/*                                                                       */
/*      By:     H. Dan. Thomas                                           */
/*                                                                       */
/*      Date:   August 31, 1993                                          */
/*                                                                       */
/*      Last Major Revision:  August 31, 1993                            */
/*      Last Minor Revision:  August 31, 1993                            */
/*                                                                       */
/*      Description:  This ANSI C program calculates and draws the optimum */
/*                    power-limited trajectory between earth and select asteroids */
/*                    (full 3-D). The trajectory is from a specific point on earth's */
/*                    orbit to a specific point on the target asteroid's orbit. The */
/*                    flight time is fixed, and the program iterates to the proper */
/*                    launch date which minimizes the fuel expenditure of the */
/*                    rocket.                                            */
/*************************************************************************/
/*************************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/device.h>


double AU = 1.4959965E+8;                /* km/au */
double GM = 3.96396415708E-14;           /* au3/s2 */
double accgrav = 9.81;                   /* m/s2 */
double dt = 0.005;
double tol = 1e-15;
double tol1 = 1e-10;
double inf = 1e+15;
double RAD;                              /* Conversion from radians to degrees */

double pos1[3], pos2[3];
double spheresize;
double saillength = 0.07, svrt1[3], svrt2[3], svrt3[3], svrt4[3];
long zval;
Object objinit = 11, obj1 = 1, obj2 = 2, objpath = 12, objsail = 13;
Object objAster = 14, objfinal = 15;
int count, interval;

double  Tfinal;          /* The date of arrival at the target planet */
double  tstar;           /* Time non-dimensionalization factor */
double  vstar;           /* Velocity non-dimensionalization factor */
```

```
double astar;              /* Acceleration non-dimensionalization factor */
double Vr;                 /* Radial velocity of sail */
double Vphi;               /* Tangential velocity of sail */
double Vpsi;               /* Normal velocity of sail */
double r;                  /* Radial position of sail */
double Phi;                /* Celestial Latitude in 3-D spherical */
double Psi;                /* Celestial Longitude in 3-D spherical */
double PVr;                /* Lagrange multiplier conjugate to Vr */
double PVphi;              /* Lagrange multiplier conjugate to Vphi */
double PVpsi;              /* Lagrange multiplier conjugate to Vpsi */
double Pr;                 /* Lagrange multiplier conjugate to radius */
double Pphi;               /* Lagrange multiplier conjugate to Phi */
double Ppsi;               /* Lagrange multiplier conjugate to Ppsi */
double rInitial;           /* Initial r of solar sail */
double rfinal;             /* Final r of solar sail */
double PhiInitial;         /* Initial Phi of solar sail */
double PsiInitial;         /* Initial Psi of solar sail */
double Phifinal;           /* Final Phi of solar sail */
double Psifinal;           /* Final Psi of solar sail */
double Vrfinal;            /* Final Vr of sail */
double Vphifinal;          /* Final Vphi of sail */
double Vpsifinal;          /* Final Vpsi of sail */
double Hamil;              /* The Hamiltonian */
double JRK;                /* Integral of a^2 dt over flight time (Runge-Kutta(4)) */
double JEUL;               /* Integral of a^2 dt over flight time (Euler's Method) */
double t;                  /* non-dimensionalized time */
double time;               /* Time ellapsed in days */
double Jt1;                /* Julian time (in centuries) */
double Jt2;                /* Julian time (in centuries) */
double Jt3;                /* Julian time (in centuries) */
double Jdt;                /* Julian stepsize (in centuries) */
double Jcentury1[4];       /* Initial Julian time (in centuries) */
double Jcentury2[4];       /* Final Julian time (in centuries) */
double ClaunchT;           /* Calendar departure date */
double CarriveT;           /* Calendar arrival date */
double ar;                 /* Thrust acceleration in the r-direction */
double aphi;               /* Thrust acceleration in the phi-direction */
double apsi;               /* Thrust acceleration in the psi-direction */
double atot;               /* Total thrust acceleration */
double spmass;             /* Specific mass of powerplant */
double eff;                /* System efficiency */
double Isp;                /* Specific impulse of the propulsion system */
double Jfunct, gam;
double JEpoch1 = 2451545.0;
double JEpoch2 = 11.051990;
double JEpoch3 = 6.191986;
double x[10];
double y[10];
double departure[8], target[8], Marselem[8];
double rdum[4], Phidum[4], Psidum[4], Vrdum[4], Vphidum[4], Vpsidum[4],
double TOF, Eclipt[4][4];
double tstar2;
```

```
double  nplan, tau;
int     pickroid;
FILE *fp;
char filestr[15];


/*************************************************************************/
/********************** GOVERNING EQUATIONS **********************/
/*************************************************************************/

double Dr (double arg4)
{
double funcval;

        funcval = arg4;
        return funcval;
}

/*************************************************************************/

double Dphi (double arg1, double arg3, double arg5)
{
double funcval;

        funcval = arg5/(arg1*cos(arg3));
        return funcval;
}

/*************************************************************************/

double Dpsi (double arg1, double arg6)
{
double funcval;

        funcval = arg6/arg1;
        return funcval;
}

/*************************************************************************/

double DVr (double arg1, double arg5, double arg6, double accr)
{
double funcval;

        funcval = (arg5*arg5 + arg6*arg6)/arg1 - 1.0/(arg1*arg1) + accr;
        return funcval;
}

/*************************************************************************/

double DVphi (double arg1, double arg3, double arg4, double arg5, double arg6, double
accphi)
```

```
{
double funcval;

        funcval = -(arg4*arg5/arg1) + (arg5*arg6*tan(arg3)/arg1) + accphi;
        return funcval;
}

/*************************************************************************/

double DVpsi (double arg1, double arg3, double arg4, double arg5, double arg6,
                double accpsi)
{
double funcval;

        funcval = -(arg4*arg6/arg1) - (arg5*arg5*tan(arg3)/arg1) + accpsi;
        return funcval;
}

/*************************************************************************/

double DPvr (double arg1, double arg5, double arg6, double arg7, double arg10,
                double arg11)
{
double funcval;

        funcval = arg10*arg5/arg1 + arg11*arg6/arg1 - arg7;
        return funcval;
}

/*************************************************************************/

double DPvphi (double arg1, double arg3, double arg4, double arg5, double arg6,
                        double arg9, double arg10, double arg11)
{
double funcval;

        funcval = -(2*arg9*arg5/arg1) + arg4*arg10/arg1 - arg10*arg6*tan(arg3)/arg1 +
                        2*arg11*arg5*tan(arg3)/arg1;
        return funcval;
}

/*************************************************************************/

double DPvpsi (double arg1, double arg3, double arg4, double arg5, double arg6, double
arg8,
                double arg9, double arg10, double arg11)
{
double funcval;

        funcval = -(2*arg9*arg6/arg1) + arg4*arg11/arg1 - arg10*arg5*tan(arg3)/arg1 -
                        arg8/arg1;
        return funcval;
```

```
}

/*************************************************************************/

double DPr (double arg1, double arg3, double arg4, double arg5, double arg6,
            double arg8, double arg9, double arg10, double arg11)
{
double C1, C2, C3, C4, C5, C6, C7, funcval;

        C1 = arg8*arg6/(arg1*arg1);
        C2 = arg9*(arg5*arg5 + arg6*arg6)/(arg1*arg1);
        C3 = 2*arg9/(arg1*arg1*arg1);
        C4 = arg10*arg4*arg5/(arg1*arg1);
        C5 = arg10*arg5*arg6*tan(arg3)/(arg1*arg1);
        C6 = arg11*arg4*arg6/(arg1*arg1);
        C7 = arg11*arg5*arg5*tan(arg3)/(arg1*arg1);
        funcval = C1 + C2 - C3 - C4 + C5 - C6 - C7;
        return funcval;
}

/*************************************************************************/

double DPpsi (double arg1, double arg3, double arg5, double arg6, double arg10,
              double arg11)
{
double funcval;

        funcval = -(arg10*arg5*arg6/(arg1*cos(arg3)*cos(arg3))) +
                        arg11*arg5*arg5/(arg1*cos(arg3)*cos(arg3));
        return funcval;
}

/*************************************************************************/

double DJ (double arg9, double arg10, double arg11)
{
double funcval;

        funcval = arg9*arg9 + arg10*arg10 + arg11*arg11;
        return funcval;
}

/*************************************************************************/

double CalcIsp(double arga, double argJ)

{
double funcval;

        funcval = (1.0/(accgrav*arga))*((2.0*eff/spmass)*(gam - gam*gam) + argJ);
        return funcval;
```

```
}

/*****************************************************************************/
/************************** PROCEDURES ***************************************/
/*****************************************************************************/

void earth(double elem[8], double argt)

        /* This procedure determines the orbital elements    */
        /* of earth at a specific time.                      */

/* Epoch = 2000 AD */

{
        elem[1] = 0.0;
        elem[2] = (100.0466449 + 36000.769823*argt + 0.000304*argt*argt)/RAD;
        elem[3] = (102.937348 + 1.719539*argt + 0.000460*argt*argt)/RAD;
        elem[4] = 0.01670862 - 0.00004204*argt;
        elem[5] = 1.00001161;
        elem[6] = 0.0;
        elem[7] = 0.0;
}

/*****************************************************************************/

void Mars(double elem[8], double argt)

        /* This procedure determines the orbital elements of */
        /* Mars at a specific time.                          */

/* Epoch = 2000 AD */

{
        elem[1] = 0.0;
        elem[2] = (355.433275 + 19141.696475*argt + 0.000311*argt*argt)/RAD;
        elem[3] = (336.060234 + 1.841044*argt + 0.000135*argt*argt)/RAD;
        elem[4] = 0.09340062 + 0.00009048*argt - 0.00000008*argt*argt;
        elem[5] = 1.52371069;
        elem[6] = (1.849726 - 0.000601*argt + 0.000013*argt*argt)/RAD;
        elem[7] = (49.558093 + 0.772096*argt + 0.000016*argt*argt)/RAD;
}

/*****************************************************************************/

void XB(double elem[8], double argt)

/* Epoch = 11.051990 */

{
        elem[1] = 16.77969/RAD;
        elem[2] = 56.89879/RAD;
```

```
        elem[3] = 91.30613/RAD;
        elem[4] = 0.4466558;
        elem[5] = 1.8355863;
        elem[6] = 3.87446/RAD;
        elem[7] = 74.52644/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch2;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void Orpheus(double elem[8], double argt)

/* Epoch = 11.051990 */

{
        elem[1] = 301.56931/RAD;
        elem[2] = 197.58723/RAD;
        elem[3] = 130.70829/RAD;
        elem[4] = 0.3226346;
        elem[5] = 1.2093243;
        elem[6] = 2.68775/RAD;
        elem[7] = 189.13898/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch2;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void McAuliffe(double elem[8], double argt)

/* Epoch = 11.051990 */

{
        elem[1] = 15.59117/RAD;
        elem[2] = 283.00658/RAD;
        elem[3] = 122.49259/RAD;
        elem[4] = 0.3694649;
        elem[5] = 1.8787307;
        elem[6] = 4.77743/RAD;
        elem[7] = 106.90142/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch2;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
```

```
}

/*************************************************************************/

void Seleucus(double elem[8], double argt)

/* Epoch = 11.051990 */

{
        elem[1] = 349.18871/RAD;
        elem[2] = 343.98574/RAD;
        elem[3] = 207.31474/RAD;
        elem[4] = 0.4571202;
        elem[5] = 2.0325801;
        elem[6] = 5.93181/RAD;
        elem[7] = 218.12603/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch2;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void Eros(double elem[8], double argt)

/* Epoch = 6.191986 */

{
        elem[1] = 178.510/RAD;
        elem[2] = 170.451/RAD;
        elem[3] = 123.006/RAD;
        elem[4] = 0.2229;
        elem[5] = 1.4582;
        elem[6] = 10.832/RAD;
        elem[7] = 304.496/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch3;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void InvConvert(double date2, double *argdate)

                        /* This procedure converts any Julian date to the standard  */
                        /* calender date.                                           */
                        /* Taken from Astronomical Formulae for Calculators,        */
```

```
                    /*Meeus, p. 26.                                          */

{
long Month;         /* The portion of the enterred date that represents the month */
double Day;         /* The portion of the enterred date that represents the day   */
long Year;          /* The protion of the enterred date that represents the year  */
long a, b, c, d, e;
double f;

        f = date2 + 0.5;
        a = trunc(f);
        f = f - a;
        if (a >= 2299161) {
                b = (a - 1867216.25)/36524.25;
                a += 1 + b - b/4;
        }
        b = a + 1524;
        c = (b - 122.1)/365.25;
        d = 365.25*c;
        e = (b - d)/30.6001;
        Day = b - d - trunc(30.6001*e) + f;
        if (e <= 13)
                Month = e - 1;
        else
                Month = e - 13;
        if (Month <= 2)
                Year = c - 4715;
        else
                Year = c - 4716;
        *argdate = trunc(Month) + 0.01*rint(Day) + 0.000001*trunc(Year);
}

/*************************************************************************/

void Convert(double *date1)

                    /* This procedure converts any date entered in the standard */
                    /* calender into a Julian date.                             */
                    /* Taken from Astronomical Formulae for Calculators,        */
                    /* Meeus, p. 24.                                            */

{
long Month;         /* The portion of the enterred date that represents the month */
long Day;           /* The portion of the enterred date that represents the day   */
long Year;          /* The protion of the enterred date that represents the year  */
long a, b;

        Month = trunc(*date1);
        Day = trunc(100*(*date1 - Month));
        Year = rint((1000000.0*(*date1)) - (Month*1000000.0) - (Day*10000.0));
        if (Month <= 2) {
                Year = Year - 1;
```

```
            Month = Month + 12;
        }
        a = Year/100;
        b = 2 - a + a/4;
        *date1 = trunc(365.25*Year) + trunc(30.6001*(Month + 1.0)) + Day +
                        1720994.5 + b;
}

/*************************************************************************/

void GetCaldate(double Juldate1, double *argdate)

{
double Juldate2, Caldate1, Caldate2;

        InvConvert(Juldate1, &Caldate1);
        Juldate2 = Caldate1;
        Convert(&Juldate2);
        *argdate = Caldate1;
        if (Juldate2 > Juldate1)
                *argdate = (*argdate) - 0.01;
}

/*************************************************************************/

void CaseEntry ()

                        /* This procedure prompts for the initial guesses for launch */
                        /* date and time of flight.                                  */

{
double JlaunchT;

        if ((JEpoch1/1000.0) < 1)
                Convert(&JEpoch1);
        if ((JEpoch2/1000.0) < 1)
                Convert(&JEpoch2);
        if ((JEpoch3/1000.0) < 1)
                Convert(&JEpoch3);

        printf("Enter the number for the desired asteroid:\n");
        printf("       1: XB-1982\n");
        printf("       2: Orpheus\n");
        printf("       3: McAuliffe\n");
        printf("       4: Seleucus\n");
        printf("       5: Eros\n");
        scanf("%i", &pickroid);

        printf("Enter the initial guess for launch date as mm.ddyyyy or as a ");
        printf("Julian date:\n");
        scanf("%lf", &JlaunchT);
        if ((JlaunchT/1000.0) < 1)
```

```
        Convert(&JlaunchT);
    x[1] = JlaunchT;
    printf("Enter the desired time of flight to target in days:\n");
    scanf("%lf", &TOF);

    printf("Enter the specific mass of the powerplant (kg/kW):\n");
    scanf("%lf", &spmass);
    spmass = spmass/1000.0;

    printf("Enter the system efficiency (percent):\n");
    scanf("%lf", &eff);
    eff = eff/100.0;

    gets(filestr);
    printf("Enter the desired output file name:\n");
    gets(filestr);
}

/*************************************************************************/

void BuildTrans1(double body[8], double Trans1[4][4])

                    /* This procedure builds the matrix which serves    */
                    /* as the transformation matrix between the         */
                    /* ecliptic and pericentric coordinate system       */

{
double SmOmega;

    SmOmega = body[3] - body[7];
    Trans1[1][1] = cos(body[7])*cos(SmOmega) -
                        sin(body[7])*sin(SmOmega)*cos(body[6]);
    Trans1[1][2] = -cos(body[7])*sin(SmOmega) -
                        sin(body[7])*cos(SmOmega)*cos(body[6]);
    Trans1[1][3] = sin(body[7])*sin(body[6]);
    Trans1[2][1] = sin(body[7])*cos(SmOmega) +
                        cos(body[7])*sin(SmOmega)*cos(body[6]);
    Trans1[2][2] = -sin(body[7])*sin(SmOmega) +
                        cos(body[7])*cos(SmOmega)*cos(body[6]);
    Trans1[2][3] = -cos(body[7])*sin(body[6]);
    Trans1[3][1] = sin(SmOmega)*sin(body[6]);
    Trans1[3][2] = cos(SmOmega)*sin(body[6]);
    Trans1[3][3] = cos(body[6]);
}

/*************************************************************************/

void BuildTrans2 (double Phip[4], double Psip[4], int index, double Trans2[4][4])

                    /* This procedure builds the matrix which serves    */
                    /* as the transformation matrix between the         */
                    /* spherical and ecliptic coordinate system         */
```

```
{
        Trans2[1][1] = cos(Phip[index])*cos(Psip[index]);
        Trans2[1][2] = sin(Phip[index])*cos(Psip[index]);
        Trans2[1][3] = sin(Psip[index]);
        Trans2[2][1] = -sin(Phip[index]);
        Trans2[2][2] = cos(Phip[index]);
        Trans2[2][3] = 0.0;
        Trans2[3][1] = -sin(Psip[index])*cos(Phip[index]);
        Trans2[3][2] = -sin(Psip[index])*sin(Phip[index]);
        Trans2[3][3] = cos(Psip[index]);
}

/*************************************************************************/

void CoordConvert(double body[8], double TransC[4][4], double nuC[4], int i,
                        double EcliptC[4][4])

{
double Peri[4][4];

        Peri[i][1] = (body[5]*(1 - body[4]*body[4])/(1 +
                        body[4]*cos(nuC[i])))*cos(nuC[i]);
        Peri[i][2] = (body[5]*(1 - body[4]*body[4])/(1 +
                        body[4]*cos(nuC[i])))*sin(nuC[i]);
        Peri[i][3] = 0.0;
        EcliptC[i][1] = TransC[1][1]*Peri[i][1] + TransC[1][2]*Peri[i][2] +
                                TransC[1][3]*Peri[i][3];
        EcliptC[i][2] = TransC[2][1]*Peri[i][1] + TransC[2][2]*Peri[i][2] +
                                TransC[2][3]*Peri[i][3];
        EcliptC[i][3] = TransC[3][1]*Peri[i][1] + TransC[3][2]*Peri[i][2] +
                                TransC[3][3]*Peri[i][3];
}

/*************************************************************************/

void PlanetPosition (double body[8], double rp[4], double Phip[4], double Psip[4],
                        double Vrp[4], double Vphip[4], double Vpsip[4],
                        double Ecliptp[4][4], int index)

{
double Ma, Mo, Ea, D, SminAx, SmOmega, VPx, VPy;
double Velp[4], nup[4], Transp1[4][4], Transp2[4][4], EcliptV[4][4];

        Ma = body[2] - body[3];
        if (Ma < 0.0)
                Ma = Ma + (2.0*M_PI);
        Ea = Ma + body[4]*(sin(Ma) + body[4]*sin(2.0*Ma)/2);
        do{
                Mo = Ea - body[4]*sin(Ea);
                D = (Ma - Mo)/(1 - body[4]*cos(Ea));
                Ea = Ea + D;
```

```
        }while (fabs(D) > tol1);
        nup[index] = 2.0*atan(sqrt((1 + body[4])/(1 - body[4]))*
                        ((sin(Ea/2.0)/cos(Ea/2.0))));
        if (nup[index] < 0.0)
                nup[index] = nup[index] + 2.0*M_PI;
        SminAx = body[5]*sqrt(1 - body[4]*body[4]);
        SmOmega = body[3] - body[7];
        BuildTrans1(body, Transp1);
        CoordConvert(body, Transp1, nup, index, Ecliptp);
        Phip[index] = atan(Ecliptp[index][2]/Ecliptp[index][1]);
        rp[index] = body[5]*(1 - body[4]*body[4])/(1 + body[4]*cos(nup[index]));
        VPx = (-body[5]*sin(Ea)/(sqrt(body[5]*body[5]*body[5])*(1 -
                body[4]*cos(Ea))));
        VPy = (SminAx*cos(Ea)/(sqrt(body[5]*body[5]*body[5])*(1 -
                body[4]*cos(Ea))));
        Velp[index] = sqrt(2.0/rp[index] - 1.0/body[5]);
        EcliptV[index][1] = Transp1[1][1]*VPx + Transp1[1][2]*VPy;
        EcliptV[index][2] = Transp1[2][1]*VPx + Transp1[2][2]*VPy;
        EcliptV[index][3] = Transp1[3][1]*VPx + Transp1[3][2]*VPy;
        if ((Ecliptp[index][1] < 0) && (Ecliptp[index][2] > 0) && (Phip[index] < 0))
                Phip[index] = Phip[index] + M_PI;
        if ((Ecliptp[index][1] < 0) && (Ecliptp[index][2] < 0) && (Phip[index] > 0))
                Phip[index] = Phip[index] + M_PI;
        if ((Ecliptp[index][1] > 0) && (Ecliptp[index][2] < 0) && (Phip[index] < 0))
                Phip[index] = Phip[index] + 2.0*M_PI;
        Psip[index] = asin(sin(SmOmega + nup[index])*sin(body[6]));
        BuildTrans2(Phip, Psip, index, Transp2);
        Vrp[index] = Transp2[1][1]*EcliptV[index][1] +
                        Transp2[1][2]*EcliptV[index][2] +
                                Transp2[1][3]*EcliptV[index][3];
        Vphip[index] = Transp2[2][1]*EcliptV[index][1] +
                                Transp2[2][2]*EcliptV[index][2] +
                                Transp2[2][3]*EcliptV[index][3];
        Vpsip[index] = Transp2[3][1]*EcliptV[index][1] +
                                Transp2[3][2]*EcliptV[index][2] +
                                Transp2[3][3]*EcliptV[index][3];
}

/*************************************************************************/

void InitialConditions (double xi[10], double yi[10])

{
double JlaunchT, JarriveT, stepdays;
double ri[4], Vri[4], Vphii[4], Vpsii[4], Phii[4], Psii[4];
double departureI[8], targetI[8], Ecldum[4][4];
int i;

        JlaunchT = xi[1];
        Jcentury1[1] = (JlaunchT - JEpoch1)/(36525.0);
        Jcentury1[2] = JlaunchT/36525.0;
        Jcentury1[3] = JlaunchT/36525.0;
```

```
JarriveT = xi[1] + TOF;
Jcentury2[1] = (JarriveT - JEpoch1)/(36525.0);
Jcentury2[2] = JarriveT/36525.0;
Jcentury2[3] = JarriveT/36525.0;

GetCaldate(JlaunchT, &ClaunchT);
GetCaldate(JarriveT, &CarriveT);

stepdays = dt*sqrt(1.0/GM)/86400.0;

Jdt = (((JlaunchT + stepdays) - JEpoch1)/36525.0) - Jcentury1[1];

earth(departureI, Jcentury1[1]);
if (pickroid == 1)
        XB(targetI, Jcentury2[2]);
else if (pickroid == 2)
        Orpheus(targetI, Jcentury2[2]);
else if (pickroid == 3)
        McAuliffe(targetI, Jcentury2[2]);
else if (pickroid == 4)
        Seleucus(targetI, Jcentury2[2]);
else if (pickroid == 5)
        Eros(targetI, Jcentury2[3]);

PlanetPosition (departureI, ri, Phii, Psii, Vri, Vphii, Vpsii, Ecldum, 1);
PlanetPosition (targetI, ri, Phii, Psii, Vri, Vphii, Vpsii, Ecldum, 3);

rInitial = ri[1];
rfinal = ri[3];
PhiInitial = Phii[1];
Phifinal = Phii[3];
PsiInitial = Psii[1];
Psifinal = Psii[3];
if (PhiInitial >= 2.0*M_PI)
        PhiInitial = PhiInitial - 2.0*M_PI;
if (Phifinal >= 2.0*M_PI)
        Phifinal = Phifinal - 2.0*M_PI;
Vrfinal = Vri[3];
Vphifinal = Vphii[3];
Vpsifinal = Vpsii[3];
Jt1 = Jcentury1[1];
Jt2 = Jcentury1[2];
Jt3 = Jcentury1[3];

t = 0.0;
time = 0.0;
Vr = Vri[1];
Vphi = Vphii[1];
Vpsi = Vpsii[1];
r = rInitial;
Phi = PhiInitial;
```

```
        Psi = PsiInitial;
        Tfinal = TOF/tstar;
        Pr = xi[2];
        PVr = xi[3];
        PVphi = xi[4];
        Ppsi = xi[5];
        PVpsi = xi[6];
        Pphi = 0.0;
        ar = PVr;
        aphi = PVphi;
        apsi = PVpsi;
        JRK = ar*ar + aphi*aphi + apsi*apsi;
        JEUL = ar*ar + aphi*aphi + apsi*apsi;
        atot = sqrt(ar*ar + aphi*aphi + apsi*apsi);
        Isp = CalcIsp(atot, JEUL);
}

/*************************************************************************/

void RungeKutta ()

{
double K1, K2, K3, K4, L1, L2, L3, L4, M1, M2, M3, M4, N1, N2, N3, N4;
double O1, O2, O3, O4, P1, P2, P3, P4, Q1, Q2, Q3, Q4, R1, R2, R3, R4;
double S1, S2, S3, S4, U1, U2, U3, U4, W1, W2, W3, W4, J1, J2, J3, J4;

        K1 = dt*Dr(Vr);
        L1 = dt*Dphi(r, Psi, Vphi);
        M1 = dt*Dpsi(r, Vpsi);
        N1 = dt*DVr(r, Vphi, Vpsi, PVr);
        O1 = dt*DVphi(r, Psi, Vr, Vphi, Vpsi, PVphi);
        P1 = dt*DVpsi(r, Psi, Vr, Vphi, Vpsi, PVpsi);
        Q1 = dt*DPvr(r, Vphi, Vpsi, Pr, PVphi, PVpsi);
        R1 = dt*DPvphi(r, Psi, Vr, Vphi, Vpsi, PVr, PVphi, PVpsi);
        S1 = dt*DPvpsi(r, Psi, Vr, Vphi, Vpsi, Ppsi, PVr, PVphi, PVpsi);
        U1 = dt*DPr(r, Psi, Vr, Vphi, Vpsi, Ppsi, PVr, PVphi, PVpsi);
        W1 = dt*DPpsi(r, Psi, Vphi, Vpsi, PVphi, PVpsi);
        J1 = dt*tstar2*DJ(PVr*astar, PVphi*astar, PVpsi*astar);

        K2 = dt*Dr(Vr + N1/2.0);
        L2 = dt*Dphi(r + K1/2.0, Psi + M1/2.0, Vphi + O1/2.0);
        M2 = dt*Dpsi(r + K1/2.0, Vpsi + P1/2.0);
        N2 = dt*DVr(r + K1/2.0, Vphi + O1/2.0, Vpsi + P1/2.0, PVr + Q1/2.0);
        O2 = dt*DVphi(r + K1/2.0, Psi + M1/2.0, Vr + N1/2.0, Vphi + O1/2.0,
                        Vpsi + P1/2.0, PVphi + R1/2.0);
        P2 = dt*DVpsi(r + K1/2.0, Psi + M1/2.0, Vr + N1/2.0, Vphi + O1/2.0,
                        Vpsi + P1/2.0, PVpsi + S1/2.0);
        Q2 = dt*DPvr(r + K1/2.0, Vphi + O1/2.0, Vpsi + P1/2.0, Pr + U1/2.0,
                    PVphi + R1/2.0,  PVpsi + S1/2.0);
        R2 = dt*DPvphi(r + K1/2.0, Psi + M1/2.0, Vr + N1/2.0, Vphi + O1/2.0,
                        Vpsi + P1/2.0, PVr + Q1/2.0, PVphi + R1/2.0,
```

```
                        PVpsi + S1/2.0);
S2 = dt*DPvpsi(r + K1/2.0, Psi + M1/2.0, Vr + N1/2.0, Vphi + O1/2.0,
                        Vpsi + P1/2.0, Ppsi + W1/2.0, PVr + Q1/2.0,
                        PVphi + R1/2.0, PVpsi + S1/2.0);
U2 = dt*DPr(r + K1/2.0, Psi + M1/2.0, Vr + N1/2.0, Vphi + O1/2.0,
                Vpsi + P1/2.0, Ppsi + W1/2.0, PVr + Q1/2.0, PVphi + R1/2.0,
                PVpsi + S1/2.0);
W2 = dt*DPpsi(r + K1/2.0, Psi + M1/2.0, Vphi + O1/2.0, Vpsi + P1/2.0,
                        PVphi + R1/2.0, PVpsi + S1/2.0);
J2 = dt*tstar2*DJ((PVr + Q1/2.0)*astar, (PVphi + R1/2.0)*astar,
                        (PVpsi + S1/2.0)*astar);


K3 = dt*Dr(Vr + N2/2.0);
L3 = dt*Dphi(r + K2/2.0, Psi + M2/2.0, Vphi + O2/2.0);
M3 = dt*Dpsi(r + K2/2.0, Vpsi + P2/2.0);
N3 = dt*DVr(r + K2/2.0, Vphi + O2/2.0, Vpsi + P2/2.0, PVr + Q2/2.0);
O3 = dt*DVphi(r + K2/2.0, Psi + M2/2.0, Vr + N2/2.0, Vphi + O2/2.0,
                        Vpsi + P2/2.0, PVphi + R2/2.0);
P3 = dt*DVpsi(r + K2/2.0, Psi + M2/2.0, Vr + N2/2.0, Vphi + O2/2.0,
                        Vpsi + P2/2.0, PVpsi + S2/2.0);
Q3 = dt*DPvr(r + K2/2.0, Vphi + O2/2.0, Vpsi + P2/2.0, Pr + U2/2.0,
                        PVphi + R2/2.0, PVpsi + S2/2.0);
R3 = dt*DPvphi(r + K2/2.0, Psi + M2/2.0, Vr + N2/2.0, Vphi + O2/2.0,
                        Vpsi + P2/2.0, PVr + Q2/2.0, PVphi + R2/2.0,
                        PVpsi + S2/2.0);
S3 = dt*DPvpsi(r + K2/2.0, Psi + M2/2.0, Vr + N2/2.0, Vphi + O2/2.0,
                        Vpsi + P2/2.0, Ppsi + W2/2.0, PVr + Q2/2.0,
                        PVphi + R2/2.0, PVpsi + S2/2.0);
U3 = dt*DPr(r + K2/2.0, Psi + M2/2.0, Vr + N2/2.0, Vphi + O2/2.0,
                Vpsi + P2/2.0,  Ppsi + W2/2.0, PVr + Q2/2.0, PVphi + R2/2.0,
                PVpsi + S2/2.0);
W3 = dt*DPpsi(r + K2/2.0, Psi + M2/2.0, Vphi + O2/2.0, Vpsi + P2/2.0,
                        PVphi + R2/2.0, PVpsi + S2/2.0);
J3 = dt*tstar2*DJ((PVr + Q2/2.0)*astar, (PVphi + R2/2.0)*astar,
                        (PVpsi + S2/2.0)*astar);


K4 = dt*Dr(Vr + N3);
L4 = dt*Dphi(r + K3, Psi + M3, Vphi + O3);
M4 = dt*Dpsi(r + K3, Vpsi + P3);
N4 = dt*DVr(r + K3, Vphi + O3, Vpsi + P3, PVr + Q3);
O4 = dt*DVphi(r + K3, Psi + M3, Vr + N3, Vphi + O3, Vpsi + P3, PVphi + R3);
P4 = dt*DVpsi(r + K3, Psi + M3, Vr + N3, Vphi + O3, Vpsi + P3, PVpsi + S3);
Q4 = dt*DPvr(r + K3, Vphi + O3, Vpsi + P3, Pr + U3, PVphi + R3, PVpsi + S3);
R4 = dt*DPvphi(r + K3, Psi + M3, Vr + N3, Vphi + O3, Vpsi + P3, PVr + Q3,
                        PVphi + R3, PVpsi + S3);
S4 = dt*DPvpsi(r + K3, Psi + M3, Vr + N3, Vphi + O3, Vpsi + P3, Ppsi + W3,
                        PVr + Q3, PVphi + R3, PVpsi + S3);
U4 = dt*DPr(r + K3, Psi + M3, Vr + N3, Vphi + O3, Vpsi + P3, Ppsi + W3,
                PVr + Q3, PVphi + R3, PVpsi + S3);
W4 = dt*DPpsi(r + K3, Psi + M3, Vphi + O3, Vpsi + P3, PVphi + R3,
```

```
                                PVpsi + S3);
        J3 = dt*tstar2*DJ((PVr + Q3)*astar, (PVphi + R3)*astar, (PVpsi + S3)*astar);


        r = r + (1.0/6.0)*(K1 + 2.0*K2 + 2.0*K3 + K4);
        Phi = Phi + (1.0/6.0)*(L1 + 2.0*L2 + 2.0*L3 + L4);
        if (Phi >= 2.0*M_PI)
                Phi = Phi - 2.0*M_PI;
        Psi = Psi + (1.0/6.0)*(M1 + 2.0*M2 + 2.0*M3 + M4);
        Vr = Vr + (1.0/6.0)*(N1 + 2.0*N2 + 2.0*N3 + N4);
        Vphi = Vphi + (1.0/6.0)*(O1 + 2.0*O2 + 2.0*O3 + O4);
        Vpsi = Vpsi + (1.0/6.0)*(P1 + 2.0*P2 + 2.0*P3 + P4);
        PVr = PVr + (1.0/6.0)*(Q1 + 2.0*Q2 + 2.0*Q3 + Q4);
        PVphi = PVphi + (1.0/6.0)*(R1 + 2.0*R2 + 2.0*R3 + R4);
        PVpsi = PVpsi + (1.0/6.0)*(S1 + 2.0*S2 + 2.0*S3 + S4);
        Pr = Pr + (1.0/6.0)*(U1 + 2.0*U2 + 2.0*U3 + U4);
        Ppsi = Ppsi + (1.0/6.0)*(W1 + 2.0*W2 + 2.0*W3 + W4);
        JRK = JRK + (1.0/6.0)*(J1 + 2.0*J2 + 2.0*J3 + J4);
        JEUL = JEUL + dt*tstar2*DJ(PVr*astar, PVphi*astar, PVpsi*astar);

        Hamil = -0.5*(PVr*PVr + PVphi*PVphi + PVpsi*PVpsi);
        Hamil = Hamil + Pr*Vr + Ppsi*Vpsi/r;
        Hamil = Hamil + PVr*((Vphi*Vphi + Vpsi*Vpsi)/r - 1/(r*r) + PVr);
        Hamil = Hamil + PVphi*(-Vr*Vphi/r + Vphi*Vpsi*tan(Psi)/r + PVphi);
        Hamil = Hamil + PVpsi*(-Vr*Vpsi/r - Vphi*Vphi*tan(Psi)/r + PVpsi);

        ar = PVr*astar;
        aphi = PVphi*astar;
        apsi = PVpsi*astar;

        atot = sqrt(ar*ar + aphi*aphi + apsi*apsi);
        Isp = CalcIsp(atot, JEUL);
}

/*************************************************************************/

void Solution (double xsol[10], double ysol[10])

{
double time1, dummy[8];

        InitialConditions(xsol, ysol);
        time1 = 0.0;
        while (t <= Tfinal) {
                t = t + dt;
                RungeKutta();
                time1 = t*tstar;
                if (time1 > 1000)
                        return;
        }
}
```

```
/*************************************************************************/

void Vecset (double xv[10], double yv[10])

{
        Solution (xv, yv);
        yv[1] = r - rfinal;
        yv[2] = Phi - Phifinal;
        yv[3] = Psi - Psifinal;
        yv[4] = Vr - Vrfinal;
        yv[5] = Vphi - Vphifinal;
        yv[6] = Vpsi - Vpsifinal;
}

/*************************************************************************/

void AugMatrix (double A[10][10], double b[10], int n)

{
int i;

        for (i = 1; i <= n; i++)
                A[i][n+1] = b[i];
}

/*************************************************************************/

void PartialPiv (double A[10][10], int n, int k)

{
int row, i, j;
double max, temp;

        max = fabs(A[k][k]);
        row = k;

        for (i = k+1; i <= n; i++) {
                if (fabs(A[i][k]) > max) {
                                row = i;
                                max = fabs(A[i][k]);
                }
        }

        if (row != k) {
                for (j = k; j <= n+1; j++) {
                        temp = A[k][j];
                        A[k][j] = A[row][j];
                        A[row][j] = temp;
                }
        }
}
```

```
/*************************************************************************/

void Gauss (double A[10][10], int n)

{
int i, j, k, k1;
double m;

        for (k = 1; k <= n-1; k++) {
                k1 = k;
                PartialPiv (A, n, k1);
                if (fabs(A[k][k]) > tol) {
                        for (i = k+1; i <= n; i++) {
                                m = (A[i][k])/(A[k][k]);
                                for (j = k+1; j <= n+1; j++)
                                        A[i][j] = A[i][j] - m*(A[k][j]);
                        }
                        for (i = k+1; i <= n; i++)
                                A[i][k] = 0.0;
                }
        }
}

/*************************************************************************/

void BackSubst (double A[10][10], int n, double xg[10])

{
int i,j;
double sum;

        xg[n] = A[n][n+1]/A[n][n];
        for (i = n-1; i >= 1; i--) {
                sum = 0.0;
                for (j = i+1; j <= n; j++)
                        sum = sum + A[i][j]*(xg[j]);
                xg[i] = (A[i][n+1] - sum)/A[i][i];
        }
}

/*************************************************************************/

void GaussianElim (double A[10][10], double b[10], double xg[10], int n)

{
        AugMatrix(A, b, n);
        Gauss(A, n);
        BackSubst(A, n, xg);
}

/*************************************************************************/
```

```
void Steep(double xs[10], double ys[10], int ndim)

{
double xbase[10], ybase[10] , z[10];
double jacob[10][10], jacobT[10][10];
double g0, g1, g2, g3, h1, h2, h3;
double a0, a1, a2, a3, z0, gmin, amin, delx;
int i, j, k, w, q;

        for(w = 1; w < 15; w++) {
                printf("\n");
                printf("Steepest descent:\n");
                printf("\n");
                printf("iteration = %2i\n",w);
                printf("\n");
                Vecset(xs, ybase);
                for(i = 1; i <= ndim; i++)
                        xbase[i] = xs[i];

                /* Calculate Jacobian */

                printf("\n");
                for(j = 1; j <= ndim; j++) {
                        if (j==1)
                                delx = 5;
                        else
                                delx = xs[j]/100.0;
                        xs[j] = xs[j] + delx;
                        printf("Calculating Jacobian: varying initial condition %1i\n",j);
                        Vecset(xs, ys);
                        for(i = 1; i <= ndim; i++)
                                jacob[i][j] = (ys[i] - ybase[i])/delx;
                        xs[j] = xs[j] - delx;
                }
                printf("\n");
                g1 = 0.0;
                for(i = 1; i <= ndim; i++)
                        g1 = g1 + ybase[i] * ybase[i];

                /* Find Transpose of Jacobian */

                for(i = 1; i <= ndim; i++) {
                        for(j = 1; j <= ndim; j++)
                                jacobT[i][j] = jacob[j][i];
                }

                /* Obtain 2*jacobT*ybase (gradient of g) */

                for(j = 1; j <= ndim; j++) {
                        z[j] = 0.0;
                        for(k = 1; k <= ndim; k++)
                                z[j] = z[j] + 2*jacobT[j][k]*ybase[k];
```

```
}

z0 = 0.0;
for(i = 1; i <= ndim; i++)
        z0 = z0 + z[i] * z[i];
if (z0 < tol / 2)  {
        printf("exit Steep\n");
        getchar();
        return;
}
for(i = 1; i <= ndim; i++)
        z[i] = z[i] / z0;
a1 = 0.0;
a3 = 1.0;
for(i = 1; i <= ndim; i++)
        x[i] = xbase[i] - a3 * z[i];
Vecset(xs, ys);
g3 = 0.0;
for(i = 1; i <= ndim; i++)
        g3 = g3 + ys[i]*ys[i];
while ((fabs(g3) - fabs(g1)) > tol / 10) {
        a3 = a3 / 2.0;
        for(i = 1; i <= ndim; i++)
                xs[i] = xbase[i] - a3 * z[i];
        Vecset(xs, ys);
        g3 = 0.0;
        for(i = 1; i <= ndim; i++)
                g3 = g3 + ys[i]*ys[i];
        printf("Snapperhead\n");
        for(i = 1; i <= ndim; i++)
                printf("x[%1i] = %.15lf\n", i, xs[i]);
        printf("g1 = %.15le\n",g1);
        printf("g3 = %.15le\n",g3);
        printf("g3 - g1 = %.15le\n",fabs(g3) - fabs(g1));
        printf("\n");
        if (a3 < 1e-15){
                printf("Steepest did not work\n");
                getchar();
                return;
        }
}
a2 = a3/2.0;
for(i = 1; i <= ndim; i++)
        xs[i] = xbase[i] - a2 * z[i];
Vecset(xs, ys);
g2 = 0.0;
for(i = 1; i <= ndim; i++ )
        g2 = g2 + ys[i] * ys[i];
h1 = (g2 - g1)/a2;
h2 = (g3 - g2)/(a3 - a2);
h3 = (h2 - h1)/a3;
printf("g1 = %.15le\n",g1);
```

```
                printf("g2 = %.15le\n",g2);
                printf("g3 = %.15le\n",g3);
                a0 = 0.5 * (a2 - h1/h3);
                for(i = 1; i <= ndim; i++)
                        xs[i] = xbase[i] - a0 * z[i];
                Vecset(xs, ys);
                g0 = 0.0;
                for(i = 1; i <= ndim; i++)
                        g0 = g0 + ys[i]*ys[i];
                gmin = g0;
                amin = a0;
                if (g1 < gmin) {
                        gmin = g1;
                        amin = a1;
                }
                if (g2 < gmin) {
                        gmin = g2;
                        amin = a2;
                }
                if (g3 < gmin) {
                        gmin = g3;
                        amin = a3;
                }
                for(i = 1; i <= ndim; i++)
                        xs[i] = xbase[i] - amin*z[i];
                printf("gmin = %le\n",gmin);
                for(i = 1; i <= ndim; i++)
                        printf("xbest[%1i] = %.15lf\n",i,xs[i]);
                for(i = 1; i <= 3; i++)
                        printf("\n");
                if (gmin < tol) {
                        printf("exit Steep\n");
                        getchar();
                        return;
                }
        }

        printf("Steepest did not work\n");
        getchar();
        getchar();
}

/*************************************************************************/

void Vecroot(double xv[10], double yv[10], int ndim)

{
double x1[10], ybase[10], xbest[10], xold[10];
double jacob[10][10];
double err, delx, errbest, errlast;
int i, j, k, w, bad;
```

```
for (k = 1; k <= ndim; k++)
        ybase[k] = yv[k];
printf("\n");
printf("ybase at entry:\n");
for(i = 1; i <= ndim; i++)
        printf("ybase[%1i] = %.15lf\n",i,ybase[i]);
errbest = inf;
errlast = inf;
for (k = 1; k <= ndim; k++)
        xbest[k] = xv[k];
for (w = 1; w <= 25; w++) {
        printf("Vecroot\n");
        printf("\n");
        for (i = 1; i <= ndim; i++)
                printf("x[%1i] = %.15lf\n",i,xv[i]);
        Vecset(xv, ybase);
        err = 0.0;
        for (i = 1; i <= ndim; i++)
                err = err + ybase[i]*ybase[i];
        err = sqrt(err);
        printf("Vecroot : %.19lf\n", err);
        printf("\n");
        if (err < errbest) {
                for (j = 1; j <= ndim; j++)
                        xbest[j] = xv[j];
                errbest = err;
                Jfunct = JEUL;
        }
        if (err < errlast)
                bad = 0;
        if (((w > 5) && (err >= errlast)) || (err > 100.0) || (err < 0.00005)) {
                bad = bad + 1;
                if ((bad == 2) || (err > 100.0) || (err < 0.00005)) {
                        for (j = 1; j <= ndim; j++)
                                xv[j] = xbest[j];
                        for (i = 1; i <= ndim; i++)
                                printf("xbest[%1i] = %.15lf\n",i,xbest[i]);
                        printf("Vecroot : %.19lf\n", errbest);
                        getchar();
                        return;
                }
        }

        printf("\n");
        for(j = 1; j <= ndim; j++) {
                if (j==1)
                        delx = 1.0;
                else
                        delx = xv[j]/100.0;
                xv[j] = xv[j] + delx;
                printf("Calculating Jacobian: varying initial condition %1i\n",j);
                Vecset(xv, yv);
```

```
                for(i = 1; i <= ndim; i++)
                        jacob[i][j] = (yv[i] - ybase[i])/delx;
                xv[j] = xv[j] - delx;
        }
        printf("\n");
        for (i = 1; i <= ndim; i++)
                ybase[i] = -ybase[i];

        GaussianElim(jacob, ybase, x1, ndim);

        for (i = 1; i <= ndim; i++)
                xv[i] = xv[i] + x1[i];
    }
    printf("Solution not found in Vecroot\n");
    for (j = 1; j <= ndim; j++)
        xv[j] = xbest[j];
    for (i = 1; i <= ndim; i++)
        printf("xbest[%1i] = %.15lf\n",i,xbest[i]);
    printf("Vecroot : %.19lf\n", errbest);
    getchar();
}

/*************************************************************************/

void Outputdata1()

{
    fprintf(fp, "time{days} radius{AU} Phi{deg} Psi{deg} Vr{km/s} Vphi{km/s} ");
    fprintf(fp, "Vpsi{km/s} ar{m/s2} aphi{m/s2} apsi{m/s2} Pr Psi PVr PVphi");
    fprintf(fp, "PVpsi H Isp{s}\r");
}

/*************************************************************************/

void Outputdata2()

{
    printf("time:%.4lf r:%.6lf Phi:%.6lf Psi:%.6lf ", t*tstar, r, Phi*RAD, Psi*RAD);
    printf("Vr:%.4lf Vphi:%.4lf Vpsi:%.4lf ar:%.6lf ", Vr, Vphi, Vpsi, ar);
    printf("aphi:%.6lf apsi:%.6lf H:%.5lf Isp:%.0lf\n", aphi, apsi, Hamil, Isp);

    fprintf(fp, "%.6lf %.6lf %.6lf %.6lf %.6lf %.6lf %.6lf %.6lf %.6lf %.6lf",
        t*tstar, r, Phi*RAD, Psi*RAD, Vr*vstar, Vphi*vstar,Vpsi*vstar, ar,
        aphi, apsi);
    fprintf(fp, "%.6lf %.6lf %.6lf %.6lf %.6lf %.6lf %.0lf\r", Pr, Ppsi, PVr, PVphi,
        PVpsi, Hamil, Isp);
}


/*************************************************************************/
/****************************** GRAPHICS *********************************/
/*************************************************************************/
```

```
/*********** Define Lighting Model ***********/

float mat[] = {
        AMBIENT, .1, .1, .1,
        DIFFUSE, 0, .369, .165,
        SPECULAR, .5, .5, .5,
        SHININESS, 10,
        LMNULL,
};

static float lm[] = {
        AMBIENT, .1, .1, .1,
        LOCALVIEWER, 1,
        LMNULL,
};

static float lt[] = {
        LCOLOR, 1, 1, 1,
        POSITION, 80, 80, 200, 0,
        LMNULL,
};


/***************** Define Axes ***************/

double axisdata1[6][3] = {
        {-2.0, 0.0, 0.0},
        {2.0, 0.0, 0.0},
        {0.0, -2.0, 0.0},
        {0.0, 2.0, 0.0},
        {0.0, 0.0, -1.3},
        {0.0, 0.0, 1.3}
};

float xaxis1 = 2.0, yaxis1 = 2.0, zaxis1 = 1.3;

double axisdata2[6][3] = {
        {-3.0, 0.0, 0.0},
        {3.0, 0.0, 0.0},
        {0.0, -3.0, 0.0},
        {0.0, 3.0, 0.0},
        {0.0, 0.0, -1.3},
        {0.0, 0.0, 1.3}
};

float xaxis2 = 3.0, yaxis2 = 3.0, zaxis2 = 1.3;


/*************** Define Colors ***************/

long colr[8][3] = {
```

```
        {255, 0, 0},            /* red */
        {0, 255, 0},            /* green */
        {0, 0, 255},            /* blue */
        {255, 0, 255},          /* magenta */
        {255, 255, 255},        /* white */
        {255, 255, 0},          /* yellow */
        {0, 255, 255},          /* cyan */
        {0, 0, 0},              /* black */
};

/*****************************************************************************/

void InitGraphics()

{
long xsize, ysize;

        printf("Enter the desired interval at which the scene is drawn:\n");
        scanf("%d", &interval);

        xsize = getgdesc(GD_XPMAX) - 15;
        ysize = getgdesc(GD_YPMAX) - 40;
        prefposition(0, xsize, 0, ysize);

        if (pickroid == 1)
                winopen("Earth ---> XB (Solar Sail)   [H. Dan. Thomas]");
        else if (pickroid == 2)
                winopen("Earth ---> Orpheus (Solar Sail)   [H. Dan. Thomas]");
        else if (pickroid == 3)
                winopen("Earth ---> McAuliffe (Solar Sail)   [H. Dan. Thomas]");
        else if (pickroid == 4)
                winopen("Earth ---> Seleucus (Solar Sail)   [H. Dan. Thomas]");
        else if (pickroid == 5)
                winopen("Earth ---> Eros (Solar Sail)   [H. Dan. Thomas]");

        mmode(MVIEWING);
        perspective(180, xsize/(float)ysize, 0.1, 400.0);

        if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                lookat(7.0, 7.0, 16.0, 0.0, 0.0, 0.0, 0);
        if ((pickroid == 2) || (pickroid == 5))
                lookat(4.0, 4.0, 10.0, 0.0, 0.0, 0.0, 0);

        zbuffer(TRUE);          /* hidden surface removal */

        RGBmode();
        doublebuffer();
        gconfig();

        linewidth(2);

        lmdef(DEFMATERIAL, 1, 0, mat);
```

```
        lmdef(DEFLIGHT, 1, 0, lt);
        lmdef(DEFLMODEL, 1, 0, lm);
        lmbind(MATERIAL, 1);
        lmbind(LMODEL, 1);
        lmbind(LIGHT0, 1);

        zval = getgdesc(GD_ZMAX);

        rotate(-900, 'x');

        lsetdepth(getgdesc(GD_ZMIN), getgdesc(GD_ZMAX));

        qdevice(LEFTMOUSE);
        qdevice(MIDDLEMOUSE);
        qdevice(RIGHTMOUSE);
        qdevice(ESCKEY);
}

/*************************************************************************/

void drawsphere(double size)

{
double inc = 15;
double theta, dtheta;
double phi, dphi;
double rsin1, rsin2;
double phicos, phisin;
double x, y, z, zorig;
double x1, y1, z1;
float n[3], v[3];
int i, j;

   zorig = 0.0;
   dtheta = M_PI/inc;
   dphi = 2.0*M_PI/inc;

   for (i = 0, theta = 0;  i < inc;  i++, theta += dtheta)  {
        z = cos(theta);
        z1 = cos(theta + dtheta);
        rsin1 = sin(theta);
        rsin2 = sin(theta + dtheta);
        bgntmesh();
          for (j = 0, phi = 0;  j <= inc;  j++, phi += dphi)  {
                if (j == 20)  phi = 0;
                phicos = cos(phi);
                phisin = sin(phi);

                x = rsin1*phicos;
                y = rsin1*phisin;

                n[0] = x; n[1] = y; n[2] = z;
```

```
                n3f(n);

                v[0] = x*size; v[1] = y*size; v[2] = (z + zorig)*size;
                v3f(v);

                x1 = rsin2*phicos;
                y1 = rsin2*phisin;

                v[0] = (x1)*size;
                v[1] = (y1)*size;
                v[2] = (z1 + zorig)*size;
                v3f(v);
            }
        endtmesh();
    }
}

/*************************************************************************/

void Drawtick(double argx, double argy, double argz, double length, Angle angrot)

{
double tick1[3], tick2[3];

        tick1[0] = -length/2.0;
        tick1[1] = 0.0;
        tick1[2] = 0.0;
        tick2[0] = length/2.0;
        tick2[1] = 0.0;
        tick2[2] = 0.0;
        pushmatrix();
                translate(argx, argy, argz);
                rot(angrot, 'z');
                bgnline();
                        v3d(tick1);
                        v3d(tick2);
                endline();
        popmatrix();
}

/*************************************************************************/

void Drawaxes1()

{
        c3i(colr[0]);
        bgnline();
                v3d(axisdata1[0]);
                v3d(axisdata1[1]);
        endline();
        Drawtick(-2.0, 0.0, 0.0, 0.17, 90);
        Drawtick(-1.5, 0.0, 0.0, 0.10, 90);
```

```
        Drawtick(-1.0, 0.0, 0.0, 0.17, 90);
        Drawtick(-0.5, 0.0, 0.0, 0.10, 90);
        Drawtick(0.5, 0.0, 0.0, 0.10, 90);
        Drawtick(1.0, 0.0, 0.0, 0.17, 90);
        Drawtick(1.5, 0.0, 0.0, 0.10, 90);
        Drawtick(2.0, 0.0, 0.0, 0.17, 90);
        c3i(colr[1]);
        cmov(xaxis1 + 0.05, 0, 0);
        charstr("x");

        c3i(colr[0]);
        bgnline();
                v3d(axisdata1[2]);
                v3d(axisdata1[3]);
        endline();
        Drawtick(0.0, -2.0, 0.0, 0.17, 0);
        Drawtick(0.0, -1.5, 0.0, 0.10, 0);
        Drawtick(0.0, -1.0, 0.0, 0.17, 0);
        Drawtick(0.0, -0.5, 0.0, 0.10, 0);
        Drawtick(0.0, 0.5, 0.0, 0.10, 0);
        Drawtick(0.0, 1.0, 0.0, 0.17, 0);
        Drawtick(0.0, 1.5, 0.0, 0.10, 0);
        Drawtick(0.0, 2.0, 0.0, 0.17, 0);
        c3i(colr[1]);
        cmov(0, yaxis1 + 0.05, 0);
        charstr("y");

        c3i(colr[0]);
        bgnline();
                v3d(axisdata1[4]);
                v3d(axisdata1[5]);
        endline();
        Drawtick(0.0, 0.0, -1.0, 0.17, 90);
        Drawtick(0.0, 0.0, -0.5, 0.10, 90);
        Drawtick(0.0, 0.0, 0.5, 0.10, 90);
        Drawtick(0.0, 0.0, 1.0, 0.17, 90);
        Drawtick(0.0, 0.0, -1.0, 0.17, 0);
        Drawtick(0.0, 0.0, -0.5, 0.10, 0);
        Drawtick(0.0, 0.0, 0.5, 0.10, 0);
        Drawtick(0.0, 0.0, 1.0, 0.17, 0);
        c3i(colr[1]);
        cmov(0, 0, zaxis1 + 0.05);
        charstr("z");

}

/*************************************************************************/

void Drawaxes2()

{
        c3i(colr[0]);
```

```
bgnline();
        v3d(axisdata2[0]);
        v3d(axisdata2[1]);
endline();
Drawtick(-3.0, 0.0, 0.0, 0.17, 90);
Drawtick(-2.5, 0.0, 0.0, 0.10, 90);
Drawtick(-2.0, 0.0, 0.0, 0.17, 90);
Drawtick(-1.5, 0.0, 0.0, 0.10, 90);
Drawtick(-1.0, 0.0, 0.0, 0.17, 90);
Drawtick(-0.5, 0.0, 0.0, 0.10, 90);
Drawtick(0.5, 0.0, 0.0, 0.10, 90);
Drawtick(1.0, 0.0, 0.0, 0.17, 90);
Drawtick(1.5, 0.0, 0.0, 0.10, 90);
Drawtick(2.0, 0.0, 0.0, 0.17, 90);
Drawtick(2.5, 0.0, 0.0, 0.10, 90);
Drawtick(3.0, 0.0, 0.0, 0.17, 90);
c3i(colr[1]);
cmov(xaxis2 + 0.05, 0, 0);
charstr("x");

c3i(colr[0]);
bgnline();
        v3d(axisdata2[2]);
        v3d(axisdata2[3]);
endline();
Drawtick(0.0, -3.0, 0.0, 0.17, 0);
Drawtick(0.0, -2.5, 0.0, 0.10, 0);
Drawtick(0.0, -2.0, 0.0, 0.17, 0);
Drawtick(0.0, -1.5, 0.0, 0.10, 0);
Drawtick(0.0, -1.0, 0.0, 0.17, 0);
Drawtick(0.0, -0.5, 0.0, 0.10, 0);
Drawtick(0.0, 0.5, 0.0, 0.10, 0);
Drawtick(0.0, 1.0, 0.0, 0.17, 0);
Drawtick(0.0, 1.5, 0.0, 0.10, 0);
Drawtick(0.0, 2.0, 0.0, 0.17, 0);
Drawtick(0.0, 2.5, 0.0, 0.10, 0);
Drawtick(0.0, 3.0, 0.0, 0.17, 0);
c3i(colr[1]);
cmov(0, yaxis2 + 0.05, 0);
charstr("y");

c3i(colr[0]);
bgnline();
        v3d(axisdata2[4]);
        v3d(axisdata2[5]);
endline();
Drawtick(0.0, 0.0, -1.0, 0.17, 90);
Drawtick(0.0, 0.0, -0.5, 0.10, 90);
Drawtick(0.0, 0.0, 0.5, 0.10, 90);
Drawtick(0.0, 0.0, 1.0, 0.17, 90);
Drawtick(0.0, 0.0, -1.0, 0.17, 0);
Drawtick(0.0, 0.0, -0.5, 0.10, 0);
```

```
        Drawtick(0.0, 0.0, 0.5, 0.10, 0);
        Drawtick(0.0, 0.0, 1.0, 0.17, 0);
        c3i(colr[1]);
        cmov(0, 0, zaxis2 + 0.05);
        charstr("z");

}

/*************************************************************************/

void drawendpt(double argr, double argPhi, double argPsi)

{
double x1, y1, vert1[3], vert2[3];

        vert1[0] = 0.0;
        vert1[1] = 0.0;
        vert1[2] = 0.0;
        vert2[0] = argr*cos(argPsi)*cos(argPhi);
        vert2[1] = argr*cos(argPsi)*sin(argPhi);
        vert2[2] = argr*sin(argPsi);
        bgnline();
                v3d(vert1);
                v3d(vert2);
        endline();
}

/*************************************************************************/

void PlanetOrbit(double body[8], int index)

{
double M, E, Mo, D, initx, inity, initz, vert1[3], vert2[3];
double xold, yold, zold, final;
double Transp[4][4], Ecliptp[4][4], nup[4];

   M = body[2] - body[3];
   if (M < 0)
        M = M + (2*M_PI);
    E = M + body[4]*(sin(M) + body[4]*sin(2*M)/2);
   do{
        Mo = E - body[4]*sin(E);
        D = (M - Mo)/(1 - body[4]*cos(E));
        E = E + D;
   }while (fabs(D) > tol1);
   nup[index] = 2*atan(sqrt((1 + body[4])/(1 - body[4]))*((sin(E/2)/cos(E/2))));

   if (nup[index] < 0)
        nup[index] = nup[index] + 2*M_PI;
   BuildTrans1(body, Transp);
   CoordConvert(body, Transp, nup, index, Ecliptp);
   initx = Ecliptp[index][1];
```

```
    inity = Ecliptp[index][2];
    initz = Ecliptp[index][3];
    final = nup[index] + 2*M_PI;
    do{
        xold = Ecliptp[index][1];
        yold = Ecliptp[index][2];
        zold = Ecliptp[index][3];
        nup[index] = nup[index] + 5.0/RAD;
        CoordConvert (body, Transp, nup, index, Ecliptp);
        vert1[0] = xold;
        vert1[1] = yold;
        vert1[2] = zold;
        vert2[0] = Ecliptp[index][1];
        vert2[1] = Ecliptp[index][2];
        vert2[2] = Ecliptp[index][3];
        bgnline();
          v3d(vert1);
          v3d(vert2);
        endline();
    }while(nup[index] < final);
    Ecliptp[index][1] = initx;
    Ecliptp[index][2] = inity;
    Ecliptp[index][3] = initz;
}

/*************************************************************************/

void GraphicStart()

{
        makeobj(objinit);

        /***************** Create Sun ********************/

                lmcolor(LMC_DIFFUSE);
                RGBcolor(255, 255, 0);
                lmcolor(LMC_COLOR);
                spheresize = 0.25;
                drawsphere(spheresize);

        /**************** Create Orbits *****************/

                if ((pickroid == 2) || (pickroid == 5))
                        Drawaxes1();
                if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                        Drawaxes2();

                c3i(colr[4]);
                PlanetOrbit(departure, 1);
                PlanetOrbit(Marselem, 2);
                PlanetOrbit(target, 3);
```

```
        /******* Create Initial and Final Position ******/

                c3i(colr[1]);
                drawendpt(rInitial, PhiInitial, PsiInitial);
                drawendpt(rfinal, Phifinal, Psifinal);

        closeobj();


/**************** Create earth *****************/

        makeobj(obj1);
                lmcolor(LMC_DIFFUSE);
                RGBcolor(0, 150, 255);
                lmcolor(LMC_COLOR);
                spheresize = 0.1;
                drawsphere(spheresize);
        closeobj();

/***************** Create Mars ******************/

        makeobj(obj2);
                lmcolor(LMC_DIFFUSE);
                RGBcolor(255, 80, 0);
                lmcolor(LMC_COLOR);
                spheresize = 0.05;
                drawsphere(spheresize);
        closeobj();

        if (pickroid == 1) {

        /***************** Create XB ******************/

                makeobj(objAster);
                        lmcolor(LMC_DIFFUSE);
                        RGBcolor(150, 50, 0);
                        lmcolor(LMC_COLOR);
                        spheresize = 0.02;
                        drawsphere(spheresize);
                closeobj();
        }
        else if (pickroid == 2) {

        /*************** Create Orpheus ***************/

                makeobj(objAster);
                        lmcolor(LMC_DIFFUSE);
                        RGBcolor(150, 50, 0);
                        lmcolor(LMC_COLOR);
                        spheresize = 0.02;
                        drawsphere(spheresize);
                closeobj();
```

```
        }
        else if (pickroid == 3) {

        /*************** Create McAuliffe **************/

                makeobj(objAster);
                        lmcolor(LMC_DIFFUSE);
                        RGBcolor(150, 50, 0);
                        lmcolor(LMC_COLOR);
                        spheresize = 0.025;
                        drawsphere(spheresize);
                closeobj();
        }
        else if (pickroid == 4) {

        /*************** Create Seleucus ***************/

                makeobj(objAster);
                        lmcolor(LMC_DIFFUSE);
                        RGBcolor(150, 50, 0);
                        lmcolor(LMC_COLOR);
                        spheresize = 0.025;
                        drawsphere(spheresize);
                closeobj();
        }
        else if (pickroid == 5) {

        /***************** Create Eros *****************/

                makeobj(objAster);
                        lmcolor(LMC_DIFFUSE);
                        RGBcolor(150, 50, 0);
                        lmcolor(LMC_COLOR);
                        spheresize = 0.03;
                        drawsphere(spheresize);
                closeobj();
        }


/************ Begin Trajectory Trace *************/

        pos1[0] = (r*cos(Psi)*cos(Phi));
        pos1[1] = (r*cos(Psi)*sin(Phi));
        pos1[2] = (r*sin(Psi));

        pos2[0] = pos1[0];
        pos2[1] = pos1[1];
        pos2[2] = pos1[2];

        makeobj(objpath);
                c3i(colr[1]);
                bgnline();
```

```
                v3d(pos1);
                v3d(pos2);
            endline();
        closeobj();
}

/*************************************************************************/

void GoGraphics()

{
        c3i(colr[6]);

/************* Draw Sun and Orbits **************/

        callobj(objinit);

/****************** Draw earth *******************/

        PlanetPosition(departure, rdum, Phidum, Psidum, Vrdum, Vphidum,
                Vpsidum, Eclipt, 1);

        pushmatrix();
            translate(Eclipt[1][1], Eclipt[1][2], Eclipt[1][3]);
            callobj(obj1);
        popmatrix();

/****************** Draw Mars *******************/

        PlanetPosition(Marselem, rdum, Phidum, Psidum, Vrdum, Vphidum,
                Vpsidum, Eclipt, 2);

        pushmatrix();
            translate(Eclipt[2][1], Eclipt[2][2], Eclipt[2][3]);
            callobj(obj2);
        popmatrix();

/***************** Draw Asteroid *****************/

        PlanetPosition(target, rdum, Phidum, Psidum, Vrdum, Vphidum,
                        Vpsidum, Eclipt, 3);

        if (pickroid == 1) {

        /******************* Draw XB ********************/

            pushmatrix();
                translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
                callobj(objAster);
                c3i(colr[6]);
                cmov(0, 0, 0.05);
```

```
            charstr("XB");
        popmatrix();
}
else if (pickroid == 2) {

/*************** Draw Orpheus ****************/

        pushmatrix();
            translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
            callobj(objAster);
            c3i(colr[6]);
            cmov(0, 0, 0.05);
            charstr("Orph");
        popmatrix();
}
else if (pickroid == 3) {

/*************** Draw McAuliffe ****************/

        pushmatrix();
            translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
            callobj(objAster);
            c3i(colr[6]);
            cmov(0, 0, 0.05);
            charstr("Mc");
        popmatrix();
}
else if (pickroid == 4) {

/*************** Draw Seleucus ****************/

        pushmatrix();
            translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
            callobj(objAster);
            c3i(colr[6]);
            cmov(0, 0, 0.05);
            charstr("Sel");
        popmatrix();
}
else if (pickroid == 5) {

/*************** Draw Eros ****************/

        pushmatrix();
            translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
            callobj(objAster);
            c3i(colr[6]);
            cmov(0, 0, 0.05);
            charstr("Eros");
        popmatrix();
}
```

```
/*************** Draw Trajectory ***************/

        callobj(objpath);
}


/*****************************************************************************/

void PathUpdate()

{
double pathang, phi, gammapath, anglep;
double length = 0.10;

        pos1[0] = pos2[0];
        pos1[1] = pos2[1];
        pos1[2] = pos2[2];
        pos2[0] = (r*cos(Psi)*cos(Phi));
        pos2[1] = (r*cos(Psi)*sin(Phi));
        pos2[2] = (r*sin(Psi));

        pathang = atan2(Vr,Vphi);
        editobj(objpath);
                c3i(colr[1]);
                bgnline();
                        v3d(pos1);
                        v3d(pos2);
                endline();
                if ((fabs(fmod(time,50) < 0.5) && (time > 49.0))) {
                        anglep = (Phi - pathang)*RAD;
                        Drawtick(pos2[0], pos2[1], pos2[2], 0.10, anglep);
                }
        closeobj();
}

/*****************************************************************************/

void PrintTOF()

{
char Tstr[30];

        c3i(colr[1]);
        sprintf(Tstr, "FLIGHT TIME = %4.2lf days", time);

        if ((pickroid == 2) || (pickroid == 5))
                cmov(-2.0, 1.25, 0.5);
        else if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                cmov(-3.0, 2.25, 0.5);

        charstr(Tstr);
}
```

```
/*************************************************************************/

void Printdates()

{
char datestr1[30];
char datestr2[30];

        c3i(colr[1]);
        sprintf(datestr1, "Departure Date:  %9.6lf", ClaunchT);
        sprintf(datestr2, "Arrival Date:    %9.6lf", CarriveT);
        if ((pickroid == 2) || (pickroid == 5))
        cmov(-1.954, 1.55, 0.8);
        else if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                cmov(-2.954, 2.55, 0.8);
        charstr(datestr1);
        if ((pickroid == 2) || (pickroid == 5))
        cmov(-1.97, 1.45, 0.7);
        else if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                cmov(-2.97, 2.45, 0.7);
        charstr(datestr2);
}


/*************************************************************************/
/************************** MAIN PROGRAM *************************/
/*************************************************************************/

main()
{
double hold1, hold2, hold3, hold4;
int i;


        RAD = 180.0/M_PI;

        tstar = sqrt(1.0/GM)/86400.0;
        vstar = sqrt(GM)*AU;
        astar = GM*AU*1000;
        tstar2= sqrt(1.0/GM);


/************** Initial Guesses ***************/

        x[2] = 1.0;             /* Pr */
        x[3] = 0.004/astar;     /* PVr */
        x[4] = 0.0005/astar;    /* PVphi */
        x[5] = -0.05;           /* Ppsi */
        x[6] = 0.00005/astar;   /* PVpsi */

/*****************************************/
```

```
        CaseEntry ();

        Steep(x, y, 6);
        Vecroot(x, y, 6);

        printf("\n");

        for(i = 1; i <= 5; i++)
                printf("y[%1i] = %.15lf\n",i,y[i]);
        printf("\n");

        gam = sqrt((spmass/2.0)*Jfunct);
        InitialConditions(x, y);
        earth(departure, Jt1);
        Mars(Marselem, Jt1);

        if (pickroid == 1)
                XB(target, Jt2);
        else if (pickroid == 2)
                Orpheus(target, Jt2);
        else if (pickroid == 3)
                McAuliffe(target, Jt2);
        else if (pickroid == 4)
                Seleucus(target, Jt2);
        else if (pickroid == 5)
                Eros(target, Jt3);

        hold1 = -0.5*(PVr*PVr + PVphi*PVphi + PVpsi*PVpsi);
        hold2 = hold1 + Pr*Vr + Ppsi*Vpsi/r;
        hold3 = hold2 + PVr*((Vphi*Vphi + Vpsi*Vpsi)/r - 1/(r*r) + PVr);
        hold4 = hold3 + PVphi*(-Vr*Vphi/r + Vphi*Vpsi*tan(Psi)/r + PVphi);
        Hamil = hold4 + PVpsi*(-Vr*Vpsi/r - Vphi*Vphi*tan(Psi)/r + PVpsi);

        printf("Tfinal = %.5lf\n", Tfinal*tstar);
        getchar();

        if ((fp = fopen(filestr, "w")) == NULL) {
                printf("cannot open file\n");
                exit(1);
        }

        InitGraphics();
        GraphicStart();

        printf("\n");
        Outputdata1();
        Outputdata2();

/*************** Compute and Draw Trajectory *************/
```

```
c3i(colr[6]);

while (t <= Tfinal) {

   count = count + 1;

   if (count == interval) {
        czclear(0x000000, zval);       /* use with zbuffer */
        GoGraphics();
        PrintTOF();
        Printdates();
        swapbuffers();
        Outputdata2();
        count = 0;
   }

   pos1[0] = pos2[0];
   pos1[1] = pos2[1];
   pos1[2] = pos2[2];

   t = t + dt;
   time = t*tstar;
   Jt1 = Jt1 + Jdt;
   Jt2 = Jt2 + Jdt;
   Jt3 = Jt3 + Jdt;

   earth(departure, Jt1);
   Mars(Marselem, Jt1);
   if (pickroid == 1)
        XB(target, Jt2);
   else if (pickroid == 2)
        Orpheus(target, Jt2);
   else if (pickroid == 3)
        McAuliffe(target, Jt2);
   else if (pickroid == 4)
        Seleucus(target, Jt2);
   else if (pickroid == 5)
        Eros(target, Jt3);

   RungeKutta();

   pos2[0] = (r*cos(Psi)*cos(Phi));
   pos2[1] = (r*cos(Psi)*sin(Phi));
   pos2[2] = (r*sin(Psi));

   PathUpdate();

   if (getbutton(ESCKEY)) {
        gexit();
        exit(0);
   }
}
```

```
/****************** Create Final Scene Object ********************/

        time = TOF;

        PathUpdate();

        makeobj(objfinal);
                GoGraphics();
                PrintTOF();
                Printdates();
        closeobj();

/************************ Draw Final Scene ************************/

        czclear(0x000000, zval);
        callobj(objfinal);
        swapbuffers();


        Outputdata2();


        printf("Rfinal: %.5lf Phifinal: %.5lf Psifinal: %.5lf ", rfinal,
                Phifinal*RAD, Psifinal*RAD);
        printf("Vrfinal: %.5lf Vphifinal: %.5lf Vpsifinal: %.5lf\n", Vrfinal,
                Vphifinal, Vpsifinal);
        printf("\n");
        printf("\n");
        printf("\n");
        printf("J = %lf {m^2/s^3} (4th order Runge-Kutta)\n", JRK);
        printf("J = %lf {m^2/s^3} (Euler's Method)\n", JEUL);
        printf("\n");
        printf("\n");
        printf("Launch Date = %.6lf\n", ClaunchT);
        printf("\n");
        fclose(fp);

        while (1) {
                if (getbutton(LEFTMOUSE)) {
                        rotate(10, 'x');
                        czclear(0x000000, zval);
                                callobj(objfinal);
                        swapbuffers();
                }
                if (getbutton(MIDDLEMOUSE)) {
                        rotate(10, 'y');
                        czclear(0x000000, zval);
                                callobj(objfinal);
                        swapbuffers();
                }
                if (getbutton(RIGHTMOUSE)) {
```

```
                rotate(10, 'z');
                czclear(0x000000, zval);
                        callobj(objfinal);
                swapbuffers();
        }
        if (getbutton(ESCKEY)) {
                gexit();
                exit(0);
        }
    }
}
```

## APPENDIX C: Solar Sail Program

```
/*************************************************************************/
/*************************************************************************/
/*      Program:      3D Solar Sail Trajectory (Asteroid)                */
/*                                                                       */
/*      By:     Brian K. Funk, A. Allen McCarley, H. Dan. Thomas         */
/*                                                                       */
/*      Date:  December 17, 1992                                         */
/*                                                                       */
/*      Last Major Revised:  April 11, 1993                              */
/*      Last Minor Revised:  April 11, 1993                              */
/*                                                                       */
/*      Description:  This ANSI C program calculates and draws the optimum solar  */
/*                    sail trajectory between earth and select asteroids (full 3-D)  */
/*                                                                       */
/*************************************************************************/
/*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/device.h>


double AU = 1.4959965E+8;              /* km/au */
double GM = 3.96396415708E-14;         /* au3/s2 */
double dt = 0.005;
double tol = 1e-15;
double tol1 = 1e-10;
double inf = 1e+15;
double RAD;                            /* Conversion from radians to degrees */

double pos1[3];
double pos2[3] = {1.0, 0.0, 0.0};
double spheresize;
double saillength = 0.07, svrt1[3], svrt2[3], svrt3[3], svrt4[3];
long zval;
Object objinit = 11, obj1 = 1, obj2 = 2, objpath = 12, objsail = 13;
Object objAster = 14, objfinal = 15;
int count, interval;

double  Tfinal;        /* The date of arrival at the target planet */
double  tstar;         /* Non-dimensionalization time factor */
double  vstar;         /* Non-dimensionalization velocity factor */
double  Vr;            /* Radial velocity of sail */
double  Vphi;          /* Tangential velocity of sail */
double  Vpsi;          /* Normal velocity of sail */
double  r;             /* Radial position of sail */
double  Phi;           /* Celestial Latitude in 3-D spherical */
```

```
double Psi;          /* Celestial Longitude in 3-D spherical */
double PVr;          /* Lagrange multiplier conjugate to Vr */
double PVphi;        /* Lagrange multiplier conjugate to Vphi */
double PVpsi;        /* Lagrange multiplier conjugate to Vpsi */
double Pr;           /* Lagrange multiplier conjugate to radius */
double Pphi;         /* Lagrange multiplier conjugate to Phi */
double Ppsi;         /* Lagrange multiplier conjugate to Ppsi */
double rInitial;     /* Initial r of solar sail */
double rfinal;       /* Final r of solar sail */
double PhiInitial;   /* Initial Phi of solar sail */
double PsiInitial;   /* Initial Psi of solar sail */
double Phifinal;     /* Final Phi of solar sail */
double Psifinal;     /* Final Psi of solar sail */
double Vrfinal;      /* Final Vr of sail */
double Vphifinal;    /* Final Vphi of sail */
double Vpsifinal;    /* Final Vpsi of sail */
double Theta;        /* The sail thrust cone angle */
double Beta;         /* The sail thrust clock angle */
double Hamil;        /* The Hamiltonian */
double t;            /* non-dimensionalized time */
double time;         /* Time ellapsed in days */
double Jt1;          /* Julian time (in centuries) */
double Jt2;          /* Julian time (in centuries) */
double Jt3;          /* Julian time (in centuries) */
double ClaunchT;     /* Calendar departure date */
double CarriveT;     /* Calendar arrival date */
double Jcentury1[4]; /* Initial Julian time (in centuries) */
double Jcentury2[4]; /* Final Julian time (in centuries) */
double Jdt;          /* Julian stepsize (in centuries) */
double ao;           /* Characteristic acceleration */

double enterao;
double x[10];
double y[10];
double departure[8], target[8], Marselem[8];
double rdum[4], Phidum[4], Psidum[4], Vrdum[4], Vphidum[4], Vpsidum[4],
       Eclipt[4][4];
double JEpoch1 = 2451545.0;
double JEpoch2 = 11.051990;
double JEpoch3 = 6.191986;
double nplan, tau;
int pickroid;
FILE *fp;
char filestr[15];


/*************************************************************************/
/********************** GOVERNING EQUATIONS **********************/
/*************************************************************************/

double CalcBeta (double arg10,double  arg11)
{
```

```
double funcval, Betac, Betas, B1;

        B1 = sqrt(arg10*arg10 + arg11*arg11);
        Betac = arg11/B1;
        Betas = arg10/B1;
        funcval = atan2(Betas, Betac);
        return funcval;
}

/*************************************************************************/

double CalcTheta (double arg9, double arg10, double arg11, double Beta)
{
double funcval, B1;

        B1 = arg10*arg10 + arg11*arg11;
        funcval = atan((sqrt(9*arg9*arg9 + 8*B1) - 3*arg9)/(4*sqrt(B1)));
        return funcval;
}

/*************************************************************************/

double Dr (double arg4)
{
double funcval;

        funcval = arg4;
        return funcval;
}

/*************************************************************************/

double Dphi (double arg1, double arg3, double arg5)
{
double funcval;

        funcval = arg5/(arg1*cos(arg3));
        return funcval;
}

/*************************************************************************/

double Dpsi (double arg1, double arg6)
{
double funcval;

        funcval = arg6/arg1;
        return funcval;
}

/*************************************************************************/
```

```
double DVr (double arg1, double arg5, double arg6, double ao, double Theta)
{
double funcval;

        funcval = (arg5*arg5 + arg6*arg6)/arg1 +
                        (ao*cos(Theta)*cos(Theta)*cos(Theta) - 1)/(arg1*arg1);
        return funcval;
}

/*************************************************************************/

double DVphi (double arg1, double arg3, double arg4, double arg5, double arg6,
                double ao, double Theta, double Beta)
{
double funcval;

        funcval = -(arg4*arg5/arg1) + (arg5*arg6*tan(arg3)/arg1) +
                        (ao*sin(Theta)*cos(Theta)*cos(Theta)*sin(Beta))/(arg1*arg1);
        return funcval;
}

/*************************************************************************/

double DVpsi (double arg1, double arg3, double arg4, double arg5, double arg6,
                double ao, double Theta, double Beta)
{
double funcval;

        funcval = -(arg4*arg6/arg1) - (arg5*arg5*tan(arg3)/arg1) +
                        (ao*sin(Theta)*cos(Theta)*cos(Theta)*cos(Beta))/(arg1*arg1);
        return funcval;
}

/*************************************************************************/

double DPvr (double arg1, double arg5, double arg6, double arg7, double arg10,
                double arg11)
{
double funcval;

        funcval = arg10*arg5/arg1 + arg11*arg6/arg1 - arg7;
        return funcval;
}

/*************************************************************************/

double DPvphi (double arg1, double arg3, double arg4, double arg5, double arg6,
                        double arg9, double arg10, double arg11)
{
double funcval;

        funcval = -(2*arg9*arg5/arg1) + arg4*arg10/arg1 - arg10*arg6*tan(arg3)/arg1 +
```

```
                    2*arg11*arg5*tan(arg3)/arg1;
        return funcval;
}

/*************************************************************************/

double DPvpsi (double arg1, double arg3, double arg4, double arg5, double arg6,
                    double arg8, double arg9, double arg10, double arg11)
{
double funcval;

        funcval = -(2*arg9*arg6/arg1) + arg4*arg11/arg1 - arg10*arg5*tan(arg3)/arg1 -
                    arg8/arg1;
        return funcval;
}

/*************************************************************************/

double DPr (double arg1, double arg3, double arg4, double arg5, double arg6,
            double arg8,double arg9, double arg10, double arg11, double ao,
            double Theta, double Beta)
{
double C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, funcval;

        C1 = arg8*arg6/(arg1*arg1);
        C2 = arg9*(arg5*arg5 + arg6*arg6)/(arg1*arg1);
        C3 = 2*arg9/(arg1*arg1*arg1);
        C4 = 2*ao*arg9*cos(Theta)*cos(Theta)*cos(Theta)/(arg1*arg1*arg1);
        C5 = arg10*arg4*arg5/(arg1*arg1);
        C6 = arg10*arg5*arg6*tan(arg3)/(arg1*arg1);
        C7 = 2*arg10*ao*cos(Theta)*cos(Theta)*sin(Theta)*sin(Beta)/(arg1*arg1*arg1);
        C8 = arg11*arg4*arg6/(arg1*arg1);
        C9 = arg11*arg5*arg5*tan(arg3)/(arg1*arg1);
        C10 = 2*arg11*ao*cos(Theta)*cos(Theta)*sin(Theta)*cos(Beta)/
            (arg1*arg1*arg1);
        funcval = C1 + C2 - C3 + C4 - C5 + C6 + C7 - C8 - C9 + C10;
        return funcval;
}

/*************************************************************************/

double DPpsi (double arg1, double arg3, double arg5, double arg6, double arg10,
            double arg11)
{
double funcval;

        funcval = -(arg10*arg5*arg6/(arg1*cos(arg3)*cos(arg3))) +
                    arg11*arg5*arg5/(arg1*cos(arg3)*cos(arg3));
        return funcval;
}
```

```
/*************************************************************************/
/***************************** PROCEDURES ********************************/
/*************************************************************************/

void earth(double elem[8], double argt)

        /* This procedure determines the orbital elements of        */
        /* earth at a specific time.                                */

/* Epoch = 2000 AD */

{
        elem[1] = 0.0;
        elem[2] = (100.0466449 + 36000.769823*argt + 0.000304*argt*argt)/RAD;
        elem[3] = (102.937348 + 1.719539*argt + 0.000460*argt*argt)/RAD;
        elem[4] = 0.01670862 - 0.00004204*argt;
        elem[5] = 1.00001161;
        elem[6] = 0.0;
        elem[7] = 0.0;
}

/*************************************************************************/

void Mars(double elem[8], double argt)

        /* This procedure determines the orbital elements of        */
        /* Mars at a specific time.                                 */

/* Epoch = 2000 AD */

{
        elem[1] = 0.0;
        elem[2] = (355.433275 + 19141.696475*argt + 0.000311*argt*argt)/RAD;
        elem[3] = (336.060234 + 1.841044*argt + 0.000135*argt*argt)/RAD;
        elem[4] = 0.09340062 + 0.00009048*argt - 0.00000008*argt*argt;
        elem[5] = 1.52371069;
        elem[6] = (1.849726 - 0.000601*argt + 0.000013*argt*argt)/RAD;
        elem[7] = (49.558093 + 0.772096*argt + 0.000016*argt*argt)/RAD;
}

/*************************************************************************/

void XB(double elem[8], double argt)

/* Epoch = 11.051990 */

{
        elem[1] = 16.77969/RAD;
        elem[2] = 56.89879/RAD;
        elem[3] = 91.30613/RAD;
        elem[4] = 0.4466558;
        elem[5] = 1.8355863;
```

```
        elem[6] = 3.87446/RAD;
        elem[7] = 74.52644/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch2;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void Orpheus(double elem[8], double argt)

/* Epoch = 11.051990 */

{
        elem[1] = 301.56931/RAD;
        elem[2] = 197.58723/RAD;
        elem[3] = 130.70829/RAD;
        elem[4] = 0.3226346;
        elem[5] = 1.2093243;
        elem[6] = 2.68775/RAD;
        elem[7] = 189.13898/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch2;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void McAuliffe(double elem[8], double argt)

/* Epoch = 11.051990 */

{
        elem[1] = 15.59117/RAD;
        elem[2] = 283.00658/RAD;
        elem[3] = 122.49259/RAD;
        elem[4] = 0.3694649;
        elem[5] = 1.8787307;
        elem[6] = 4.77743/RAD;
        elem[7] = 106.90142/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch2;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/
```

```
void Seleucus(double elem[8], double argt)

/* Epoch = 11.051990 */

{
        elem[1] = 349.18871/RAD;
        elem[2] = 343.98574/RAD;
        elem[3] = 207.31474/RAD;
        elem[4] = 0.4571202;
        elem[5] = 2.0325801;
        elem[6] = 5.93181/RAD;
        elem[7] = 218.12603/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch2;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void Eros(double elem[8], double argt)

/* Epoch = 6.191986 */

{
        elem[1] = 178.510/RAD;
        elem[2] = 170.451/RAD;
        elem[3] = 123.006/RAD;
        elem[4] = 0.2229;
        elem[5] = 1.4582;
        elem[6] = 10.832/RAD;
        elem[7] = 304.496/RAD;

        nplan = (sqrt(GM/(elem[5]*elem[5]*elem[5])))*86400;
        tau = -elem[2]/nplan + JEpoch3;
        elem[2] = nplan*((argt*36525.0) - tau);
        elem[2] = elem[2] + elem[3];
}

/*************************************************************************/

void InvConvert(double date2, double *argdate)

                        /* This procedure converts any Julian date to the      */
                        /* standard calender date.                             */
                        /* Taken from Astronomical Formulae for Calculators,   */
                        /* Meeus, p. 26.                                       */

{
long Month;             /* The portion of the enterred date that represents the month */
```

```
double Day;             /* The portion of the enterred date that represents the day */
long Year;              /* The protion of the enterred date that represents the year */
long a, b, c, d, e;
double f;

        f = date2 + 0.5;
        a = trunc(f);
        f = f - a;
        if (a >= 2299161) {
                b = (a - 1867216.25)/36524.25;
                a += 1 + b - b/4;
        }
        b = a + 1524;
        c = (b - 122.1)/365.25;
        d = 365.25*c;
        e = (b - d)/30.6001;
        Day = b - d - trunc(30.6001*e) + f;
        if (e <= 13)
                Month = e - 1;
        else
                Month = e - 13;
        if (Month <= 2)
                Year = c - 4715;
        else
                Year = c - 4716;
        *argdate = trunc(Month) + 0.01*rint(Day) + 0.000001*trunc(Year);
}

/*************************************************************************/

void Convert(double *date1)

                        /* This procedure converts any date entered in the      */
                        /* standard calender into a Julian date.                */
                        /* Taken from Astronomical Formulae for Calculators,    */
                        /* Meeus, p. 24.                                        */

{
long Month;             /* The portion of the enterred date that represents the month */
long Day;               /* The portion of the enterred date that represents the day */
long Year;              /* The protion of the enterred date that represents the year */
long a, b;

        Month = trunc(*date1);
        Day = trunc(100*(*date1 - Month));
        Year = rint((1000000.0*(*date1)) - (Month*1000000.0) - (Day*10000.0));
        if (Month <= 2) {
                Year = Year - 1;
                Month = Month + 12;
        }
        a = Year/100;
```

```
        b = 2 - a + a/4;
        *date1 = trunc(365.25*Year) + trunc(30.6001*(Month + 1.0)) + Day +
                        1720994.5 + b;
}

/*************************************************************************/

void GetCaldate(double Juldate1, double *argdate)

{
double Juldate2, Caldate1, Caldate2;

        InvConvert(Juldate1, &Caldate1);
        Juldate2 = Caldate1;
        Convert(&Juldate2);
        *argdate = Caldate1;
        if (Juldate2 > Juldate1)
                *argdate = (*argdate) - 0.01;
}

/*************************************************************************/

void CaseEntry ()

                /* This procedure prompts for the initial guesses for launch      */
                /* date and time of flight.                                       */

{
double JlaunchT, TOF;
int i;

        if ((JEpoch1/1000.0) < 1)
                Convert(&JEpoch1);
        if ((JEpoch2/1000.0) < 1)
                Convert(&JEpoch2);
        if ((JEpoch3/1000.0) < 1)
                Convert(&JEpoch3);

        printf("Enter the number for the desired asteroid:\n");
        printf("       1: XB-1982\n");
        printf("       2: Orpheus\n");
        printf("       3: McAuliffe\n");
        printf("       4: Seleucus\n");
        printf("       5: Eros\n");
        scanf("%i", &pickroid);

        printf("Enter the characteristic sail acceleration in mm/sec^2 (1 or 2):\n");
        scanf("%lf", &enterao);
        ao = enterao*((0.000001/AU)/GM);

        printf("Enter the initial guess for launch date as mm.ddyyyy or as a ");
```

```
        printf("Julian date:\n");
        scanf("%lf", &JlaunchT);
        if ((JlaunchT/1000.0) < 1)
                Convert(&JlaunchT);
        x[1] = JlaunchT;
        printf("Enter the initial guess for time of flight to target in days:\n");
        scanf("%lf", &TOF);
        x[2] = TOF/tstar;

        gets(filestr);
        printf("Enter the desired output file name:\n");
        gets(filestr);
}

/*************************************************************************/

void BuildTrans1(double body[8], double Trans1[4][4])

                        /* This procedure builds the matrix which serves */
                        /* as the transformation matrix between the */
                        /* ecliptic and pericentric coordinate system */

{
double SmOmega;

   SmOmega = body[3] - body[7];
   Trans1[1][1] = cos(body[7])*cos(SmOmega) -
                        sin(body[7])*sin(SmOmega)*cos(body[6]);
   Trans1[1][2] = -cos(body[7])*sin(SmOmega) -
                        sin(body[7])*cos(SmOmega)*cos(body[6]);
   Trans1[1][3] = sin(body[7])*sin(body[6]);
   Trans1[2][1] = sin(body[7])*cos(SmOmega) +
                        cos(body[7])*sin(SmOmega)*cos(body[6]);
   Trans1[2][2] = -sin(body[7])*sin(SmOmega) +
                        cos(body[7])*cos(SmOmega)*cos(body[6]);
   Trans1[2][3] = -cos(body[7])*sin(body[6]);
   Trans1[3][1] = sin(SmOmega)*sin(body[6]);
   Trans1[3][2] = cos(SmOmega)*sin(body[6]);
   Trans1[3][3] = cos(body[6]);
}

/*************************************************************************/

void BuildTrans2 (double Phip[4], double Psip[4], int index, double Trans2[4][4])

                        /* This procedure builds the matrix which serves */
                        /* as the transformation matrix between the */
                        /* spherical and ecliptic coordinate system */

{
        Trans2[1][1] = cos(Phip[index])*cos(Psip[index]);
        Trans2[1][2] = sin(Phip[index])*cos(Psip[index]);
```

```
        Trans2[1][3] = sin(Psip[index]);
        Trans2[2][1] = -sin(Phip[index]);
        Trans2[2][2] = cos(Phip[index]);
        Trans2[2][3] = 0.0;
        Trans2[3][1] = -sin(Psip[index])*cos(Phip[index]);
        Trans2[3][2] = -sin(Psip[index])*sin(Phip[index]);
        Trans2[3][3] = cos(Psip[index]);
}

/*************************************************************************/

void CoordConvert(double body[8], double TransC[4][4], double nuC[4], int i,
                        double EcliptC[4][4])

{
double Peri[4][4];

   Peri[i][1] = (body[5]*(1 - body[4]*body[4])/(1 + body[4]*cos(nuC[i])))*cos(nuC[i]);
   Peri[i][2] = (body[5]*(1 - body[4]*body[4])/(1 + body[4]*cos(nuC[i])))*sin(nuC[i]);
   Peri[i][3] = 0.0;
   EcliptC[i][1] = TransC[1][1]*Peri[i][1] + TransC[1][2]*Peri[i][2] +
                        TransC[1][3]*Peri[i][3];
   EcliptC[i][2] = TransC[2][1]*Peri[i][1] + TransC[2][2]*Peri[i][2] +
                        TransC[2][3]*Peri[i][3];
   EcliptC[i][3] = TransC[3][1]*Peri[i][1] + TransC[3][2]*Peri[i][2] +
                        TransC[3][3]*Peri[i][3];
}

/*************************************************************************/

void PlanetPosition (double body[8], double rp[4], double Phip[4], double Psip[4],
                        double Vrp[4], double Vphip[4], double Vpsip[4],
                        double Ecliptp[4][4], int index)

{
double Ma, Mo, Ea, D, SminAx, SmOmega, VPx, VPy;
double Velp[4], nup[4], Transp1[4][4], Transp2[4][4], EcliptV[4][4];

        Ma = body[2] - body[3];
        if (Ma < 0.0)
                Ma = Ma + (2.0*M_PI);
        Ea = Ma + body[4]*(sin(Ma) + body[4]*sin(2.0*Ma)/2);
        do{
                Mo = Ea - body[4]*sin(Ea);
                D = (Ma - Mo)/(1 - body[4]*cos(Ea));
                Ea = Ea + D;
        }while (fabs(D) > tol1);
        nup[index] = 2.0*atan(sqrt((1 + body[4])/(1 - body[4]))*
                        ((sin(Ea/2.0)/cos(Ea/2.0))));
        if (nup[index] < 0.0)
                nup[index] = nup[index] + 2.0*M_PI;
        SminAx = body[5]*sqrt(1 - body[4]*body[4]);
```

```
        SmOmega = body[3] - body[7];
        BuildTrans1(body, Transp1);
        CoordConvert(body, Transp1, nup, index, Ecliptp);
        Phip[index] = atan(Ecliptp[index][2]/Ecliptp[index][1]);
        rp[index] = body[5]*(1 - body[4]*body[4])/(1 + body[4]*cos(nup[index]));
        VPx = (-body[5]*sin(Ea)/(sqrt(body[5]*body[5]*body[5])*(1 -
                body[4]*cos(Ea))));
        VPy = (SminAx*cos(Ea)/(sqrt(body[5]*body[5]*body[5])*(1 -
                body[4]*cos(Ea))));
        Velp[index] = sqrt(2.0/rp[index] - 1.0/body[5]);
        EcliptV[index][1] = Transp1[1][1]*VPx + Transp1[1][2]*VPy;
        EcliptV[index][2] = Transp1[2][1]*VPx + Transp1[2][2]*VPy;
        EcliptV[index][3] = Transp1[3][1]*VPx + Transp1[3][2]*VPy;
        if ((Ecliptp[index][1] < 0) && (Ecliptp[index][2] > 0) && (Phip[index] < 0))
                Phip[index] = Phip[index] + M_PI;
        if ((Ecliptp[index][1] < 0) && (Ecliptp[index][2] < 0) && (Phip[index] > 0))
                Phip[index] = Phip[index] + M_PI;
        if ((Ecliptp[index][1] > 0) && (Ecliptp[index][2] < 0) && (Phip[index] < 0))
                Phip[index] = Phip[index] + 2.0*M_PI;
        Psip[index] = asin(sin(SmOmega + nup[index])*sin(body[6]));
        BuildTrans2(Phip, Psip, index, Transp2);
        Vrp[index] = Transp2[1][1]*EcliptV[index][1] +
                        Transp2[1][2]*EcliptV[index][2] +
                        Transp2[1][3]*EcliptV[index][3];
        Vphip[index] = Transp2[2][1]*EcliptV[index][1] +
                                Transp2[2][2]*EcliptV[index][2] +
                                Transp2[2][3]*EcliptV[index][3];
        Vpsip[index] = Transp2[3][1]*EcliptV[index][1] +
                                Transp2[3][2]*EcliptV[index][2] +
                                Transp2[3][3]*EcliptV[index][3];
}

/*************************************************************************/

void InitialConditions (double xi[10], double yi[10])

{
double JlaunchT, JarriveT, stepdays;
double ri[4], Vri[4], Vphii[4], Vpsii[4], Phii[4], Psii[4];
double departureI[8], targetI[8], Ecldum[4][4];
int i;

        JlaunchT = xi[1];

        Jcentury1[1] = (JlaunchT - JEpoch1)/(36525.0);
        Jcentury1[2] = JlaunchT/36525.0;
        Jcentury1[3] = JlaunchT/36525.0;

        JarriveT = JlaunchT + xi[2]*tstar;

        Jcentury2[1] = (JarriveT - JEpoch1)/(36525.0);
        Jcentury2[2] = JarriveT/36525.0;
```

```
Jcentury2[3] = JarriveT/36525.0;

GetCaldate(JlaunchT, &ClaunchT);
GetCaldate(JarriveT, &CarriveT);

stepdays = dt*sqrt(1.0/GM)/86400.0;

Jdt = (((JlaunchT + stepdays) - JEpoch1)/36525.0) - Jcentury1[1];

earth(departureI, Jcentury1[1]);
if (pickroid == 1)
        XB(targetI, Jcentury2[2]);
else if (pickroid == 2)
        Orpheus(targetI, Jcentury2[2]);
else if (pickroid == 3)
        McAuliffe(targetI, Jcentury2[2]);
else if (pickroid == 4)
        Seleucus(targetI, Jcentury2[2]);
else if (pickroid == 5)
        Eros(targetI, Jcentury2[3]);

PlanetPosition (departureI, ri, Phii, Psii, Vri, Vphii, Vpsii, Ecldum, 1);
PlanetPosition (targetI, ri, Phii, Psii, Vri, Vphii, Vpsii, Ecldum, 3);
rInitial = ri[1];
rfinal = ri[3];
PhiInitial = Phii[1];
Phifinal = Phii[3];
PsiInitial = Psii[1];
Psifinal = Psii[3];
if (PhiInitial >= 2.0*M_PI)
        PhiInitial = PhiInitial - 2.0*M_PI;
if (Phifinal >= 2.0*M_PI)
        Phifinal = Phifinal - 2.0*M_PI;
Vrfinal = Vri[3];
Vphifinal = Vphii[3];
Vpsifinal = Vpsii[3];
Jt1 = Jcentury1[1];
Jt2 = Jcentury1[2];
Jt3 = Jcentury1[3];

t = 0.0;
time = 0.0;
Vr = Vri[1];
Vphi = Vphii[1];
Vpsi = Vpsii[1];
r = rInitial;
Phi = PhiInitial;
Psi = PsiInitial;
Tfinal = xi[2];
Pr = xi[3];
PVr = xi[4];
PVphi = xi[5];
```

```
        Ppsi = xi[6];
        PVpsi = xi[7];
        Pphi = 0.0;
}

/*************************************************************************/

void RungeKutta ()

{
double K1, K2, K3, K4, L1, L2, L3, L4, M1, M2, M3, M4, N1, N2, N3, N4;
double O1, O2, O3, O4, P1, P2, P3, P4, Q1, Q2, Q3, Q4, R1, R2, R3, R4;
double S1, S2, S3, S4, U1, U2, U3, U4, W1, W2, W3, W4;

        Beta = CalcBeta (PVphi, PVpsi);
        Theta = CalcTheta (PVr, PVphi, PVpsi, Beta);
        K1 = dt*Dr(Vr);
        L1 = dt*Dphi(r, Psi, Vphi);
        M1 = dt*Dpsi(r, Vpsi);
        N1 = dt*DVr(r, Vphi, Vpsi, ao, Theta);
        O1 = dt*DVphi(r, Psi, Vr, Vphi, Vpsi, ao, Theta, Beta);
        P1 = dt*DVpsi(r, Psi, Vr, Vphi, Vpsi, ao, Theta, Beta);
        Q1 = dt*DPvr(r, Vphi, Vpsi, Pr, PVphi, PVpsi);
        R1 = dt*DPvphi(r, Psi, Vr, Vphi, Vpsi, PVr, PVphi, PVpsi);
        S1 = dt*DPvpsi(r, Psi, Vr, Vphi, Vpsi, Ppsi, PVr, PVphi, PVpsi);
        U1 = dt*DPr(r, Psi, Vr, Vphi, Vpsi, Ppsi, PVr, PVphi, PVpsi, ao, Theta, Beta);
        W1 = dt*DPpsi(r, Psi, Vphi, Vpsi, PVphi, PVpsi);

        Beta = CalcBeta (PVphi + R1/2.0, PVpsi + S1/2.0);
        Theta = CalcTheta(PVr + Q1/2.0, PVphi + R1/2.0, PVpsi + S1/2.0, Beta);
        K2 = dt*Dr(Vr + N1/2.0);
        L2 = dt*Dphi(r + K1/2.0, Psi + M1/2.0, Vphi + O1/2.0);
        M2 = dt*Dpsi(r + K1/2.0, Vpsi + P1/2.0);
        N2 = dt*DVr(r + K1/2.0, Vphi + O1/2.0, Vpsi + P1/2.0, ao, Theta);
        O2 = dt*DVphi(r + K1/2.0, Psi + M1/2.0, Vr + N1/2.0, Vphi + O1/2.0, Vpsi +
                        P1/2.0, ao, Theta, Beta);
        P2 = dt*DVpsi(r + K1/2.0, Psi + M1/2.0, Vr + N1/2.0, Vphi + O1/2.0, Vpsi +
                        P1/2.0, ao, Theta, Beta);
        Q2 = dt*DPvr(r + K1/2.0, Vphi + O1/2.0, Vpsi + P1/2.0, Pr + U1/2.0, PVphi +
                R1/2.0, PVpsi + S1/2.0);
        R2 = dt*DPvphi(r + K1/2.0, Psi + M1/2.0, Vr + N1/2.0, Vphi + O1/2.0, Vpsi +
                        P1/2.0, PVr + Q1/2.0, PVphi + R1/2.0, PVpsi + S1/2.0);
        S2 = dt*DPvpsi(r + K1/2.0, Psi + M1/2.0, Vr + N1/2.0, Vphi + O1/2.0, Vpsi +
                        P1/2.0, Ppsi + W1/2.0, PVr + Q1/2.0, PVphi + R1/2.0,
                        PVpsi + S1/2.0);
        U2 = dt*DPr(r + K1/2.0, Psi + M1/2.0, Vr + N1/2.0, Vphi + O1/2.0, Vpsi +
                P1/2.0, Ppsi + W1/2.0, PVr + Q1/2.0, PVphi + R1/2.0, PVpsi +
                S1/2.0, ao, Theta, Beta);
        W2 = dt*DPpsi(r + K1/2.0, Psi + M1/2.0, Vphi + O1/2.0, Vpsi + P1/2.0,
                        PVphi + R1/2.0, PVpsi + S1/2.0);

        Beta = CalcBeta (PVphi + R2/2.0, PVpsi + S2/2.0);
```

```
Theta = CalcTheta(PVr + Q2/2.0, PVphi + R2/2.0, PVpsi + S2/2.0, Beta);
K3 = dt*Dr(Vr + N2/2.0);
L3 = dt*Dphi(r + K2/2.0, Psi + M2/2.0, Vphi + O2/2.0);
M3 = dt*Dpsi(r + K2/2.0, Vpsi + P2/2.0);
N3 = dt*DVr(r + K2/2.0, Vphi + O2/2.0, Vpsi + P2/2.0, ao, Theta);
O3 = dt*DVphi(r + K2/2.0, Psi + M2/2.0, Vr + N2/2.0, Vphi + O2/2.0, Vpsi +
                        P2/2.0, ao, Theta, Beta);
P3 = dt*DVpsi(r + K2/2.0, Psi + M2/2.0, Vr + N2/2.0, Vphi + O2/2.0, Vpsi +
                        P2/2.0, ao, Theta, Beta);
Q3 = dt*DPvr(r + K2/2.0, Vphi + O2/2.0, Vpsi + P2/2.0, Pr + U2/2.0, PVphi +
                R2/2.0, PVpsi + S2/2.0);
R3 = dt*DPvphi(r + K2/2.0, Psi + M2/2.0, Vr + N2/2.0, Vphi + O2/2.0, Vpsi +
                        P2/2.0, PVr + Q2/2.0, PVphi + R2/2.0, PVpsi + S2/2.0);
S3 = dt*DPvpsi(r + K2/2.0, Psi + M2/2.0, Vr + N2/2.0, Vphi + O2/2.0, Vpsi +
                        P2/2.0, Ppsi + W2/2.0, PVr + Q2/2.0, PVphi + R2/2.0,
                        PVpsi + S2/2.0);
U3 = dt*DPr(r + K2/2.0, Psi + M2/2.0, Vr + N2/2.0, Vphi + O2/2.0, Vpsi +
                P2/2.0, Ppsi + W2/2.0, PVr + Q2/2.0, PVphi + R2/2.0, PVpsi +
                S2/2.0, ao, Theta, Beta);
W3 = dt*DPpsi(r + K2/2.0, Psi + M2/2.0, Vphi + O2/2.0, Vpsi + P2/2.0,
                PVphi + R2/2.0, PVpsi + S2/2.0);

Beta = CalcBeta (PVphi + R3, PVpsi + S3);
Theta = CalcTheta(PVr + Q3, PVphi + R3, PVpsi + S3, Beta);
K4 = dt*Dr(Vr + N3);
L4 = dt*Dphi(r + K3, Psi + M3, Vphi + O3);
M4 = dt*Dpsi(r + K3, Vpsi + P3);
N4 = dt*DVr(r + K3, Vphi + O3, Vpsi + P3, ao, Theta);
O4 = dt*DVphi(r + K3, Psi + M3, Vr + N3, Vphi + O3, Vpsi + P3, ao, Theta,
                        Beta);
P4 = dt*DVpsi(r + K3, Psi + M3, Vr + N3, Vphi + O3, Vpsi + P3, ao, Theta,
                        Beta);
Q4 = dt*DPvr(r + K3, Vphi + O3, Vpsi + P3, Pr + U3, PVphi + R3, PVpsi + S3);
R4 = dt*DPvphi(r + K3, Psi + M3, Vr + N3, Vphi + O3, Vpsi + P3, PVr + Q3,
                        PVphi + R3, PVpsi + S3);
S4 = dt*DPvpsi(r + K3, Psi + M3, Vr + N3, Vphi + O3, Vpsi + P3, Ppsi + W3,
                        PVr + Q3, PVphi + R3, PVpsi + S3);
U4 = dt*DPr(r + K3, Psi + M3, Vr + N3, Vphi + O3, Vpsi + P3, Ppsi + W3,
                PVr + Q3, PVphi + R3, PVpsi + S3, ao, Theta, Beta);
W4 = dt*DPpsi(r + K3, Psi + M3, Vphi + O3, Vpsi + P3, PVphi + R3,
                        PVpsi + S3);

r = r + (1.0/6.0)*(K1 + 2.0*K2 + 2.0*K3 + K4);
Phi = Phi + (1.0/6.0)*(L1 + 2.0*L2 + 2.0*L3 + L4);
if (Phi >= 2.0*M_PI)
        Phi = Phi - 2.0*M_PI;
Psi = Psi + (1.0/6.0)*(M1 + 2.0*M2 + 2.0*M3 + M4);
Vr = Vr + (1.0/6.0)*(N1 + 2.0*N2 + 2.0*N3 + N4);
Vphi = Vphi + (1.0/6.0)*(O1 + 2.0*O2 + 2.0*O3 + O4);
Vpsi = Vpsi + (1.0/6.0)*(P1 + 2.0*P2 + 2.0*P3 + P4);
PVr = PVr + (1.0/6.0)*(Q1 + 2.0*Q2 + 2.0*Q3 + Q4);
PVphi = PVphi + (1.0/6.0)*(R1 + 2.0*R2 + 2.0*R3 + R4);
```

```
        PVpsi = PVpsi + (1.0/6.0)*(S1 + 2.0*S2 + 2.0*S3 + S4);
        Pr = Pr + (1.0/6.0)*(U1 + 2.0*U2 + 2.0*U3 + U4);
        Ppsi = Ppsi + (1.0/6.0)*(W1 + 2.0*W2 + 2.0*W3 + W4);
        Beta = CalcBeta (PVphi, PVpsi);
        Theta = CalcTheta(PVr, PVphi, PVpsi, Beta);

        Hamil = Pr*Vr + Ppsi*Vpsi/r;
        Hamil = Hamil + PVr*((Vphi*Vphi + Vpsi*Vpsi)/r - 1/(r*r)) +
                        PVr*(ao*cos(Theta)*cos(Theta)*cos(Theta)/(r*r));
        Hamil = Hamil + PVphi*(-Vr*Vphi/r + Vphi*Vpsi*tan(Psi)/r) +
                        PVphi*(ao*sin(Theta)*cos(Theta)*cos(Theta)*sin(Beta)/(r*r));
        Hamil = Hamil + PVpsi*(-Vr*Vpsi/r - Vphi*Vphi*tan(Psi)/r) +
                        PVpsi*(ao*sin(Theta)*cos(Theta)*cos(Theta)*cos(Beta)/(r*r));
}

/*************************************************************************/

void Solution (double xsol[10], double ysol[10])

{
double time1, dummy[8];

        InitialConditions(xsol, ysol);
        time1 = 0.0;
        while (t <= Tfinal) {
                t = t + dt;
                RungeKutta();
                time1 = t*tstar;
                if (time1 > 1000)
                        return;
        }
}

/*************************************************************************/

void Vecset (double xv[10], double yv[10])

{
        Solution (xv, yv);
        yv[1] = r - rfinal;
        yv[2] = Phi - Phifinal;
        yv[3] = Psi - Psifinal;
        yv[4] = Vr - Vrfinal;
        yv[5] = Vphi - Vphifinal;
        yv[6] = Vpsi - Vpsifinal;
        yv[7] = Hamil - 1;
}

/*************************************************************************/

void AugMatrix (double A[10][10], double b[10], int n)
```

```
{
int i;

        for (i = 1; i <= n; i++)
                A[i][n+1] = b[i];
}

/*************************************************************************/

void PartialPiv (double A[10][10], int n, int k)

{
int row, i, j;
double max, temp;

        max = fabs(A[k][k]);
        row = k;

        for (i = k+1; i <= n; i++) {
                if (fabs(A[i][k]) > max) {
                                row = i;
                                max = fabs(A[i][k]);
                }
        }

        if (row != k) {
                for (j = k; j <= n+1; j++) {
                        temp = A[k][j];
                        A[k][j] = A[row][j];
                        A[row][j] = temp;
                }
        }
}

/*************************************************************************/

void Gauss (double A[10][10], int n)

{
int i, j, k, k1;
double m;

        for (k = 1; k <= n-1; k++) {
                k1 = k;
                PartialPiv (A, n, k1);
                if (fabs(A[k][k]) > tol) {
                        for (i = k+1; i <= n; i++) {
                                m = (A[i][k])/(A[k][k]);
                                for (j = k+1; j <= n+1; j++)
                                        A[i][j] = A[i][j] - m*(A[k][j]);
                        }
                        for (i = k+1; i <= n; i++)
```

```
                        A[i][k] = 0.0;
                }
        }
}

/*************************************************************************/

void BackSubst (double A[10][10], int n, double xg[10])

{
int i,j;
double sum;

        xg[n] = A[n][n+1]/A[n][n];
        for (i = n-1; i >= 1; i--) {
                sum = 0.0;
                for (j = i+1; j <= n; j++)
                        sum = sum + A[i][j]*(xg[j]);
                xg[i] = (A[i][n+1] - sum)/A[i][i];
        }
}

/*************************************************************************/

void GaussianElim (double A[10][10], double b[10], double xg[10], int n)

{
        AugMatrix(A, b, n);
        Gauss(A, n);
        BackSubst(A, n, xg);
}

/*************************************************************************/

void Steep(double xs[10], double ys[10], int ndim)

{
double xbase[10], ybase[10] , z[10];
double jacob[10][10], jacobT[10][10];
double g0, g1, g2, g3, h1, h2, h3;
double a0, a1, a2, a3, z0, gmin, amin, delx;
int i, j, k, w, q;

        for(w = 1; w < 15; w++) {
                printf("\n");
                printf("Steepest descent:\n");
                printf("\n");
                printf("iteration = %2i\n",w);
                printf("\n");
                Vecset(xs, ybase);
                for(i = 1; i <= ndim; i++)
                        xbase[i] = xs[i];
```

```
/* Calculate Jacobian */

printf("\n");
for(j = 1; j <= ndim; j++) {
        if (j==1)
                delx = 5;
        else
                delx = xs[j]/100.0;
        xs[j] = xs[j] + delx;
        printf("Calculating Jacobian: varying initial condition %1i\n",j);
        Vecset(xs, ys);
        for(i = 1; i <= ndim; i++)
                jacob[i][j] = (ys[i] - ybase[i])/delx;
        xs[j] = xs[j] - delx;
}
printf("\n");
g1 = 0.0;
for(i = 1; i <= ndim; i++)
        g1 = g1 + ybase[i] * ybase[i];

/* Find Transpose of Jacobian */

for(i = 1; i <= ndim; i++) {
        for(j = 1; j <= ndim; j++)
                jacobT[i][j] = jacob[j][i];
}

/* Obtain 2*jacobT*ybase (gradient of g) */

for(j = 1; j <= ndim; j++) {
        z[j] = 0.0;
        for(k = 1; k <= ndim; k++)
                z[j] = z[j] + 2*jacobT[j][k]*ybase[k];
}

z0 = 0.0;
for(i = 1; i <= ndim; i++)
        z0 = z0 + z[i] * z[i];
if (z0 < tol / 2)  {
        printf("exit Steep\n");
        getchar();
        return;
}
for(i = 1; i <= ndim; i++)
        z[i] = z[i] / z0;
a1 = 0.0;
a3 = 1.0;
for(i = 1; i <= ndim; i++)
        x[i] = xbase[i] - a3 * z[i];
Vecset(xs, ys);
g3 = 0.0;
```

```
for(i = 1; i <= ndim; i++)
        g3 = g3 + ys[i]*ys[i];
while ((fabs(g3) - fabs(g1)) > tol / 10) {
        a3 = a3 / 2.0;
        for(i = 1; i <= ndim; i++)
                xs[i] = xbase[i] - a3 * z[i];
        Vecset(xs, ys);
        g3 = 0.0;
        for(i = 1; i <= ndim; i++)
                g3 = g3 + ys[i]*ys[i];
        printf("Snapperhead\n");
        for(i = 1; i <= ndim; i++)
                printf("x[%1i] = %.15lf\n", i, xs[i]);
        printf("g1 = %.15le\n",g1);
        printf("g3 = %.15le\n",g3);
        printf("g3 - g1 = %.15le\n",fabs(g3) - fabs(g1));
        printf("\n");
        if (a3 < 1e-15){
                printf("Steepest did not work\n");
                getchar();
                return;
        }
}
a2 = a3/2.0;
for(i = 1; i <= ndim; i++)
        xs[i] = xbase[i] - a2 * z[i];
Vecset(xs, ys);
g2 = 0.0;
for(i = 1; i <= ndim; i++ )
        g2 = g2 + ys[i] * ys[i];
h1 = (g2 - g1)/a2;
h2 = (g3 - g2)/(a3 - a2);
h3 = (h2 - h1)/a3;
printf("g1 = %.15le\n",g1);
printf("g2 = %.15le\n",g2);
printf("g3 = %.15le\n",g3);
a0 = 0.5 * (a2 - h1/h3);
for(i = 1; i <= ndim; i++)
        xs[i] = xbase[i] - a0 * z[i];
Vecset(xs, ys);
g0 = 0.0;
for(i = 1; i <= ndim; i++)
        g0 = g0 + ys[i]*ys[i];
gmin = g0;
amin = a0;
if (g1 < gmin) {
        gmin = g1;
        amin = a1;
}
if (g2 < gmin) {
        gmin = g2;
        amin = a2;
```

```
                }
                if (g3 < gmin) {
                        gmin = g3;
                        amin = a3;
                }
                for(i = 1; i <= ndim; i++)
                        xs[i] = xbase[i] - amin*z[i];
                printf("gmin = %le\n",gmin);
                for(i = 1; i <= ndim; i++)
                        printf("xbest[%1i] = %.15lf\n",i,xs[i]);
                for(i = 1; i <= 3; i++)
                        printf("\n");
                if (gmin < tol) {
                        printf("exit Steep\n");
                        getchar();
                        return;
                }
        }

        printf("Steepest did not work\n");
        getchar();
        getchar();
}

/*************************************************************************/

void Vecroot(double xv[10], double yv[10], int ndim)

{
double x1[10], ybase[10], xbest[10], xold[10];
double jacob[10][10];
double err, delx, errbest, errlast;
int i, j, k, w, bad;

        for (k = 1; k <= ndim; k++)
                ybase[k] = yv[k];
        printf("\n");
        printf("ybase at entry:\n");
        for(i = 1; i <= ndim; i++)
                printf("ybase[%1i] = %.15lf\n",i,ybase[i]);
        errbest = inf;
        errlast = inf;
        for (k = 1; k <= ndim; k++)
                xbest[k] = xv[k];
        for (w = 1; w <= 25; w++) {
                printf("Vecroot\n");
                printf("\n");
                for (i = 1; i <= ndim; i++)
                        printf("x[%1i] = %.15lf\n",i,xv[i]);
                Vecset(xv, ybase);
                err = 0.0;
                for (i = 1; i <= ndim; i++)
```

```
                err = err + ybase[i]*ybase[i];
        err = sqrt(err);
        printf("Vecroot : %.19lf\n", err);
        printf("\n");
        if (err < errbest) {
                for (j = 1; j <= ndim; j++)
                        xbest[j] = xv[j];
                errbest = err;
        }
        if (err < errlast)
                bad = 0;
        if (((w > 5) && (err >= errlast)) || (err > 10.0)) {
                bad = bad + 1;
                if ((bad == 2) || (err > 10.0)) {
                        for (j = 1; j <= ndim; j++)
                                xv[j] = xbest[j];
                        for (i = 1; i <= ndim; i++)
                                printf("xbest[%1i] = %.15lf\n",i,xbest[i]);
                        printf("Vecroot : %.19lf\n", errbest);
                        getchar();
                        return;
                }
        }

        printf("\n");
        for(j = 1; j <= ndim; j++) {
                if (j==1)
                        delx = 1.0;
                else
                        delx = xv[j]/100.0;
                xv[j] = xv[j] + delx;
                printf("Calculating Jacobian: varying initial condition %1i\n",j);
                Vecset(xv, yv);
                for(i = 1; i <= ndim; i++)
                        jacob[i][j] = (yv[i] - ybase[i])/delx;
                xv[j] = xv[j] - delx;
        }
        printf("\n");
        for (i = 1; i <= ndim; i++)
                ybase[i] = -ybase[i];

        GaussianElim(jacob, ybase, x1, ndim);

        for (i = 1; i <= ndim; i++)
                xv[i] = xv[i] + x1[i];
    }
    printf("Solution not found in Vecroot\n");
}

/*************************************************************************/

void Outputdata1()
```

```
{
        fprintf(fp, "time{days} radius{AU} Phi{deg} Psi{deg} Vr{km/s} Vphi{km/s} ");
        fprintf(fp, "Vpsi{km/s} Theta{deg} Beta{deg} Pr Psi PVr PVphi PVpsi H\r");
}

/*************************************************************************/

void Outputdata2()

{
        printf("time:%.4lf r:%.6lf Phi:%.2lf  Psi:%.2lf ", t*tstar, r, Phi*RAD, Psi*RAD) ;
        printf("Vr:%.4lf Vphi:%.4lf Vpsi:%.4lf H:%.5lf\n",Vr*vstar, Vphi*vstar,
                Vpsi*vstar, Hamil);
        fprintf(fp, "%.6lf %.6lf %.6lf %.6lf %.6lf %.6lf %.6lf %.6lf ",  t*tstar, r,
                Phi*RAD, Psi*RAD, Vr*vstar, Vphi*vstar,Vpsi*vstar, Theta*RAD,
                Beta*RAD);
        fprintf(fp, "%.6lf %.6lf %.6lf %.6lf %.6lf %.6lf\r", Pr, Ppsi, PVr, PVphi, PVpsi,
                Hamil);
}


/*************************************************************************/
/***************************** GRAPHICS **********************************/
/*************************************************************************/

/*********** Define Lighting Model ***********/

float mat[] = {
        AMBIENT, .1, .1, .1,
        DIFFUSE, 0, .369, .165,
        SPECULAR, .5, .5, .5,
        SHININESS, 10,
        LMNULL,
};

static float lm[] = {
        AMBIENT, .1, .1, .1,
        LOCALVIEWER, 1,
        LMNULL,
};

static float lt[] = {
        LCOLOR, 1, 1, 1,
        POSITION, 4, 4, 9, 0,
        LMNULL,
};


/***************** Define Axes ***************/

double axisdata1[6][3] = {
```

```
        {-2.0, 0.0, 0.0},
        {2.0, 0.0, 0.0},
        {0.0, -2.0, 0.0},
        {0.0, 2.0, 0.0},
        {0.0, 0.0, -1.3},
        {0.0, 0.0, 1.3}
};

float xaxis1 = 2.0, yaxis1 = 2.0, zaxis1 = 1.3;

double axisdata2[6][3] = {
        {-3.0, 0.0, 0.0},
        {3.0, 0.0, 0.0},
        {0.0, -3.0, 0.0},
        {0.0, 3.0, 0.0},
        {0.0, 0.0, -1.3},
        {0.0, 0.0, 1.3}
};

float xaxis2 = 3.0, yaxis2 = 3.0, zaxis2 = 1.3;


/*************** Define Colors ***************/

long colr[8][3] = {
        {255, 0, 0},            /* red */
        {0, 255, 0},            /* green */
        {0, 0, 255},            /* blue */
        {255, 0, 255},                  /* magenta */
        {255, 255, 255},        /* white */
        {255, 255, 0},                  /* yellow */
        {0, 255, 255},                  /* cyan */
        {0, 0, 0},              /* black */
};

/*********************************************************************/

void InitGraphics()

{
long xsize, ysize;

        printf("Enter the desired interval at which the scene is drawn:\n");
        scanf("%d", &interval);

        xsize = getgdesc(GD_XPMAX) - 15;
        ysize = getgdesc(GD_YPMAX) - 40;
        prefposition(0, xsize, 0, ysize);

        if (pickroid == 1)
                winopen("Earth ---> XB (Solar Sail)   [H. Dan. Thomas]");
        else if (pickroid == 2)
```

```
            winopen("Earth ---> Orpheus (Solar Sail)    [H. Dan. Thomas]");
        else if (pickroid == 3)
            winopen("Earth ---> McAuliffe (Solar Sail)    [H. Dan. Thomas]");
        else if (pickroid == 4)
            winopen("Earth ---> Seleucus (Solar Sail)    [H. Dan. Thomas]");
        else if (pickroid == 5)
            winopen("Earth ---> Eros (Solar Sail)    [H. Dan. Thomas]");

        mmode(MVIEWING);
        perspective(180, xsize/(float)ysize, 0.1, 400.0);

        if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
            lookat(7.0, 7.0, 16.0, 0.0, 0.0, 0.0, 0);
        if ((pickroid == 2) || (pickroid == 5))
            lookat(4.0, 4.0, 10.0, 0.0, 0.0, 0.0, 0);

        zbuffer(TRUE);        /* hidden surface removal */

        RGBmode();
        doublebuffer();
        gconfig();

        linewidth(2);

        lmdef(DEFMATERIAL, 1, 0, mat);
        lmdef(DEFLIGHT, 1, 0, lt);
        lmdef(DEFLMODEL, 1, 0, lm);
        lmbind(MATERIAL, 1);
        lmbind(LMODEL, 1);
        lmbind(LIGHT0, 1);

        zval = getgdesc(GD_ZMAX);

        rotate(-900, 'x');

        lsetdepth(getgdesc(GD_ZMIN), getgdesc(GD_ZMAX));

        qdevice(LEFTMOUSE);
        qdevice(MIDDLEMOUSE);
        qdevice(RIGHTMOUSE);
        qdevice(ESCKEY);
}

/*****************************************************************************/

void drawsphere(double size)

{
double inc = 15;
double theta, dtheta;
double phi, dphi;
double rsin1, rsin2;
```

```
double phicos, phisin;
double x, y, z, zorig;
double x1, y1, z1;
float n[3], v[3];
int i, j;

   zorig = 0.0;
   dtheta = M_PI/inc;
   dphi = 2.0*M_PI/inc;

   for (i = 0, theta = 0; i < inc; i++, theta += dtheta) {
         z = cos(theta);
         z1 = cos(theta + dtheta);
         rsin1 = sin(theta);
         rsin2 = sin(theta + dtheta);
         bgntmesh();
            for (j = 0, phi = 0; j <= inc; j++, phi += dphi) {
                  if (j == 20) phi = 0;
                  phicos = cos(phi);
                  phisin = sin(phi);

                  x = rsin1*phicos;
                  y = rsin1*phisin;

                  n[0] = x; n[1] = y; n[2] = z;
                  n3f(n);

                  v[0] = x*size; v[1] = y*size; v[2] = (z + zorig)*size;
                  v3f(v);

                  x1 = rsin2*phicos;
                  y1 = rsin2*phisin;

                  v[0] = (x1)*size;
                  v[1] = (y1)*size;
                  v[2] = (z1 + zorig)*size;
                  v3f(v);
            }
         endtmesh();
   }
}

/*************************************************************************/

void Drawtick(double argx, double argy, double argz, double length, Angle angrot)

{
double tick1[3], tick2[3];

         tick1[0] = -length/2.0;
         tick1[1] = 0.0;
         tick1[2] = 0.0;
```

```
        tick2[0] = length/2.0;
        tick2[1] = 0.0;
        tick2[2] = 0.0;
        pushmatrix();
                translate(argx, argy, argz);
                rot(angrot, 'z');
                bgnline();
                        v3d(tick1);
                        v3d(tick2);
                endline();
        popmatrix();
}

/*****************************************************************************/

void Drawaxes1()

{
        c3i(colr[0]);
        bgnline();
                v3d(axisdata1[0]);
                v3d(axisdata1[1]);
        endline();
        Drawtick(-2.0, 0.0, 0.0, 0.17, 90);
        Drawtick(-1.5, 0.0, 0.0, 0.10, 90);
        Drawtick(-1.0, 0.0, 0.0, 0.17, 90);
        Drawtick(-0.5, 0.0, 0.0, 0.10, 90);
        Drawtick(0.5, 0.0, 0.0, 0.10, 90);
        Drawtick(1.0, 0.0, 0.0, 0.17, 90);
        Drawtick(1.5, 0.0, 0.0, 0.10, 90);
        Drawtick(2.0, 0.0, 0.0, 0.17, 90);
        c3i(colr[1]);
        cmov(xaxis1 + 0.05, 0, 0);
        charstr("x");

        c3i(colr[0]);
        bgnline();
                v3d(axisdata1[2]);
                v3d(axisdata1[3]);
        endline();
        Drawtick(0.0, -2.0, 0.0, 0.17, 0);
        Drawtick(0.0, -1.5, 0.0, 0.10, 0);
        Drawtick(0.0, -1.0, 0.0, 0.17, 0);
        Drawtick(0.0, -0.5, 0.0, 0.10, 0);
        Drawtick(0.0, 0.5, 0.0, 0.10, 0);
        Drawtick(0.0, 1.0, 0.0, 0.17, 0);
        Drawtick(0.0, 1.5, 0.0, 0.10, 0);
        Drawtick(0.0, 2.0, 0.0, 0.17, 0);
        c3i(colr[1]);
        cmov(0, yaxis1 + 0.05, 0);
        charstr("y");
```

```
        c3i(colr[0]);
        bgnline();
                v3d(axisdata1[4]);
                v3d(axisdata1[5]);
        endline();
        Drawtick(0.0, 0.0, -1.0, 0.17, 90);
        Drawtick(0.0, 0.0, -0.5, 0.10, 90);
        Drawtick(0.0, 0.0, 0.5, 0.10, 90);
        Drawtick(0.0, 0.0, 1.0, 0.17, 90);
        Drawtick(0.0, 0.0, -1.0, 0.17, 0);
        Drawtick(0.0, 0.0, -0.5, 0.10, 0);
        Drawtick(0.0, 0.0, 0.5, 0.10, 0);
        Drawtick(0.0, 0.0, 1.0, 0.17, 0);
        c3i(colr[1]);
        cmov(0, 0, zaxis1 + 0.05);
        charstr("z");

}

/*************************************************************************/

void Drawaxes2()

{
        c3i(colr[0]);
        bgnline();
                v3d(axisdata2[0]);
                v3d(axisdata2[1]);
        endline();
        Drawtick(-3.0, 0.0, 0.0, 0.17, 90);
        Drawtick(-2.5, 0.0, 0.0, 0.10, 90);
        Drawtick(-2.0, 0.0, 0.0, 0.17, 90);
        Drawtick(-1.5, 0.0, 0.0, 0.10, 90);
        Drawtick(-1.0, 0.0, 0.0, 0.17, 90);
        Drawtick(-0.5, 0.0, 0.0, 0.10, 90);
        Drawtick(0.5, 0.0, 0.0, 0.10, 90);
        Drawtick(1.0, 0.0, 0.0, 0.17, 90);
        Drawtick(1.5, 0.0, 0.0, 0.10, 90);
        Drawtick(2.0, 0.0, 0.0, 0.17, 90);
        Drawtick(2.5, 0.0, 0.0, 0.10, 90);
        Drawtick(3.0, 0.0, 0.0, 0.17, 90);
        c3i(colr[1]);
        cmov(xaxis2 + 0.05, 0, 0);
        charstr("x");

        c3i(colr[0]);
        bgnline();
                v3d(axisdata2[2]);
                v3d(axisdata2[3]);
        endline();
        Drawtick(0.0, -3.0, 0.0, 0.17, 0);
        Drawtick(0.0, -2.5, 0.0, 0.10, 0);
```

```
        Drawtick(0.0, -2.0, 0.0, 0.17, 0);
        Drawtick(0.0, -1.5, 0.0, 0.10, 0);
        Drawtick(0.0, -1.0, 0.0, 0.17, 0);
        Drawtick(0.0, -0.5, 0.0, 0.10, 0);
        Drawtick(0.0, 0.5, 0.0, 0.10, 0);
        Drawtick(0.0, 1.0, 0.0, 0.17, 0);
        Drawtick(0.0, 1.5, 0.0, 0.10, 0);
        Drawtick(0.0, 2.0, 0.0, 0.17, 0);
        Drawtick(0.0, 2.5, 0.0, 0.10, 0);
        Drawtick(0.0, 3.0, 0.0, 0.17, 0);
        c3i(colr[1]);
        cmov(0, yaxis2 + 0.05, 0);
        charstr("y");

        c3i(colr[0]);
        bgnline();
                v3d(axisdata2[4]);
                v3d(axisdata2[5]);
        endline();
        Drawtick(0.0, 0.0, -1.0, 0.17, 90);
        Drawtick(0.0, 0.0, -0.5, 0.10, 90);
        Drawtick(0.0, 0.0, 0.5, 0.10, 90);
        Drawtick(0.0, 0.0, 1.0, 0.17, 90);
        Drawtick(0.0, 0.0, -1.0, 0.17, 0);
        Drawtick(0.0, 0.0, -0.5, 0.10, 0);
        Drawtick(0.0, 0.0, 0.5, 0.10, 0);
        Drawtick(0.0, 0.0, 1.0, 0.17, 0);
        c3i(colr[1]);
        cmov(0, 0, zaxis2 + 0.05);
        charstr("z");

}

/*************************************************************************/

void drawendpt(double argr, double argPhi, double argPsi)

{
double x1, y1, vert1[3], vert2[3];

        vert1[0] = 0.0;
        vert1[1] = 0.0;
        vert1[2] = 0.0;
        vert2[0] = argr*cos(argPsi)*cos(argPhi);
        vert2[1] = argr*cos(argPsi)*sin(argPhi);
        vert2[2] = argr*sin(argPsi);
        bgnline();
                v3d(vert1);
                v3d(vert2);
        endline();
}
```

```
/*************************************************************************/

void PlanetOrbit(double body[8], int index)

{
double M, E, Mo, D, initx, inity, initz, vert1[3], vert2[3];
double xold, yold, zold, final;
double Transp[4][4], Ecliptp[4][4], nup[4];

   M = body[2] - body[3];
   if (M < 0)
        M = M + (2*M_PI);
    E = M + body[4]*(sin(M) + body[4]*sin(2*M)/2);
   do{
        Mo = E - body[4]*sin(E);
        D = (M - Mo)/(1 - body[4]*cos(E));
        E = E + D;
   }while (fabs(D) > tol1);
   nup[index] = 2*atan(sqrt((1 + body[4])/(1 - body[4]))*((sin(E/2)/cos(E/2))));

   if (nup[index] < 0)
        nup[index] = nup[index] + 2*M_PI;
   BuildTrans1(body, Transp);
   CoordConvert(body, Transp, nup, index, Ecliptp);
   initx = Ecliptp[index][1];
   inity = Ecliptp[index][2];
   initz = Ecliptp[index][3];
   final = nup[index] + 2*M_PI;
   do{
        xold = Ecliptp[index][1];
        yold = Ecliptp[index][2];
        zold = Ecliptp[index][3];
        nup[index] = nup[index] + 5.0/RAD;
        CoordConvert (body, Transp, nup, index, Ecliptp);
        vert1[0] = xold;
        vert1[1] = yold;
        vert1[2] = zold;
        vert2[0] = Ecliptp[index][1];
        vert2[1] = Ecliptp[index][2];
        vert2[2] = Ecliptp[index][3];
        bgnline();
          v3d(vert1);
          v3d(vert2);
        endline();
   }while(nup[index] < final);
   Ecliptp[index][1] = initx;
   Ecliptp[index][2] = inity;
   Ecliptp[index][3] = initz;
}

/*************************************************************************/
```

```
void GraphicStart()

{
        makeobj(objinit);

        /***************** Create Sun ******************/

                lmcolor(LMC_DIFFUSE);
                RGBcolor(255, 255, 0);
                lmcolor(LMC_COLOR);
                spheresize = 0.25;
                drawsphere(spheresize);

        /**************** Create Orbits *****************/

                if ((pickroid == 2) || (pickroid == 5))
                        Drawaxes1();
                if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                        Drawaxes2();

                c3i(colr[4]);
                PlanetOrbit(departure, 1);
                PlanetOrbit(Marselem, 2);
                PlanetOrbit(target, 3);

        /******* Create Initial and Final Position ******/

                c3i(colr[1]);
                drawendpt(rInitial, PhiInitial, PsiInitial);
                drawendpt(rfinal, Phifinal, Psifinal);

        closeobj();


/**************** Create earth *****************/

        makeobj(obj1);
                lmcolor(LMC_DIFFUSE);
                RGBcolor(0, 150, 255);
                lmcolor(LMC_COLOR);
                spheresize = 0.1;
                drawsphere(spheresize);
        closeobj();

/***************** Create Mars ******************/

        makeobj(obj2);
                lmcolor(LMC_DIFFUSE);
                RGBcolor(255, 80, 0);
                lmcolor(LMC_COLOR);
                spheresize = 0.05;
                drawsphere(spheresize);
```

```
closeobj();

if (pickroid == 1) {

/***************** Create XB ******************/

        makeobj(objAster);
                lmcolor(LMC_DIFFUSE);
                RGBcolor(150, 50, 0);
                lmcolor(LMC_COLOR);
                spheresize = 0.02;
                drawsphere(spheresize);
        closeobj();
}
else if (pickroid == 2) {

/*************** Create Orpheus ***************/

        makeobj(objAster);
                lmcolor(LMC_DIFFUSE);
                RGBcolor(150, 50, 0);
                lmcolor(LMC_COLOR);
                spheresize = 0.02;
                drawsphere(spheresize);
        closeobj();
}
else if (pickroid == 3) {

/*************** Create McAuliffe **************/

        makeobj(objAster);
                lmcolor(LMC_DIFFUSE);
                RGBcolor(150, 50, 0);
                lmcolor(LMC_COLOR);
                spheresize = 0.025;
                drawsphere(spheresize);
        closeobj();
}
else if (pickroid == 4) {

/*************** Create Seleucus ***************/

        makeobj(objAster);
                lmcolor(LMC_DIFFUSE);
                RGBcolor(150, 50, 0);
                lmcolor(LMC_COLOR);
                spheresize = 0.025;
                drawsphere(spheresize);
        closeobj();
}
else if (pickroid == 5) {
```

```
        /***************** Create Eros *****************/

                makeobj(objAster);
                        lmcolor(LMC_DIFFUSE);
                        RGBcolor(150, 50, 0);
                        lmcolor(LMC_COLOR);
                        spheresize = 0.03;
                        drawsphere(spheresize);
                closeobj();
        }


/************ Begin Trjectory Trace *************/

        pos1[0] = (r*cos(Psi)*cos(Phi));
        pos1[1] = (r*cos(Psi)*sin(Phi));
        pos1[2] = (r*sin(Psi));

        pos2[0] = pos1[0];
        pos2[1] = pos1[1];
        pos2[2] = pos1[2];

        makeobj(objpath);
                c3i(colr[1]);
                bgnline();
                        v3d(pos1);
                        v3d(pos2);
                endline();
        closeobj();

/***************** Create Sail ******************/

        svrt1[0] = 0.0;
        svrt1[1] = saillength;
        svrt1[2] = saillength;

        svrt2[0] = 0.0;
        svrt2[1] = saillength;
        svrt2[2] = -saillength;

        svrt3[0] = 0.0;
        svrt3[1] = -saillength;
        svrt3[2] = -saillength;

        svrt4[0] = 0.0;
        svrt4[1] = -saillength;
        svrt4[2] = saillength;

        makeobj(objsail);
                c3i(colr[3]);
                bgnpolygon();
                        v3d(svrt1);
```

```
                v3d(svrt2);
                v3d(svrt3);
                v3d(svrt4);
            endpolygon();
        closeobj();
}

/*************************************************************************/

void GoGraphics()

{
        c3i(colr[6]);

/************** Draw Sun and Orbits **************/

        callobj(objinit);

/****************** Draw earth *******************/

        PlanetPosition(departure, rdum, Phidum, Psidum, Vrdum, Vphidum,
                Vpsidum, Eclipt, 1);

        pushmatrix();
            translate(Eclipt[1][1], Eclipt[1][2], Eclipt[1][3]);
            callobj(obj1);
        popmatrix();

/****************** Draw Mars ******************/

        PlanetPosition(Marselem, rdum, Phidum, Psidum, Vrdum, Vphidum,
                Vpsidum, Eclipt, 2);

        pushmatrix();
            translate(Eclipt[2][1], Eclipt[2][2], Eclipt[2][3]);
            callobj(obj2);
        popmatrix();

/****************** Draw Asteroid ******************/

        PlanetPosition(target, rdum, Phidum, Psidum, Vrdum, Vphidum,
                Vpsidum, Eclipt, 3);

        if (pickroid == 1) {

        /******************* Draw XB *******************/

            pushmatrix();
                translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
                callobj(objAster);
                c3i(colr[6]);
```

```
            cmov(0, 0, 0.05);
            charstr("XB");
        popmatrix();
}
else if (pickroid == 2) {

/*************** Draw Orpheus ****************/

        pushmatrix();
            translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
            callobj(objAster);
            c3i(colr[6]);
            cmov(0, 0, 0.05);
            charstr("Orph");
        popmatrix();
}
else if (pickroid == 3) {

/*************** Draw McAuliffe ****************/

        pushmatrix();
            translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
            callobj(objAster);
            c3i(colr[6]);
            cmov(0, 0, 0.05);
            charstr("Mc");
        popmatrix();
}
else if (pickroid == 4) {

/*************** Draw Seleucus ****************/

        pushmatrix();
            translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
            callobj(objAster);
            c3i(colr[6]);
            cmov(0, 0, 0.05);
            charstr("Sel");
        popmatrix();
}
else if (pickroid == 5) {

/*************** Draw Eros ****************/

        pushmatrix();
            translate(Eclipt[3][1], Eclipt[3][2], Eclipt[3][3]);
            callobj(objAster);
            c3i(colr[6]);
            cmov(0, 0, 0.05);
            charstr("Eros");
        popmatrix();
}
```

```
/*************** Draw Trajectory **************/

        callobj(objpath);

/*************** Draw Spacecraft **************/

        pushmatrix();
                translate(pos2[0], pos2[1], pos2[2]);
                rotate((Phi + Theta)*(180.0/M_PI)*10, 'z');
                callobj(objsail);
        popmatrix();
}

/*****************************************************************************/

void PathUpdate()

{
double pathang, phi, gammapath, anglep;
double length = 0.10;

        pathang = atan2(Vr,Vphi);
        editobj(objpath);
                c3i(colr[1]);
                bgnline();
                        v3d(pos1);
                        v3d(pos2);
                endline();
                if ((fabs(fmod(time,50) < 0.5) && (time > 49.0))) {
                        anglep = (Phi - pathang)*RAD;
                        Drawtick(pos2[0], pos2[1], pos2[2], 0.10, anglep);
                }
        closeobj();
}

/*****************************************************************************/

void PrintTOF()

{
char Tstr[30];

        c3i(colr[1]);
        sprintf(Tstr, "FLIGHT TIME = %4.2lf days", time);

        if ((pickroid == 2) || (pickroid == 5))
                cmov(-2.0, 1.25, 0.5);
        else if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                cmov(-3.0, 2.25, 0.5);

        charstr(Tstr);
```

```
}

/*************************************************************************/

void Printdates()

{
char datestr1[30];
char datestr2[30];

        c3i(colr[1]);
        sprintf(datestr1, "Departure Date:  %9.6lf", ClaunchT);
        sprintf(datestr2, "Arrival Date:    %9.6lf", CarriveT);
        if ((pickroid == 2) || (pickroid == 5))
        cmov(-1.954, 1.55, 0.8);
        else if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                cmov(-2.954, 2.55, 0.8);
        charstr(datestr1);
        if ((pickroid == 2) || (pickroid == 5))
        cmov(-1.97, 1.45, 0.7);
        else if ((pickroid == 1) || (pickroid == 3) || (pickroid == 4))
                cmov(-2.97, 2.45, 0.7);
        charstr(datestr2);
}


/*************************************************************************/
/************************** MAIN PROGRAM *************************/
/*************************************************************************/

main()
{
double hold1, hold2, hold3;
int i;

double dd;
double ddpsi, ddVpsi;

        RAD = 180.0/M_PI;

        tstar = sqrt(1.0/GM)/86400.0;
        vstar = sqrt(GM)*AU;


        CaseEntry();

/*-------------------- Initial Guesses ---------------------*/

        x[3] = 2.450727238135749;  /* Pr   */
        x[4] = 2.576378441602833;  /* PVr  */
        x[5] = 2.743603263944799;  /* PVphi */
        x[6] = 0.501329632898275;  /* Ppsi  */
```

```
        x[7] = 1.111247868106898;  /* PVpsi */

/*-------------------------------------------------------------*/

        Steep(x, y, 7);
        Vecroot(x, y, 7);

        printf("\n");

        for(i = 1; i <= 5; i++)
                printf("y[%1i] = %.15lf\n",i,y[i]);
        printf("\n");

        InitialConditions(x, y);
        earth(departure, Jt1);
        Mars(Marselem, Jt1);

        if (pickroid == 1)
                XB(target, Jt2);
        else if (pickroid == 2)
                Orpheus(target, Jt2);
        else if (pickroid == 3)
                McAuliffe(target, Jt2);
        else if (pickroid == 4)
                Seleucus(target, Jt2);
        else if (pickroid == 5)
                Eros(target, Jt3);

        Beta = CalcBeta(PVphi, PVpsi);
        Theta = CalcTheta(PVr, PVphi, PVpsi, Beta);

        hold1 = Pr*Vr + Ppsi*Vpsi/r;
        hold2 = hold1 + PVr*((Vphi*Vphi + Vpsi*Vpsi)/r - 1/(r*r)) +
                        PVr*(ao*cos(Theta)*cos(Theta)*cos(Theta)/(r*r));
        hold3 = hold2 + PVphi*(-Vr*Vphi/r + Vphi*Vpsi*tan(Psi)/r) +
                        PVphi*(ao*sin(Theta)*cos(Theta)*cos(Theta)*sin(Beta)/(r*r));
        Hamil = hold3 + PVpsi*(-Vr*Vpsi/r - Vphi*Vphi*tan(Psi)/r) +
                        PVpsi*(ao*sin(Theta)*cos(Theta)*cos(Theta)*cos(Beta)/(r*r));

        printf("Tfinal = %.5lf\n", Tfinal*tstar);
        getchar();

        if ((fp = fopen(filestr, "w")) == NULL) {
                printf("cannot open file\n");
                exit(1);
        }

        InitGraphics();
        GraphicStart();

        Outputdata1();
        Outputdata2();
```

```
/*************** Compute and Draw Trajectory **************/

        c3i(colr[6]);

        while (t <= Tfinal) {

           count = count + 1;

           if (count == interval) {
                czclear(0x000000, zval);        /* use with zbuffer */
                GoGraphics();
                PrintTOF();
                Printdates();
                swapbuffers();
                Outputdata2();
                count = 0;
           }

           pos1[0] = pos2[0];
           pos1[1] = pos2[1];
           pos1[2] = pos2[2];

           t = t + dt;
           time = t*tstar;
           Jt1 = Jt1 + Jdt;
           Jt2 = Jt2 + Jdt;
           Jt3 = Jt3 + Jdt;

           earth(departure, Jt1);
           Mars(Marselem, Jt1);
           if (pickroid == 1)
                XB(target, Jt2);
           else if (pickroid == 2)
                Orpheus(target, Jt2);
           else if (pickroid == 3)
                McAuliffe(target, Jt2);
           else if (pickroid == 4)
                Seleucus(target, Jt2);
           else if (pickroid == 5)
                Eros(target, Jt3);

           RungeKutta();

           pos2[0] = (r*cos(Psi)*cos(Phi));
           pos2[1] = (r*cos(Psi)*sin(Phi));
           pos2[2] = (r*sin(Psi));

           PathUpdate();

           if (getbutton(ESCKEY)) {
                gexit();
```

```
            exit(0);
        }
    }

/****************** Create Final Scene Object ********************/

    time = Tfinal*tstar;

    PathUpdate();

    makeobj(objfinal);
        GoGraphics();
        PrintTOF();
        Printdates();
    closeobj();

/************************ Draw Final Scene ************************/

    czclear(0x000000, zval);
    callobj(objfinal);
    swapbuffers();


    Outputdata2();

    printf("Rfinal: %.5lf Phifinal: %.5lf Psifinal: %.5lf Vrfinal: %.5lf ", rfinal,
        Phifinal*RAD, Psifinal*RAD, Vrfinal*vstar);
    printf("Vphifinal: %.5lf Vpsifinal: %.5lf\n"", Vphifinal*vstar, Vpsifinal*vstar);

    fclose(fp);

    while (1) {
        if (getbutton(LEFTMOUSE)) {
            rotate(10, 'x');
            czclear(0x000000, zval);
                callobj(objfinal);
            swapbuffers();
        }
        if (getbutton(MIDDLEMOUSE)) {
            rotate(10, 'y');
            czclear(0x000000, zval);
                callobj(objfinal);
            swapbuffers();
        }
        if (getbutton(RIGHTMOUSE)) {
            rotate(10, 'z');
            czclear(0x000000, zval);
                callobj(objfinal);
            swapbuffers();
        }
        if (getbutton(ESCKEY)) {
            gexit();
```

```
                    exit(0);
                }
            }
        }
```

# VITA

H. Dan. Thomas was born April 15, 1968 in Campsprings, Maryland. At this time his father, Herbert Thomas, served in the United States Air Force at Andrews AFB. After Dan's father retired as a Senior Master Sergeant in 1975, the family moved to Knoxville, Tennessee. After graduating from South Young High School in 1986, Dan attended the University of Tennessee in Knoxville and worked part time as an undergraduate researcher for the Material Science and Engineering department. In May 1991, Dan graduated with a Bachelor of Science degree in Aerospace Engineering with highest honors. Following graduation, Dan pursued a Master's degree in Aerospace Engineering at the University of Tennessee Space Institute in Tullahoma where he worked as a Graduate Research Assistant to Dr. Gary A. Flandro. The primary area of his research was orbital mechanics and interplanetary mission design. Dan received his Master of Science degree in December of 1993.