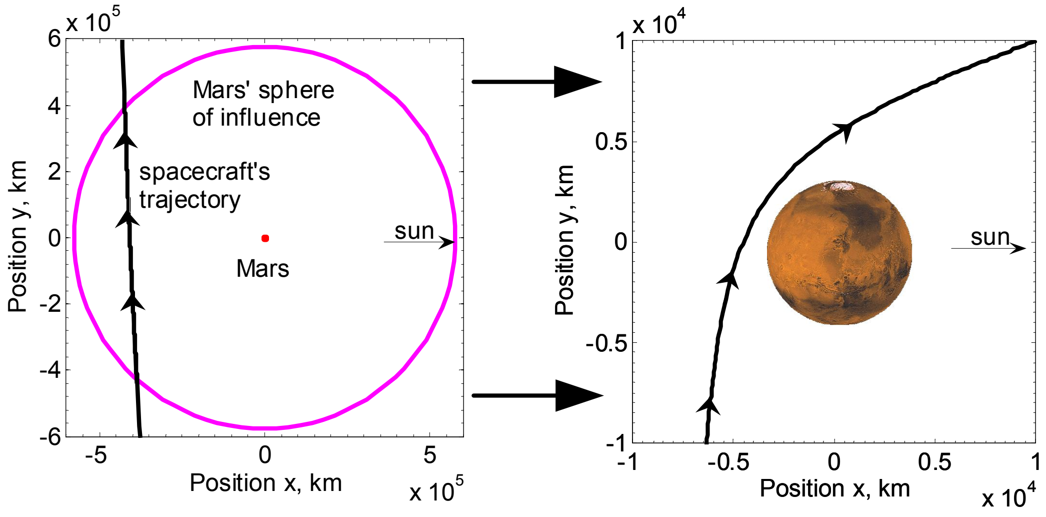
Interplanetary Trajectory Development: Sensitivities of the Restricted Four-Body Problem

Thomas R. Reppert
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

December 08, 2006



Abstract

This report presents an investigation of how sensitivities may be used to attain a particular arrival orbit after a Hohmann transfer from Earth to Mars. The equations of motion are derived from the restricted four-body orbit setup. A spacecraft's motion is integrated with respect to Earth, Mars, and the sun. The advantages of using the C programming language to carry out multiple iterations of the orbit integration are discussed. Subsequently, an introduction to the concept of sensitivity matrices is provided. The state transition matrix, a particular type of sensitivity matrix, is used to relate the initial conditions and final state of a standard spring-mass system. The concept of state transition matrices is then extended to the restricted four-body problem. Three sensitivities are used to map perturbations in the initial Earth-relative departure speed to changes in the arrival orbit: analytical $\frac{\partial d_a}{\partial \nu_0}$, numerical $\frac{\partial d_a}{\partial \nu_0}$, and numerical $\frac{\partial r_3}{\partial \nu_0}$. The methods for implementing each perturbation scheme are discussed, and the convergence performances of the schemes are compared by the computation time required to achieve a particular accuracy. It is concluded that all three methods perform fairly equally during the first 12 seconds of computation time. However, after 12 seconds, the indirect miss distance perturbation techniques plateau at a significant level of periapsis error, whereas the direct periapsis technique continues to drop in error. Therefore, it is concluded that the numerical $\frac{\partial r_3}{\partial \nu_0}$ scheme is the most efficient correction method for altering the arrival periapsis of the restricted four-body Hohmann transfer.

Nomenclature

a	Semi-major axis
$[A]$	Coefficient matrix
c	Chord length
d_a	Perpendicular miss distance of the arrival orbit
d_e	Error in miss distance calculation
d^*	Desired miss distance
ϵ	Perturbation in ν_0 departure speed
i	Iteration counter
$[I]$	Identity matrix
k	Spring constant
λ	Eigenvalue of the coefficient matrix
μ	Gravitational coefficient
$[\Phi(t, t_0)]$	State transition matrix
r_3	Arrival orbit periapsis radius
r_{3_e}	Error in periapsis calculation
$\boldsymbol{r}_s$	Position of the spacecraft
$r_\oplus$	Earth's mean heliocentric orbit radius
$r_♂$	Mars' mean heliocentric orbit radius
$\boldsymbol{v}_s$	Heliocentric velocity of the spacecraft
$\boldsymbol{\nu}_s$	Planet-relative velocity of the spacecraft
ω_n	Natural frequency
$\boldsymbol{x}(t)$	State vector

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The objective of this report is to present the use of sensitivities as a method for developing interplanetary trajectories. The interplanetary trajectory focused on during this paper is a Hohmann transfer from Earth to Mars. A restricted four-body setup is used to describe the motion of the spacecraft. Both Earth's and Mars' orbits are constrained to be circular. The only gravitational forces taken into consideration are those of the sun, Earth, and Mars on the spacecraft. In addition, all orbits are constrained to be planar. As previously shown by Reppert, the restricted four-body orbit setup yields a more accurate representation of the Hohmann transfer than the patched-conic approximation.[4]

The Hohmann transfer was already computed using initial conditions derived from the patched-conic approximation.[5] This paper presents an analysis of how sensitivities may be used to refine the initial propagation of the transfer orbit. These refinements are used to provide desired arrival orbit conditions, such as a chosen Mars periapsis radius. The sensitivities are computed both analytically and numerically. The analysis provides an approximation to how perturbations in initial conditions of the departure orbit affect the arrival orbit geometry. Such information is valuable when trying to attain a particular arrival orbit.

Before assessing sensitivities of the four-body problem, the paper begins with a discussion of coding the four-body integrator in both Matlab and C. The advantages and disadvantages of using each language to perform the four-body integration are given. Thereafter, Chapter 3 provides an analysis of state transition matrices and their relation to sensitivity matrices. The analysis given in Chapter 3 is an introduction to the concept of sensitivities. This introduction provides preparation for the application of sensitivities to the four-body problem in Chapter 4.

The two sensitivities that are used to examine the restricted four-body problem are $\frac{\partial d_a}{\partial \nu_0}$ and $\frac{\partial r_3}{\partial \nu_0}$. The variable ν_0 is the Earth-relative departure speed, whereas d_a and r_3 are the arrival miss distance and periapsis radius, respectively. These sensitivities are computed both analytically and numerically in order to provide a means of attaining a desired arrival orbit geometry. The convergence performances of the perturbation methods are compared by the computation time required to achieve a particular accuracy.

Chapter 2

Converting Code from Matlab to C

This past semester, a set of Matlab code was used to propagate the spacecraft's Hohmann transfer from Earth to Mars. This set of code is displayed in the appendix of the corresponding semester report.[5] Using Matlab's many built-in functions helped make the coding process easier to understand. However, the user-friendly advantages of coding in Matlab came at a cost.

The last objective of this past semester's research was to optimize the Hohmann transfer from Earth to Mars. In order to perform this optimization, multiple integrations of the transfer orbit were required. As is shown in the semester report, seven iterations of the Runge-Kutta integration were necessary to reach the stopping criterion for the Martian periapsis optimization.[5] Computing seven iterations of the orbit took approximately two minutes in Matlab. It was realized that, in order to compute any lengthier orbit optimizations, coding in another language would be much more efficient. The C programming language was chosen as the tool to be used for computing the lengthier optimization problems.

Having a previously developed set of Matlab code allowed for an easier transition to the C code. If a certain function in C did not provide the expected output, its contents could be checked in a line-by-line fashion with the corresponding Matlab code. In this manner, a set of C code was developed to complement the previously developed Matlab code. A listing of the C code is provided in Appendix A.

The only critical difference between the contents of the Matlab code and the C code is the method used to calculate the miss distance d_a of the arrival orbit. The Matlab code utilizes the built-in function `polyfit` to perform a linear regression on the spacecraft's position relative to Mars upon entering the sphere of influence. A total of 10 points are used to calculate the linear fit of the spacecraft's propagated position.

In contrast, the C code uses the value of the spacecraft's velocity vector upon entering Mars' sphere of influence to calculate the miss distance d_a. Figure 2.1 provides an illustration of the geometry. As the spacecraft crosses Mars' sphere of influence, the its velocity is propagated tangentially as follows:

$$y_{\tan} = r_{s_y} + \frac{\nu_{s_y}}{\nu_{s_x}}(x_{\tan} - r_{s_x}) \tag{2.1}$$

where $\boldsymbol{r}_s$ and $\boldsymbol{v}_s$ denote the spacecraft's position and velocity relative to Mars, respectively. However, problems can arise when the value of v_{s_y}/v_{s_x} becomes large. In the case of the

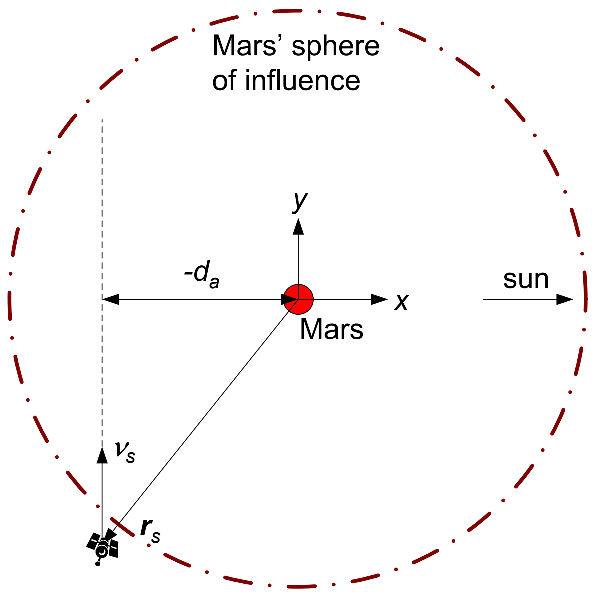


Figure 2.1: Depiction of the arrival orbit geometry used to derive the C miss distance d_a calculation algorithm.

Hohmann transfer, the value of this slope will always be large (as shown in Figure 2.1). Therefore, a better way to create the tangential extension of the velocity vector is to use $x_{\tan}$ as the dependent variable. The velocity vector is then propagated as

$$x_{\tan} = r_{s_x} + \frac{\nu_{s_x}}{\nu_{s_t}}(y_{\tan} - r_{s_y}) \tag{2.2}$$

Equation (2.2) is a more reliable way of calculating the linear extension of the velocity vector. The value of the slope v_{s_x}/v_{s_y} is always less than one for the Hohmann transfer. The `opt_r3.c` source file shown in Appendix A gives the C code used to propagate the spacecraft's velocity vector.

Once the velocity propagation is complete, the perpendicular miss distance d_a may be computed as the minimum distance between the center of Mars and the propagated velocity. This process for calculating the miss distance is more accurate than the process previously used in the Matlab code. The increase in accuracy is due to using the actual velocity vector instead of an approximated velocity vector to create the linear fit.

Most importantly, the C code computes the same optimization that took Matlab two minutes in approximately 10 seconds. This savings in time is critical when attempting to perform sensitive optimization problems. Matlab is valuable for computing and visualizing one transfer orbit, whereas C is valuable for calculating multiple transfer orbits in a computationally intense optimization problem. For a listing of the C code used to perform the sensitivity analyses discussed in Chapter 4, contact the author at `treppert@vt.edu`.

Chapter 3

State Transition Matrices

Sensitivity matrices provide a description of how perturbations of one variable cause changes in the value of another variable. The state transition matrix $[\Phi(t, t_0)]$ is a particular type of sensitivity matrix that describes how perturbations of an initial state vector $\boldsymbol{r}(t_0)$ lead to changes in the final state vector $\boldsymbol{r}(t)$. The state transition matrix can be seen as the sensitivity matrix of the current state to the initial conditions. One of the many applications of this matrix is to calculate how initial trajectory errors evolve over time.[6] This application is later discussed with respect to the interplanetary transfer optimization problem. Chapter 3 presents fundamental state transition matrix theory as it applies to linear, homogeneous dynamic systems. The theory given is later extended to perform optimizations of the restricted four-body problem.

3.1 Linear, Homogeneous Dynamic Systems

Consider the homogeneous vector-matrix differential equation case:

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \dot{\boldsymbol{x}} = [A]\boldsymbol{x}, \qquad \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \tag{3.1}$$

where the coefficient matrix $[A]$ is constant and $\boldsymbol{x}(t)$ is an n-dimensional state vector. Schaub and Junkins show that, using a Taylor series solution, the $\boldsymbol{x}(t)$ state vector may be computed in terms of the initial conditions as

$$\boldsymbol{x}(t) = \left([I] + \sum_{n=1}^{\infty} A^n \frac{(t - t_0)^n}{n!} \right) \boldsymbol{x}(t_0) \tag{3.2}$$

The expression between the large parenthesis is exactly the definition of the matrix exponential function.[6] Therefore, the general solution for $\boldsymbol{x}(t)$ may be written as

$$\boldsymbol{x}(t) = e^{[A](t-t_0)}\boldsymbol{x}(t_0) \tag{3.3}$$

Now consider a classical result that, if $[A]$ has distinct eigenvalues, transforms the computation of the matrix exponential function into a trivial exercise. A transformation to a different n-dimensional state vector $\boldsymbol{\eta}$ may be written as

$$\boldsymbol{x}(t) = [T]\boldsymbol{\eta}(t) \tag{3.4}$$

where $[T]$ is a constant, non-singular $n \times n$ matrix. Substituting Equation (3.4) into Equation (3.1) yields

$$\dot{\boldsymbol{\eta}} = ([T]^{-1}[A][T])\boldsymbol{\eta} \tag{3.5}$$

Schaub and Junkins show that the matrix multiplication of $[T]^{-1}[A][T]$ is diagonal when the columns of the $[T]$ matrix are the eigenvectors of $[A]$.[6] This state transformation converts the originally coupled set of n differential equations into n uncoupled differential equations, given by

$$\dot{\boldsymbol{\eta}}(t) = \lambda_i \boldsymbol{\eta}(t) \tag{3.6}$$

where each λ_i is an eigenvalue of $[A]$. Using the convenient choice of the matrix $[T]$ allows for the computation of the matrix exponential function as follows:

$$e^{[A](t-t_0)} = [T][\text{diag}(e^{\lambda_i(t-t_0)})][T]^{-1} \tag{3.7}$$

where $[\text{diag}(e^{\lambda_i(t-t_0)})]$ is a diagonal matrix with each diagonal entry given using the corresponding eigenvalue λ_i. The solution for the state transition matrix of the linear, homogeneous dynamic system is now applied to the specific case of the spring-mass system.

3.2 Spring-Mass System; Direct Analytical Solution

The equation of motion governing the standard spring-mass system is

$$m\ddot{x} + kx = 0 \tag{3.8}$$

where m denotes the mass and k is the spring constant. Various methods may be used to find the analytical solution for $x(t)$ as

$$x(t) = x_0 \cos(\omega_n t) + \frac{\dot{x}_0}{\omega_n} \sin(\omega_n t) \tag{3.9}$$

where x_0 and $\dot{x_0}$ are the initial position and speed, respectively. Differentiating Equation (3.9) with respect to time yields

$$\dot{x}(t) = -\omega_n x_0 \sin(\omega_n t) + \dot{x}_0 \cos(\omega_n t) \tag{3.10}$$

where $\omega_n = \sqrt{\frac{k}{m}}$ is the natural frequency of the spring-mass system. The values of $x(t)$ and $\dot{x}(t)$ may be combined into the state vector $\boldsymbol{x}(t)$, which is written in terms of the initial conditions as

$$\boldsymbol{x}(t) = \begin{pmatrix} x(t) \\ \dot{x}(t) \end{pmatrix} = \begin{bmatrix} \cos(\omega_n t) & \frac{\sin(\omega_n t)}{\omega_n} \\ -\omega_n \sin(\omega_n t) & \cos(\omega_n t) \end{bmatrix} \begin{pmatrix} x_0 \\ \dot{x}_0 \end{pmatrix} \tag{3.11}$$

Note that Equation (3.11) gives the state vector $\boldsymbol{x}(t)$ in terms of the initial state vector $\boldsymbol{x}_0(t)$ multiplied by a matrix. This matrix is the state transition matrix for the spring-mass system. Therefore, Equation (3.11) may be written equivalently as

$$\boldsymbol{x}(t) = [\Phi(t, t_0)]\boldsymbol{x}(t_0) \tag{3.12}$$

The previous process shows how, for some very simple systems, the state transition matrix may be derived by solving directly for the state vector $\boldsymbol{x}(t)$ in terms of the initial conditions. However, not all systems lend themselves to this method of solution. Thus two more robust methods are used to solve for the state transition matrix of the spring-mass system. The solutions of the two following methods are compared with the previous solution for $[\Phi(t, t_0)]$.

3.3 Spring-Mass System; State Transformation Solution

The state transition matrix of the spring-mass system may also be determined by computing the eigenvalues and eigenvectors of the system's $[A]$ coefficient matrix, as described earlier. In order to do so, the homogeneous vector-matrix differential equation for the spring-mass system is written as follows:

$$\dot{\boldsymbol{X}}(t) = \begin{pmatrix} \dot{x}(t) \\ \ddot{x}(t) \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & 0 \end{bmatrix} \boldsymbol{X}(t) \equiv [A]\boldsymbol{X}(t), \qquad \boldsymbol{X}(t_0) = \boldsymbol{X}_0 \tag{3.13}$$

If the $[A]$ coefficient matrix is determined to have distinct eigenvalues, then the computation of the matrix exponential function for this system is trivial.

Before computing the eigenvalues of the spring-mass system, a brief review of the eigenvalue problem is given. This problem is stated mathematically as

$$[A]\boldsymbol{s} = \lambda \boldsymbol{s} \tag{3.14}$$

where each λ_i is an eigenvalue of the $[A]$ matrix. Equation (3.14) may also be written as follows:

$$([A] - \lambda[I])\boldsymbol{s} = \boldsymbol{0} \tag{3.15}$$

If the matrix $([A] - \lambda[I])$ is invertible, then the only solution is $\boldsymbol{s} = 0$. However, this trivial solution is not permissible for the eigenvalue problem. The matrix $([A] - \lambda[I])$ must have no inverse, and therefore its determinant must be zero:

$$\det([A] - \lambda[I]) = 0 \tag{3.16}$$

Solving this characteristic equation for the spring-mass system yields two distinct eigenvalues, given as $\lambda_{1,2} = i\omega_n, -i\omega_n$. These eigenvalues are used to solve for the two corresponding eigenvectors of the $[A]$ matrix. While the eigenvalues of the system are unique, the eigenvectors for a given $[A]$ matrix are not unique. For instance, if $\boldsymbol{x}$ is an eigenvector of $[A]$, then any constant non-zero multiple of $\boldsymbol{x}$ is also an eigenvector of $[A]$.[2] The two eigenvectors of the standard spring-mass system are determined to be

$$\boldsymbol{s}_1 = \begin{pmatrix} 1 \\ i\omega_n \end{pmatrix}, \qquad \boldsymbol{s}_2 = \begin{pmatrix} 1 \\ -i\omega_n \end{pmatrix} \tag{3.17}$$

As previously shown, because the $[A]$ matrix for the spring-mass system has distinct eigenvalues, the matrix exponential function for the system may be computed as

$$e^{[A](t-t_0)} = [T][\mathrm{diag}(e^{\lambda_i(t-t_0)})][T]^{-1} \tag{3.18}$$

where, in this case, the two columns of $[T]$ are the eigenvectors $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$. Thus the state transition matrix for the spring-mass system is given by

$$[\Phi(t,t_0)] = e^{[A](t-t_0)} = \begin{bmatrix} 1 & 1 \\ i\omega_n & -i\omega_n \end{bmatrix} \begin{bmatrix} e^{i\omega_n(t-t_0)} & 0 \\ 0 & e^{-i\omega_n(t-t_0)} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ i\omega_n & -i\omega_n \end{bmatrix}^{-1} \tag{3.19}$$

where the inverse $[T]^{-1}$ is determined to be

$$[T]^{-1} = \begin{bmatrix} 1 & 1 \\ i\omega_n & -i\omega_n \end{bmatrix}^{-1} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2i\omega_n} \\ \frac{1}{2} & \frac{-1}{2i\omega_n} \end{bmatrix} \tag{3.20}$$

The reduced state transition matrix for the spring-mass system is calculated as

$$[\Phi(t,t_0)] = \begin{bmatrix} \frac{1}{2}(e^{i\omega_n(t-t_0)} + e^{-i\omega_n(t-t_0)}) & \frac{1}{2i\omega_n}(e^{i\omega_n(t-t_0)} - e^{-i\omega_n(t-t_0)}) \\ \frac{i\omega_n}{2}(e^{i\omega_n(t-t_0)} - e^{-i\omega_n(t-t_0)}) & \frac{1}{2}(e^{i\omega_n(t-t_0)} + e^{-i\omega_n(t-t_0)}) \end{bmatrix} \tag{3.21}$$

Euler's equation may be used to transform the matrix entries of Equation (3.21) into sine and cosine format.[1] In this manner, the state transition matrix is written as

$$[\Phi(t,t_0)] = \begin{bmatrix} \cos(\omega_n(t-t_0)) & \frac{\sin(\omega_n(t-t_0))}{\omega_n} \\ -\omega_n \sin(\omega_n(t-t_0)) & \cos(\omega_n(t-t_0)) \end{bmatrix} \tag{3.22}$$

Note that the state transition matrix of Equation (3.22) matches the matrix of Equation (3.11) exactly. While this second method used to solve for $[\Phi(t,t_0)]$ is more time-consuming, it can be applied to systems for which $\dot{\boldsymbol{X}}(t)$ cannot be determined as a function of $\boldsymbol{X}(t)$ directly.

3.4 Spring-Mass System; Matrix Exponential Solution

One additional way to solve for the state transition matrix of the spring-mass system is to use the definition of the matrix exponential function to solve for each entry in series format. The problem is given as follows:

$$e^{[A](t-t_0)} = \left([I] + \sum_{n=1}^{\infty} A^n \frac{(t-t_0)^n}{n!}\right), \qquad [A] = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & 0 \end{bmatrix} \tag{3.23}$$

Carrying out the series for the (1,1) entry of the state transition matrix yields

$$[\Phi(t,t_0)](1,1) = 1 + 0 - \frac{(\omega_n(t-t_0))^2}{2!} + 0 + \frac{(\omega_n(t-t_0))^4}{4!} + \dots \tag{3.24}$$

The series representation of the cosine function is given as[7]

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \dots \tag{3.25}$$

Thus, by comparing Equations (3.24) and (3.25), it can be seen that, in the limit, the (1,1) entry of the state transition matrix may be written as

$$[\Phi(t,t_0)](1,1) = \cos(\omega_n(t-t_0)) \tag{3.26}$$

which matches the (1,1) entries of the matrices given in Equations (3.11) and (3.22). In a similar fashion, the other three entries of $[\Phi(t,t_0)]$ may be derived from their series representations. Their derivations using this method will also yield values that match the entries given in Equations (3.11) and (3.22). Computing the state transition matrix by carrying out the matrix exponential function serves as a good exercise in working with series representations of functions. It also supports the findings of the previous two methods used to solve for $[\Phi(t,t_0)]$.

3.5 Using State Transition Matrices to Apply Corrections to Initial Conditions

The state transition matrix provides a way to map perturbations in the initial state into changes in the final state. The mathematical representation of this mapping is repeated here:

$$\boldsymbol{x}(t) = [\Phi(t, t_0)]\boldsymbol{x}(t_0) \tag{3.27}$$

Note that the state transition matrix may be partitioned into smaller submatrices as follows:

$$[\Phi] = \begin{bmatrix} \Phi_{11} & \Phi_{12} \\ \Phi_{21} & \Phi_{22} \end{bmatrix} \tag{3.28}$$

Let the state vector $\boldsymbol{x}$ be defined as

$$\boldsymbol{x} = \begin{pmatrix} \boldsymbol{r} \\ \dot{\boldsymbol{r}} \end{pmatrix} \tag{3.29}$$

where $\dot{\boldsymbol{r}}$ is the inertial derivative of the position vector $\boldsymbol{r}$. Using these definitions of the state transition matrix and the state vector, Equation (3.27) may be rewritten as

$$\begin{pmatrix} \boldsymbol{r} \\ \dot{\boldsymbol{r}} \end{pmatrix} = \begin{bmatrix} \Phi_{11} & \Phi_{12} \\ \Phi_{21} & \Phi_{22} \end{bmatrix} \begin{pmatrix} \boldsymbol{r}_0 \\ \dot{\boldsymbol{r}}_0 \end{pmatrix} \tag{3.30}$$

where the initial state vector is composed of $\boldsymbol{r}_0$ and $\dot{\boldsymbol{r}}_0$. Suppose that, for a particular system, a change in the position vector $\boldsymbol{r}$ at time t is desired. This problem is applied to the restricted four-body orbit setup in Chapter 4. In order to change $\boldsymbol{r}$, perturbations in either $\boldsymbol{r}_0$ or $\dot{\boldsymbol{r}}_0$ may be used.

Suppose that changing the initial velocity vector is more suitable for the given system. The effect of perturbing the initial velocity $\dot{\boldsymbol{r}}_0$ by some value $\delta\dot{\boldsymbol{r}}_0$ on the final position $\boldsymbol{r}$ is calculated using the state transition matrix as follows:

$$\delta\boldsymbol{r} = [\Phi_{12}]\delta\dot{\boldsymbol{r}}_0 \tag{3.31}$$

Each of the four submatrices that comprise the general state transition matrix may be used to relate changes in initial conditions to changes in the final state vector. The question of which submatrix to use depends upon the nature of the particular problem. If changes in initial position are more feasible than changes in initial velocity, then $[\Phi_{11}]$ and $[\Phi_{21}]$ are the two submatrices that should be considered for perturbations in the final state. Chapter 4 includes an examination of how to apply this sensitivity analysis to the restricted four-body problem.

Chapter 4

Sensitivity Analysis of the Restricted Four-Body Problem

The previous analysis of state transition matrices offers an introduction to the concept of sensitivities. As is stated in Chapter 3, the state transition matrix is the sensitivity matrix of the current state with respect to the initial conditions. It maps perturbations in the initial conditions into changes in the final state at time t. This concept of relating changes in initial conditions to changes in the final state is now extended to the restricted four-body problem.

Chapter 4 uses a Hohmann transfer from Earth to Mars as the restricted four-body system. Reppert provides a detailed explanation of the Hohmann transfer orbit setup.[4] The orbits of both Earth and Mars are restricted to be circular. In addition, the only gravitational effects considered are the effects of the sun, Earth, and Mars on the spacecraft. All motion is constrained to be planar.

This chapter provides an analysis of how changes in the transfer orbit's initial conditions affect certain parameters of the Martian arrival orbit. These sensitivities of the Hohmann transfer are calculated both analytically and numerically. The calculations of the sensitivies provide an efficient means of estimating the initial conditions required to achieve a desired arrival orbit geometry.

4.1 Analytical Solution for $\frac{\partial d_a}{\partial \nu_0}$

This section presents the development of an analytical approximation to the sensitivity of the arrival miss distance d_a with respect to the initial Earth-relative speed ν_0. In other words, perturbations in the velocity ν_0 are mapped into changes in the miss distance d_a. The process for deriving the $\frac{\partial d_a}{\partial \nu_0}$ sensitivity is very methodical. A series of partial derivates are used to work backward from the miss distance to the initial Earth-relative speed. Then the chain rule for differentiation is used to relate each of the separate partial derivatives. The partial derivatives used to calculate the sensitivity analytically are derived from patched-conic equations. Therefore, the analytical solution for the sensitivity $\frac{\partial d_a}{\partial \nu_0}$ derived in this section is an approximation to the true non-Keplerian sensitivity. Nonetheless, as is later shown, the approximate sensitivity provides a sufficient means of calculating the necessary initial conditions for a desired arrival orbit.

4.1.1 Derivation of the Analytical Solution

The first required partial derivative is that of the miss distance d_a with respect to the Hohmann semi-major axis a_H. For the Hohmann transfer from Earth to Mars, the chord c may be approximated as

$$c \approx 2a_H - d_a \tag{4.1}$$

assuming that the spacecraft's Mars-relative arrival velocity $\boldsymbol{\nu}_2$ is parallel to Mars' velocity vector $\boldsymbol{v}_♂$. Figure 4.1 provides an illustration of the required arrival orbit geometry. Note

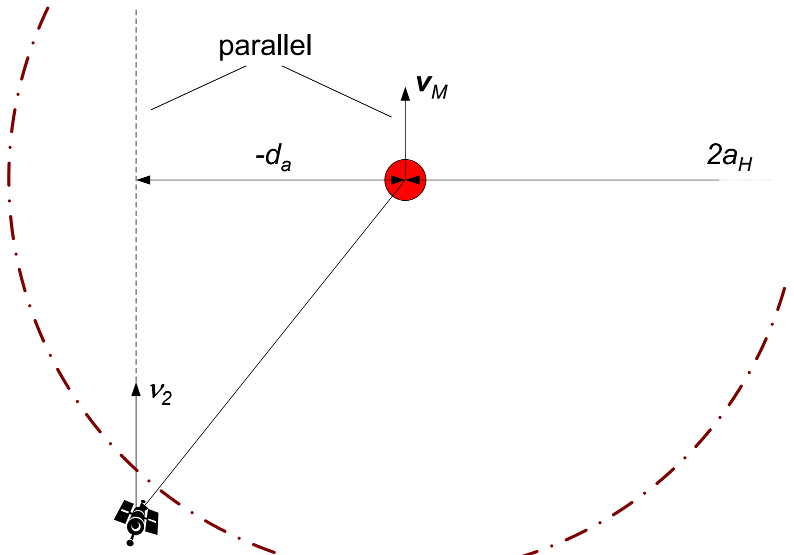


Figure 4.1: Example of an arrival orbit velocity $\boldsymbol{v}_2$ necessary for the approximation of the Hohmann chord length.

how the miss distance d_a is treated as a signed scalar. For the Hohmann transfer from Earth to Mars, the arrival velocity $\boldsymbol{\nu}_2$ is always nearly parallel to $\boldsymbol{v}_♂$. Thus Equation (4.1) presents a good approximation for the problem being studied. Using Equation (4.1), the partial of the miss distance with respect to the semi-major axis is calculated as follows:

$$\frac{\partial d_a}{\partial a_H} = \left(\frac{\partial c}{\partial d_a}\right)^{-1}\left(\frac{\partial c}{\partial a_H}\right) = (-1)^{-1}(2) = -2 \tag{4.2}$$

Thus if a_H is perturbed by some value n, then the corresponding change in d_a is predicted to be -2n.

The next partial derivative to be computed is that of the semi-major axis a_H with respect to the heliocentric departure velocity v_1. Figure 4.2 offers an illustration of the hyperbolic departure orbit. By applying the energy equation at the time t_1 (when the spacecraft reaches

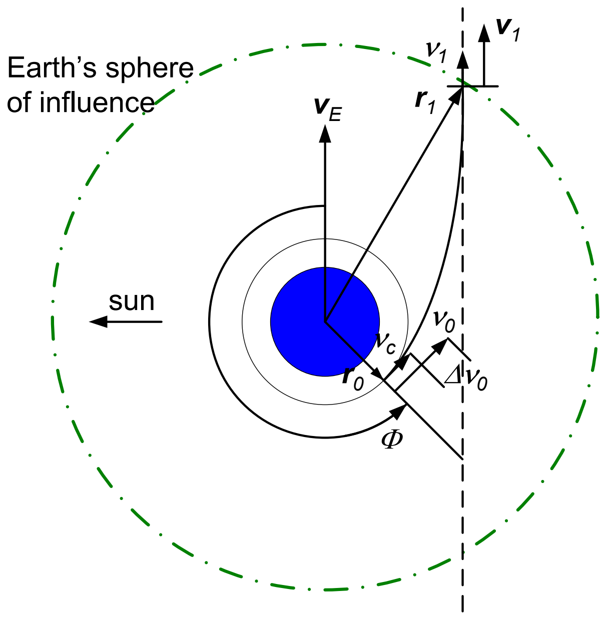


Figure 4.2: Depiction of the position and velocity notation used to describe the spacecraft's hyperbolic departure orbit from Earth.

Earth's sphere of influence), the semi-major axis is related to v_1 as

$$a_H = \frac{r_\oplus \mu_\odot}{2\mu_\odot - r_\oplus v_1^2} \tag{4.3}$$

where $r_\oplus$ is the mean orbit radius of Earth and $\mu_\odot$ is the gravitational coefficient of the sun. By Equation (4.3), the partial of a_H with respect to v_1 is determined to be

$$\frac{\partial a_H}{\partial v_1} = \frac{2r_\oplus^2 v_1 \mu_\odot}{(2\mu_\odot - r_\oplus v_1^2)^2} \tag{4.4}$$

Continuing to work back to the initial Earth-relative velocity ν_0, the next required partial derivative is that of the heliocentric v_1 with respect to the Earth-relative ν_1. Both of these velocities are displayed in Figure 4.2. In order to attain a relation between v_1 and ν_1, it is assumed that $\boldsymbol{\nu}_s$ converges parallel to $\boldsymbol{v}_\oplus$ at time t_1. Making this assumption, it follows that

$$v_1 = \nu_1 + v_\oplus \tag{4.5}$$

Thus the partial derivative $\partial v_1/\partial \nu_1$ is equal to one.

Finally, the partial derivative of v_1 with respect to the initial Earth-relative speed ν_0 is computed. In order to do so, the energy equation is applied at time t_0. In addition, it is assumed that $r_1 \approx \infty$.[4] By applying the energy equation, ν_1 is expressed as a function of ν_0 as follows:

$$\nu_1 = \sqrt{\nu_0^2 - \frac{2\mu_\oplus}{r_0}} \tag{4.6}$$

where r_0 is the initial parking radius of the hyperbolic departure. By Equation (4.6), the partial of ν_1 with respect to ν_0 is expressed as

$$\frac{\partial \nu_1}{\partial \nu_0} = \frac{\nu_0}{\sqrt{\nu_0^2 - \frac{2\mu_\oplus}{r_0}}} \tag{4.7}$$

In order to determine the partial of the miss distance d_a with respect to ν_0, the four previously calculated partial derivates are related using the chain rule for differentiation. The process is given as follows:

$$\frac{\partial d_a}{\partial \nu_0} = \frac{\partial d_a}{\partial a_H}\frac{\partial a_H}{\partial v_1}\frac{\partial v_1}{\partial \nu_1}\frac{\partial \nu_1}{\partial \nu_0} = \frac{-4r_\oplus^2 v_1 \nu_0 \mu_\odot}{(2\mu_\odot - r_\oplus v_1^2)^2\sqrt{\nu_0^2 - \frac{2\mu_\oplus}{r_0}}} \tag{4.8}$$

Equation (4.8) gives the analytical solution for the sensitivity of the miss distance d_a with respect to the initial Earth-relative velocity ν_0. This sensitivity may be used to relate perturbations in ν_0 to predicted changes in d_a.

4.1.2 Application Using the Four-Body Integrator

The analytical approximation to $\partial d_a/\partial \nu_0$ is now applied to the four-body problem using the previously developed variable step Runge-Kutta integrator.[5] Let the desired miss distance d^* correspond to a desired arrival periapsis of 4000 km. Using the patched-conic approximation, some initial guess for the required velocity ν_0 is computed. Reppert previously showed that this initial guess yields an arrival orbit that penetrates Mars' sphere of influence. The current goal is to use the analytical solution for $\partial d_a/\partial \nu_0$ to refine ν_0 such that it produces the desired miss distance d^*. Let the miss distance error at a given iteration be defined as

$$d_e = d_a - d^* \tag{4.9}$$

where d_a is the actual miss distance computed using the four-body integrator. By Equation (4.9), after a given iteration, the applied change in miss distance should be $\Delta d_a = -d_e$. Using the applied change in d_a, the required perturbation in ν_0 is computed as follows:

$$\Delta \nu_0 = \left(\frac{\partial d_a}{\partial \nu_0}\right)^{-1} \Delta d_a = -\left(\frac{\partial d_a}{\partial \nu_0}\right)^{-1} d_e = \frac{(2\mu_\odot - r_\oplus v_1^2)^2\sqrt{\nu_0^2 - \frac{2\mu_\oplus}{r_0}}}{4r_\oplus^2 v_1 \nu_0 \mu_\odot}\ d_e \tag{4.10}$$

Equation (4.10) is used to make corrections in the initial Earth-relative velocity ν_0 for each iteration of the Hohmann transfer. The desired result is an arrival orbit that has a miss distance d_a corresponding to a periapsis radius r_3 of 4000 km. Because d_a is a signed scalar, this application can be used to generate both posigrade and retrograde arrival orbits. If the arrival orbit direction were not significant, then the desired miss distance d^* could be assigned the same sign as d_a. This process would yield the desired arrival geometry closest to the initial integrated orbit.

The code used to apply the ν_0 correction scheme to the restricted four-body problem is developed in C. The reasons for programming the iterative process in C are given in Chapter

2. On the first iteration of the correction loop, the patched-conic approximation for the ν_0 required to arrive at Mars is calculated. This value is used as the first guess for the required departure speed. The first iteration also includes a calculation of the desired miss distance d^* based upon the specified Mars arrival periapsis r_3 (4000 km). Thereafter, for each iteration, the value of ν_0 is corrected using the approximate sensitivity given in Equation (4.10). The corrections are performed until the difference between the actual miss distance d_a and the desired miss distance d^* is less than 10 km. A limit of 15 iterations is placed on the correction scheme to avoid excessive computation. Figure 4.3 provides a flowchart outlining the iterative process that uses the analytical approximation for $\partial d_a/\partial \nu_0$ to apply corrections to ν_0.

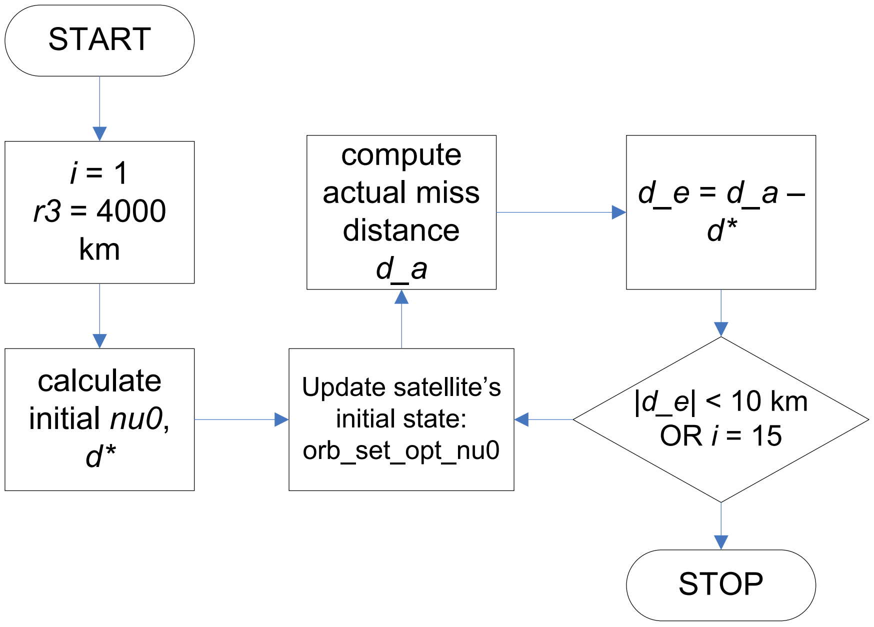

START
i = 1
r3 = 4000 km
calculate initial nu0, d*
Update satellite's initial state: orb_set_opt_nu0
compute actual miss distance d_a
d_e = d_a – d*
|d_e| < 10 km OR i = 15
STOP

Figure 4.3: Flowchart describing the iterative process used to apply the analytical approximation for $\partial d_a/\partial \nu_0$ to the restricted four-body problem. The sensitivity is used to make corrections in the initial speed ν_0.

Using the analytical approximation to the miss distance sensitivity, the system does indeed converge. As shown by Table 4.1, the miss distance d_a converges to within 10 km of the desired miss distance d^* after seven iterations. It is important to note that the value of d^* never changes throughout the iterative process. The desired periapsis radius r_3 and the overall Hohmann orbit geometry are the two main factors that affect the computation of d^*. Because the patched-conic approximation is used to derive a relation for d^*, the initial departure speed ν_0 is taken to have no effect on the desired miss distance. However, the value of ν_0 is taken to have an effect on the departure angle Φ (Figure 4.2). For each correction in ν_0, patched-conic relations are used to compute a corresponding perturbation in the departure angle. If this angle were not corrected along with the departure speed, then

Table 4.1: Listing of the miss distance values calculated for each step of the correction scheme using an analytical $\partial d_a / \partial \nu_0$. Note that the iterative process stops after the miss distance error $|d_e|$ drops below 10 km.

Iteration	d_a, km	d^*, km	d_e, km
1	-4.09307e+005	-8.05187e+003	-4.01255e+005
2	-9.27212e+004	-8.05187e+003	-8.46693e+004
3	-1.97402e+004	-8.05187e+003	-1.16883e+004
4	-9.49203e+003	-8.05187e+003	-1.44016e+003
5	-8.22640e+003	-8.05187e+003	-1.74537e+002
6	-8.07298e+003	-8.05187e+003	-2.11189e+001
7	-8.05443e+003	-8.05187e+003	-2.56708e+000

the spacecraft would not exit Earth's sphere of influence at the desired hyperbolic trajectory. In this case, controlled adjustments to ν_0 could not be made, and the iterative process would most likely fail to determine the necessary departure velocity.

Another noteable feature of the data given in Table 4.1 is that both the actual and error miss distances d_a and d_e are always negative. The spacecraft always penetrates Mars' sphere of influence on the negative side of the Mars-relative coordinate frame, as is later shown in Figure 4.5. In other words, the analytical approximation to the sensitivity never yields a correction that overshoots the desired miss distance. Each corrected value of ν_0 consistently approaches the desired arrival geometry from the same side.

As both Table 4.4 and Figure 4.4 show, the magnitude of the miss distance error d_e decreases by approximately one order of magnitude for every iteration. From the first itera-

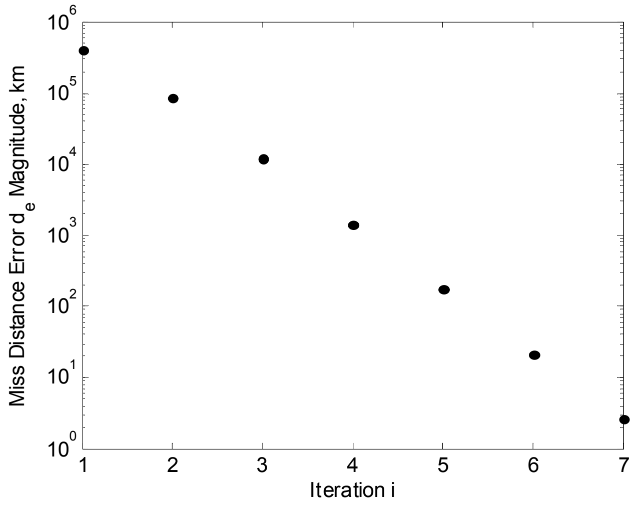


Figure 4.4: Plot of the decreasing miss distance error values d_e for the ν_0 correction scheme using an analytical sensitivity approximation. Note that d_e decreases by approximately one order of magnitude at every iteration.

tion to the seventh iteration, the magnitude drops from 4.01×10^5 km to 2.57 km. After the seventh iteration, the 10-km error stopping criterion is achieved, and the iterative process is terminated.

In summary, the analytical approximation to the miss distance sensitivity performs very well when used to correct ν_0 for the given Hohmann transfer orbit. The actual miss distance consistently approaches the desired miss distance, with the miss distance error decreasing by an order of magnitude at each step. Using this correction scheme, it is determined that an initial speed ν_0 of approximately 10.71824 km/s is necessary to achieve the desired miss distance. Figure 4.5 displays the corrected Mars arrival orbit geometry. As Figure 4.5 shows,

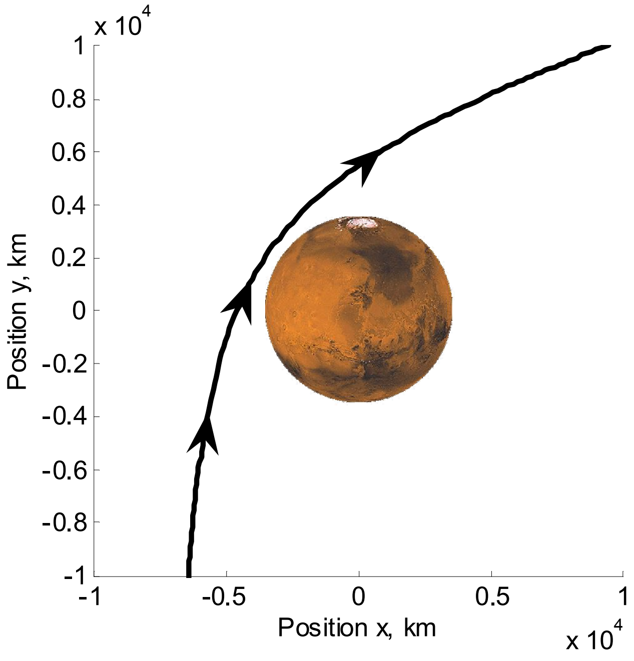


Figure 4.5: Depiction of the Mars-relative orbit geometry that results from using the analytical approximation to the miss distance sensitivity $\partial d_a/\partial \nu_0$. Positions x and y are given relative to the center of the Mars frame $\mathcal{M}$.

the arrival orbit achieves a periapsis radius r_3 that is very close to 4000 km. However, the correction scheme presented in this section is only used to compute corrections in the miss distance d_a. Patched-conic relations are then used to relate the miss distance to the final periapsis radius r_3. A more exact correction process would incorporate the gravitational influences of Earth and the sun into the calculated of the actual periapsis radius r_3. In this manner, corrections could be made directly to the arrival periapsis r_3. Nonetheless, Figure 4.5 illustrates that the analytical approximation to $\partial d_a/\partial \nu_0$ provides a sufficient means of achieving a good estimate of the necessary departure speed ν_0.

4.2 Numerical Solution for $\frac{\partial d_a}{\partial \nu_0}$

The numerical miss distance correction scheme uses approximations to the $\frac{\partial d_a}{\partial \nu_0}$ sensitivity. At each iteration, a current value of ν_0 is used to integrate the spacecraft's Hohmann transfer to Mars. The resulting miss distance d_a upon entering Mars' sphere of influence is recorded. The current ν_0 is then perturbed by a value ϵ as follows:

$$\nu_0' = \nu_0 + \epsilon \tag{4.11}$$

where ν_0' is the Earth-centric departure speed after the small perturbation. The importance of choosing a decent ϵ value is discussed later in this section. Once the perturbed speed ν_0' is obtained, the spacecraft's Hohmann transfer is integrated again. This second orbit is slightly different from the first, resulting in a different miss distance d_a'. The sensitivity $\frac{\partial d_a}{\partial \nu_0}$ can then be approximated as:

$$\frac{\partial d_a}{\partial \nu_0} \approx \frac{d_a' - d_a}{\nu_0' - \nu_0} = \frac{d_a' - d_a}{\epsilon} \tag{4.12}$$

This approximate sensitivity is used to compute a new Earth-centric departure speed as follows:

$$\nu_{0_{i+1}} = \nu_{0_i} - \left(\frac{\partial d_a}{\partial \nu_0}\right)^{-1} d_e \tag{4.13}$$

where d_e represents the error in the miss distance at the current iteration i.

In this manner, an approximate sensitivity is computed for each iteration of the correction scheme, and therefore two complete integrations of the Hohmann transfer are necessary for one iteration i. Each iteration of the numerical sensitivity correction scheme takes approximately twice as long as an iteration using the analytical sensitivity. However, this property does not necessarily mean that the numerical solution takes longer to converge to a desired miss distance criterion. A comparison of integration time using the two methods is provided in Section 4.4.

At this point, it is important to note that a good initial guess for the necessary departure speed ν_0 is required due to the nonlinearity of the spacecraft's motion. If the initial guess is not within a sufficient range of the required value, the numerical correction scheme fails. For this problem, the patched-conic approximation to the necessary ν_0 for the transfer is sufficient for convergence to a solution. When solving linear systems of equations, the initial guess is not very important when considering convergence. But for nonlinear problems such as the restricted four-body setup, the initial guess must be chosen with care.[3]

Table 4.2 provides a listing of the number of iterations k and final miss distance error values $|d_e|$ corresponding to the use of a range of perturbation values ϵ. The value of ϵ has a significant effect on the rate of convergence of the numerical correction scheme. For instance, any values of ϵ greater than or equal to 10^{-2} km/s are too large to yield convergence. Such large perturbations incur excessively large corrections to the Hohmann transfer. Conversely, any values of ϵ equal to or smaller than 10^{-10} km/s are too small to yield convergence. Using a 32-bit processor, the truncation error involved with such calculations makes the correction

Table 4.2: Tabulated values for number of iterations k and final miss distance error magnitude $|d_e|$ versus perturbation ϵ using the numerical $\frac{\partial d_a}{\partial \nu_0}$. At ϵ equal to 1e-02 and 1e-10, the iterations do not converge (DNC).

ϵ, km/s	1e-02	1e-03	1e-04	1e-05	1e-06	1e-07	1e-08	1e-09	1e-10
k	DNC	5	4	4	4	4	4	5	DNC
$\lvert d_e \rvert$, km	DNC	4.16e-1	8.16e-1	7.29e-2	4.85e-2	1.98e-1	6.43e-1	1.09e-1	DNC

scheme ineffective. Using a numerical approximation to $\frac{\partial d_a}{\partial \nu_0}$, the number of iterations necessary for convergence to the stopping criterion ($d_e < 10$ km) plateaus at four for ϵ between 10^{-4} km/s and 10^{-8} km/s.

Figure 4.6 provides a plot of the number of iterations k necessary to satisfy the stopping criterion using the different values of ϵ. As is noted in the figure, the minimum final miss

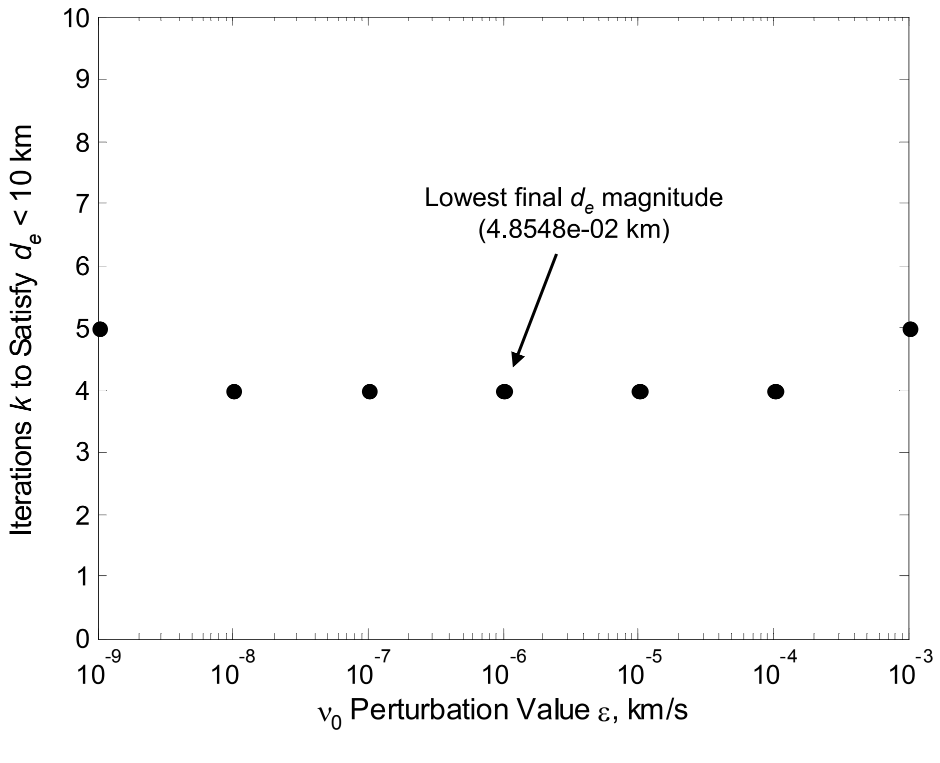


Figure 4.6: Plot of the number of iterations k versus the ϵ perturbation value in ν_0. Note that $\epsilon = 10^{-6}$ km/s yields the lowest final miss distance error magnitude.

distance error magnitude occurs at $\epsilon = 10^{-6}$ km/s. However, all values of ϵ between 10^{-4} km/s and 10^{-8} km/s provide convergence within the same minimum amount of time for the given stopping criterion.

4.3 Numerical Solution for $\frac{\partial r_3}{\partial \nu_0}$

The numerical periapsis correction scheme uses approximations to the $\frac{\partial r_3}{\partial \nu_0}$ sensitivity. The algorithm for using the sensitivity approximation to apply corrections to the Earth-relative departure speed ν_0 is identical to that used for the miss distance numerical scheme. The only difference in this case is that the sensitivity is being used to correct the periapsis radius r_3 instead of the miss distance d_a. Thus, after ν_0 is perturbed by some value ϵ, the resulting orbit yields a periapsis radius r_3'. This perturbed periapsis radius is used with the unperturbed value to approximate the sensitivity as:

$$\frac{\partial r_3}{\partial \nu_0} \approx \frac{r_3' - r_3}{\nu_0' - \nu_0} = \frac{r_3' - r_3}{\epsilon} \tag{4.14}$$

The approximation to the sensitivity can then be used to apply corrections to ν_0 according to:

$$\nu_{0_{i+1}} = \nu_{0_i} - \left(\frac{\partial r_3}{\partial \nu_0}\right)^{-1} r_{3_e} \tag{4.15}$$

where r_{3_e} corresponds to the error of the current periapsis radius relative to the desired value (4000 km for this report). The value of ν_0 is corrected until the specified periapsis error tolerance is achieved.

The direct periapsis correction scheme is an even more sensitive nonlinear problem than the miss distance correction scheme. Therefore, the initial guess for ν_0 is more important for the periapsis correction algorithm. For the restricted four-body problem, using the patched-conic approximation to the necessary departure speed ν_0 is sufficient to provide convergence for a range of perturbations ϵ. If the patched-conic approximation were not sufficiently accurate for an initial guess, the solution of the miss distance correction scheme could have been used as an initial guess for the periapsis scheme. It is important to note how a very sensitive, nonlinear problem can be brought to a particular solution by gradually increasing the complexity and robustness of the integration.

Figure 4.7 provides a plot of the number of iterations k taken to converge within the specified periapsis tolerance ($r_{3_e} < 10$ km) for different values of perturbation ϵ. Similar to the miss distance correction case, any values of ϵ less than or equal to 10^{-10} km/s result in correction schemes that do not converge. The lack of convergence is again due to the significant truncation error that occurs when using such fine perturbations to make corrections in ν_0. If a 64-bit processor were used to make the calculations, then the lower limit for ϵ values would most likely decrease due to the enhanced ability to handle computations with smaller numbers. All values of ϵ equal to or greater than 1 km/s cause divergence as well. In this case, the corrections to ν_0 are so large that they cause excessive overshooting of the desired departure speed.

As Figure 4.7 shows, the number of iterations k plateaus for values of ϵ between 10^{-9} km/s and 10^{-5} km/s. Interestingly, the periapsis correction scheme takes 38 iterations to converge for $\epsilon = 10^{-4}$ km/s. For some reason, this seemingly standard value of ϵ results in extremely slow convergence to the desired orbit geometry. The particular reasons for this behavior are left for a subsequent study. The minimum error magnitude $|r_{3_e}|$ (0.5455 km) occurs for a perturbation value of 10^{-5} km/s. However, using perturbations of 10^{-9} km/s

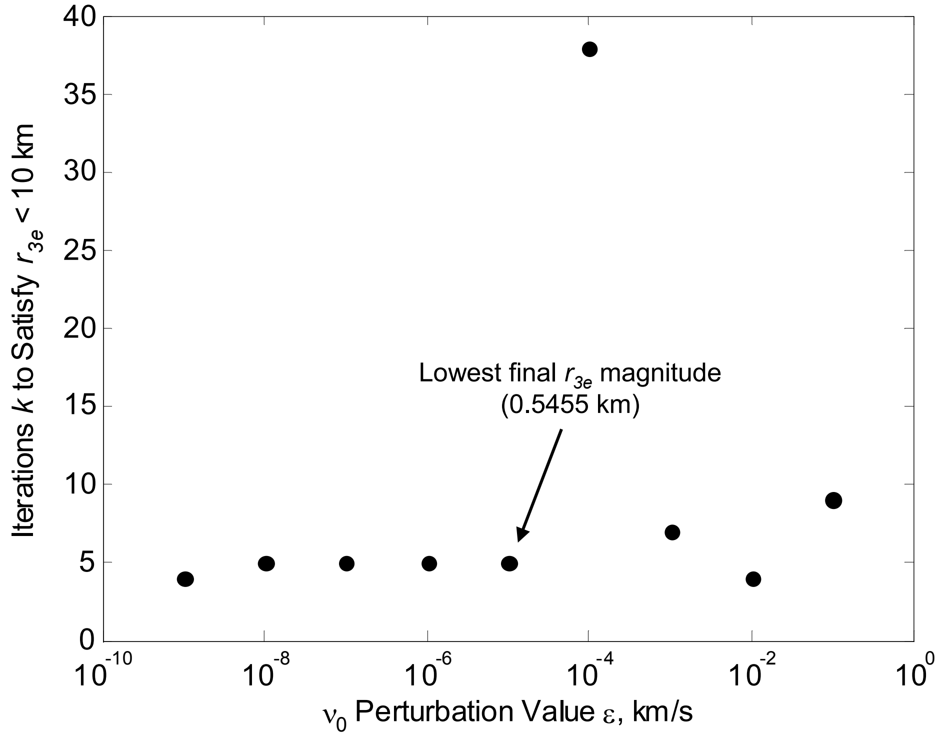


Figure 4.7: Plot of the number of iterations k versus the ϵ perturbation value in ν_0. Note that $\epsilon = 10^{-5}$ km/s yields the lowest final miss distance error magnitude.

and 10^{-4} km/s result in the fastest rates of convergence, as the correction schemes finish within 4 iterations.

4.4 Integration Time Comparison

The number of iterations is not the best means of comparing the convergence rates of the three previously discussed correction algorithms. Both of the numerical sensitivity algorithms require two complete transfer orbit integrations for each iteration. The first integration is used to compute the current sensitivity value, and the second integration provides a means of calculating a new departure speed. In contrast, the analytical miss distance sensitivity algorithm requires only one integration per iteration. While the use of the analytical sensitivity requires more iterations to achieve a particular miss distance accuracy, it does not necessarily require more computational time.

Therefore, the central processor unit (CPU) time required to converge to within a particular accuracy is a better measure of the rate of convergence for the correction methods. The C function `clock` is used to provide an estimate of the amount of time that each of the three previously described methods takes to achieve the stopping criterion. The accuracy of the time estimates is 0.001 seconds. For both of the miss distance algorithms, the stopping criterion is set to $d_e \leq 10^{-3}$ km, whereas that of the direct periapsis algorithm is set to $r_{3_e} \leq 10^{-3}$ km. For each of the numerical approximation methods, the perturbation ϵ is set to 10^{-6} km/s.

Figure 4.8 provides plots of the time taken to provide different levels of accuracy in the arrival periapsis r_3. One of the most notable features of Figure 4.8 is the plateau in periapsis

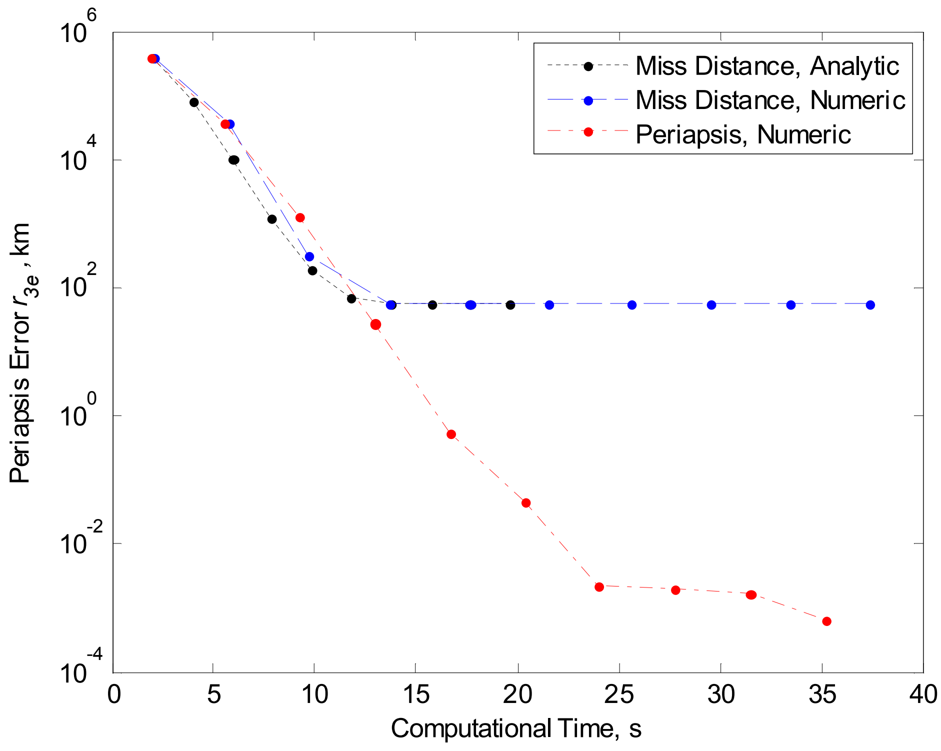


Figure 4.8: Plot of the integration time used for the analytical and numerical $\frac{\partial d_a}{\partial \nu_0}$ approximation algorithms, as well as the numerical $\frac{\partial r_3}{\partial \nu_0}$ algorithm. For each of the numerical approximation methods, the perturbation ϵ is set to 10^{-6} km/s.

error r_{3_e} that occurs for both of the miss distance sensitivity algorithms. This plateau suggests that the two methods are unable to achieve an error in the periapsis radius less than approximately 190 km. The reason for the two methods' plateau is the approximation built into the patched-conic relation between the miss distance d_a and periapsis r_3. The patched-conic relation between d_a and r_3 is built upon the two-body assumption. However, the sun and the Earth still have minor effects upon the trajectory of the spacecraft after it penetrates Mars' sphere of influence. In this manner, the patched-conic prediction for the miss distance necessary to achieve $r_3 = 4000$ km involves approximately 190-km error from the actual periapsis radius.

In contrast, the algorithm using the numerical approximation to the periapsis sensitivity continues to approach zero error after a computational time of approximately 12 seconds. This third algorithm achieves periapsis errors roughly five orders of magnitude smaller by 24 seconds. In other words, the $\frac{\partial r_3}{\partial \nu_0}$ correction algorithm achieves roughly meter-level periapsis accuracy by 24 seconds. The other two algorithms simply cannot achieve this accuracy. While all three methods perform fairly equally during the first 12 seconds of CPU time, the direct periapsis correction scheme performs much better when continuing the correction process. The reason for this better performance is that it does not depend upon a patched-conic relation between d_a and r_3 for a specified accuracy. Thus, of the three examined methods, the $\frac{\partial r_3}{\partial \nu_0}$ correction algorithm provides the best performance for correcting the arrival periapsis r_3.

Chapter 5

Conclusions

This report presents an investigation of the use of sensitivities to optimize the arrival trajectory of a Hohmann transfer from Earth to Mars. The equations of motion are derived from a restricted four-body setup, with Earth, Mars, and the sun as the primary celestial bodies. Chapter 3 presents an introduction to the concept of sensitivities and, in particular, state transition matrices. The usefulness of sensitivity matrices in providing a description of how perturbations of one variable cause changes in another variable are discussed. The theory presented in Chapter 3 is then extended to perform optimizations of the restricted four-body problem. Chapter 4 discusses the use of three sensitivities to optimize the arrival orbit geometry of the Hohmann transfer: analytical $\frac{\partial d_a}{\partial \nu_0}$, numerical $\frac{\partial d_a}{\partial \nu_0}$, and numerical $\frac{\partial r_3}{\partial \nu_0}$. The application of each sensitivity to a corresponding orbit correction scheme is presented. Additionally, the convergence performances of the perturbation methods are compared by the computation time required to achieve a particular accuracy. The following conclusions are drawn:

1. The C programming language is more efficient than Matlab for calculating multiple transfer orbits in a computationally intense optimization problem.
2. Matlab provides user-friendly techniques for creating visualizations of the C output.
3. Sensitivity matrices provide a way to map perturbations in one variable into changes in another variable.
4. State transition matrices, a subset of sensitivity matrices, provide a way to map perturbations in initial conditions to changes in the final state.
5. For a stopping criterion of $d_e \leq 10$ km, the use of the analytical sensitivity $\frac{\partial d_a}{\partial \nu_0}$ provides convergence within seven iterations.
6. For a stopping criterion of $d_e \leq 10$ km, the use of the numerical sensitivity $\frac{\partial d_a}{\partial \nu_0}$ provides convergence within four iterations for a series of departure speed perturbations ϵ.
7. For a stopping criterion of $r_3 \leq 10$ km, the use of the numerical sensitivity $\frac{\partial r_3}{\partial \nu_0}$ provides convergence within four iterations for two values of departure speed perturbation ϵ.
8. During the first 12 seconds of computational time, the three perturbation techniques perform equally well.
9. After 12 seconds, the direct r_3 correction scheme performs much better than the indirect d_a correction schemes.
10. The r_3 correction scheme performs better because it does not depend on the patched-conic approximation to the relation between the miss distance and the periapsis radius.

Appendix A

C Integration Code

A.1 C Source Files

opt_r3.c

```
/* Integrate the equations of motion of a satellite on a Hohmann transfer
   from Earth to Mars, taking into consideration the gravity of Earth,
   Mars, and the sun for all time t */

// Thomas Reppert 08/31/06

#include     <stdio.h>
#include    <stdlib.h>
#include      <math.h>
#include "ArrayMath.h"
#include "AstroConstants.h"
#include "MathConstants.h"

/* note: all array math performed disregarding the zeroth index in
   an attempt to make the C code more compatible with MatLab */

// declare the orbit setup function
double orb_set_opt_r3(double r_0, double* y_0, double r_2, double r_3);

// declare the Runge-Kutta integrator
double RK_4_4body(const double ti, const double tf, double* y);

// declare derivative calculator for each slope estimate
void dydt_4body(double offset, double t, double* y, double* dydt);

// declare the Earth-centric satellite position calculator
void SatvsEarth(double t, double* y, double* R_2);
```

```
// declare the Mars-centric satellite position calculator
void SatvsMars(double offset, double t, double* y, double* R_3);

// declare the Earth position calculator
void EarthPos(double t, double* PosVec);

// declare the Mars position and velocity calculator
void MarsPosVel(double offset, double t, double* PosAndVel);

// declare the arrival miss distance calculator
double MissDistance(double* SatWrtMars);

int main(void)
{
// initialize the initial time and final time (s)
  double InitTime = 0.0, FinTime = 22376000.0;
// initialize the departure parking orbit radius (km)
double InitPark = 7500.0;
// initialize the satellite state vector at time t_0
double InitState[7] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
// initialize heliocentric and arrival radii, arrival orbit miss
// distance values
double r_2, r_3, d_a, d_star, d_error;
// initialize loop counter
int i = 1;
// initialize pointer to file for writing
FILE* foutput_0;

// open the file for writing
if((foutput_0 = fopen("OptArrive.txt", "w")) == NULL)
  {
    printf("*** OptArrive.txt could not be opened.\n");
    exit(EXIT_FAILURE);
  }

// set the initial value of r_2 and the desired arrival parking radius r_3
r_2 = SMA_MARS;
r_3 = 4000.0;

while(1)
{
// compute the necessary miss distance d_star
d_star = orb_set_opt_r3(InitPark, InitState, r_2, r_3);

// compute the actual miss distance d_a
d_a = RK_4_4body(InitTime, FinTime, InitState);

// calculat the error in miss distance
d_error = d_a - d_star;

// print the iteration to the write file
fprintf(foutput_0,"%2i %15.10e %15.10e %15.10e\n",i,d_a,d_star,d_error);

// once the error drops below 20 km (or have done 15 iters),
// stop iterating
if((fabs(d_error) < 1.e1) || (i == 15)) break;
// else, correct the value of the parameter r_2
else r_2 = r_2 + d_error;

// step the iteration variable
i = i + 1;
}

return 0;
}


// orb_set_opt_r3:
// computes the initial heliocentric position and velocity of a satellite
// on a Hohmann transfer from Earth to Mars, in addition to the necessary
// miss distance for a desired final parking radius r_3

// input:
// initial parking orbit radius r_0
// initialized satellite state vector y_0

// output:
// populated initial satellite state vector y_0
// necessary miss distance d_star, given desired Mars parking radius r_3

// Thomas Reppert 08/30/06

double orb_set_opt_r3(double r_0, double* y_0, double r_2, double r_3)
{
// initialize values
double c, v_1, n_earth, v_earth, nu_1, a, nu_0, nu_c, e, Phi;
double rx_0, ry_0, rz_0, xdot_0, ydot_0, zdot_0, d_star;
double v_2, sigma_2, n_mars, v_mars, nu_2;

// Hohmann transfer chord length (km)
```

```
c = SMA_EARTH + r_2;

// spacecraft's velocity when departing Earth's SOI (km/s)
v_1 = sqrt((2.0*MU_SUN*r_2)/(c*SMA_EARTH));

// Earth's mean orbit rate (rad/s)
n_earth = sqrt(MU_SUN/pow(SMA_EARTH, 3));

// Earth's heliocentric velocity (km/s)
v_earth = SMA_EARTH*n_earth;

// spacecraft's Earth-centric departure velocity at t_1 (km/s)
nu_1 = v_1 - v_earth;

// hyperbolic semi-major axis (km)
a = -MU_EARTH/pow(nu_1,2);

// spacecraft's Earth-centric departure velocity at t_0 (km/s)
nu_0 = sqrt(2.0*MU_EARTH/r_0 - MU_EARTH/a);

// initial Earth-centric critical velocity (km/s)
nu_c = sqrt(MU_EARTH/r_0);

// departure orbit eccentricity
e = r_0*pow(nu_0,2)/MU_EARTH - 1.0;

// initial burn angle Phi, rad
Phi = acos(1.0/e) + PI;

// initial satellite state vector
// ------------------------------

// heliocentric position (km)
rx_0 = SMA_EARTH + r_0*cos(3*PI/2-Phi);
ry_0 =            -r_0*sin(3*PI/2-Phi);
rz_0 =                               0;

// heliocentric velocity (km/s)
xdot_0 =           nu_0*cos(Phi-PI);
ydot_0 = v_earth + nu_0*sin(Phi-PI);
zdot_0 =                          0;

// populate the state vector array
y_0[1] = rx_0;
y_0[2] = ry_0;
```

```
y_0[3] = rz_0;
y_0[4] = xdot_0;
y_0[5] = ydot_0;
y_0[6] = zdot_0;

// arrival orbit parameters
// ------------------------

// spacecraft's heliocentric velocity when arriving at Mars's SOI, km/s
v_2 = sqrt(2*MU_SUN*((SMA_EARTH - r_2)/(SMA_EARTH*r_2)) + v_1*v_1);
n_mars = (SMA_EARTH*v_1)/(r_2*v_2);
// heading angle between the spacecraft's helio velocity and the sun
// normal, rad
if(fabs((SMA_EARTH*v_1)/(r_2*v_2) - 1.0) < 1.e-6) sigma_2 = 0.0;
else sigma_2 = acos((SMA_EARTH*v_1)/(r_2*v_2));

// Mars's mean orbit rate, rad/s
n_mars = sqrt(MU_SUN/pow(SMA_MARS, 3));

// Mars's heliocentric velocity, km/s
v_mars = SMA_MARS*n_mars;

// spacecraft's Mars-centric arrival velocity at t_2, km/s
nu_2 = sqrt(v_2*v_2 + v_mars*v_mars - 2*v_2*v_mars*cos(sigma_2));

// arrival orbit eccentricity
e = r_3*nu_2*nu_2/MU_MARS + 1.0;

// necessary miss distance for desired r_3, km
d_star = -sqrt((e*e - 1.0)*(MU_MARS/(nu_2*nu_2))*(MU_MARS/(nu_2*nu_2)));

return d_star;
}


// RK_4_4body:
// uses the Fourth-Order Runge-Kutta technique to integrate the equations
// of motion of a satellite in a planar Hohmann transfer from Earth to
// Mars (type dydt = f(t,y)), taking into consideration the gravity of
// the sun, Earth, and Mars for all time t

// input:
// initial time ti
// final time tf
// initial state y0
// output:
// final satellite state vector y
// file SatwrtSun.txt with propagated heliocentric state vector
// file SatwrtEarth.txt with propagated Earth-centric state vector
// file SatwrtMars.txt with propagated Mars-centric state vector
// arrival orbit miss distance d_a, km

// Thomas Reppert 08/31/06

double RK_4_4body(const double ti, const double tf, double* y)
{
// initialize slope estimates k_i, averaged slope estimate phi
  double k1[7], k2[7], k3[7], k4[7], phi[7];
// initialize scaled slope estimates k_sc_i and scaled average phi_sc
double k1_sc[7], k2_sc[7], k3_sc[7], phi_sc[7];
// initialize iterated time t, time step h, temporary and old y place
// holders
  double t, h, y_temp[7], y_old[7];
// initialize celestial parameters and arrival miss distance
double n_mars, a, P, offset, rE_SOI, rM_SOI, d_a;
// initialize the satellite's position vector with respect to each planet
double SvsE[5], SvsM[9], SvsM_old[9];
// initialize pointers to files for writing
  FILE* foutput_1;
FILE* foutput_2;
FILE* foutput_3;

// check to make sure the output file opens correctly
  // if it doesn't open, leave the program
if((foutput_1 = fopen("SatwrtSun.txt", "w")) == NULL)
  {
    printf("*** SatwrtSun.txt could not be opened.\n");
    exit(EXIT_FAILURE);
  }
else if((foutput_2 = fopen("SatwrtEarth.txt", "w")) == NULL)
  {
    printf("*** SatwrtEarth.txt could not be opened.\n");
    exit(EXIT_FAILURE);
  }
else if((foutput_3 = fopen("SatwrtMars.txt", "w")) == NULL)
  {
    printf("*** SatwrtMars.txt could not be opened.\n");
    exit(EXIT_FAILURE);
  }
```

```
// calculate the offset angle between Earth and Mars
n_mars = sqrt(MU_SUN/pow(SMA_MARS,3)); // Mars's mean orbit rate, rad/s
a = (SMA_EARTH + SMA_MARS)/2; // semi-major axis of the transfer, km
P = PI*sqrt(pow(a,3)/MU_SUN); // period of the Hohmann transfer, s
offset = PI - n_mars*P; // offset angle between Earth and Mars, rad

SatvsEarth(ti, y, SvsE);
SatvsMars(offset, ti, y, SvsM);

// write the initial heliocentric conditions to SatwrtSun.txt
  fprintf(foutput_1,"%15.10e %15.10e %15.10e %15.10e %15.10e %15.10e"
                    "%15.10e\n",ti ,y[1],y[2],y[3],y[4],y[5],y[6]);

// write the initial Earth-centric conditions to SatwrtEarth.txt
  fprintf(foutput_2,"%15.10e %15.10e %15.10e %15.10e %15.10e\n",
                    ti ,SvsE[1],SvsE[2],SvsE[3],SvsE[4]);

// write the initial Mars-centric conditions to SatwrtMars.txt
  fprintf(foutput_3,"%15.10e %15.10e %15.10e %15.10e %15.10e\n",
                    ti ,SvsM[1],SvsM[2],SvsM[3],SvsM[4]);

// perform the Runge-Kutta integration
// -----------------------------------

rE_SOI = 916600; // Earth's SOI radius, km
rM_SOI = 577400; // Mars's SOI radius, km

// assign the initial time to time variable t
t = ti;

while(1)
{
// loop while the current time t is still less than the final time tf
    if(t >= tf) break;

// use a variable time step to speed up the integration: increase the
    // time step when the satellite's position wrt both Earth and Mars is
    // greater than 1.5*r_SOI of both Earth and Mars, respectively
if((SvsE[4] > 1.5*rE_SOI) && (SvsM[4] > 1.5*rM_SOI)) h = 50000.0;
    else h = 50.0;

// estimate slope at t and assign to k1
    dydt_4body(offset, t, y, k1);
```

```
// scale the k1 slope estimate by h/2
    scale_array(7, h/2, k1, k1_sc);

// assign (y + k1*h/2) value to temporary state y_temp
    add_2_arrays(7, y, k1_sc, y_temp);

// use the new y_temp to estimate slope at (t + h/2) as k2
    dydt_4body(offset, t + h/2, y_temp, k2);

// scale the k2 slope estimate by h/2
scale_array(7, h/2, k2, k2_sc);

// assign (y + k2*h/2) value to temporary state y_temp
add_2_arrays(7, y, k2_sc, y_temp);

// use the new y_temp to estimate slope at (t + h/2) as k3
dydt_4body(offset, t + h/2, y_temp, k3);

// scale the k3 slope estimate by h
scale_array(7, h, k3, k3_sc);

// assign (y + k3*h) value to temporary state y_temp
add_2_arrays(7, y, k3_sc, y_temp);

// use the new y_temp to estimate slope at (t + h) as k4
dydt_4body(offset, t + h, y_temp, k4);

// scale the k2 slope estimate to twice its original value
scale_array(7, 2, k2, k2_sc);

// scale the k3 slope estimate to twice its original value
scale_array(7, 2, k3, k3_sc);

// calculate the averaged slope estimate phi
// note: phi weighted as (k1 + 2*k2 + 2*k3 + k4)/6
add_4_arrays(7, k1, k2_sc, k3_sc, k4, phi_sc);
scale_array(7, h/6, phi_sc, phi);

// assign current state y to y_old in preparation for calculating the
// new state
equal(7, y_old, y);

// add the old state y_old to quantity phi*h for the new state y
add_2_arrays(7, y_old, phi, y);
```

```
// assign current state SvsM to SvsM_old (eventually find where crosses
// Mars' SOI)
equal(9, SvsM_old, SvsM);

// increment the time variable
t = t + h;

// compute the new satellite position wrt both Earth and Mars
    SatvsEarth(t, y, SvsE);
    SatvsMars(offset, t, y, SvsM);

// write the propagated satellite state vectors to each respective file
fprintf(foutput_1,"%15.10e %15.10e %15.10e %15.10e %15.10e %15.10e"
                  "%15.10e\n",t ,y[1],y[2],y[3],y[4],y[5],y[6]);
fprintf(foutput_2,"%15.10e %15.10e %15.10e %15.10e %15.10e\n",
t ,SvsE[1],SvsE[2],SvsE[3],SvsE[4]);
fprintf(foutput_3,"%15.10e %15.10e %15.10e %15.10e %15.10e\n",
t ,SvsM[1],SvsM[2],SvsM[3],SvsM[4]);

// check for when the satellite crosses Mars' SOI
if((SvsM[4] <= rM_SOI) && (SvsM_old[4] > rM_SOI))
{
  d_a = MissDistance(SvsM);
}
}

  fclose(foutput_1);
  fclose(foutput_2);
  fclose(foutput_3);

  return d_a;
}


// MissDistance:
// upon entry into Mars' sphere of influence, calculates the miss distance
// d_a, km

// input:
// satellite's integrated Mars-centric state vector SatWrtMars

// output:
// miss distance d_miss, km

// Thomas Reppert 09/01/06
```

```
double MissDistance(double* SatWrtMars)
{
// initialize necessary values
int i, index[2];
double d_miss, step, x_tan[40001], y_tan[40001], r_tan[40001];

// set the step size
step = 10.0;

// if the entering slope is less than 1.0
if(fabs(SatWrtMars[6]/SatWrtMars[5]) <= 1.0)
{
// set the initial values of the tangent line (use x_tan as independent
// variable)
x_tan[1] = -4.e5;
y_tan[1] = SatWrtMars[2] + (SatWrtMars[6]/SatWrtMars[5])*(x_tan[1]
           - SatWrtMars[1]);
r_tan[1] = sqrt(x_tan[1]*x_tan[1] + y_tan[1]*y_tan[1]);
// iterate for the remaining values of the tangent line
for(i = 2; i <= 40000; ++i)
{
x_tan[i] = x_tan[i-1] + step;
y_tan[i] = SatWrtMars[2] + (SatWrtMars[6]/SatWrtMars[5])*(x_tan[i]
           - SatWrtMars[1]);
r_tan[i] = sqrt(x_tan[i]*x_tan[i] + y_tan[i]*y_tan[i]);
}
}
// if the entering slope is greater than 1.0
else
{
// set the initial values of the tangent line (use y_tan as independent
// variable)
y_tan[1] = -4.e5;
x_tan[1] = SatWrtMars[1] + (SatWrtMars[5]/SatWrtMars[6])*(y_tan[1]
           - SatWrtMars[2]);
r_tan[1] = sqrt(x_tan[1]*x_tan[1] + y_tan[1]*y_tan[1]);
// iterate for the remaining values of the tangent line
for(i = 2; i <= 40000; ++i)
{
y_tan[i] = y_tan[i-1] + step;
x_tan[i] = SatWrtMars[1] + (SatWrtMars[5]/SatWrtMars[6])*(y_tan[i]
           - SatWrtMars[2]);
r_tan[i] = sqrt(x_tan[i]*x_tan[i] + y_tan[i]*y_tan[i]);
}
}

// calculate the closest point to Mars and the corresponding index
d_miss = MinValue(i, index, r_tan);

// give the miss distance a sign relative to the center of Mars
if(x_tan[index[1]] < 0.0)
{
d_miss = -d_miss;
}

return d_miss;
}


// dydt_4body:
// calculates the value of each slope estimate k
// for the Fourth-Order Runge-Kutta integration

// input:
// initial offset angle between Earth and Mars
// current time t
// current satellite heliocentric state y

// output:
// slope estimate vector dydt

// Thomas Reppert 08/31/06

void dydt_4body(double offset, double t, double* y, double* dydt)
{
// initialize planet position vectors
double E[4], M[7];
// initialize satellite position parameters
double x_1, y_1, z_1, r_1, x_2, y_2, z_2, r_2, x_3, y_3, z_3, r_3;

// compute Earth's heliocentric position vector at the current time t
EarthPos(t, E);

// compute Mars's heliocentric position vector at the current time t
MarsPosVel(offset, t, M);

// satellite's position wrt the sun r_1, km
// ----------------------------------------
x_1 = y[1];
```

```
y_1 = y[2];
z_1 = y[3];
r_1 = sqrt(x_1*x_1 + y_1*y_1 + z_1*z_1);

// satellite's position wrt Earth r_2, km
// ------------------------------------
x_2 = y[1] - E[1];
y_2 = y[2] - E[2];
z_2 = y[3] - E[3];
r_2 = sqrt(x_2*x_2 + y_2*y_2 + z_2*z_2);

// satellite's position wrt Mars r_3, km
// -----------------------------------
x_3 = y[1] - M[1];
y_3 = y[2] - M[2];
z_3 = y[3] - M[3];
r_3 = sqrt(x_3*x_3 + y_3*y_3 + z_3*z_3);

// calculate the dydt slope estimate components 1:6
// ---------------------------------------------

dydt[1] = y[4];
dydt[2] = y[5];
dydt[3] = y[6];

dydt[4] = -MU_SUN/pow(r_1,3)*x_1 - MU_EARTH/pow(r_2,3)*x_2
                                 - MU_MARS/pow(r_3,3)*x_3;
dydt[5] = -MU_SUN/pow(r_1,3)*y_1 - MU_EARTH/pow(r_2,3)*y_2
                                 - MU_MARS/pow(r_3,3)*y_3;
dydt[6] = -MU_SUN/pow(r_1,3)*z_1 - MU_EARTH/pow(r_2,3)*z_2
                                 - MU_MARS/pow(r_3,3)*z_3;

return;
}


// SatvsEarth:
// computes the satellite's position with respect to Earth at the
// specified time t during the Hohmann transfer

// input:
// current time t
// current satellite heliocentric state y

// output:
```

```
// satellite's position with respect to Earth R_2

// Thomas Reppert 08/31/06

void SatvsEarth(double t, double* y, double* R_2)
{
// initialize position vector parameters
double x_2, y_2, z_2, r_2, E[4];

// compute Earth's heliocentric position vector at the current time t
EarthPos(t, E);

// satellite's position wrt Earth, km
// ----------------------------------

x_2 = y[1] - E[1];
y_2 = y[2] - E[2];
z_2 = y[3] - E[3];
r_2 = sqrt(pow(x_2,2) + pow(y_2,2) + pow(z_2,2));

R_2[1] = x_2;
R_2[2] = y_2;
R_2[3] = z_2;
R_2[4] = r_2;

return;
}


// SatvsMars:
// computes the satellite's position with respect to Mars at the specified
// time t during the Hohmann transfer

// input:
// initial offset angle between Earth and Mars
// current time t
// current satellite heliocentric state y

// output:
// satellite's position with respect to Mars R_3

// Thomas Reppert 08/31/06

void SatvsMars(double offset, double t, double* y, double* R_3)
{
```

```
// initialize position vector parameters
double x_3, y_3, z_3, r_3, vx_3, vy_3, vz_3, v_3, M[7];

// compute Mars's heliocentric position vector at the current time t
MarsPosVel(offset, t, M);

// satellite's position and velocity wrt Mars, km
// ----------------------------------------------

x_3 = y[1] - M[1];
y_3 = y[2] - M[2];
z_3 = y[3] - M[3];
r_3 = sqrt(pow(x_3,2) + pow(y_3,2) + pow(z_3,2));

vx_3 = y[4] - M[4];
vy_3 = y[5] - M[5];
vz_3 = y[6] - M[6];
v_3 = sqrt(pow(vx_3,2) + pow(vy_3,2) + pow(vz_3,2));

R_3[1] = x_3;   R_3[2] = y_3;
R_3[3] = z_3;   R_3[4] = r_3;
R_3[5] = vx_3;  R_3[6] = vy_3;
R_3[7] = vz_3;  R_3[8] = v_3;

return;
}


// EarthPos:
// calculates the current Earth position vector at the
// desired time t for a circular orbit about the sun

// input:
// current time t

// output:
// Earth's heliocentric position vector PosVec

// Thomas Reppert 08/31/06

void EarthPos(double t, double* PosVec)
{
double n_earth = sqrt(MU_SUN/pow(SMA_EARTH,3)); // Earth's mean
// orbit rate, rad/s
// Earth's current heliocentric position vector, km
// ------------------------------------------------

PosVec[1] = SMA_EARTH*cos(n_earth*t);
PosVec[2] = SMA_EARTH*sin(n_earth*t);
PosVec[3] =    0.0;

return;
}


// MarsPosVel:
// calculates Mars's position and velocity vectors at the specified
// time t for a circular orbit about the sun

// input:
// initial offset angle between Earth and Mars
// current time t

// output:
// Mars's heliocentric state vector PosAndVel

// Thomas Reppert 09/03/06

void MarsPosVel(double offset, double t, double* PosAndVel)
{
double n_mars = sqrt(MU_SUN/pow(SMA_MARS,3)); // Mars's mean orbit
// rate, rad/s

// Mars's current heliocentric position vector, km
// -----------------------------------------------
PosAndVel[1] = SMA_MARS*cos(n_mars*t + offset);
PosAndVel[2] = SMA_MARS*sin(n_mars*t + offset);
PosAndVel[3] =  0.0;

// Mars's current heliocentric velocity vector, km/s
// -------------------------------------------------
PosAndVel[4] = -SMA_MARS*n_mars*sin(n_mars*t + offset);
PosAndVel[5] =  SMA_MARS*n_mars*cos(n_mars*t + offset);
PosAndVel[6] =          0.0;

return;
}
```

ArrayMath.c

```
// Header file for performing mathematical operations on single- and
// multi-dimensional arrays.

// Thomas Reppert 09/03/06

#include <math.h>

// add_2_arrays:
// adds two arrays of the same length

void add_2_arrays(int length, double* array1, double* array2,
double* final)
{
  int i;
for(i = 0; i < length; ++i)
{
    final[i] = array1[i] + array2[i];
}
return;
}

// add_4_arrays:
// adds four arrays of the same length

void add_4_arrays(int length, double* array1, double* array2,
                  double* array3, double* array4, double* final)
{
  int i;
for(i = 0; i < length; ++i)
{
    final[i] = array1[i] + array2[i] + array3[i] + array4[i];
}
return;
}

// equal:
// assigns all values of one array to another array

void equal(int length, double* array1, double* array2)
{
int i;
for(i = 0; i < length; ++i)
{
```

```
array1[i] = array2[i];
}
return;
}

// mag:
// calculates the magnitude of a one-dimentional array

double mag(int length, double* array)
{
int i;
double final = 0.0;
for(i = 0; i < length; ++i)
{
final = final + pow(array[i], 2);
}
final = sqrt(final);
return final;
}

// MinValue:
// finds the minimum value and the corresponding index in the given array
// note: disregards the zeroth entry of the array

double MinValue(int length, int* index, double* array)
{
int i;
double min = 1.e10;
for(i = 1; i < length; ++i)
{
if(array[i] < min)
{
min = array[i];
index[1] = i;
}
}
return min;
}

// mtrx_tms_vctr:
// performs matrix multiplication between a multi-dimensional array and a
// single-dimensional array
// note: will need to change A matrix dimensions as necessary
// note: does not multiply the zeroth entries of the arrays
```

```
void mtrx_tms_vctr(int n_rows, int n_cols, double A[7][7], double* b,
                   double* final)
{
  int i, j;
for(i = 1; i < n_rows; ++i)
  {
    final[i] = 0.0;
    for(j = 1; j < n_cols; ++j)
    {
      final[i] += A[i][j] * b[j];
    }
  }
  return;
}

// scale_array:
// multiplies each component of an array by a scalar factor

void scale_array(int length, double scalar, double* array1, double* array2)
{
  int i;
  for(i = 0; i < length; ++i)
  {
    array2[i] = array1[i] * scalar;
  }
  return;
}
```

A.2 C Header Files

ArrayMath.h

```
#ifndef ARRAYMATH
#define ARRAYMATH

void add_2_arrays(int length, double *array1, double *array2, double *final);
void add_4_arrays(int length, double *array1, double *array2, double *array3,
                  double *array4, double *final);
void equal(int length, double *array1, double *array2);
double mag(int length, double* array);
double MinValue(int length, int* index, double* array);
void mtrx_tms_vctr(int n_rows, int n_cols, double A[7][7], double *b,
                   double *final);
void scale_array(int length, double scalar, double *array1, double *array2);
```

```
#endif
```

AstroConstants.h

```
/*
 *  AstroConstants.h
 *  OrbitalMotion
 *
 *  Created by Hanspeter Schaub on 06/19/05.
 *  Modified by Thomas Reppert on 09/01/06.
 *
 * Defines common astronomical constants using in the
 *  Keplerian 2-body problem.
 *
 */

#include <math.h>

#ifndef ASTROCONSTANTS
#define ASTROCONSTANTS

/*
  universal gravitational constant
  units are in km^3/s^2/kg
*/
#define G_UNIVERSIAL 6.67259e-20

/*
  astronomical unit in units of kilometers
*/
#define AU    149597870.691

/*
  common conversions
*/
#ifndef M_PI
#define M_PI 3.141592653589793
#endif

#ifndef D2R
#define D2R M_PI/180.0
#endif
#ifndef R2D
#define R2D 180.0/M_PI
```

```
#endif

/*
  Gravitational constants mu = G*m, where m is the planet of the
  attracting planet.  All units are km^3/s^2.
  values are obtained from http://ssd.jpl.nasa.gov/astro_constants.html
*/
#define MU_SUN       1.32712440018e11
#define MU_MERCURY   2.2032e4
#define MU_VENUS     3.2485859e5
#define MU_EARTH     3.986004418e5
#define MU_MOON      4.9027988e3
#define MU_MARS      4.28283e4
#define MU_JUPITER   1.2671277e8
#define MU_SATURN    3.79406e7
#define MU_URANUS    5.79455e6
#define MU_NEPTUNE   6.83653e6
#define MU_PLUTO     983.0

/*
  Planet information for major solar system bodies.  Units are in km.
  data taken from http://nssdc.gsfc.nasa.gov/planetary/planets.html
*/

/* Sun:  */
#define REQ_SUN 695000.0

/* Mercury: */
#define REQ_MERCURY 2439.7
#define J2_MERCURY 60.0e-6
#define SMA_MERCURY       0.38709893*AU
#define I_MERCURY         7.00487*D2R
#define E_MERCURY         0.20563069

/* Venus:   */
#define REQ_VENUS 6051.8
#define J2_VENUS          4.458e-6
#define SMA_VENUS         0.72333199*AU
#define I_VENUS           3.39471*D2R
#define E_VENUS           0.00677323

/* Earth:  */
#define REQ_EARTH 6378.14
#define SMA_EARTH         1.00000011*AU
#define I_EARTH           0.00005*D2R
```

```
#define E_EARTH           0.01671022

/* Moon:  */
#define REQ_MOON 1737.4
#define J2_MOON 202.7e-6
#define SMA_MOON          0.3844e6
#define E_MOON            0.0549

/* Mars:  */
#define REQ_MARS 3397.2
#define J2_MARS 1960.45e-6
#define SMA_MARS          1.52366231*AU
#define I_MARS            1.85061*D2R
#define E_MARS            0.09341233

/* Jupiter:  */
#define REQ_JUPITER 71492.0
#define J2_JUPITER 14736.e-6
#define SMA_JUPITER       5.20336301*AU
#define I_JUPITER         1.30530*D2R
#define E_JUPITER         0.04839266

/* Saturn:  */
#define REQ_SATURN 60268.
#define J2_SATURN 16298.e-6
#define SMA_SATURN        9.53707032*AU
#define I_SATURN          2.48446*D2R
#define E_SATURN          0.05415060

/* Uranus:   */
#define REQ_URANUS 25559.0
#define J2_URANUS 3343.43e-6
#define SMA_URANUS 19.19126393*AU
#define I_URANUS          0.76986*D2R
#define E_URANUS          0.04716771

/* Neptune:   */
#define REQ_NEPTUNE 24746.0
#define J2_NEPTUNE 3411.e-6
#define SMA_NEPTUNE 30.06896348*AU
#define I_NEPTUNE         1.76917*D2R
#define E_NEPTUNE         0.00858587

/* Pluto:  */
#define REQ_PLUTO 1137.0
```

```
#define SMA_PLUTO 39.48168677*AU
#define I_PLUTO 17.14175*D2R
#define E_PLUTO           0.24880766

/*
  Zonal gravitational harmonics Ji for the Earth
*/
#define J2  1082.63e-6
#define J3    -2.52e-6
#define J4    -1.61e-6
#define J5    -0.15e-6
#define J6     0.57e-6

#endif
```

MathConstants.h

```
/*
 *  MathConstants.h
 *
 *  Created by Thomas Reppert on 09/01/06.
 *
 * Defines common mathematical constants.
 *
 */

#ifndef MATHCONSTANTS
#define MATHCONSTANTS

#define PI 3.14159265358979

#endif
```

Bibliography

[1] William L. Hallauer Jr. *Introduction to Linear, Time-Invariant, Dynamic Systems.* William L. Hallauer, Jr., Blacksburg, VA, 2005.

[2] Werner Kohler and Lee Johnson. *Elementary Differential Equations with Boundary Value Problems.* Pearson Education, Inc., New York, NY, 2004.

[3] Tao Lin. *Class Notes: Introduction to Numerical Analysis.* Virginia Polytechnic Institute and State University, Blacksburg, VA, 2006.

[4] Thomas R. Reppert. Extending the patched-conic approximation to the restricted four-body problem. *AIAA Student Journal*, 44(2):1–11, May–Aug. 2006.

[5] Thomas R. Reppert. Extending the patched-conic approximation to the restricted four-body problem. Technical report, Virginia Polytechnic Institute and State University, Blacksburg, VA 24060, May 2006.

[6] Hanspeter Schaub and John L. Junkins. *Analytical Mechanics of Space Systems.* American Institute of Aeronautics and Astronautics, Inc., Reston, VA, 2003.

[7] James Stewart. *Calculus: Early Transcendentals.* Thomson Learning, Inc., Belmont, CA, 2003.