Table of Contents

APPENDIX: MATLAB SCRIPT

Isaac Hertzog and Vittorio Baraldi

```
clear all
close all

% Let's take the year, month and day of the mission
y=input("Insert the year: \n");
m=input("Insert the month: \n");
D=input("Insert the day: \n");
%conversion of the departure date in seconds
d = (367*y - 7 * ( y + (m+9)/12 ) / 4 + 275*m/9 + D - 730530)*86400;

%All the following orbital elements are based on the ecliptic
 coordinate
%system
%Earth orbital elements
aE=149598023;                          %semi-major axis [km]
eE=0.0167086;                          %eccentricity
iE=0.00005*pi/180;                     %inclination [rad]
wE=114.20783*pi/180;                   %argument of perihelion [rad]
omegaE=(-11.26064+360)*pi/180;         %RAAN[rad]
muS=1.32712440018e11;                  %gravitational parameter for
 Sun
TE=365.256363004*86400;                %earth period [s]
muE=398600;                            %gravitational parameter for
 Earth
nE=2*pi/TE;                            %mean orbit rate for Earth
 [rad/s]
ME=(nE*d);                             %Earth's mean anomaly @ given
 time
muM=4.282837e3;                        %gravitational parameter for
 Mars

%Ceres orbital elements
aC=414261000;                          %[km]
eC=0.07600902910;
iC=10.59406704*pi/180;                 %[rad]
omegaC=80.3055316*pi/180;              %[rad]
wC=73.5976941*pi/180;                  %[rad]
TC=1683.14570801*86400;                %ceres period [s]
muC=62.6325;                           %gravitational parameter for
 Ceres
```

```
nC=2*pi/TC;                                     %[rad/s]
MC=nC*d;                                        %[rad]

%checking on mean anomalies to be less than 360°
for i=1:1000
    if MC<=(2*pi) && ME<=(2*pi)
        break
    elseif MC>(2*pi) && ME>(2*pi)
        MC=MC-2*pi;
        ME=ME-2*pi;
    elseif MC>(2*pi) && ME<=(2*pi)
        MC=MC-2*pi;
    elseif MC<=(2*pi) && ME>(2*pi)
        ME=ME-2*pi;
    end
end

%calculating eccentric anomalies via Newton-Raphson numerical method
fE=@(x) x-eE*sin(x)-ME;
dfE=@(x) 1-eE*cos(x);
x0=MC;
fC=@(x) x-eC*sin(x)-MC;
dfC=@(x) 1-eC*cos(x);
[EE,succ1]=NewtonRhapsonSolver(fE,dfE,x0,10e-10);
[EC,succ2]=NewtonRhapsonSolver(fC,dfC,x0,10e-10);

%calculating position and velocity of the planet given the eccentric
%anomaly, and therefore we can obtain the true anomaly at given
 departure
%time

%perifocal earth coordinates
xvE=aE*(cos(EE)-eE);
yvE=aE*(sqrt(1-eE^2)*sin(EE));
%earth true anomaly
thetaE=atan2(yvE,xvE);
%radius
rE=sqrt(xvE^2+yvE^2);
%perifocal Ceres coordinates
xvC=aC*(cos(EC)-eC);
yvC=aC*(sqrt(1-eC^2)*sin(EC));
%ceres true anomaly
thetaC=atan2(yvC,xvC);
%radius
rC=sqrt(xvC^2+yvC^2);
%converting from perifocal coordinates to geocentric, via eulerian
%matrixes
QE=perifocaltogeo(wE,omegaE,iE);
QC=perifocaltogeo(wC,omegaC,iC);
rEvec=QE*[xvE yvE 0]';
rCvec=QC*[xvC yvC 0]';

%hohmann transfer time. For both bodies, we consider the mean distance
%from the sun, since the eccentricity is low enough to give us such
```

```
%approximation.
r2=(152100000+147095000)/2;
r1=(445749000+382774000)/2;;
aH=(r1+r2)/2;
```

Hohmann Transfer

```
%Hohmann transfer calculation combined with the plane change
%initial and final velocities for planets' orbits
vi=sqrt(muS/r2);
vf=sqrt(muS/r1);
%initial and final velocities for transfer orbit
vit=sqrt(((2*muS)/r2)-(muS/aH));
vft=sqrt(((2*muS)/r1)-(muS/aH));

%We want to go from a 300km LEO to a 300km altitude circular orbit to
%Ceres'. In order to do that we need to calculate delta v's to escape
 the
%inital orbit and to capture the final one. We use the hyperbolic
 escape
%trajectory theory for this Hohmann transfer in order to make it more
%realistic and accurate
D2=vf-vft;
D1=vit-vi;
Vinf1=D1;
%initial LEO velocity
vorbit1=sqrt(muE/(6378+300));
%escape hyperbolic velocity
v_hyp_1=sqrt(Vinf1^2+2*muE/(6378+300));
D_escape=abs(v_hyp_1-vorbit1);
Vinf2=D2;
%final 300km altitude orbit velocity at Ceres
vorbit2=sqrt(muC/(469.73+300));
%capture hyperbolic velocity
v_hyp_2=sqrt(Vinf2^2+2*muC/(469.73+300));
D_capture=abs(v_hyp_2-vorbit2);

%plane change and longitude of the ascending node change
delta_o=abs(omegaC-omegaE);
theta_plane=acos(cos(iE)*cos(iC)+sin(iE)*sin(iC)*cos(delta_o));
D_plane=2*vorbit2*sin(theta_plane/2);

%transfer time
tT=0.5*sqrt(4*pi^2*aH^3/muS);
%wait time to launch
tW=((thetaC-thetaE)+nC*tT-pi)/(nE-nC);

%check on wait time to be positive
for i=1:1000
    if tW<0
        tW=((thetaC-thetaE)+nC*tT-pi+2*pi*i)/(nE-nC);
    else
        break
```

```
    end
end


%Total Delta-v required and total mission time
Total_DV=D_plane+D_capture+D_escape;
Total_time=tW+tT;
y_m=Total_time/86400;


fprintf('Mission starts: %i/%i/%i',m,D,y)
fprintf('\nDelta-V required for Hohmann Transfer: %.3f km/s',Total_DV)
fprintf('\nTotal mission time for Hohmann Transfer: %.2f days',y_m)
```

Lambert's solution with Mars Flyby

```
[coeE, RE, VE, jdE] = planet_elements_and_sv(3, y, m, D, 0, 0, 0,
 muS);
[coeM, RM, VM, jdM] = planet_elements_and_sv(4, y+1, m-4, D, 0, 0, 0,
 muS);
tof=(jdM-jdE)*24*60*60;
[vd,va,dTheta1] = LambertSolver( RE', RM', tof, muS );
%excess velocity at departure (patched conic assumption)
v_inf_d=vd-VE';
vinfd=norm(v_inf_d);
%excess velocity at arrival
v_inf_a=va-VM';
vinfa=norm(v_inf_a);
%transfer orbit COE
[at,it,Wt,wt,et,tht,ht,Nvec,evec] = OrbitalElementsFromRV( RE', vd,
 muS );
%delta-v necessary to put the spacecraft in a hyperbolic escape
 trajectory
%from a 300km circular parking orbit
vp=sqrt(vinfd^2+(2*muE)/(300+6378));
vc=sqrt(muE/(6378+300));
dv_fly=vp-vc;
%arrival at Mars, for flyby
%spacecraft velocity at approach
v1=v_inf_a+VM';
v1n=norm(v1);
alpha=acos((v1'*VM')/(norm(VM)*v1n));
%spacecraft coordinates in heliocentric frame
v1m=v1n*cos(alpha);
v1s=v1n*sin(alpha);
%picking a 300km perigee approach radius
h_a=(3389.5+300)*sqrt(vinfa^2+2*muM/(3389.5+300));
e_a=1+((3389.5+300)*vinfa^2)/muM;
%turn angle for flyby
delta1=2*asin(1/(1+((3389.5+300)*vinfa^2/muM)));
%angle between planet's heliocentric velocity and hyperbolic excess
%velocity of the spacecraft at approach
phi1=atan(v1s/(v1m-norm(VM)));
```

```
%outbound crossing angle
phi2=phi1+delta1;
%hyperbolic excess velocity at the outbound crossing
v_inf_3=[vinfa*cos(phi2);vinfa*sin(phi2)];
vinf3=norm(v_inf_3);
%spacecraft velocity at the end of flyby (heliocentric frame)
v3=[norm(VM)+vinfa*cos(phi2);vinfa*sin(phi2)];
%elements of the new heliocentric departure trajectory from Mars
h2=norm(RM)*(norm(VM)+vinfa*cos(phi2));
h2vec=cross(RM',(VM'+v_inf_a*cos(phi2)));
i2=acos(h2vec(3)/h2);
normR=norm(RM);
Vr=-vinfa*sin(phi2);
th2=atan((h2*Vr)/(muS*((h2^2/muS/normR)-1)));
e2=((h2^2/muS/normR)-1)/cos(th2);
a2=(h2^2/muS)/(1-e2^2);
%now we can assume a Hohmann-like type of transfer from Mars to Ceres
 after
%flyby. The transfer time would be approximately:
Tt_fly=0.5*sqrt(4*pi^2*a2^3/muS);
%now we are going towards Ceres and we want to park at an elliptical
 orbit
%we set an eccentricity high enough in order to have the less delta-v
%possible, but not extreme. We have assumed that there was no
 perturbations
%or propulsion used during transfer, so the hyperbolic excess velocity
 at
%Ceres approach is the same of the after-flyby hyp. excess velocity.
ec=0.1;

%speed of spacecraft at periapsis at arriving hyperbola
v_hyp=sqrt(vinf3^2+(2*muC)/(469.73+300));
%speed at periapsis of capture orbit
v_cap=sqrt(muC*(1+ec)/(469.73+300));

%delta-v for capturing orbit (we're still aiming at a 300km altitude
 orbit)
dv2_fly=v_hyp-v_cap;
%plane adjustment once at Ceres
di_fly=iC-i2;
d_plane_fly=abs(2*v_cap*sin(di_fly/2));
Total_DV_fly=dv_fly+dv2_fly+d_plane_fly;
y_m_fly=(jdM-jdE)+Tt_fly/86400;
fprintf('\nDelta-V required for Lambert: %.3f km/s',Total_DV_fly)
fprintf('\nTotal mission time for Lambert: %.2f days',y_m_fly)

% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Notes

```
%{
 In both cases the delta-v's were probably a little underestimated
 since
```

```
 some approximations were taken:
- For the Hohmann Transfer,we assumed both Earth's and Ceres' orbits
 to be
circular, since e<0.1 for both of them. This is a fair approximation
 widely
used in actual mission designs.
- For the flyby, the total mission delta-v and mission time
is not a trivial analysis in this
case, depending on which angle the spacecraft would leave Mars
 (outbound
crossing angle). That would require a deeper analysis,because
 spacecraft's
direction after flyby and Ceres' orbit might not be in the perfect
alignment for rendezvous. In this case we are going to simplify and
assume a "perfect catch". In reality, the spacecraft would first
arrive and phase with Ceres' orbit and then
perform a chase maneuver (which would not require a huge delta-v.)
Another procedure is to determine which departure date is the best
from the very beginning of the mission in order to obtain the perfect
catch. The purpose of this example is to compare the energy required
 for
two different types of orbit trasnfers, so we need to pick the same
"mission start" date (which was randomly picked).

Despite those approximations, comparing the results with similar
 approaches
in the literature and previous missions, we can affirm that these
 results
look realistic.

In the following functions _planetary_elements_and_sv.m_ and
_planetary_elements.m_ are found in the "Orbital Mechanics for
 Engineers"
by Howard D. Curtis (Appendix D.34) and the Lamberts solver function
used the algorithm at the Appendix D.25 of the same publication as a
starting point.Universal variables were used to solve the problem in
 this
situation.

References
- F.J. Hale, Introduction to Space Flight, Prentice-Hall, 1994.
- [2] M.H. Kaplan, Modern Spacecraft Dynamics and Control, John Wiley
 and
Sons, 1976
- Howard D. Curtis, Orbital Mechanics for Engineers, 2010
%}

% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Functions

```
function [x,success] = NewtonRhapsonSolver( fun, dfun, x0, tol )
```

```
% Use the Newton-Rhapson method to find the zero crossing for a
 function.
%
%   Inputs:
%     fun     Function handle. We want to find x where fun(x) = 0.
%     dfun    Derivative function handle.
%     x0      Initial guess.
%     tol     Tolerance
%
%   Outputs:
%     x         Solution to fun(x) = 0.
%     success   Flag indicating whether it returned with a solution
 (1)
%                 or not (0).

if nargin<4
  tol = 1e-8;
end
maxCount = 100;
count = 0;

% initial guess given
x = x0;

dist = tol+1;
while dist>tol && count<maxCount

  % compute function value and function derivative value
  f = feval(fun,x);
  fp = feval(dfun,x);

  % distance between current and next iterate
  dist = abs(f/fp);

  x = x-f/fp;
  count = count + 1;
end

if( dist>tol && count==maxCount )
  success = 0;
  warning('Terminated after %d iterations before reaching desired
 tolerance.',count);
else
  success = 1;
end

end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

function Q=perifocaltogeo(w,o,i)

q1=[cos(w) sin(w) 0; -sin(w) cos(w) 0; 0 0 1];
q2=[1 0 0; 0 cos(i) sin(i); 0 -sin(i) cos(i)];
```

```
q3=[cos(o) sin(o) 0; -sin(o) cos(o) 0; 0 0 1];

Q=q1*q2*q3;
Q=Q';
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


function [a,inc,W,w,e,th,h,Nvec,evec] = OrbitalElementsFromRV( r, v,
 mu )
%
% Compute 6 orbital elements from position and velocity vectors
%
% Inputs:
%   r     (3,1)   Position vector     [km]
%   v     (3,1)   Velocity vector     [km/s]
%
% Outputs:
%   a             Semi major axis     [km]
%   inc           Inclination         [rad]
%   W             Right ascension     [rad]
%   w             Argument of perigee [rad]
%   e             Eccentricity
%   th            True anomaly        [rad]
%

tol = 2*eps;

% Default value for mu (if not provided)
if( nargin<3 )
  mu = 398600.44;
end

% compute the magntiude of r and v vectors:
rMag  = sqrt(r'*r);
vMag  = sqrt(v'*v);

% radial component of velocity
vr    = v'*r/rMag;  % (Note that v'*r is equivalent to dot(v,r)

% Compute the specific angular momentum
hvec  = cross(r,v);
h     = sqrt(hvec'*hvec);

% inclination
inc = acos( hvec(3)/h ); % equivalent to acos( dot(hvec/h,[0;0;1]) )

% node line
Nvec  = cross([0;0;1],hvec);
N     = sqrt(Nvec'*Nvec);

% if the node line is not well defined from the angular momentum
 vector
```

```
% (this occurs at i=0 and i=pi) then define it to be along the
% inertial x vector
if( N<tol )
  Nvec = [1;0;0];
  N = 1.0;
end

% right ascension
if( abs(inc)<tol || abs(inc-pi)<tol )
  W = 0; % R.A. not defined for zero-inclination orbits.
elseif( Nvec(2)>=0 )
  W = acos(Nvec(1)/N);
else
  W = 2*pi-acos(Nvec(1)/N);
end

% Eccentricity
evec = 1/mu*( (vMag^2-mu/rMag)*r-rMag*vr*v );
e = sqrt(evec'*evec);

% Look out for special cases
equatorial = ( abs(inc)<tol || abs(inc-pi)<tol );
circular   = e < tol;

% Argument of perigee
%  - The angle between the eccentricity vector and the line of nodes
if( circular )
  % circular case
  % there is no definition for Arg. of perigee in this case.
  % just set it to zero
  w = 0;
else
  % non-circular cases...

  if( equatorial )
    % equatorial

    arg = evec(1)/ e;
    w = acos( arg );
    if( abs(inc) < tol && evec(2) < 0 )
      w = 2*pi - w;
    elseif (abs(inc - pi) < tol) && evec(2) > 0
      % retrograde: change sign of test
      w = 2*pi - w;
    end
  else
      % non-equatorial

    arg = Nvec'*evec/N/e;
    % check the argument to prevent issues with numerical rounding
    if( arg>=1 )
      w = 0;
    elseif( arg<= -1 )
      w = pi;
```

```
    else
      w = acos(arg);
    end

    % check for retrograde case!
    if( evec(3)<0 )
      w = 2*pi-w;
    elseif( abs(inc-pi)<tol && evec(2)>0 )
      w = 2*pi-w;
    end

  end

end


% True anomaly
%  - The angle between the eccentricity vector and the position vector
if( circular && equatorial )
  % circular and equatorial case
  arg = r(1)/rMag;
  th = acos(arg);
  if( r(2)<0 )
    th = 2*pi-th;
  end
else
  if( circular )
    % circular and inclined case
    % (measure TA from line of nodes)
    evecDir = Nvec/N;
  else
    % non-circular case
    evecDir = evec/e;
  end
  arg = evecDir'*r/rMag;
  if( arg>= 1.0 )
    arg = 1.0;
  elseif( arg <= -1.0 )
    arg = -1.0;
  end

  if( vr>=0 )
    th = acos(arg); % T.A. is defined to lie between 0 and 2*PI
  else
    th = 2*pi-acos(arg);
  end

end


% Semi major axis
a = h^2/mu/(1-e^2);
```

```
% return all six elements into a vector if only one output is
 requested
if( nargout==1 )
  a = [a,inc,W,w,e,th];
end

end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

function [coe, r, v, jd] = planet_elements_and_sv ...
                (planet_id, year, month, day, hour, minute, second,
 mu)
%{
  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  RA  incl  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)
               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 - 2050
  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

%}
if( nargin<8 )
  mu = 1.327e11;
end

deg    = pi/180;

%...Equation 5.48:
j0     = J0(year, month, day);

ut     = (hour + minute/60 + second/3600)/24;

%...Equation 5.47
jd     = j0 + ut;

%...Obtain the data for the selected planet from Table 8.1:
[J2000_coe, rates] = planetary_elements(planet_id);

%...Equation 8.93a:
t0     = (jd - 2451545)/36525;

%...Equation 8.93b:
elements = J2000_coe + rates*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:
incl   = elements(3);
RA     = mod(elements(4),360);
w_hat  = mod(elements(5),360);
L      = mod(elements(6),360);
w      = mod(w_hat - RA ,360);
M      = mod(L - w_hat  ,360);

%...Algorithm 3.1 (for which M must be in radians)
E      = kepler_E(e, M*deg); %rad

%...Equation 3.13 (converting the result to degrees):
```

```
TA     = mod(2*atand(sqrt((1 + e)/(1 - e))*tan(E/2)),360);

coe    = [h e RA*deg incl*deg w*deg TA*deg];

%...Algorithm 4.5:
[r, v] = sv_from_coe(coe, mu);
return
  end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  function [J2000_coe, rates] = planetary_elements(planet_id)
%{
  This function extracts a planet's J2000 orbital elements and
  centennial rates from Table 8.1.

  planet_id      - 1 through 9, for Mercury through Pluto

  J2000_elements - 9 by 6 matrix of J2000 orbital elements for the
 nine
                   planets Mercury through Pluto. The columns of each
                   row are:
                     a     = semimajor axis (AU)
                     e     = eccentricity
                     i     = inclination (degrees)
                     RA    = right ascension of the ascending
                             node (degrees)
                     w_hat = longitude of perihelion (degrees)
                     L     = mean longitude (degrees)

  cent_rates     - 9 by 6 matrix of the rates of change of the
                   J2000_elements per Julian century (Cy). Using "dot"
                   for time derivative, the columns of each row are:
                     a_dot     (AU/Cy)
                     e_dot     (1/Cy)
                     i_dot     (deg/Cy)
                     RA_dot    (deg/Cy)
                     w_hat_dot (deg/Cy)
                     Ldot      (deg/Cy)

  J2000_coe      - row vector of J2000_elements corresponding
                   to "planet_id", with au converted to km
  rates          - row vector of cent_rates corresponding to
                   "planet_id", with au converted to km

  au             - astronomical unit (149597871 km)
%}
% --------------------------------------------------------------------

%---- a --------- e -------- i -------- RA --------- w_hat ------- L
 ------

J2000_elements = ...
```

```
[0.38709927  0.20563593  7.00497902  48.33076593  77.45779628
 252.25032350
 0.72333566  0.00677672  3.39467605  76.67984255 131.60246718
 181.97909950
 1.00000261  0.01671123 -0.00001531   0.0        102.93768193
 100.46457166
 1.52371034  0.09339410  1.84969142  49.55953891 -23.94362959
 -4.55343205
 5.20288700  0.04838624  1.30439695 100.47390909  14.72847983
 34.39644501
 9.53667594  0.05386179  2.48599187 113.66242448  92.59887831
 49.95424423
19.18916464  0.04725744  0.77263783  74.01692503 170.95427630
 313.23810451
30.06992276  0.00859048  1.77004347 131.78422574  44.96476227
 -55.12002969
39.48211675  0.24882730 17.14001206 110.30393684 224.06891629
 238.92903833];

cent_rates = ...
[0.00000037  0.00001906 -0.00594749 -0.12534081  0.16047689
 149472.67411175
 0.00000390 -0.00004107 -0.00078890 -0.27769418  0.00268329
 58517.81538729
 0.00000562 -0.00004392 -0.01294668  0.0         0.32327364
 35999.37244981
 0.0001847   0.00007882 -0.00813131 -0.29257343  0.44441088
 19140.30268499
-0.00011607 -0.00013253 -0.00183714  0.20469106  0.21252668
 3034.74612775
-0.00125060 -0.00050991  0.00193609 -0.28867794 -0.41897216
 1222.49362201
-0.00196176 -0.00004397 -0.00242939  0.04240589  0.40805281
 428.48202785
 0.00026291  0.00005105  0.00035372 -0.00508664 -0.32241464
 218.45945325
-0.00031596  0.00005170  0.00004818 -0.01183482 -0.04062942
 145.20780515];

J2000_coe      = J2000_elements(planet_id,:);
rates          = cent_rates(planet_id,:);

%...Convert from AU to km:
au             = 149597871;
J2000_coe(1)   = J2000_coe(1)*au;
rates(1)       = rates(1)*au;

end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

function [v1,v2,dTheta] = LambertSolver( r1, r2, TOF, mu, dir )

% Solve Lambert's problem.
```

```
%
%  Given two position vectors, r1 and r2, and the time of flight
 between
%  them, TOF, find the corresponding Keplerian orbit.

% Input checking

% direction
if( nargin<5 )
  dir = 'pro';
end

% Check validity of "dir" input
if( ~ischar(dir) )
  error('Input for direction "dir" must be a string, either
 ''pro'' ...or ''retro''.');
end

% mu
if( nargin<4 )
  mu = 398600.44;
  warning('Using Earth gravitational constant, mu = %f.',mu);
end

% TOF
if( TOF<=0 )
  error('The time of flight TOF must be >0.')
end

% Calculate the position vector magnitudes
r1m = sqrt(r1'*r1);
r2m = sqrt(r2'*r2);

% Calculate delta-theta
r1CrossR2 = cross(r1,r2);
arg = r1'*r2/r1m/r2m;
if( arg>1 )
  angle = 2*pi;
elseif( arg<-1 )
  angle = pi;
else
  angle = acos( arg );
end
switch lower(dir)
  case {'pro','prograde'}
    if( r1CrossR2(3) >= 0 )
      dTheta = angle;
    else
      dTheta = 2*pi-angle;
    end
  case {'retro','retrograde'}
    if( r1CrossR2(3) < 0 )
      dTheta = angle;
    else
```

```
      dTheta = 2*pi-angle;
    end
  otherwise
    error('Unrecognized direction. Use either ''pro'' or ''retro''.')
end

if( abs( abs(dTheta)-2*pi ) < 1e-8 )
  % this is a rectilinear orbit... not supported.
  v1 = nan;
  v2 = nan;
  return;
end

% Calculate "A"
A = sin(dTheta)*sqrt(r1m*r2m/(1-cos(dTheta)));

% Inline function handles, all functions of "z"
yf  = @(z) r1m+r2m+A*( z.*stumpS(z)-1 )./sqrt(stumpC(z));
Ff  = @(z) ( yf(z)./stumpC(z) ).^1.5 .* stumpS(z) + A*sqrt(yf(z)) -...
    sqrt(mu)*TOF;
dFf = @(z) LambertFPrimeOfZ( r1m, r2m, dTheta, z );

% Solve for z ...

% Initial guess for z...
z0s = FindInitialZGuessForLambert(yf,Ff);

% consider each approx. crossing and solve for exact z value at
 each...
% terminate as soon as we have a good answer.
success = 0;
for i=1:length(z0s)
  [zs,success] = NewtonRhapsonSolver( Ff, dFf, z0s(i), 1e-6 );
  if( success )
    break;
  end
end
if( ~success )
  warning('Newton method did not find a solution in LambertSolver.')
  v1=nan;
  v2=nan;
  return;
end


% Calculate "y"
%y = yf(zs);

% Calculate the Lagrange Coefficients
[f,g,~,gdot] = LagrangeCoeffZ(r1m,r2m,dTheta,zs,mu);

% Calculate v1 and v2
v1 = 1/g*(r2-f*r1);
v2 = 1/g*(gdot*r2-r1);
```

```
% Check result...

% Calculate the COE from [r1,v1]
[a1,i1,W1,w1,e1,th1] = OrbitalElementsFromRV(r1,v1,mu);

% Calculate the COE from [r2,v2]
[a2,i2,W2,w2,e2,th2] = OrbitalElementsFromRV(r2,v2,mu);

if( norm([1,cos([i1,W1,w1]),e1]-[a2/a1,cos([i2,W2,w2]),e2])>1e-8 )
  warning('Orbital elements from [r1,v1] do not match with those from
 [r2,v2].')
  disp([a1,i1,W1,w1,e1,th1; a2,i2,W2,w2,e2,th2]')
end
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

function [f,g,fdot,gdot] = LagrangeCoeffZ( r1m, r2m, dTheta, z, mu )

% Compute the Lagrange coefficients in terms of universal variable z
%
%   Inputs:
%     r1m     Magnitude of position vector r1
%     r2m     Magnitude of position vector r2
%     dTheta  Angle between vector r1 and r2
%     z       Universal variable "z"
%     mu      Gravitational constant
%
%  Outputs:
%     f
%     g
%     fdot
%     gdot
%


C     = stumpC(z);
S     = stumpS(z);
A     = sin(dTheta)*sqrt(r1m*r2m/(1-cos(dTheta)));
y     = r1m+r2m+A*( z*S-1 )/sqrt(C);
f     = 1-y/r1m;
g     = A*sqrt(y/mu);
fdot  = sqrt(mu)/(r1m*r2m)*sqrt(y/C)*(z*S-1);
gdot  = 1-y/r2m;
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

function z0 = FindInitialZGuessForLambert( yf, Ff )

% Find an initial guess for "z" to solve Universal Keplers equation.
% For use in the Lambert method.
%
```

```
% This method finds all zero-crossings for y(z), and looks for z-
values in
% the ranges where y(z)>0 where F(z) is near zero.
%
%
%   Inputs:
%     yf      Function handle for y(z).
%     Ff      Function handle for F(z)
%
%   Outputs:
%     z0      Array of initial guesses to consider.

zmin = -1e3;
zmax = 1e3;

% first find any y(z)=0 crossings, if they exists
zz = linspace(zmin,zmax,1e4);
yz = yf(zz);
y1 = yz(1:end-1);
y2 = yz(2:end);
kyz = find(sign(y1).*sign(y2)<0);

if( isempty(kyz) )
  %disp('No y(z)=0 crossing found...')
  zz = linspace(zmin,zmax,1e4);
  z0 = FindApproxFZero( Ff, zz );

else
  %disp(sprintf('%d y(z)=0 crossings found...',length(kyz)))
  z0 = [];
  for i=1:length(kyz)

    zy0 = fzero(yf,zz(kyz(i)));
    zz = linspace(zy0,zy0+zmax,1e4);
    z0i = FindApproxFZero( Ff, zz );
    if( ~isempty(z0i) )
      z0 = [z0, z0i];
    end

  end
end

%disp(z0)
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

function z0 = FindApproxFZero( Ff, zz )

  f = Ff(zz);                  % F(z) where y(z) > 0

  % find approx. F(z)=0 crossing
  f1 = f(1:end-1);
  f2 = f(2:end);
```

```
  ks = find(sign(f1).*sign(f2)<0);
  z0 = zz(ks);
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

function dF = LambertFPrimeOfZ( r1m, r2m, dTheta, z )

% Compute the derivative of F(z)
%
%   Given two position vectors, r1 and r2, and the time of flight
 between
%   them, TOF, find the corresponding Keplerian orbit.
%
%   Inputs:
%     r1m     Magnitude of position vector r1
%     r2m     Magnitude of position vector r2
%     dTheta  Angle between vector r1 and r2
%     z       Universal variable "z"
%
%  Outputs:
%     dF      Derivative of F(z) at z

A = sin(dTheta)*sqrt(r1m*r2m/(1-cos(dTheta)));

if( abs(z)<eps )
  y0 = r1m+r2m-sqrt(2)*A;
  dF = sqrt(2)/40*y0^1.5 + A/8*( sqrt(y0) + A*sqrt(1/2/y0));
else
  S = stumpS(z);
  C = stumpC(z);
  y = r1m+r2m+A*( z*S-1 )/sqrt(C);
  dF = (y/C)^1.5 * ( 1/(2*z) * (C-3*S/(2*C))+3*S^2 / (4*C) ) + ...
    (A/8)*( 3*S/C*sqrt(y) + A*sqrt(C/y) );
end
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

function S = stumpS( z )

if( length(z)>1 )

  % vectorized
  kp = find(z>0);
  kn = find(z<0);
  ke = z==0;
  sz = zeros(size(z));
  S = zeros(size(z));
  sz(kp) = sqrt(z(kp));
  sz(kn) = sqrt(-z(kn));
  sz(ke) = 0;
  S(kp) = (sz(kp)-sin(sz(kp)))./(sz(kp).^3);
  S(kn) = (sinh(sz(kn))-sz(kn))./(sz(kn).^3);
```

```
    S(ke) = 1/6;

  else

    if( z>0 )
      sz = sqrt(z);
      S = (sz-sin(sz))/(sz^3);
    elseif( z<0 )
      sz = sqrt(-z);
      S = (sinh(sz)-sz)/(sz^3);
    else
      S = 1/6;
    end

  end
  end
  % ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  % ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  function C = stumpC( z )

  if( length(z) > 1 )

    % vectorized
    kp = find(z>0);
    kn = find(z<0);
    ke = z==0;
    C = zeros(size(z));
    C(kp) = (1-cos(sqrt(z(kp)))))./z(kp);
    C(kn) = (cosh(sqrt(-z(kn)))-1)./(-z(kn));
    C(ke) = 0.5;

  else

    if( z>0 )
      C = (1-cos(sqrt(z)))./z;
    elseif( z<0 )
      C = (cosh(sqrt(-z))-1)/(-z);
    else
      C = 0.5;
    end

  end
  end

  % ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  % ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Published with MATLAB® R2019b