Table of Contents

HW3

Aero 557 Liam Hood

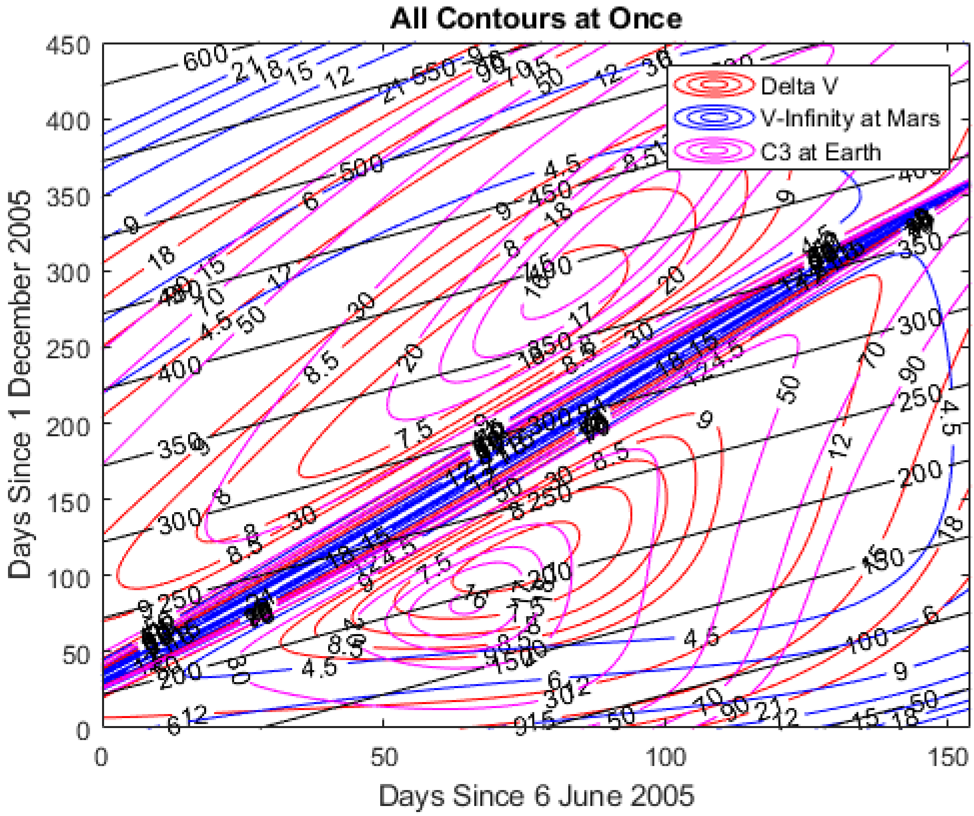
```
function Aero_557_HW3()
```

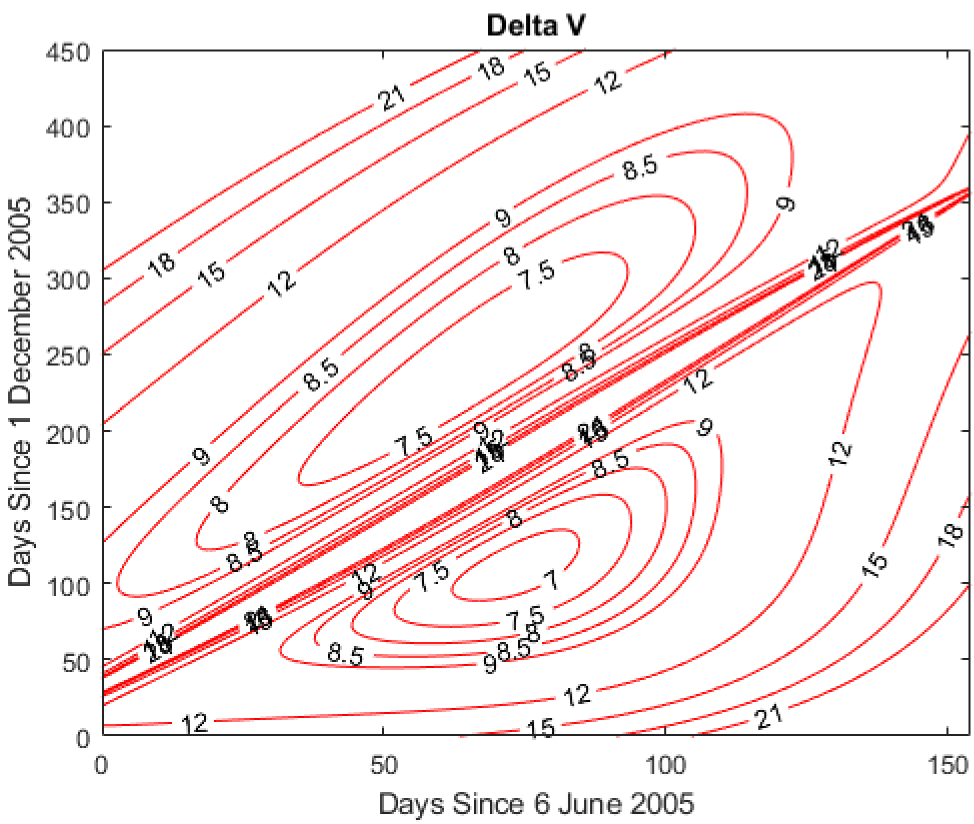
```
clear ; close all ; clc
```

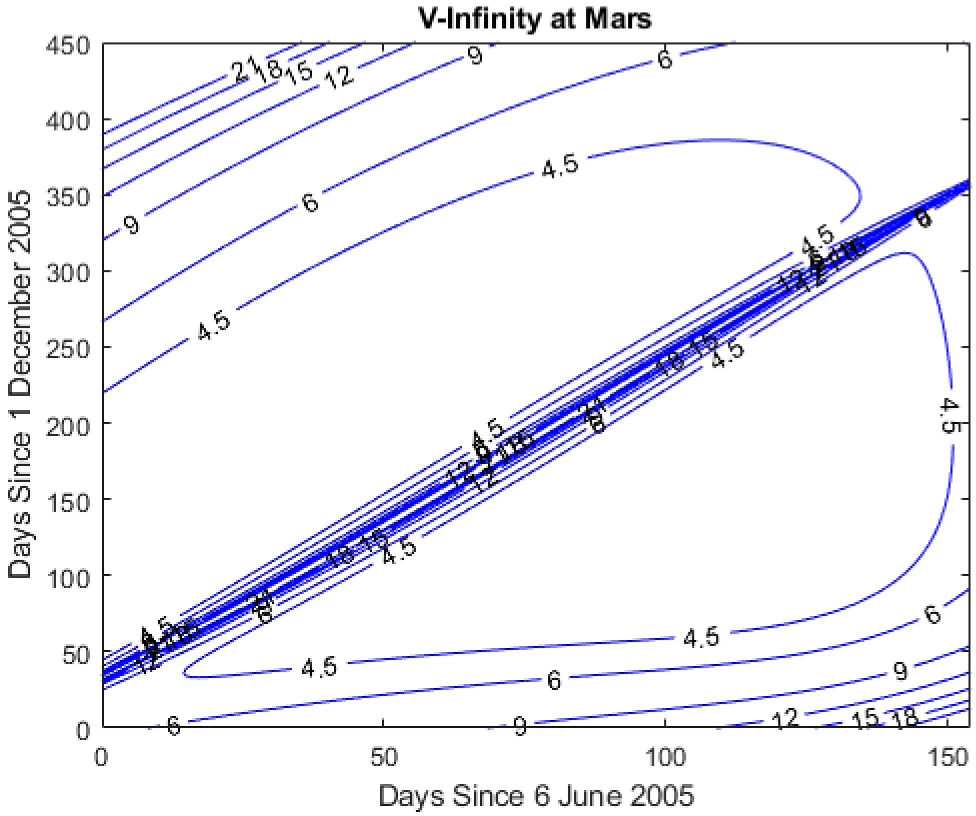
1

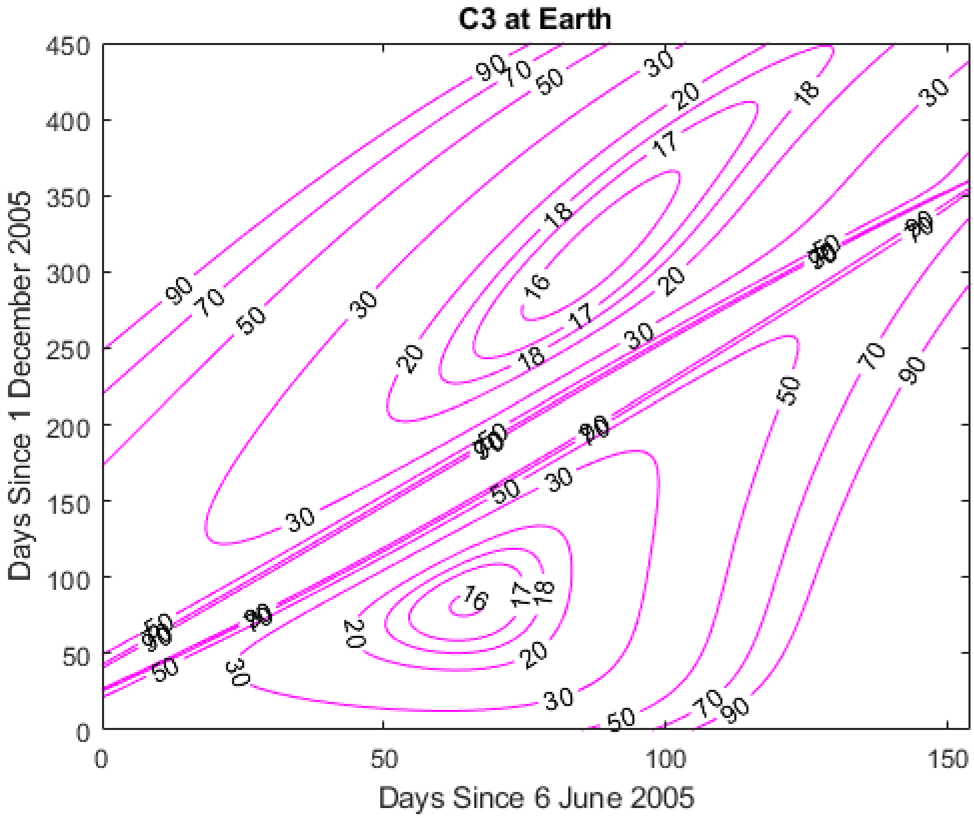
```
fprintf( 'Problem 1 \n' )
PorkChopPlots() ;
```

```
Problem 1
A mission trying to minimize delta v would leave on 19 Aug 2005
and arrive at Mars on 22 Mar 2006 using 6.786769 km/s of delta-v,
a C3 of 17.380316 km^2/s^2, and V-inf of 2.617799 km^2/s^2.
This minimum delta-v maneuver may not be achievable because
the launch C3 value needed, the V-infinite at capture, or
the time of transfer. Thinking about keeping the v-infinity
manageable at under 4.5 km/s and the time of flight under 150
leads us to need to do a type I transfer instead of the type II
used to minimize delta v. With these constraints means a mission
should leave on 18 Sep 2005 and arrive on 8 Feb 2006
with a delta v of 10.497436 km/s, C3 of 42.226208 km^2/s^2
and V-inf of 3.999267 km^2/s^2. This second transfer has less
schedule margin and may more easily exceed its time or
v-inf limits
```


All Contours at Once
Delta V
V-Infinity at Mars
C3 at Earth
Days Since 1 December 2005
Days Since 6 June 2005


Delta V
Days Since 1 December 2005
Days Since 6 June 2005

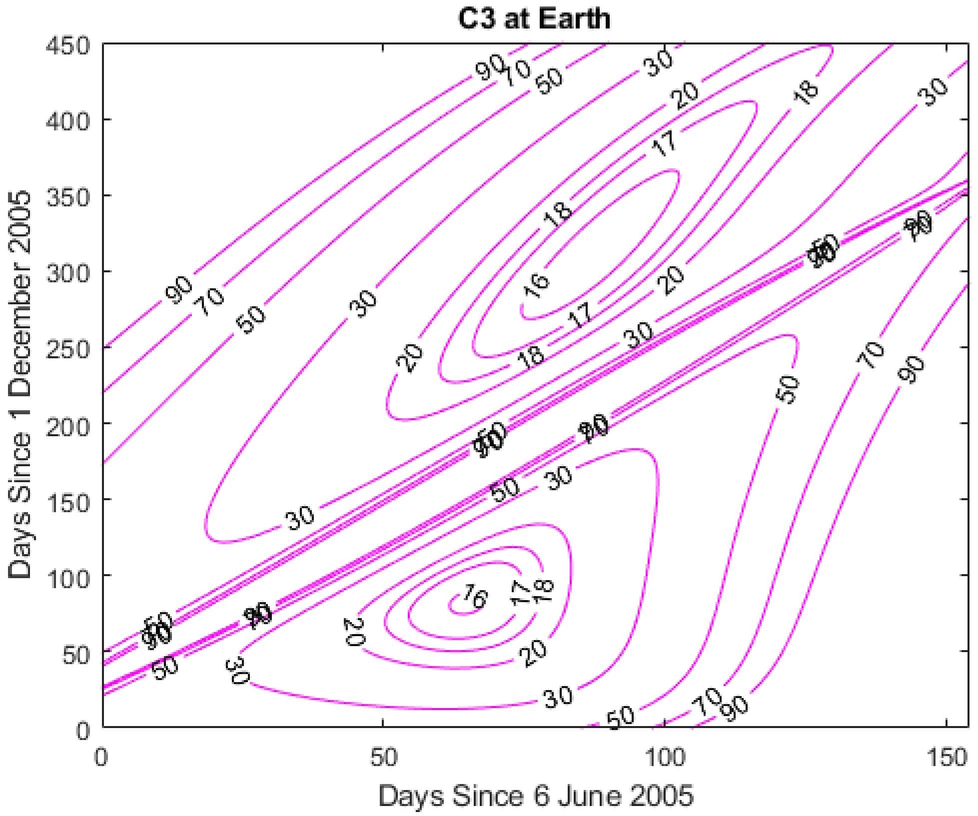

V-Infinity at Mars
Days Since 1 December 2005
Days Since 6 June 2005

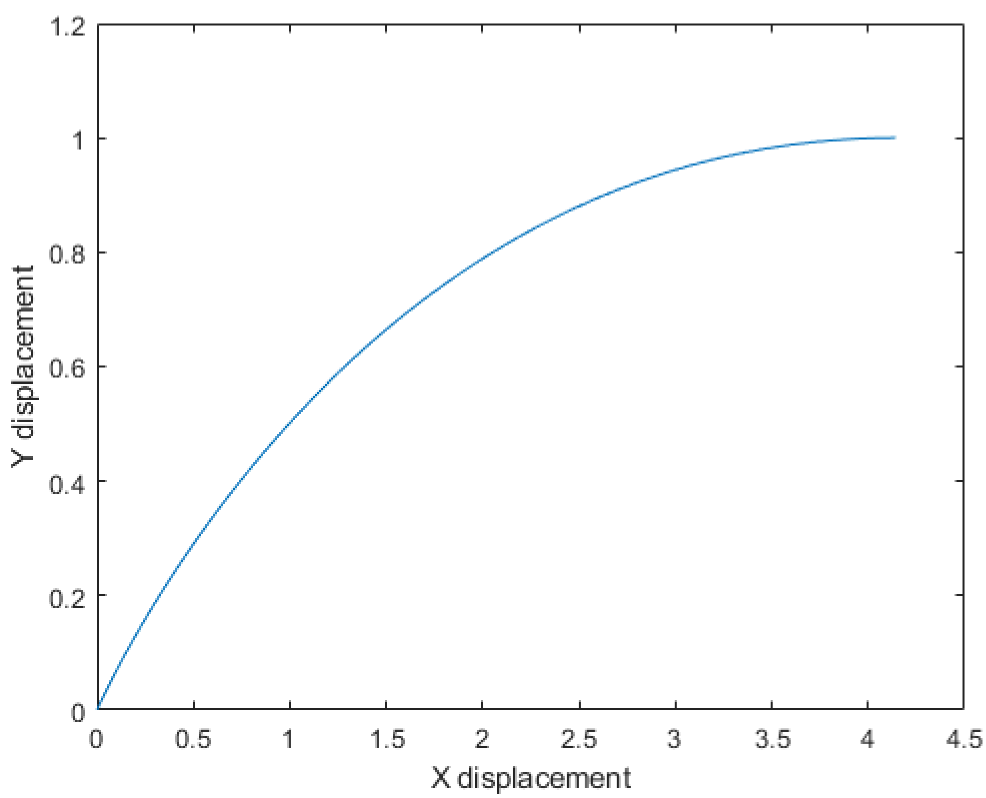

C3 at Earth
Days Since 1 December 2005
Days Since 6 June 2005

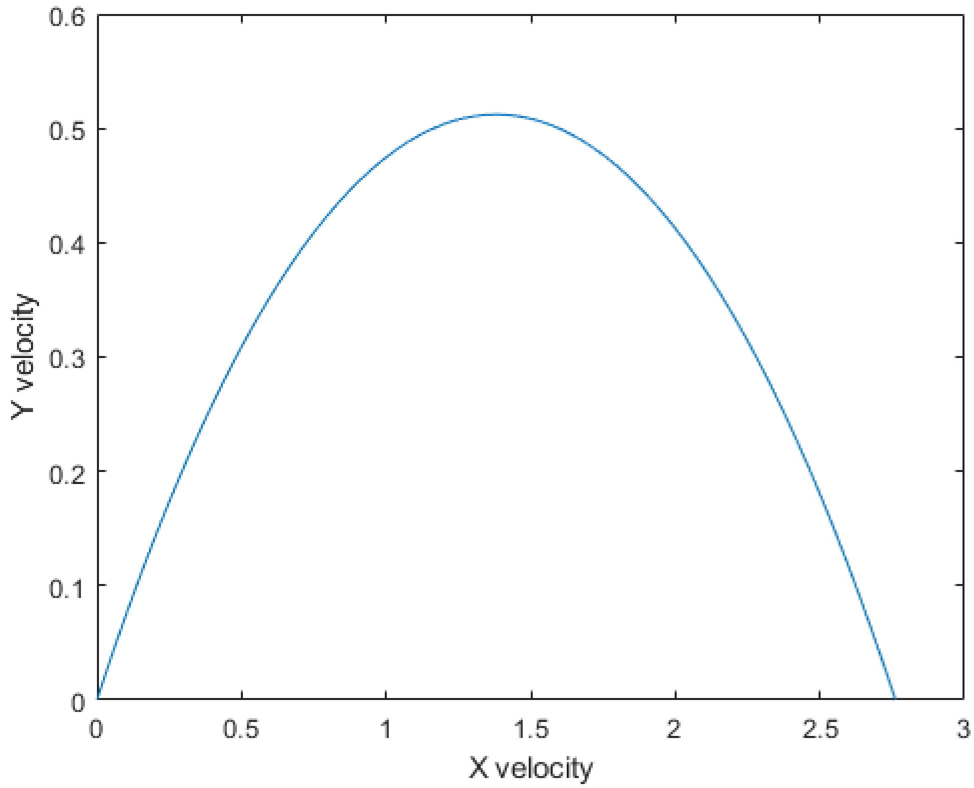
3

```
fprintf( '\nProblem 2 \n' )
HW3P3() ;
```

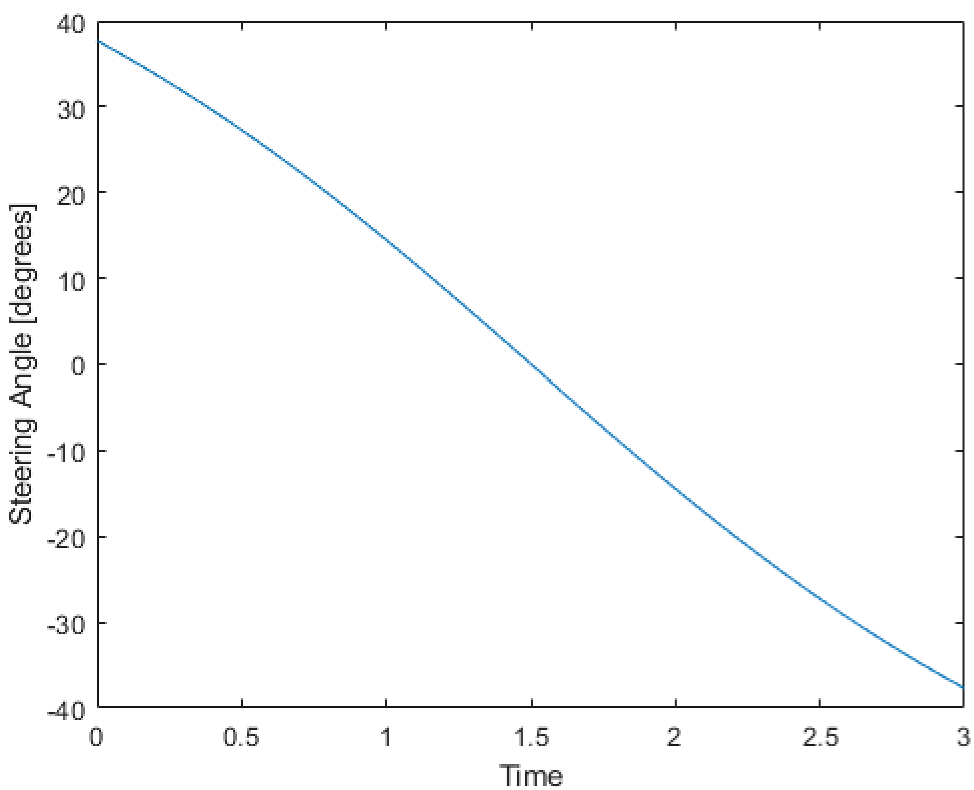
```
Problem 2

                                         Norm of      First-order
 Trust-region
 Iteration  Func-count     f(x)          step          optimality
 radius
     0          9         10.9179                           30
        1
     1         10         10.9179              1            30
        1
     2         19         8.12564           0.25          7.97
     0.25
     3         28        0.871144          0.625          2.88
    0.625
     4         37       0.0039103       0.406261         0.323
     1.56
     5         46     4.62778e-07      0.0198554       0.00328
     1.56
     6         55     4.38414e-14    0.000617351      8.92e-07
     1.56

Equation solved. The sum of squared function values, r = 4.384140e-14,
 is less than
sqrt(options.FunctionTolerance) = 1.000000e-04. The relative norm of
 the gradient of r,
8.917299e-07, is less than options.OptimalityTolerance = 1.000000e-06.

The final x position is 4.141683 and final y position is 1.000000
The final x velocity is 2.761122 and final y velocity is 0.000000
The steering angle begins at 37.703075 and ends at -37.703069
This makes sense because the steering angle is always forward
beginning with some vertical acceleration and then ending in
a position to eliminate all vertical velocity while still adding
to the horizontal velocity
```


C3 at Earth
Days Since 1 December 2005
Days Since 6 June 2005


Y displacement
X displacement



Y velocity
X velocity



Steering Angle [degrees]
Time

Work

```
function PorkChopPlots()
    mu = 1.327124e11 ;
    JDbegE = 2453528.0 ;
    JDendE = 2453682.0 ;
    JDbegM = 2453706.0 ;
    JDendM = 2454156.0 ;
    JDE = JDbegE:1:JDendE ;
    JDM = JDbegM:1:JDendM ;
    n = length( JDE ) ;
    m = length( JDM ) ;
    for ii = 1:n
        for jj = 1:m
    %          datiE = julian2date( JDE(ii) ) ;
    %          datiM = julian2date( JDM(jj) ) ;
            [~ , rEv(:,ii) , vEv(:,ii) , ~ ] =
AERO557planetcoe_and_sv( 3 , JDE(ii) ) ;
            [~ , rMv(:,jj) , vMv(:,jj) , ~ ] =
AERO557planetcoe_and_sv( 4 , JDM(jj) ) ;
            rE = rEv(:,ii) ;
            vE = vEv(:,ii) ;
            rM = rMv(:,jj) ;
            vM = vMv(:,jj) ;
            dt(ii,jj) = ( JDM(jj) - JDE(ii) )*86400 ;
            [ vEt , vMt ] = Lambert_Izzo( rE , rM , dt(ii,jj) ,
1 , mu ) ;
            dvE = norm( vEt - vE ) ;
            dvM = norm( vMt - vM ) ;
            dv(ii,jj) = dvE + dvM ;
            vinf(ii,jj) = dvM ;
            c3(ii,jj) = dvE^2 ;
            [ vEtL , vMtL ] = Lambert_Izzo( rE , rM , dt(ii,jj) ,
-1 , mu ) ;
            dvEL = norm( vEtL - vE ) ;
            dvML = norm( vMtL - vM ) ;
            dvL(ii,jj) = dvEL + dvML ;
            vinfL(ii,jj) = dvML ;
            c3L(ii,jj) = dvEL^2 ;
            if dvL(ii,jj) < dv(ii,jj)
                dv(ii,jj) = dvL(ii,jj) ;
                vinf(ii,jj) = vinfL(ii,jj) ;
                c3(ii,jj) = c3L(ii,jj) ;
            end

        end
    end
    levdv = [ 1:.5:9 , 12:3:21 ] ;
    levvinf = [ 4.5 , 6:3:21 ] ;
    levc3 = [ 0:1:18 , 20 , 30:20:100 ] ;
    levdt = 0:50:600 ;

    figure
```

```
        contour( JDE - JDbegE , JDM - JDbegM , dv' , levdv , 'r' ,
 'ShowText' , 'on'  )
        hold on
        contour( JDE - JDbegE , JDM - JDbegM , vinf' , levvinf , 'b'
, 'ShowText' , 'on' )
        contour( JDE - JDbegE , JDM - JDbegM , c3' , levc3 , 'm'
, 'ShowText' , 'on' )
        contour( JDE - JDbegE , JDM - JDbegM , dt'./86400 ,
levdt , 'k' , 'ShowText' , 'on' )
        title( 'All Contours at Once' )
        xlabel( 'Days Since 6 June 2005' )
        ylabel( 'Days Since 1 December 2005' )
        legend( 'Delta V' , 'V-Infinity at Mars' , 'C3 at Earth' )

        figure
        contour( JDE - JDbegE , JDM - JDbegM , dv' , levdv , 'r' ,
 'ShowText' , 'on'  )
        xlabel( 'Days Since 6 June 2005' )
        ylabel( 'Days Since 1 December 2005' )
        title( 'Delta V' )
        figure
        contour( JDE - JDbegE , JDM - JDbegM , vinf' , levvinf , 'b'
, 'ShowText' , 'on' )
        xlabel( 'Days Since 6 June 2005' )
        ylabel( 'Days Since 1 December 2005' )
        title( 'V-Infinity at Mars' )
        figure
        contour( JDE - JDbegE , JDM - JDbegM , c3' , levc3 , 'm'
, 'ShowText' , 'on' )
        xlabel( 'Days Since 6 June 2005' )
        ylabel( 'Days Since 1 December 2005' )
        title( 'C3 at Earth' )
%         figure
%         contour( JDE - JDbegE , JDM - JDbegM , dt'./86400 , levdt ,
 'k' , 'ShowText' , 'on' )
%         xlabel( 'Days Since 6 June 2005' )
%         ylabel( 'Days Since 1 December 2005' )
        % figure
        % hold on
        % plot3( rEv(1,:) , rEv(2,:) , rEv(3,:) )
        % plot3( rMv(1,:) , rMv(2,:) , rMv(3,:) )
        mindv = min( min( dv ) ) ;
        [ I , J ] = find( dv == mindv ) ;
        minDepart = julian2date( JDE(I) ) ;
        minArrive = julian2date( JDM(J) ) ;
        X = 105 ;
        Y = 70 ;
        realDepart = julian2date( JDE(X) ) ;
        realArrive = julian2date( JDM(Y) ) ;
        month = ["Jan" , "Feb" , "Mar" , "Apr" , "May" , "Jun" , "Jul"
, "Aug" , "Sep" , "Oct" , "Nov" , "Dec" ] ;
        fprintf( 'A mission trying to minimize delta v would leave on
%i %s %i \n' , minDepart(3) , month( minDepart(2) ) , minDepart(1) )
```

```
        fprintf( 'and arrive at Mars on %i %s %i using %f km/s of
delta-v, \n' , minArrive(3) , month( minArrive(2) ) , minArrive(1) ,
mindv )
        fprintf( 'a C3 of %f km^2/s^2, and V-inf of %f km^2/s^2. \n' ,
c3( I , J ) , vinf( I , J ) )
        fprintf( 'This minimum delta-v maneuver may not be achievable
because \n' )
        fprintf( 'the launch C3 value needed, the V-infinite at
capture, or \n' )
        fprintf( 'the time of transfer. Thinking about keeping the v-
infinity \n' )
        fprintf( 'manageable at under 4.5 km/s and the time of flight
under 150 \n' )
        fprintf( 'leads us to need to do a type I transfer instead of
the type II \n' )
        fprintf( 'used to minimize delta v. With these constraints
means a mission \n' )
        fprintf( 'should leave on %i %s %i and arrive on %i %s %i
\n' , realDepart(3) , month( realDepart(2) ) , realDepart(1) ,
realArrive(3) , month( realArrive(2) ) , realArrive(1))
        fprintf( 'with a delta v of %f km/s, C3 of %f km^2/s^2 \n' ,
dv( X,Y ) , c3( X,Y ) )
        fprintf( 'and V-inf of %f km^2/s^2. This second transfer has
less \n' , vinf( X , Y ) )
        fprintf( 'schedule margin and may more easily exceed its time
or \n' )
        fprintf( 'v-inf limits \n' )
    end

    function HW3P3()
        x = 0 ;
        y = 0 ;
        xdot = 0 ;
        ydot = 0 ;
        lambda = [ -1 ; -1 ; -1 ; -1 ] ;
        s0 = [ x ; y ; xdot ; ydot ; lambda ] ;
        ttrans = 3 ;

        opts = optimoptions( 'fsolve' , 'Display' , 'iter-detailed'
, 'FunctionTolerance' , 1e-8 , 'StepTolerance' , 1e-8 ) ; % ,
'Algorithm' , 'levenberg-marquardt'
        [ s , F ] = fsolve( @SteeringAngle1 , s0 , opts ) ;
        optsode = odeset( 'RelTol' , 1e-8 , 'AbsTol' , 1e-8 ) ;
        tspan = [ 0 , 3 ] ;
        [ tall , sall ] = ode45( @SteeringAngle1EOM , tspan , s ,
optsode ) ;
        for ii = 1:length( tall )
            bot = sqrt( s(7)^2 + s(8)^2 ) ;
            cosc(ii) = -sall(ii,7)/bot ;
        %     if cosc > 0
        %         cosc = -cosc ;
        %     end
            sinc(ii) = -sall(ii,8)/bot ;
        %     if sinc > 0
```

```
        %          sinc = -sinc ;
        %      end
            c(ii) = atan2d( sinc(ii) , cosc(ii) ) ;
        end
        figure
        plot( sall(:,1) , sall(:,2) )
        xlabel( 'X displacement' )
        ylabel( 'Y displacement' )
        figure
        plot( sall(:,3) , sall(:,4) )
        xlabel( 'X velocity' )
        ylabel( 'Y velocity' )
        figure
        plot( tall , c )
        xlabel( 'Time' )
        ylabel( 'Steering Angle [degrees]' )

        fprintf( 'The final x position is %f and final y position is
%f \n' , sall(end,1) , sall(end,2) )
        fprintf( 'The final x velocity is %f and final y velocity is
%f \n' , sall(end,3) , sall(end,4) )
        fprintf( 'The steering angle begins at %f and ends at %f \n' ,
c(1) , c(end) )
        fprintf( 'This makes sense because the steering angle is
always forward \n' )
        fprintf( 'beginning with some vertical acceleration and then
ending in \n' )
        fprintf( 'a position to eliminate all vertical velocity while
still adding \n' )
        fprintf( 'to the horizontal velocity \n' )

        function F = SteeringAngle1( s0 )
            tspan = [ 0 , 3 ] ;


            opts = odeset( 'RelTol' , 1e-10 , 'AbsTol' , 1e-10 ) ;
            [ t , s ] = ode45( @SteeringAngle1EOM , tspan , s0 ,
opts ) ;
            sf = s(end,:) ;

            % force intitial conditions
            F(1,1) = s0(1) ;
            F(2,1) = s0(2) ;
            F(3,1) = s0(3) ;
            F(4,1) = s0(4) ;

            % omega
            F(5,1) = sf(2) - 1 ;
            F(6,1) = sf(4) ;

            % lambda final
            F(7,1) = sf(5) ;
            F(8,1) = sf(7) + 1 ;
            % DONT NEED LAMBDAS EQUAL TO LITTLE OMEGAS
```

```
    %     w1 = s0(6) ;
    %     w2 = s0(6)*tspan(2) + s0(8) ;

    end

    function ds = SteeringAngle1EOM( t , s )
        bot = sqrt( s(7)^2 + s(8)^2 ) ;
        cosc = -s(7)/bot ;
        sinc = -s(8)/bot ;

        ds = zeros( 8 , 1 ) ;
        ds(1) = s(3) ;
        ds(2) = s(4) ;
        ds(3) = cosc ;
        ds(4) = sinc ;
        ds(5) = 0 ;
        ds(6) = 0 ;
        ds(7) = -s(5) ;
        ds(8) = -s(6) ;
    end
end
```

Functions

```
% function [coe, r, v, jd] = AERO557planetcoe_and_sv ...
% (planet_id, jd )
% % ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% %{
% This function calculates the orbital elements and the state
% vector of a planet from the date (year, month, day)
% and universal time (hour, minute, second).
% mu - gravitational parameter of the sun (km^3/s^2)
% deg - conversion factor between degrees and radians
% pi - 3.1415926...
% coe - vector of heliocentric orbital elements
% [h e RAAN inc w TA a w_hat L M E],
% where
% h = angular momentum (km^2/s)
% e = eccentricity
% RA = right ascension (deg)
% incl = inclination (deg)
% w = argument of perihelion (deg)
% TA = true anomaly (deg)
% a = semimajor axis (km)
% w_hat = longitude of perihelion ( = RA + w) (deg)
% Appendix D Page 85 of 101 10/27/09 9:07 AM
% L = mean longitude ( = w_hat + M) (deg)
% M = mean anomaly (deg)
% E = eccentric anomaly (deg)
% planet_id - planet identifier:
% 1 = Mercury
% 2 = Venus
% 3 = Earth
```

```
% 4 = Mars
% 5 = Jupiter
% 7 = Uranus
% 8 = Neptune
% 9 = Pluto
% year - range: 1901 - 2099
% month - range: 1 - 12
% day - range: 1 - 31
% hour - range: 0 - 23
% minute - range: 0 - 60
% second - range: 0 - 60
% j0 - Julian day number of the date at 0 hr UT
% ut - universal time in fractions of a day
% jd - julian day number of the date and time
% J2000_coe - row vector of J2000 orbital elements from Table 9.1
% rates - row vector of Julian centennial rates from Table 9.1
% t0 - Julian centuries between J2000 and jd
% elements - orbital elements at jd
% r - heliocentric position vector
% v - heliocentric velocity vector
% User M-functions required: J0, kepler_E, sv_from_coe
% User subfunctions required: planetary_elements, zero_to_360
% %}
% %
 --------------------------------------------------------------------
% mu = 1.327124e11; %km3/s2
% deg = pi/180;
% %...Equation 5.48:
% % jd = JDcalc(year,month,day,hour,minute,second);
%
% %...Equation 8.93a:
% t0 = (jd - 2451545)/36525;
%
% %...Equation 8.93b:
% %...Obtain the data for the selected planet:
% elements = AERO451planetary_elements2(planet_id, t0);
% a = elements(1);
% e = elements(2);
% %...Equation 2.71:
% h = sqrt(mu*a*(1 - e^2));
% %...Reduce the angular elements to within the range 0 - 360 degrees:
% inc = elements(3);
% RAAN = zero_to_360(elements(4));
% w_hat = zero_to_360(elements(5));
% L = zero_to_360(elements(6));
% w = zero_to_360(w_hat - RAAN);
% M = zero_to_360((L - w_hat));
% %...Algorithm 3.1 (for which M must be in radians)
% % %use an initial estimate E
% ecc=e;
% M = M*pi/180;
% if (M <pi)
%      E = M+ecc/2;
% else
```

```
%      E=M-ecc/2;
% end
% %
% % %set an error tolerance
%  tol = 1e-6;
%  nmax = 50;
% %
% % %Use Newton's to iterate
%  ratio = 1;
%  m=0;
%  count =1;
%  while (abs(ratio)>tol)&&(m<=nmax)
%      m=m+1;
%      [funcFE,funcFEdot]=FfuncE(E,M,ecc);
%      ratio = funcFE/funcFEdot;
%      E=E-ratio;
%      count = count+1;
%  end
%
% %...Equation 3.13 (converting the result to degrees):
% TA = zero_to_360...
% (2*atan(sqrt((1 + e)/(1 - e))*tan(E/2))/deg);
% coe = [h e RAAN inc w TA a w_hat L M E];
% %...Algorithm 4.5 (for which all angles must be in radians):
% [r,v]=...
%     COEStoRV2sun(h,ecc,inc,RAAN, w,TA);
%
% %% COEStoRV with sun
% function [rvect,vvect]=...
%     COEStoRV2sun(h,ecc,inc,raan, omega,theta)
%
% %compute r and v from COES
% mu = 1.327124e11; %km3/s2
%
% theta = theta*pi/180;
% omega = omega*pi/180;
% raan = raan*pi/180;
%
% %T = (2*pi/sqrt(mu))*a^(1.5);
%
% %energy = -mu/(2*a);
%
% %n = mu^(0.5)/a^(1.5);
%
% %p = a*(1-ecc^2);
%
% %h= (mu*p)^(1/2);
%
% %E = 2*atan(sqrt((1-ecc/1+ecc))*tan(theta/2));
%
% %book step
% rvectx = (h^2/mu)*(1/(1+ecc*cos(theta))).*[cos(theta);sin(theta);0];
% vvectx = (mu/h).*[-sin(theta);ecc+cos(theta);0];
%
```

```
% %matrix conversion back into geocentric
% term1 =[cos(omega) sin(omega) 0;...
%         -sin(omega) cos(omega) 0; 0 0 1];
% term2 =[1 0 0; 0 cosd(inc) sind(inc); 0 -sind(inc) cosd(inc)];
% term3 =[cos(raan) sin(raan) 0;...
%         -sin(raan) cos(raan) 0; 0 0 1];
%
% convmat = term1*term2*term3;
% %invconvmat = inv(convmat);
%
% rvect = convmat\rvectx;
% vvect = convmat\vvectx;
% end
%
%
% %% planetary elements
%
% function [planet_coes] = AERO451planetary_elements2(planet_id,T)
% % Planetary Ephemerides from Meeus (1991:202-204) and J2000.0
% % Output:
% % planet_coes
% % a = semimajor axis (km)
% % ecc = eccentricity
% % inc = inclination (degrees)
% % raan = right ascension of the ascending node (degrees)
% % w_hat = longitude of perihelion (degrees)
% % L = mean longitude (degrees)
%
% % Inputs:
% % planet_id - planet identifier:
% % 1 = Mercury
% % 2 = Venus
% % 3 = Earth
% % 4 = Mars
% % 5 = Jupiter
% % 6 = Saturn
% % 7 = Uranus
% % 8 = Neptune
%
% if planet_id == 1
%     a = 0.387098310; % AU but in km later
%     ecc = 0.20563175 + 0.000020406*T - 0.0000000284*T^2 -
 0.00000000017*T^3;
%     inc = 7.004986 - 0.0059516*T + 0.00000081*T^2 + 0.000000041*T^3;
 %degs
%     raan = 48.330893 - 0.1254229*T-0.00008833*T^2 - 0.000000196*T^3;
 %degs
%     w_hat = 77.456119 +0.1588643*T -0.00001343*T^2+0.000000039*T^3;
 %degs
%     L = 252.250906+149472.6746358*T-0.00000535*T^2+0.000000002*T^3;
 %degs
% elseif planet_id == 2
%     a = 0.723329820; % AU
```

```
%     ecc = 0.00677188 - 0.000047766*T + 0.000000097*T^2 +
 0.00000000044*T^3;
%     inc = 3.394662 - 0.0008568*T - 0.00003244*T^2 + 0.000000010*T^3;
 %degs
%     raan = 76.679920 - 0.2780080*T-0.00014256*T^2 - 0.000000198*T^3;
 %degs
%     w_hat = 131.563707 +0.0048646*T -0.00138232*T^2-0.000005332*T^3;
 %degs
%     L = 181.979801+58517.8156760*T+0.00000165*T^2-0.000000002*T^3;
 %degs
% elseif planet_id == 3
%     a = 1.000001018; % AU
%     ecc = 0.01670862 - 0.000042037*T - 0.0000001236*T^2 +
 0.00000000004*T^3;
%     inc = 0.0000000 + 0.0130546*T - 0.00000931*T^2 -
 0.000000034*T^3; %degs
%     raan = 0.0; %degs
%     w_hat = 102.937348 + 0.3225557*T + 0.00015026*T^2 +
 0.000000478*T^3; %degs
%     L = 100.466449 + 35999.372851*T - 0.00000568*T^2 +
 0.000000000*T^3; %degs
% elseif planet_id == 4
%     a = 1.523679342; % AU
%     ecc = 0.09340062 + 0.000090483*T - 0.00000000806*T^2 -
 0.00000000035*T^3;
%     inc = 1.849726 - 0.0081479*T - 0.00002255*T^2 - 0.000000027*T^3;
 %degs
%     raan = 49.558093 - 0.2949846*T-0.00063993*T^2 - 0.000002143*T^3;
 %degs
%     w_hat = 336.060234 +0.4438898*T -0.00017321*T^2+0.000000300*T^3;
 %degs
%     L = 355.433275+19140.2993313*T+0.00000261*T^2-0.000000003*T^3;
 %degs
% elseif planet_id == 5
%     a = 5.202603191 + 0.0000001913*T; % AU
%     ecc = 0.04849485+0.000163244*T - 0.0000004719*T^2 +
 0.00000000197*T^3;
%     inc = 1.303270 - 0.0019872*T + 0.00003318*T^2 + 0.000000092*T^3;
 %degs
%     raan = 100.464441 + 0.1766828*T+0.00090387*T^2 -
 0.000007032*T^3; %degs
%     w_hat = 14.331309 +0.2155525*T +0.00072252*T^2-0.000004590*T^3;
 %degs
%     L = 34.351484+3034.9056746*T-0.00008501*T^2+0.000000004*T^3;
 %degs
% elseif planet_id == 6
%     a = 9.5549009596 - 0.0000021389*T; % AU
%     ecc = 0.05550862 - 0.000346818*T -0.0000006456*T^2 +
 0.00000000338*T^3;
%     inc = 2.488878 + 0.0025515*T - 0.00004903*T^2 + 0.000000018*T^3;
 %degs
%     raan = 113.665524 - 0.2566649*T-0.00018345*T^2 +
 0.000000357*T^3; %degs
```

```
%     w_hat = 93.056787 +0.5665496*T +0.00052809*T^2-0.000004882*T^3;
 %degs
%     L = 50.077471+1222.1137943*T+0.00021004*T^2-0.000000019*T^3;
 %degs
% elseif planet_id == 7
%     a = 19.218446062-0.0000000372*T+0.00000000098*T^2; % AU
%     ecc = 0.04629590 - 0.000027337*T + 0.0000000790*T^2 +
 0.00000000025*T^3;
%     inc = 0.773196 - 0.0016869*T + 0.00000349*T^2 +
 0.00000000016*T^3; %degs
%     raan = 74.005947 + 0.0741461*T+0.00040540*T^2 +0.000000104*T^3;
 %degs
%     w_hat = 173.005159 +0.0893206*T -0.00009470*T^2+0.000000413*T^3;
 %degs
%     L = 314.055005+428.4669983*T-0.00000486*T^2-0.000000006*T^3;
 %degs
% elseif planet_id == 8
%     a = 30.110386869-0.0000001663*T+0.00000000069*T^2; % AU
%     ecc = 0.00898809 + 0.000006408*T -0.0000000008*T^2;
%     inc = 1.769952 +0.0002557*T +0.00000023*T^2 -0.0000000000*T^3;
 %degs
%     raan = 131.784057 - 0.0061651*T-0.00000219*T^2 -
 0.000000078*T^3; %degs
%     w_hat = 48.123691 +0.0291587*T +0.00007051*T^2-0.000000000*T^3;
 %degs
%     L = 304.348665+218.4862002*T+0.00000059*T^2-0.000000002*T^3;
 %degs
% end
%
% planet_coes = [a;ecc;inc;raan;w_hat;L];
% %Convert to km:
% au = 149597870;
% planet_coes(1) = planet_coes(1)*au;
% end
%
% %% JD calc
%
% function JD1 = JDcalc(Y,M,D,UThr,UTmin,UTsec)
%
%
% UT=UThr+UTmin/60+UTsec/60;
%
%
% JD1 = 367*Y - floor((7*(Y+floor((M+9)/12)))/4) + ...
%     floor((275*M)/9) + D + 1721013.5 + UT/24;
% end
%
% %% FfuncE
% function [funcF,funcFdot] =FfuncE(E,M,ecc)
%     funcF = E -ecc*sin(E)-M;
%     funcFdot = 1-ecc*cos(E);
% end
%
% %% zero to 360
```

```
% % ~~~~~~~~~~~~~~~~~~~~~~~~~~~
% function y = zero_to_360(x)
% % ~~~~~~~~~~~~~~~~~~~~~~~~~~~
% %{
% This function reduces an angle to lie in the range 0 - 360 degrees.
% x - the original angle in degrees
% y - the angle reduced to the range 0 - 360 degrees
% %}
% % ---------------------------
% if x >= 360
% x = x - fix(x/360)*360;
% elseif x < 0
% x = x - (fix(x/360) - 1)*360;
% end
% y = x;
% end %zero_to_360
%
% end %main
%
% function [ V1 , V2 ] = Lambert_Izzo( r1 , r2 , tf , tm , mu )
%     [ V1r , V2r , ~ , ~ ] = Lambert_IzzoFun( r1' , r2' , tm*tf/
(24*60*60) , 0 , mu ) ;
%     V1 = V1r' ;
%     V2 = V2r' ;
%     function [V1,...
%               V2, ...
%               extremal_distances,...
%               exitflag] = Lambert_IzzoFun(r1vec,...
%                                           r2vec,...
%                                           tf,...
%                                           m,...
%                                           muC) %#coder
%     % original documentation:
%     %{
%      This routine implements a new algorithm that solves Lambert's
 problem. The
%      algorithm has two major characteristics that makes it favorable
 to other
%      existing ones.
%      1) It describes the generic orbit solution of the boundary
 condition
%      problem through the variable X=log(1+cos(alpha/2)). By doing so
 the
%      graph of the time of flight become defined in the entire real
 axis and
%      resembles a straight line. Convergence is granted within few
 iterations
%      for all the possible geometries (except, of course, when the
 transfer
%      angle is zero). When multiple revolutions are considered the
 variable is
%      X=tan(cos(alpha/2)*pi/2).
%      2) Once the orbit has been determined in the plane, this
 routine
```

```
%      evaluates the velocity vectors at the two points in a way that
 is not
%      singular for the transfer angle approaching to pi (Lagrange
 coefficient
%      based methods are numerically not well suited for this
 purpose).
%      As a result Lambert's problem is solved (with multiple
 revolutions
%      being accounted for) with the same computational effort for all
%      possible geometries. The case of near 180 transfers is also
 solved
%      efficiently.
%       We note here that even when the transfer angle is exactly
 equal to pi
%      the algorithm does solve the problem in the plane (it finds X),
 but it
%      is not able to evaluate the plane in which the orbit lies. A
 solution
%      to this would be to provide the direction of the plane
 containing the
%      transfer orbit from outside. This has not been implemented in
 this
%      routine since such a direction would depend on which
 application the
%      transfer is going to be used in.
%      please report bugs to dario.izzo@esa.int
%     %}
%     % adjusted documentation:
%     %{
%      By default, the short-way solution is computed. The long way
 solution
%      may be requested by giving a negative value to the
 corresponding
%      time-of-flight [tf].
%      For problems with |m| > 0, there are generally two solutions.
 By
%      default, the right branch solution will be returned. The left
 branch
%      may be requested by giving a negative value to the
 corresponding
%      number of complete revolutions [m].
%     %}
%     % Authors
%     % .-`-.-`-.-`-.-`-.-`-.-`-.-`-.-`-.-`-.-`-.-`-.-`-.-`-.-`-.-`-.
%     % Name       : Dr. Dario Izzo
%     % E-mail     : dario.izzo@esa.int
%     % Affiliation: ESA / Advanced Concepts Team (ACT)
%     % Made more readible and optimized for speed by Rody P.S.
 Oldenhuis
%     % Code available in MGA.M on   http://www.esa.int/gsp/ACT/inf/
op/globopt.htm
%     % last edited 12/Dec/2009
%     % ADJUSTED FOR EML-COMPILATION 24/Dec/2009
%         % initial values
```

```
%          tol = 1e-14;    bad = false;     days = 86400;
%          % work with non-dimensional units
%          r1 = sqrt(r1vec*r1vec.');  r1vec = r1vec/r1;
%          V = sqrt(muC/r1);          r2vec = r2vec/r1;
%          T = r1/V;                  tf    = tf*days/T; % also
 transform to seconds
%          % relevant geometry parameters (non dimensional)
%          mr2vec = sqrt(r2vec*r2vec.');
%          % make 100% sure it's in (-1 <= dth <= +1)
%          dth = acos( max(-1, min(1, (r1vec*r2vec.')/mr2vec)) );
%          % decide whether to use the left or right branch (for multi-
revolution
%          % problems), and the long- or short way
%          leftbranch = sign(m);   longway = sign(tf);
%          m = abs(m);             tf = abs(tf);
%          if (longway < 0), dth = 2*pi - dth; end
%          % derived quantities
%          c      = sqrt(1 + mr2vec^2 - 2*mr2vec*cos(dth)); % non-
dimensional chord
%          s      = (1 + mr2vec + c)/2;                     % non-
dimensional semi-perimeter
%          a_min  = s/2;                                    % minimum
 energy ellipse semi major axis
%          Lambda = sqrt(mr2vec)*cos(dth/2)/s;              % lambda
 parameter (from BATTIN's book)
%          crossprd = [r1vec(2)*r2vec(3) - r1vec(3)*r2vec(2),...
%                      r1vec(3)*r2vec(1) - r1vec(1)*r2vec(3),...% non-
dimensional normal vectors
%                      r1vec(1)*r2vec(2) - r1vec(2)*r2vec(1)];
%          mcr       = sqrt(crossprd*crossprd.');           % magnitues
 thereof
%          nrmunit   = crossprd/mcr;                        % unit
 vector thereof
%          % Initial values
%          % ---------------------------------------------------------
%          % ELMEX requires this variable to be declared OUTSIDE the
 IF-statement
%          logt = log(tf); % avoid re-computing the same value
%          % single revolution (1 solution)
%          if (m == 0)
%              % initial values
%              inn1 = -0.5233;      % first initial guess
%              inn2 = +0.5233;      % second initial guess
%              x1   = log(1 + inn1);% transformed first initial guess
%              x2   = log(1 + inn2);% transformed first second guess
%              % multiple revolutions (0, 1 or 2 solutions)
%              % the returned soltuion depends on the sign of [m]
%          else
%              % select initial values
%              if (leftbranch < 0)
%                  inn1 = -0.5234; % first initial guess, left branch
%                  inn2 = -0.2234; % second initial guess, left branch
%              else
%                  inn1 = +0.7234; % first initial guess, right branch
```

```
%                 inn2 = +0.5234; % second initial guess, right branch
%             end
%             x1 = tan(inn1*pi/2);% transformed first initial guess
%             x2 = tan(inn2*pi/2);% transformed first second guess
%         end
%         % since (inn1, inn2) < 0, initial estimate is always ellipse
%         xx   = [inn1, inn2];  aa = a_min./(1 - xx.^2);
%         bbeta = longway * 2*asin(sqrt((s-c)/2./aa));
%         % make 100.4% sure it's in (-1 <= xx <= +1)
%         aalfa = 2*acos(  max(-1, min(1, xx)) );
%         % evaluate the time of flight via Lagrange expression
%         y12  = aa.*sqrt(aa).*((aalfa - sin(aalfa)) - (bbeta-
sin(bbeta)) + 2*pi*m);
%         % initial estimates for y
%         if m == 0
%             y1 = log(y12(1)) - logt;
%             y2 = log(y12(2)) - logt;
%         else
%             y1 = y12(1) - tf;
%             y2 = y12(2) - tf;
%         end
%         % Solve for x
%         % ---------------------------------------------------------
%         % Newton-Raphson iterations
%         % NOTE - the number of iterations will go to infinity in
 case
%         % m > 0  and there is no solution. Start the other routine
 in
%         % that case
%         err = inf;  iterations = 0; xnew = 0;
%         while (err > tol)
%             % increment number of iterations
%             iterations = iterations + 1;
%             % new x
%             xnew = (x1*y2 - y1*x2) / (y2-y1);
%             % copy-pasted code (for performance)
%             if m == 0, x = exp(xnew) - 1; else x = atan(xnew)*2/pi;
 end
%             a = a_min/(1 - x^2);
%             if (x < 1) % ellipse
%                 beta = longway * 2*asin(sqrt((s-c)/2/a));
%                 % make 100.4% sure it's in (-1 <= xx <= +1)
%                 alfa = 2*acos( max(-1, min(1, x)) );
%             else % hyperbola
%                 alfa = 2*acosh(x);
%                 beta = longway * 2*asinh(sqrt((s-c)/(-2*a)));
%             end
%             % evaluate the time of flight via Lagrange expression
%             if (a > 0)
%                 tof = a*sqrt(a)*((alfa - sin(alfa)) - (beta-
sin(beta)) + 2*pi*m);
%             else
%                 tof = -a*sqrt(-a)*((sinh(alfa) - alfa) - (sinh(beta)
 - beta));
```

```
%               end
%               % new value of y
%               if m ==0, ynew = log(tof) - logt; else ynew = tof - tf;
 end
%               % save previous and current values for the next
 iterarion
%               % (prevents getting stuck between two values)
%               x1 = x2;  x2 = xnew;
%               y1 = y2;  y2 = ynew;
%               % update error
%               err = abs(x1 - xnew);
%               % escape clause
%               if (iterations > 15), bad = true; break; end
%           end
%           % If the Newton-Raphson scheme failed, try to solve the
 problem
%           % with the other Lambert targeter.
%           if bad
%               % NOTE: use the original, UN-normalized quantities
%               [V1, V2, extremal_distances, exitflag] = ...
%                   lambert_LancasterBlanchard(r1vec*r1, r2vec*r1,
 longway*tf*T, leftbranch*m, muC);
%               return
%           end
%           % convert converged value of x
%           if m==0, x = exp(xnew) - 1; else x = atan(xnew)*2/pi; end
%           %{
%             The solution has been evaluated in terms of log(x+1) or
 tan(x*pi/2), we
%             now need the conic. As for transfer angles near to pi the
 Lagrange-
%             coefficients technique goes singular (dg approaches a
 zero/zero that is
%             numerically bad) we here use a different technique for
 those cases. When
%             the transfer angle is exactly equal to pi, then the ih
 unit vector is not
%             determined. The remaining equations, though, are still
 valid.
%           %}
%           % Solution for the semi-major axis
%           a = a_min/(1-x^2);
%           % Calculate psi
%           if (x < 1) % ellipse
%               beta = longway * 2*asin(sqrt((s-c)/2/a));
%               % make 100.4% sure it's in (-1 <= xx <= +1)
%               alfa = 2*acos( max(-1, min(1, x)) );
%               psi  = (alfa-beta)/2;
%               eta2 = 2*a*sin(psi)^2/s;
%               eta  = sqrt(eta2);
%           else       % hyperbola
%               beta = longway * 2*asinh(sqrt((c-s)/2/a));
%               alfa = 2*acosh(x);
%               psi  = (alfa-beta)/2;
```

```
%             eta2 = -2*a*sinh(psi)^2/s;
%             eta  = sqrt(eta2);
%         end
%         % unit of the normalized normal vector
%         ih = longway * nrmunit;
%         % unit vector for normalized [r2vec]
%         r2n = r2vec/mr2vec;
%         % cross-products
%         % don't use cross() (emlmex() would try to compile it, and
 this way it
%         % also does not create any additional overhead)
%         crsprd1 = [ih(2)*r1vec(3)-ih(3)*r1vec(2),...
%                    ih(3)*r1vec(1)-ih(1)*r1vec(3),...
%                    ih(1)*r1vec(2)-ih(2)*r1vec(1)];
%         crsprd2 = [ih(2)*r2n(3)-ih(3)*r2n(2),...
%                    ih(3)*r2n(1)-ih(1)*r2n(3),...
%                    ih(1)*r2n(2)-ih(2)*r2n(1)];
%         % radial and tangential directions for departure velocity
%         Vr1 = 1/eta/sqrt(a_min) * (2*Lambda*a_min - Lambda - x*eta);
%         Vt1 = sqrt(mr2vec/a_min/eta2 * sin(dth/2)^2);
%         % radial and tangential directions for arrival velocity
%         Vt2 = Vt1/mr2vec;
%         Vr2 = (Vt1 - Vt2)/tan(dth/2) - Vr1;
%         % terminal velocities
%         V1 = (Vr1*r1vec + Vt1*crsprd1)*V;
%         V2 = (Vr2*r2n + Vt2*crsprd2)*V;
%         % exitflag
%         exitflag = 1; % (success)
%         % also compute minimum distance to central body
%         % NOTE: use un-transformed vectors again!
%         extremal_distances = ...
%             minmax_distances(r1vec*r1, r1, r2vec*r1, mr2vec*r1, dth,
 a*r1, V1, V2, m, muC);
%     end
%     %
 -----------------------------------------------------------------
%     % Lancaster & Blanchard version, with improvements by Gooding
%     % Very reliable, moderately fast for both simple and complicated
 cases
%     %
 -----------------------------------------------------------------
%     function [V1,...
%               V2,...
%               extremal_distances,...
%               exitflag] = lambert_LancasterBlanchard(r1vec,...
%                                                      r2vec,...
%                                                      tf,...
%                                                      m,...
%                                                      muC) %#coder
%     %{
%     LAMBERT_LANCASTERBLANCHARD       High-Thrust Lambert-targeter
%     lambert_LancasterBlanchard() uses the method developed by
%     Lancaster & Blancard, as described in their 1969 paper. Initial
 values,
```

```
%     and several details of the procedure, are provided by R.H.
 Gooding,
%     as described in his 1990 paper.
%     %}
%     % Please report bugs and inquiries to:
%     %
%     % Name       : Rody P.S. Oldenhuis
%     % E-mail     : oldenhuis@gmail.com
%     % Licence    : 2-clause BSD (see License.txt)
%     % If you find this work useful, please consider a donation:
%     % https://www.paypal.me/RodyO/3.5
%         % ADJUSTED FOR EML-COMPILATION 29/Sep/2009
%         % manipulate input
%         tol     = 1e-12;                            % optimum for
 numerical noise v.s. actual precision
%         r1      = sqrt(r1vec*r1vec.');              % magnitude of
 r1vec
%         r2      = sqrt(r2vec*r2vec.');              % magnitude of
 r2vec
%         r1unit  = r1vec/r1;                         % unit vector of
 r1vec
%         r2unit  = r2vec/r2;                         % unit vector of
 r2vec
%         crsprod = cross(r1vec, r2vec, 2);           % cross product
 of r1vec and r2vec
%         mcrsprd = sqrt(crsprod*crsprod.');          % magnitude of
 that cross product
%         th1unit = cross(crsprod/mcrsprd, r1unit);   % unit vectors
 in the tangential-directions
%         th2unit = cross(crsprod/mcrsprd, r2unit);
%         % make 100.4% sure it's in (-1 <= x <= +1)
%         dth = acos( max(-1, min(1, (r1vec*r2vec.')/r1/r2)) ); % turn
 angle
%         % if the long way was selected, the turn-angle must be
 negative
%         % to take care of the direction of final velocity
%         longway = sign(tf); tf = abs(tf);
%         if (longway < 0), dth = dth-2*pi; end
%         % left-branch
%         leftbranch = sign(m); m = abs(m);
%         % define constants
%         c  = sqrt(r1^2 + r2^2 - 2*r1*r2*cos(dth));
%         s  = (r1 + r2 + c) / 2;
%         T  = sqrt(8*muC/s^3) * tf;
%         q  = sqrt(r1*r2)/s * cos(dth/2);
%         % general formulae for the initial values (Gooding)
%         % -------------------------------------------------
%         % some initial values
%         T0  = LancasterBlanchard(0, q, m);
%         Td  = T0 - T;
%         phr = mod(2*atan2(1 - q^2, 2*q), 2*pi);
%         % initial output is pessimistic
%         V1 = NaN(1,3);    V2 = V1;    extremal_distances = [NaN,
 NaN];
```

```
%         % single-revolution case
%         if (m == 0)
%             x01 = T0*Td/4/T;
%             if (Td > 0)
%                 x0 = x01;
%             else
%                 x01 = Td/(4 - Td);
%                 x02 = -sqrt( -Td/(T+T0/2) );
%                 W   = x01 + 1.7*sqrt(2 - phr/pi);
%                 if (W >= 0)
%                     x03 = x01;
%                 else
%                     x03 = x01 + (-W).^(1/16).*(x02 - x01);
%                 end
%                 lambda = 1 + x03*(1 + x01)/2 - 0.03*x03^2*sqrt(1 +
 x01);
%                 x0 = lambda*x03;
%             end
%             % this estimate might not give a solution
%             if (x0 < -1), exitflag = -1; return; end
%         % multi-revolution case
%         else
%             % determine minimum Tp(x)
%             xMpi = 4/(3*pi*(2*m + 1));
%             if (phr < pi)
%                 xM0 = xMpi*(phr/pi)^(1/8);
%             elseif (phr > pi)
%                 xM0 = xMpi*(2 - (2 - phr/pi)^(1/8));
%             % EMLMEX requires this one
%             else
%                 xM0 = 0;
%             end
%             % use Halley's method
%             xM = xM0;  Tp = inf;  iterations = 0;
%             while abs(Tp) > tol
%                 % iterations
%                 iterations = iterations + 1;
%                 % compute first three derivatives
%                 [dummy, Tp, Tpp, Tppp] = LancasterBlanchard(xM, q,
 m);%#ok
%                 % new value of xM
%                 xMp = xM;
%                 xM  = xM - 2*Tp.*Tpp ./ (2*Tpp.^2 - Tp.*Tppp);
%                 % escape clause
%                 if mod(iterations, 7), xM = (xMp+xM)/2; end
%                 % the method might fail. Exit in that case
%                 if (iterations > 25), exitflag = -2; return; end
%             end
%             % xM should be elliptic (-1 < x < 1)
%             % (this should be impossible to go wrong)
%             if (xM < -1) || (xM > 1), exitflag = -1; return; end
%             % corresponding time
%             TM = LancasterBlanchard(xM, q, m);
%             % T should lie above the minimum T
```

```
%              if (TM > T), exitflag = -1; return; end
%              % find two initial values for second solution (again
 with lambda-type patch)
%              %
 ---------------------------------------------------------------------------
%              % some initial values
%              TmTM  = T - TM;   T0mTM = T0 - TM;
%              [dummy, Tp, Tpp] = LancasterBlanchard(xM, q, m);%#ok
%              % first estimate (only if m > 0)
%              if leftbranch > 0
%                  x   = sqrt( TmTM / (Tpp/2 + TmTM/(1-xM)^2) );
%                  W   = xM + x;
%                  W   = 4*W/(4 + TmTM) + (1 - W)^2;
%                  x0  = x*(1 - (1 + m + (dth - 1/2)) / ...
%                      (1 + 0.15*m)*x*(W/2 + 0.03*x*sqrt(W))) + xM;
%                  % first estimate might not be able to yield possible
 solution
%                  if (x0 > 1), exitflag = -1; return; end
%              % second estimate (only if m > 0)
%              else
%                  if (Td > 0)
%                      x0 = xM - sqrt(TM/(Tpp/2 - TmTM*(Tpp/2/T0mTM -
 1/xM^2)));
%                  else
%                      x00 = Td / (4 - Td);
%                      W = x00 + 1.7*sqrt(2*(1 - phr));
%                      if (W >= 0)
%                          x03 = x00;
%                      else
%                          x03 = x00 - sqrt((-W)^(1/8))*(x00 + sqrt(-
Td/(1.5*T0 - Td)));
%                      end
%                      W      = 4/(4 - Td);
%                      lambda = (1 + (1 + m + 0.24*(dth - 1/2)) / ...
%                          (1 + 0.15*m)*x03*(W/2 - 0.03*x03*sqrt(W)));
%                      x0     = x03*lambda;
%                  end
%                  % estimate might not give solutions
%                  if (x0 < -1), exitflag = -1; return; end
%              end
%          end
%          % find root of Lancaster & Blancard's function
%          % ------------------------------------------
%          % (Halley's method)
%          x = x0; Tx = inf; iterations = 0;
%          while abs(Tx) > tol
%              % iterations
%              iterations = iterations + 1;
%              % compute function value, and first two derivatives
%              [Tx, Tp, Tpp] = LancasterBlanchard(x, q, m);
%              % find the root of the *difference* between the
%              % function value [T_x] and the required time [T]
%              Tx = Tx - T;
%              % new value of x
```

```
%               xp = x;
%               x  = x - 2*Tx*Tp ./ (2*Tp^2 - Tx*Tpp);
%               % escape clause
%               if mod(iterations, 7), x = (xp+x)/2; end
%               % Halley's method might fail
%               if iterations > 25, exitflag = -2; return; end
%           end
%           % calculate terminal velocities
%           % -----------------------------
%           % constants required for this calculation
%           gamma = sqrt(muC*s/2);
%           if (c == 0)
%               sigma = 1;
%               rho   = 0;
%               z     = abs(x);
%           else
%               sigma = 2*sqrt(r1*r2/(c^2)) * sin(dth/2);
%               rho   = (r1 - r2)/c;
%               z     = sqrt(1 + q^2*(x^2 - 1));
%           end
%           % radial component
%           Vr1    = +gamma*((q*z - x) - rho*(q*z + x)) / r1;
%           Vr1vec = Vr1*r1unit;
%           Vr2    = -gamma*((q*z - x) + rho*(q*z + x)) / r2;
%           Vr2vec = Vr2*r2unit;
%           % tangential component
%           Vtan1    = sigma * gamma * (z + q*x) / r1;
%           Vtan1vec = Vtan1 * th1unit;
%           Vtan2    = sigma * gamma * (z + q*x) / r2;
%           Vtan2vec = Vtan2 * th2unit;
%           % Cartesian velocity
%           V1 = Vtan1vec + Vr1vec;
%           V2 = Vtan2vec + Vr2vec;
%           % exitflag
%           exitflag = 1; % (success)
%           % also determine minimum/maximum distance
%           a = s/2/(1 - x^2); % semi-major axis
%           extremal_distances = minmax_distances(r1vec, r1, r1vec, r2,
 dth, a, V1, V2, m, muC);
%       end
%       % Lancaster & Blanchard's function, and three derivatives
 thereof
%       function [T, Tp, Tpp, Tppp] = LancasterBlanchard(x, q, m)
%           % protection against idiotic input
%           if (x < -1) % impossible; negative eccentricity
%               x = abs(x) - 2;
%           elseif (x == -1) % impossible; offset x slightly
%               x = x + eps;
%           end
%           % compute parameter E
%           E  = x*x - 1;
%           % T(x), T'(x), T''(x)
%           if x == 1 % exactly parabolic; solutions known exactly
%               % T(x)
```

```
%             T = 4/3*(1-q^3);
%             % T'(x)
%             Tp = 4/5*(q^5 - 1);
%             % T''(x)
%             Tpp = Tp + 120/70*(1 - q^7);
%             % T'''(x)
%             Tppp = 3*(Tpp - Tp) + 2400/1080*(q^9 - 1);
%         elseif abs(x-1) < 1e-2 % near-parabolic; compute with series
%             % evaluate sigma
%             [sig1, dsigdx1, d2sigdx21, d3sigdx31] = sigmax(-E);
%             [sig2, dsigdx2, d2sigdx22, d3sigdx32] = sigmax(-E*q*q);
%             % T(x)
%             T = sig1 - q^3*sig2;
%             % T'(x)
%             Tp = 2*x*(q^5*dsigdx2 - dsigdx1);
%             % T''(x)
%             Tpp = Tp/x + 4*x^2*(d2sigdx21 - q^7*d2sigdx22);
%             % T'''(x)
%             Tppp = 3*(Tpp-Tp/x)/x + 8*x*x*(q^9*d3sigdx32 -
 d3sigdx31);
%         else % all other cases
%             % compute all substitution functions
%             y  = sqrt(abs(E));
%             z  = sqrt(1 + q^2*E);
%             f  = y*(z - q*x);
%             g  = x*z - q*E;
%             % BUGFIX: (Simon Tardivel) this line is incorrect for
 E==0 and f+g==0
%             % d  = (E < 0)*(atan2(f, g) + pi*m) + (E >
 0)*log( max(0, f + g) );
%             % it should be written out like so:
%             if (E<0)
%                 d = atan2(f, g) + pi*m;
%             elseif (E==0)
%                 d = 0;
%             else
%                 d = log(max(0, f+g));
%             end
%             % T(x)
%             T = 2*(x - q*z - d/y)/E;
%             %  T'(x)
%             Tp = (4 - 4*q^3*x/z - 3*x*T)/E;
%             % T''(x)
%             Tpp = (-4*q^3/z * (1 - q^2*x^2/z^2) - 3*T - 3*x*Tp)/E;
%             % T'''(x)
%             Tppp = (4*q^3/z^2*((1 - q^2*x^2/z^2) + 2*q^2*x/z^2*(z -
 x)) - 8*Tp - 7*x*Tpp)/E;
%         end
%     end
%     % series approximation to T(x) and its derivatives
%     % (used for near-parabolic cases)
%     function [sig, dsigdx, d2sigdx2, d3sigdx3] = sigmax(y)
%         % preload the factors [an]
%         % (25 factors is more than enough for 16-digit accuracy)
```

```
%         persistent an;
%         if isempty(an)
%             an = [
%                 4.000000000000000e-001;     2.142857142857143e-001;
    4.629629629629630e-002
%                 6.628787878787879e-003;     7.211538461538461e-004;
    6.365740740740740e-005
%                 4.741479925303455e-006;     3.059406328320802e-007;
    1.742836409255060e-008
%                 8.892477331109578e-010;     4.110111531986532e-011;
    1.736709384841458e-012
%                 6.759767240041426e-014;     2.439123386614026e-015;
    8.203411614538007e-017
%                 2.583771576869575e-018;     7.652331327976716e-020;
    2.138860629743989e-021
%                 5.659959451165552e-023;     1.422104833817366e-024;
    3.401398483272306e-026
%                 7.762544304774155e-028;     1.693916882090479e-029;
    3.541295006766860e-031
%                 7.105336187804402e-033];
%         end
%         % powers of y
%         powers = y.^(1:25);
%         % sigma itself
%         sig = 4/3 + powers*an;
%         % dsigma / dx (derivative)
%         dsigdx = ( (1:25).*[1, powers(1:24)] ) * an;
%         % d2sigma / dx2 (second derivative)
%         d2sigdx2 = ( (1:25).*(0:24).*[1/y, 1, powers(1:23)] ) * an;
%         % d3sigma / dx3 (third derivative)
%         d3sigdx3 = ( (1:25).*(0:24).*(-1:23).*[1/y/y, 1/y, 1,
 powers(1:22)] ) * an;
%     end
%     %
 ------------------------------------------------------------------
%     % Helper functions
%     %
 ------------------------------------------------------------------
%     % compute minimum and maximum distances to the central body
%     function extremal_distances = minmax_distances(r1vec, r1,...
%                                                    r2vec, r2,...
%                                                    dth,...
%                                                    a,...
%                                                    V1, V2,...
%                                                    m,...
%                                                    muC)
%         % default - minimum/maximum of r1,r2
%         minimum_distance = min(r1,r2);
%         maximum_distance = max(r1,r2);
%         % was the longway used or not?
%         longway = abs(dth) > pi;
%         % eccentricity vector (use triple product identity)
%         evec = ((V1*V1.')*r1vec - (V1*r1vec.')*V1)/muC - r1vec/r1;
%         % eccentricity
```

```
%         e = sqrt(evec*evec.');
%         % apses
%         pericenter = a*(1-e);
%         apocenter  = inf;                   % parabolic/hyperbolic
 case
%         if (e < 1), apocenter = a*(1+e); end % elliptic case
%         % since we have the eccentricity vector, we know exactly
 where the
%         % pericenter lies. Use this fact, and the given value of
 [dth], to
%         % cross-check if the trajectory goes past it
%         if (m > 0) % obvious case (always elliptical and both apses
 are traversed)
%             minimum_distance = pericenter;
%             maximum_distance = apocenter;
%         else % less obvious case
%             % compute theta1&2 ( use (AxB)-(CxD) = (C·B)(D·A) -
 (C·A)(B·D) ))
%             pm1 = sign( r1*r1*(evec*V1.') -
 (r1vec*evec.')*(r1vec*V1.') );
%             pm2 = sign( r2*r2*(evec*V2.') -
 (r2vec*evec.')*(r2vec*V2.') );
%             % make 100.4% sure it's in (-1 <= theta12 <= +1)
%             theta1 = pm1*acos( max(-1, min(1, (r1vec/r1)*(evec/
e).')) );
%             theta2 = pm2*acos( max(-1, min(1, (r2vec/r2)*(evec/
e).')) );
%             % points 1&2 are on opposite sides of the symmetry axis
 -- minimum
%             % and maximum distance depends both on the value of
 [dth], and both
%             % [theta1] and [theta2]
%             if (theta1*theta2 < 0)
%                 % if |th1| + |th2| = turnangle, we know that the
 pericenter was
%                 % passed
%                 if abs(abs(theta1) + abs(theta2) - dth) < 5*eps(dth)
%                     minimum_distance = pericenter;
%                 % this condition can only be false for elliptic
 cases, and
%                 % when it is indeed false, we know that the orbit
 passed
%                 % apocenter
%                 else
%                     maximum_distance = apocenter;
%                 end
%             % points 1&2 are on the same side of the symmetry axis.
 Only if the
%             % long-way was used are the min. and max. distances
 different from
%             % the min. and max. values of the radii (namely, equal
 to the apses)
%             elseif longway
%                 minimum_distance = pericenter;
```

```
%                 if (e < 1), maximum_distance = apocenter; end
%             end
%         end
%         % output argument
%         extremal_distances = [minimum_distance, maximum_distance];
%     end
% end
%

end
```

Published with MATLAB® R2019a