APPENDIX

# MATLAB Scripts

# D

## D.1 Introduction

This appendix lists MATLAB scripts that implement all of the numbered algorithms presented throughout the text. The programs use only the most basic features of MATLAB and are liberally commented so as to make reading the code as easy as possible. To "drive" the various algorithms, one can use MATLAB to create graphical user interfaces (GUIs). However, in the interest of simplicity and keeping our focus on the algorithms rather than elegant programming techniques, GUIs were not developed. Furthermore, the scripts do not use files to import and export data. Data is defined in declaration statements within the scripts. All output is to the screen, that is, to the MATLAB Command Window. It is hoped that interested students will embellish these simple scripts or use them as a springboard toward generating their own programs.

Each algorithm is illustrated by a MATLAB coding of a related example problem in the text. The actual output of each of these examples is also listed.

It would be helpful to have MATLAB documentation at hand. There are a number of practical references on the subject, including Hahn (2002), Kermit and Davis (2002), and Magrab (2000). MATLAB documentation may also be found at The MathWorks Web site (www.mathworks.com). Should it be necessary to do so, it is a fairly simple matter to translate these programs into other software languages.

These programs are presented solely as an alternative to carrying out otherwise lengthy hand computations and are intended for academic use only. They are all based exclusively on the introductory material presented in this text.

## Chapter 1

## D.2 Algorithm 1.1: Numerical integration by Runge–Kutta methods RK1, RK2, RK3, or RK4

### Function file rkf1_4.m

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function [tout, yout] = rk1_4(ode_function, tspan, y0, h, rk)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function uses a selected Runge-Kutta procedure to integrate
  a system of first-order differential equations dy/dt = f(t,y).
```

```
  y             - column vector of solutions
  f             - column vector of the derivatives dy/dt
  t             - time
  rk            - = 1 for RK1; = 2 for RK2; = 3 for RK3; = 4 for RK4
  n_stages      - the number of points within a time interval that
                  the derivatives are to be computed
  a             - coefficients for locating the solution points within
                  each time interval
  b             - coefficients for computing the derivatives at each
                  interior point
  c             - coefficients for the computing solution at the end of
                  the time step
  ode_function  - handle for user M-function in which the derivatives f
                  are computed
  tspan         - the vector [t0 tf] giving the time interval for the
                  solution
  t0            - initial time
  tf            - final time
  y0            - column vector of initial values of the vector y
  tout          - column vector of times at which y was evaluated
  yout          - a matrix, each row of which contains the components of y
                  evaluated at the corresponding time in tout
  h             - time step
  ti            - time at the beginning of a time step
  yi            - values of y at the beginning of a time step
  t_inner       - time within a given time step
  y_inner       - values of y within a given time step

  User M-function required: ode_function
%}
% ------------------------------------------------------------------------

%...Determine which of the four Runge-Kutta methods is to be used:
switch rk
    case 1
        n_stages = 1;
        a = 0;
        b = 0;
        c = 1;
    case 2
        n_stages = 2;
        a = [0 1];
        b = [0 1]';
        c = [1/2 1/2];
    case 3
        n_stages = 3;
        a = [0 1/2 1];
        b = [ 0  0
             1/2 0
             -1  2];
```

```
        c = [1/6 2/3 1/6];
    case 4
        n_stages = 4;
        a = [0 1/2 1/2 1];
        b = [ 0   0   0
             1/2  0   0
              0  1/2  0
              0   0   1];
        c = [1/6 1/3 1/3 1/6];
    otherwise
         error('The parameter  rk  must have the value 1, 2, 3 or 4.')
end

t0   = tspan(1);
tf   = tspan(2);
t    = t0;
y    = y0;
tout = t;
yout = y';

while t < tf
    ti = t;
    yi = y;
    %...Evaluate the time derivative(s) at the 'n_stages' points within the
    %   current interval:
    for i = 1:n_stages
        t_inner = ti + a(i)*h;
        y_inner = yi;
        for j = 1:i-1
            y_inner = y_inner + h*b(i,j)*f(:,j);
        end
        f(:,i)  = feval(ode_function, t_inner, y_inner);
    end

    h    = min(h, tf-t);
    t    = t + h;
    y    = yi + h*f*c';
    tout = [tout;t];  % adds t to the bottom of the column vector tout
    yout = [yout;y']; % adds y' to the bottom of the matrix yout
end

end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## **Function file:** `Example_1_18.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function Example_1_18
% ~~~~~~~~~~~~~~~~~~~~
```

```
%{
  This function uses the RK1 through RK4 methods with two
  different time steps each to solve for and plot the response
  of a damped single degree of freedom spring-mass system to
  a sinusoidal forcing function, represented by

  x" + 2*z*wn*x' + wn^2*x = (Fo/m)*sin(w*t)

  The numerical integration is done by the external
  function 'rk1_4', which uses the subfunction 'rates'
  herein to compute the derivatives.

  This function also plots the exact solution for comparison.

  x            - displacement (m)
  '            - shorthand for d/dt
  t            - time (s)
  wn           - natural circular frequency (radians/s)
  z            - damping factor
  wd           - damped natural frequency
  Fo           - amplitude of the sinusoidal forcing function (N)
  m            - mass (kg)
  w            - forcing frequency (radians/s)
  t0           - initial time (s)
  tf           - final time (s)
  h            - uniform time step (s)
  tspan        - a row vector containing t0 and tf
  x0           - value of x at t0 (m)
  x_dot0       - value of dx/dt at t0 (m/s)
  f0           - column vector containing x0 and x_dot0
  rk           - = 1 for RK1; = 2 for RK2; = 3 for RK3; = 4 for RK4
  t            - solution times for the exact solution
  t1, ...,t4   - solution times for RK1,...,RK4 for smaller
  t11,...,t41  - solution times for RK1,...,RK4 for larger h
  f1, ...,f4   - solution vectors for RK1,...,RK4 for smaller h
  f11,...,f41  - solution vectors for RK1,...,RK4 for larger h

  User M-functions required:  rk1_4
  User subfunctions required: rates
%}
% ------------------------------------------------------------------------

clear all; close all; clc

%...Input data:
m       = 1;
z       = 0.03;
wn      = 1;
```

```
Fo     = 1;
w      = 0.4*wn;

x0     = 0;
x_dot0 = 0;
f0     = [x0; x_dot0];

t0     = 0;
tf     = 110;
tspan  = [t0 tf];
%...End input data

%...Solve using RK1 through RK4, using the same and a larger
%   time step for each method:
rk = 1;
h  = .01; [t1,  f1]  = rk1_4(@rates, tspan, f0, h, rk);
h  = 0.1; [t11, f11] = rk1_4(@rates, tspan, f0, h, rk);

rk = 2;
h  = 0.1; [t2,  f2]  = rk1_4(@rates, tspan, f0, h, rk);
h  = 0.5; [t21, f21] = rk1_4(@rates, tspan, f0, h, rk);

rk = 3;
h  = 0.5; [t3,  f3]  = rk1_4(@rates, tspan, f0, h, rk);
h  = 1.0; [t31, f31] = rk1_4(@rates, tspan, f0, h, rk);

rk = 4;
h  = 1.0; [t4,  f4]  = rk1_4(@rates, tspan, f0, h, rk);
h  = 2.0; [t41, f41] = rk1_4(@rates, tspan, f0, h, rk);

output

return

% ~~~~~~~~~~~~~~~~~~~~~~~~
function dfdt = rates(t,f)
% ------------------------
%{
  This function calculates first and second time derivatives
  of x as governed by the equation

  x" + 2*z*wn*x' + wn^2*x = (Fo/m)*sin(w*t)

  Dx   - velocity (x')
  D2x  - acceleration (x")
  f    - column vector containing x  and Dx  at time t
  dfdt - column vector containing Dx and D2x at time t
```

```
  User M-functions required: none
%}
% ~~~~~~~~~~~~~~~~~~~~~~~~~

x    = f(1);
Dx   = f(2);
D2x  = Fo/m*sin(w*t) - 2*z*wn*Dx - wn^2*x;
dfdt = [Dx; D2x];
end %rates

% ~~~~~~~~~~~~~
function output
% -------------
%...Exact solution:
wd  = wn*sqrt(1 - z^2);
den = (wn^2 - w^2)^2 + (2*w*wn*z)^2;
C1  = (wn^2 - w^2)/den*Fo/m;
C2  = -2*w*wn*z/den*Fo/m;
A   = x0*wn/wd + x_dot0/wd +(w^2 + (2*z^2 - 1)*wn^2)/den*w/wd*Fo/m;
B   = x0 + 2*w*wn*z/den*Fo/m;

t   = linspace(t0, tf, 5000);
x   = (A*sin(wd*t) + B*cos(wd*t)).*exp(-wn*z*t) ...
      + C1*sin(w*t) + C2*cos(w*t);

%...Plot solutions
%   Exact:
subplot(5,1,1)
plot(t/max(t),     x/max(x),              'k',  'LineWidth',1)
grid off
axis tight
title('Exact')

%   RK1:
subplot(5,1,2)
plot(t1/max(t1),   f1(:,1)/max(f1(:,1)),   '-r', 'LineWidth',1)
hold on
plot(t11/max(t11), f11(:,1)/max(f11(:,1)), '-k')
grid off
axis tight
title('RK1')
legend('h = 0.01', 'h = 0.1')

%   RK2:
subplot(5,1,3)
plot(t2/max(t2),   f2(:,1)/max(f2(:,1)),   '-r', 'LineWidth',1)
hold on
plot(t21/max(t21), f21(:,1)/max(f21(:,1)), '-k')
```

```
grid off
axis tight
title('RK2')
legend('h = 0.1', 'h = 0.5')

%   RK3:
subplot(5,1,4)
plot(t3/max(t3),   f3(:,1)/max(f3(:,1)),   '-r', 'LineWidth',1)
hold on
plot(t31/max(t31), f31(:,1)/max(f31(:,1)), '-k')
grid off
axis tight
title('RK3')
legend('h = 0.5', 'h = 1.0')

%   RK4:
subplot(5,1,5)
plot(t4/max(t4),   f4(:,1)/max(f4(:,1)),   '-r', 'LineWidth',1)
hold on
grid off
plot(t41/max(t41), f41(:,1)/max(f41(:,1)), '-k')
axis tight
title('RK4')
legend('h = 1.0', 'h = 2.0')
end %output

end %Example_1_18
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## D.3 Algorithm 1.2: Numerical integration by Heun's predictor-corrector method

### Function file: heun.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function [tout, yout] = heun(ode_function, tspan, y0, h)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function uses the predictor-corrector method to integrate a system
  of first-order differential equations dy/dt = f(t,y).

  y             - column vector of solutions
  f             - column vector of the derivatives dy/dt
  ode_function  - handle for the user M-function in which the derivatives
                  f are computed
  t             - time
```

```
  t0           - initial time
  tf           - final time
  tspan        - the vector [t0 tf] giving the time interval for the
                 solution
  h            - time step
  y0           - column vector of initial values of the vector y
  tout         - column vector of the times at which y was evaluated
  yout         - a matrix, each row of which contains the components of y
                 evaluated at the corresponding time in tout
  feval        - a built-in MATLAB function which executes 'ode_function'
                 at the arguments t and y
  tol          - Maximum allowable relative error for determining
                 convergence of the corrector
  itermax      - maximum allowable number of iterations for corrector
                 convergence
  iter         - iteration number in the corrector convergence loop
  t1           - time at the beginning of a time step
  y1           - value of y at the beginning of a time step
  f1           - derivative of y at the beginning of a time step
  f2           - derivative of y at the end of a time step
  favg         - average of f1 and f2
  y2p          - predicted value of y at the end of a time step
  y2           - corrected value of y at the end of a time step
  err          - maximum relative error (for all components) between y2p
                 and y2 for given iteration
  eps          - unit roundoff error (the smallest number for which
                 1 + eps > 1). Used to avoid a zero denominator.

  User M-function required: ode_function
%}
% ----------------------------------------------------------------------

tol      = 1.e-6;
itermax  = 100;

t0       = tspan(1);
tf       = tspan(2);
t        = t0;
y        = y0;
tout     = t;
yout     = y';

while t < tf
    h    = min(h, tf-t);
    t1   = t;
    y1   = y;
    f1   = feval(ode_function, t1, y1);
    y2   = y1 + f1*h;
```

```
    t2   = t1 + h;
    err  = tol + 1;
    iter = 0;
    while err > tol && iter <= itermax
        y2p  = y2;
        f2   = feval(ode_function, t2, y2p);
        favg = (f1 + f2)/2;
        y2   = y1 + favg*h;
        err  = max(abs((y2 - y2p)./(y2 + eps)));
        iter = iter + 1;
    end

    if iter > itermax
        fprintf('\n Maximum no. of iterations (%g)',itermax)
        fprintf('\n exceeded at time = %g',t)
        fprintf('\n in function "heun."\n\n')
        return
    end

    t    = t + h;
    y    = y2;
    tout = [tout;t];  % adds t to the bottom of the column vector tout
    yout = [yout;y']; % adds y' to the bottom of the matrix yout
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Function file: Example_1_19.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function Example_1_19
% ~~~~~~~~~~~~~~~~~~~~~~~
%{
  This program uses Heun's method with two different time steps to solve
  for and plot the response of a damped single degree of freedom
  spring-mass system to a sinusoidal forcing function, represented by

  x" + 2*z*wn*x' + wn^2*x = (Fo/m)*sin(w*t)

  The numerical integration is done in the external function 'heun',
  which uses the subfunction 'rates' herein to compute the derivatives.

  x      - displacement (m)
  '      - shorthand for d/dt
  t      - time (s)
  wn     - natural circular frequency (radians/s)
  z      - damping factor
  Fo     - amplitude of the sinusoidal forcing function (N)
```

```
  m     - mass (kg)
  w     - forcing frequency (radians/s)
  t0    - initial time (s)
  tf    - final time (s)
  h     - uniform time step (s)
  tspan - row vector containing t0 and tf
  x0    - value of x at t0 (m)
  Dx0   - value of dx/dt at t0 (m/s)
  f0    - column vector containing x0 and Dx0
  t     - column vector of times at which the solution was computed
  f     - a matrix whose columns are:
          column 1: solution for x at the times in t
          column 2: solution for x' at the times in t

  User M-functions required:  heun
  User subfunctions required: rates
%}
% ----------------------------------------------------------------------

clear all; close all; clc

%...System properties:
m       = 1;
z       = 0.03;
wn      = 1;
Fo      = 1;
w       = 0.4*wn;

%...Time range:
t0      = 0;
tf      = 110;
tspan   = [t0 tf];

%...Initial conditions:
x0  = 0;
Dx0 = 0;
f0  = [x0; Dx0];

%...Calculate and plot the solution for h = 1.0:
h        = 1.0;
[t1, f1] = heun(@rates, tspan, f0, h);

%...Calculate and plot the solution for h = 0.1:
h        = 0.1;
[t2, f2] = heun(@rates, tspan, f0, h);

output
```

```
return

% ~~~~~~~~~~~~~~~~~~~~~~~~
function dfdt = rates(t,f)
% ~~~~~~~~~~~~~~~~~~~~~~~~
%
% This function calculates first and second time derivatives of x
% for the forced vibration of a damped single degree of freedom
% system represented by the 2nd order differential equation
%
% x" + 2*z*wn*x' + wn^2*x = (Fo/m)*sin(w*t)
%
% Dx   - velocity
% D2x  - acceleration
% f    - column vector containing x and Dx at time t
% dfdt - column vector containing Dx and D2x at time t
%
% User M-functions required: none
% ------------------------
x    = f(1);
Dx   = f(2);
D2x  = Fo/m*sin(w*t) - 2*z*wn*Dx - wn^2*x;
dfdt = [Dx; D2x];
end %rates

% ~~~~~~~~~~~~~
function output
% ~~~~~~~~~~~~~
plot(t1, f1(:,1), '-r', 'LineWidth',0.5)
xlabel('time, s')
ylabel('x, m')
grid
axis([0 110 -2 2])
hold on
plot(t2, f2(:,1), '-k', 'LineWidth',1)
legend('h = 1.0','h = 0.1')
end %output

end %Example_1_19
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Function file: rkf45.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function [tout, yout] = rkf45(ode_function, tspan, y0, tolerance)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
```

```
  This function uses the Runge-Kutta-Fehlberg 4(5) algorithm to
  integrate a system of first-order differential equations
  dy/dt = f(t,y).

  y             - column vector of solutions
  f             - column vector of the derivatives dy/dt
  t             - time
  a             - Fehlberg coefficients for locating the six solution
                  points (nodes) within each time interval.
  b             - Fehlberg coupling coefficients for computing the
                  derivatives at each interior point
  c4            - Fehlberg coefficients for the fourth-order solution
  c5            - Fehlberg coefficients for the fifth-order solution
  tol           - allowable truncation error
  ode_function  - handle for user M-function in which the derivatives f
                  are computed
  tspan         - the vector [t0 tf] giving the time interval for the
                  solution
  t0            - initial time
  tf            - final time
  y0            - column vector of initial values of the vector y
  tout          - column vector of times at which y was evaluated
  yout          - a matrix, each row of which contains the components of y
                  evaluated at the corresponding time in tout
  h             - time step
  hmin          - minimum allowable time step
  ti            - time at the beginning of a time step
  yi            - values of y at the beginning of a time step
  t_inner       - time within a given time step
  y_inner       - values of y within a given time step
  te            - truncation error for each y at a given time step
  te_allowed    - allowable truncation error
  te_max        - maximum absolute value of the components of te
  ymax          - maximum absolute value of the components of y
  tol           - relative tolerance
  delta         - fractional change in step size
  eps           - unit roundoff error (the smallest number for which
                  1 + eps > 1)
  eps(x)        - the smallest number such that x + eps(x) = x

  User M-function required: ode_function
%}
% ----------------------------------------------------------------

a = [0 1/4 3/8 12/13 1 1/2];
```

```
b = [     0           0           0           0          0
        1/4          0           0           0          0
        3/32       9/32          0           0          0
     1932/2197 -7200/2197  7296/2197        0          0
      439/216       -8      3680/513   -845/4104      0
       -8/27         2     -3544/2565  1859/4104  -11/40];

c4 = [25/216  0  1408/2565    2197/4104   -1/5    0  ];
c5 = [16/135  0  6656/12825  28561/56430  -9/50  2/55];

if nargin < 4
    tol  = 1.e-8;
else
    tol = tolerance;
end

t0   = tspan(1);
tf   = tspan(2);
t    = t0;
y    = y0;
tout = t;
yout = y';
h    = (tf - t0)/100; % Assumed initial time step.

while t < tf
    hmin = 16*eps(t);
    ti   = t;
    yi   = y;
    %...Evaluate the time derivative(s) at six points within the current
    %   interval:
    for i = 1:6
        t_inner = ti + a(i)*h;
        y_inner = yi;
        for j = 1:i-1
            y_inner = y_inner + h*b(i,j)*f(:,j);
        end
        f(:,i) = feval(ode_function, t_inner, y_inner);
    end

    %...Compute the maximum truncation error:
    te     = h*f*(c4' - c5'); % Difference between 4th and
                              % 5th order solutions
    te_max = max(abs(te));

    %...Compute the allowable truncation error:
    ymax       = max(abs(y));
    te_allowed = tol*max(ymax,1.0);
```

```
    %...Compute the fractional change in step size:
    delta = (te_allowed/(te_max + eps))^(1/5);

    %...If the truncation error is in bounds, then update the solution:
    if te_max <= te_allowed
        h     = min(h, tf-t);
        t     = t + h;
        y     = yi + h*f*c5';
        tout  = [tout;t];
        yout  = [yout;y'];
    end

    %...Update the time step:
    h  = min(delta*h, 4*h);
    if h < hmin
        fprintf(['\n\n Warning: Step size fell below its minimum\n'...
                 ' allowable value (%g) at time %g.\n\n'], hmin, t)
        return
    end
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## **Function file:** `Example_1_20.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function Example_1_20
% ~~~~~~~~~~~~~~~~~~~~~
%{
  This program uses RKF4(5) with adaptive step size control
  to solve the differential equation

  x" + mu/x^2 = 0

  The numerical integration is done by the function 'rkf45' which uses
  the subfunction 'rates' herein to compute the derivatives.

  x     - displacement (km)
  '     - shorthand for d/dt
  t     - time (s)
  mu    - = go*RE^2 (km^3/s^2), where go is the sea level gravitational
          acceleration and RE is the radius of the earth.
  x0    - initial value of x
  v0    = initial value of the velocity (x')
  y0    - column vector containing x0 and v0
  t0    - initial time
  tf    - final time
  tspan - a row vector with components t0 and tf
  t     - column vector of the times at which the solution is found
```

```
  f     - a matrix whose columns are:
          column 1: solution for x at the times in t
          column 2: solution for x' at the times in t

  User M-function required:  rkf45
  User subfunction required: rates
%}
% ---------------------------------------------------------------------

clear all; close all; clc

mu      = 398600;
minutes = 60;  %Conversion from minutes to seconds

x0 = 6500;
v0 = 7.8;
y0 = [x0; v0];
t0 = 0;
tf = 70*minutes;

[t,f] = rkf45(@rates, [t0 tf], y0);
plotit
return

% ~~~~~~~~~~~~~~~~~~~~~~~~~
function dfdt = rates(t,f)
% ------------------------
%{
  This function calculates first and second time derivatives of x
  governed by the equation of two-body rectilinear motion

  x" + mu/x^2 = 0

  Dx   - velocity x'
  D2x  - acceleration x"
  f    - column vector containing x and Dx at time t
  dfdt - column vector containing Dx and D2x at time t

  User M-functions required: none
%}
% ~~~~~~~~~~~~~~~~~~~~~~~~~
x    = f(1);
Dx   = f(2);
D2x  = -mu/x^2;
dfdt = [Dx; D2x];
end %rates
```

```
% ~~~~~~~~~~~~~
function plotit
% ~~~~~~~~~~~~~

%...Position vs time:
subplot(2,1,1)
plot(t/minutes,f(:,1), '-ok')
xlabel('time, minutes')
ylabel('position, km')
grid on
axis([-inf inf 5000 15000])

%...Velocity versus time:
subplot(2,1,2)
plot(t/minutes,f(:,2), '-ok')
xlabel('time, minutes')
ylabel('velocity, km/s')
grid on
axis([-inf inf -10 10])
end %plotit

end %Example_1_20
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

# Chapter 2

## D.5 Algorithm 2.1: Numerical solution of the two-body problem relative to an inertial frame

### Function file: `twobody3d.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function twobody3d
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function solves the inertial two-body problem in three dimensions
  numerically using the RKF4(5) method.

  G             - universal gravitational constant (km^3/kg/s^2)
  m1,m2         - the masses of the two bodies (kg)
  m             - the total mass (kg)
  t0            - initial time (s)
  tf            - final time (s)
  R1_0,V1_0     - 3 by 1 column vectors containing the components of tbe
                  initial position (km) and velocity (km/s) of m1
```

```
  R2_0,V2_0     - 3 by 1 column vectors containing the components of the
                  initial position (km) and velocity (km/s) of m2
  y0            - 12 by 1 column vector containing the initial values
                  of the state vectors of the two bodies:
                  [R1_0; R2_0; V1_0; V2_0]
  t             - column vector of the times at which the solution is found
  X1,Y1,Z1      - column vectors containing the X,Y and Z coordinates (km)
                  of m1 at the times in t
  X2,Y2,Z2      - column vectors containing the X,Y and Z coordinates (km)
                  of m2 at the times in t
  VX1, VY1, VZ1 - column vectors containing the X,Y and Z components
                  of the velocity (km/s) of m1 at the times in t
  VX2, VY2, VZ2 - column vectors containing the X,Y and Z components
                  of the velocity (km/s) of m2 at the times in t
  y             - a matrix whose 12 columns are, respectively,
                  X1,Y1,Z1; X2,Y2,Z2; VX1,VY1,VZ1; VX2,VY2,VZ2
  XG,YG,ZG      - column vectors containing the X,Y and Z coordinates (km)
                  the center of mass at the times in t

  User M-function required:   rkf45
  User subfunctions required: rates, output
%}
% ------------------------------------------------------------------------
clc; clear all; close all
G = 6.67259e-20;

%...Input data:
m1   = 1.e26;
m2   = 1.e26;
t0   = 0;
tf   = 480;

R1_0 = [   0;    0;   0];
R2_0 = [3000;    0;   0];

V1_0 = [  10;   20;  30];
V2_0 = [   0;   40;   0];
%...End input data

y0 = [R1_0; R2_0; V1_0; V2_0];

%...Integrate the equations of motion:
[t,y] = rkf45(@rates, [t0 tf], y0);

%...Output the results:
output

return
```

```
% ~~~~~~~~~~~~~~~~~~~~~~~~
function dydt = rates(t,y)
% ~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function calculates the accelerations in Equations 2.19.

  t      - time
  y      - column vector containing the position and velocity vectors
           of the system at time t
  R1, R2 - position vectors of m1 & m2
  V1, V2 - velocity vectors of m1 & m2
  r      - magnitude of the relative position vector
  A1, A2 - acceleration vectors of m1 & m2
  dydt   - column vector containing the velocity and acceleration
           vectors of the system at time t
%}
% -----------------------
R1   = [y(1); y(2); y(3)];
R2   = [y(4); y(5); y(6)];

V1   = [y(7); y(8); y(9)];
V2   = [y(10); y(11); y(12)];

r    = norm(R2 - R1);

A1   = G*m2*(R2 - R1)/r^3;
A2   = G*m1*(R1 - R2)/r^3;

dydt = [V1; V2; A1; A2];

end %rates
% ~~~~~~~~~~~~~~~~~~

% ~~~~~~~~~~~~~
function output
% ~~~~~~~~~~~~~
%{
  This function calculates the trajectory of the center of mass and
  plots
  (a) the motion of m1, m2 and G relative to the inertial frame
  (b) the motion of m2 and G relative to m1
  (c) the motion of m1 and m2 relative to G

  User sub function required: common_axis_settings
%}
% -------------
```

```
%...Extract the particle trajectories:
X1 = y(:,1); Y1 = y(:,2); Z1 = y(:,3);
X2 = y(:,4); Y2 = y(:,5); Z2 = y(:,6);

%...Locate the center of mass at each time step:
XG = []; YG = []; ZG = [];
for i = 1:length(t)
    XG = [XG; (m1*X1(i) + m2*X2(i))/(m1 + m2)];
    YG = [YG; (m1*Y1(i) + m2*Y2(i))/(m1 + m2)];
    ZG = [ZG; (m1*Z1(i) + m2*Z2(i))/(m1 + m2)];
end

%...Plot the trajectories:
figure (1)
title('Figure 2.3: Motion relative to the inertial frame')
hold on
plot3(X1, Y1, Z1, '-r')
plot3(X2, Y2, Z2, '-g')
plot3(XG, YG, ZG, '-b')
common_axis_settings

figure (2)
title('Figure 2.4a: Motion of m2 and G relative to m1')
hold on
plot3(X2 - X1, Y2 - Y1, Z2 - Z1, '-g')
plot3(XG - X1, YG - Y1, ZG - Z1, '-b')
common_axis_settings

figure (3)
title('Figure 2.4b: Motion of m1 and m2 relative to G')
hold on
plot3(X1 - XG, Y1 - YG, Z1 - ZG, '-r')
plot3(X2 - XG, Y2 - YG, Z2 - ZG, '-g')
common_axis_settings

% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function common_axis_settings
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function establishes axis properties common to the several plots.
%}
% ---------------------------
text(0, 0, 0, 'o')
axis('equal')
view([2,4,1.2])
grid on
axis equal
xlabel('X (km)')
```

```
ylabel('Y (km)')
zlabel('Z (km)')
end %common_axis_settings

end %output

end %twobody3d
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## D.6 Algorithm 2.2: Numerical solution of the two-body relative motion problem

### Function file: `orbit.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function orbit
% ~~~~~~~~~~~~~
%{
  This function computes the orbit of a spacecraft by using rkf45 to
  numerically integrate Equation 2.22.

  It also plots the orbit and computes the times at which the maximum
  and minimum radii occur and the speeds at those times.

  hours     - converts hours to seconds
  G         - universal gravitational constant (km^3/kg/s^2)
  m1        - planet mass (kg)
  m2        - spacecraft mass (kg)
  mu        - gravitational parameter (km^3/s^2)
  R         - planet radius (km)
  r0        - initial position vector (km)
  v0        - initial velocity vector (km/s)
  t0,tf     - initial and final times (s)
  y0        - column vector containing r0 and v0
  t         - column vector of the times at which the solution is found
  y         - a matrix whose columns are:
                 columns 1, 2 and 3:
                    The solution for the x, y and z components of the
                    position vector r at the times in t
                 columns 4, 5 and 6:
                    The solution for the x, y and z components of the
                    velocity vector v at the times in t
  r         - magnitude of the position vector at the times in t
  imax      - component of r with the largest value
  rmax      - largest value of r
  imin      - component of r with the smallest value
```

```
  rmin      - smallest value of r
  v_at_rmax - speed where r = rmax
  v_at_rmin - speed where r = rmin

  User M-function required:   rkf45
  User subfunctions required: rates, output
%}
% ------------------------------------------------------------------

clc; close all; clear all

hours = 3600;
G     = 6.6742e-20;

%...Input data:
%   Earth:
m1 = 5.974e24;
R  = 6378;
m2 = 1000;

r0 = [8000 0 6000];
v0 = [0 7 0];

t0 = 0;
tf = 4*hours;
%...End input data


%...Numerical integration:
mu    = G*(m1 + m2);
y0    = [r0 v0]';
[t,y] = rkf45(@rates, [t0 tf], y0);

%...Output the results:
output

return

% ~~~~~~~~~~~~~~~~~~~~~~~~~
function dydt = rates(t,f)
% ~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function calculates the acceleration vector using Equation 2.22.

  t          - time
  f          - column vector containing the position vector and the
               velocity vector at time t
  x, y, z    - components of the position vector r
```

```
  r          - the magnitude of the position vector
  vx, vy, vz - components of the velocity vector v
  ax, ay, az - components of the acceleration vector a
  dydt       - column vector containing the velocity and acceleration
               components
%}
% -----------------------
x    = f(1);
y    = f(2);
z    = f(3);
vx   = f(4);
vy   = f(5);
vz   = f(6);

r    = norm([x y z]);

ax   = -mu*x/r^3;
ay   = -mu*y/r^3;
az   = -mu*z/r^3;

dydt = [vx vy vz ax ay az]';
end %rates


% ~~~~~~~~~~~~~
function output
% ~~~~~~~~~~~~~
%{
  This function computes the maximum and minimum radii, the times they
  occur and the speed at those times. It prints those results to
  the command window and plots the orbit.

  r         - magnitude of the position vector at the times in t
  imax      - the component of r with the largest value
  rmax      - the largest value of r
  imin      - the component of r with the smallest value
  rmin      - the smallest value of r
  v_at_rmax - the speed where r = rmax
  v_at_rmin - the speed where r = rmin

  User subfunction required: light_gray
%}
% -------------
for i = 1:length(t)
    r(i) = norm([y(i,1) y(i,2) y(i,3)]);
end

[rmax imax] = max(r);
```

```
[rmin imin] = min(r);

v_at_rmax   = norm([y(imax,4) y(imax,5) y(imax,6)]);
v_at_rmin   = norm([y(imin,4) y(imin,5) y(imin,6)]);

%...Output to the command window:
fprintf('\n\n--------------------------------------------------------\n')
fprintf('\n Earth Orbit\n')
fprintf(' %s\n', datestr(now))
fprintf('\n The initial position is [%g, %g, %g] (km).',...
                                             r0(1), r0(2), r0(3))
fprintf('\n   Magnitude = %g km\n', norm(r0))
fprintf('\n The initial velocity is [%g, %g, %g] (km/s).',...
                                             v0(1), v0(2), v0(3))
fprintf('\n   Magnitude = %g km/s\n', norm(v0))
fprintf('\n Initial time = %g h.\n Final time   = %g h.\n',0,tf/hours)
fprintf('\n The minimum altitude is %g km at time = %g h.',...
            rmin-R, t(imin)/hours)
fprintf('\n The speed at that point is %g km/s.\n', v_at_rmin)
fprintf('\n The maximum altitude is %g km at time = %g h.',...
            rmax-R, t(imax)/hours)
fprintf('\n The speed at that point is %g km/s\n', v_at_rmax)
fprintf('\n--------------------------------------------------------\n\n')

%...Plot the results:
%   Draw the planet
[xx, yy, zz] = sphere(100);
surf(R*xx, R*yy, R*zz)
colormap(light_gray)
caxis([-R/100 R/100])
shading interp

%   Draw and label the X, Y and Z axes
line([0 2*R],   [0 0],   [0 0]); text(2*R,   0,   0, 'X')
line(  [0 0], [0 2*R],   [0 0]); text(  0, 2*R,   0, 'Y')
line(  [0 0],   [0 0], [0 2*R]); text(  0,   0, 2*R, 'Z')

%   Plot the orbit, draw a radial to the starting point
%   and label the starting point (o) and the final point (f)
hold on
plot3(  y(:,1),    y(:,2),    y(:,3),'k')
line([0 r0(1)], [0 r0(2)], [0 r0(3)])
text(   y(1,1),    y(1,2),    y(1,3), 'o')
text( y(end,1),  y(end,2),  y(end,3), 'f')

%   Select a view direction (a vector directed outward from the origin)
view([1,1,.4])
```

```
%   Specify some properties of the graph
grid on
axis equal
xlabel('km')
ylabel('km')
zlabel('km')

% ~~~~~~~~~~~~~~~~~~~~~~~~
function map = light_gray
% ~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function creates a color map for displaying the planet as light
  gray with a black equator.

  r - fraction of red
  g - fraction of green
  b - fraction of blue

%}
% ----------------------
r = 0.8; g = r; b = r;
map = [r g b
       0 0 0
       r g b];
end %light_gray

end %output

end %orbit
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## D.7 Calculation of the Lagrange *f* and *g* functions and their time derivatives in terms of change in true anomaly

**Function file:** `f_and_g_ta.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function [f, g] = f_and_g_ta(r0, v0, dt, mu)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function calculates the Lagrange f and g coefficients from the
  change in true anomaly since time t0.

  mu  - gravitational parameter (km^3/s^2)
  dt  - change in true anomaly (degrees)
  r0  - position vector at time t0 (km)
```

```
  v0  - velocity vector at time t0 (km/s)
  h   - angular momentum (km^2/s)
  vr0 - radial component of v0 (km/s)
  r   - radial position after the change in true anomaly
  f   - the Lagrange f coefficient (dimensionless)
  g   - the Lagrange g coefficient (s)

  User M-functions required:  None
%}
% --------------------------------------------

h   = norm(cross(r0,v0));
vr0 = dot(v0,r0)/norm(r0);
r0  = norm(r0);
s   = sind(dt);
c   = cosd(dt);

%...Equation 2.152:
r   = h^2/mu/(1 + (h^2/mu/r0 - 1)*c - h*vr0*s/mu);

%...Equations 2.158a & b:
f   = 1 - mu*r*(1 - c)/h^2;
g   = r*r0*s/h;

end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Function file: fDot_and_gDot_ta.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function [fdot, gdot] = fDot_and_gDot_ta(r0, v0, dt, mu)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function calculates the time derivatives of the Lagrange
  f and g coefficients from the change in true anomaly since time t0.

  mu   - gravitational parameter (km^3/s^2)
  dt   - change in true anomaly (degrees)
  r0   - position vector at time t0 (km)
  v0   - velocity vector at time t0 (km/s)
  h    - angular momentum (km^2/s)
  vr0  - radial component of v0 (km/s)
  fdot - time derivative of the Lagrange f coefficient (1/s)
  gdot - time derivative of the Lagrange g coefficient (dimensionless)

  User M-functions required:  None
%}
```

```
% --------------------------------------------------------

h   = norm(cross(r0,v0));
vr0 = dot(v0,r0)/norm(r0);
r0  = norm(r0);
c   = cosd(dt);
s   = sind(dt);

%...Equations 2.158c & d:
fdot = mu/h*(vr0/h*(1 - c) - s/r0);
gdot = 1 - mu*r0/h^2*(1 - c);

end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## D.8 Algorithm 2.3: Calculate the state vector from the initial state vector and the change in true anomaly

### Function file: rv_from_r0v0_ta.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function [r,v] = rv_from_r0v0_ta(r0, v0, dt, mu)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function computes the state vector (r,v) from the
  initial state vector (r0,v0) and the change in true anomaly.

  mu - gravitational parameter (km^3/s^2)
  r0 - initial position vector (km)
  v0 - initial velocity vector (km/s)
  dt - change in true anomaly (degrees)
  r  - final position vector (km)
  v  - final velocity vector (km/s)

  User M-functions required: f_and_g_ta, fDot_and_gDot_ta
%}
% ------------------------------------------------------------------

%global mu

%...Compute the f and g functions and their derivatives:
[f, g]       =       f_and_g_ta(r0, v0, dt, mu);
[fdot, gdot] = fDot_and_gDot_ta(r0, v0, dt, mu);

%...Compute the final position and velocity vectors:
r =    f*r0 +    g*v0;
```

```
v = fdot*r0 + gdot*v0;

end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Script file: Example_2_13.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_2_13
% ~~~~~~~~~~~~~
%{
  This program computes the state vector [R,V] from the initial
  state vector [R0,V0] and the change in true anomaly, using the
  data in Example 2.13.

  mu - gravitational parameter (km^3/s^2)
  R0 - the initial position vector (km)
  V0 - the initial velocity vector (km/s)
  r0 - magnitude of R0
  v0 - magnitude of V0
  R  - final position vector (km)
  V  - final velocity vector (km/s)
  r  - magnitude of R
  v  - magnitude of V
  dt - change in true anomaly (degrees)

 User M-functions required: rv_from_r0v0_ta

%}
% --------------------------------------------------

clear all; clc
mu = 398600;

%...Input data:
R0 = [8182.4 -6865.9 0];
V0 = [0.47572 8.8116 0];
dt = 120;
%...End input data

%...Algorithm 2.3:
[R,V] = rv_from_r0v0_ta(R0, V0, dt, mu);

r  = norm(R);
v  = norm(V);
r0 = norm(R0);
v0 = norm(V0);
```

```
fprintf('---------------------------------------------------------')
fprintf('\n Example 2.9 \n')
fprintf('\n Initial state vector:\n')
fprintf('\n   r = [%g, %g, %g] (km)', R0(1), R0(2), R0(3))
fprintf('\n     magnitude = %g\n', norm(R0))

fprintf('\n   v = [%g, %g, %g] (km/s)', V0(1), V0(2), V0(3))
fprintf('\n     magnitude = %g', norm(V0))

fprintf('\n\n State vector after %g degree change in true anomaly:\n', dt)
fprintf('\n   r = [%g, %g, %g] (km)', R(1), R(2), R(3))
fprintf('\n     magnitude = %g\n', norm(R))
fprintf('\n   v = [%g, %g, %g] (km/s)', V(1), V(2), V(3))
fprintf('\n     magnitude = %g', norm(V))
fprintf('\n-----------------------------------------------------------\n')
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

***Output from*** `Example_2_13.m`

```
-----------------------------------------------------------
 Example 2.9

 Initial state vector:

   r = [8182.4, -6865.9, 0] (km)
     magnitude = 10681.4

   v = [0.47572, 8.8116, 0] (km/s)
     magnitude = 8.82443

 State vector after 120 degree change in true anomaly:

   r = [1454.99, 8251.47, 0] (km)
     magnitude = 8378.77

   v = [-8.13238, 5.67854, -0] (km/s)
     magnitude = 9.91874
-----------------------------------------------------------
```

## D.9 Algorithm 2.4: Find the root of a function using the bisection method

### Function file: `bisect.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function root = bisect(fun, xl, xu)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
```

```
  This function evaluates a root of a function using
  the bisection method.

  tol  - error to within which the root is computed
  n    - number of iterations
  xl   - low end of the interval containing the root
  xu   - upper end of the interval containing the root
  i    - loop index
  xm   - mid-point of the interval from xl to xu
  fun  - name of the function whose root is being found
  fxl  - value of fun at xl
  fxm  - value of fun at xm
  root - the computed root

  User M-functions required: none
%}
% ---------------------------------------------

tol = 1.e-6;
n   = ceil(log(abs(xu - xl)/tol)/log(2));

for i = 1:n
    xm  = (xl + xu)/2;
    fxl = feval(fun, xl);
    fxm = feval(fun, xm);
    if fxl*fxm > 0
        xl = xm;
    else
        xu = xm;
    end
end

root = xm;

end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

### Function file: Example_2_16.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function Example_2_16
% ~~~~~~~~~~~~~~~~~~~
%{
  This program uses the bisection method to find the three roots of
  Equation 2.204 for the earth-moon system.

  m1  - mass of the earth (kg)
```

```
  m2  - mass of the moon (kg)
  r12 - distance from the earth to the moon (km)
  p   - ratio of moon mass to total mass
  xl  - vector containing the low-side estimates of the three roots
  xu  - vector containing the high-side estimates of the three roots
  x   - vector containing the three computed roots

  User M-function required: bisect
  User subfunction required: fun
%}
% ----------------------------------------------

clear all; clc

%...Input data:
m1  = 5.974e24;
m2  = 7.348e22;
r12 = 3.844e5;

xl  = [-1.1  0.5  1.0];
xu  = [-0.9  1.0  1.5];
%...End input data

p   = m2/(m1 + m2);

for i = 1:3
    x(i) = bisect(@fun, xl(i), xu(i));
end

%...Output the results
output

return

% ~~~~~~~~~~~~~~~~~
function f = fun(z)
% -----------------
%{
  This subroutine evaluates the function in Equation 2.204.

  z - the dimensionless x-coordinate
  p - defined above
  f - the value of the function

%}
% ~~~~~~~~~~~~~~~~~
f = (1 - p)*(z + p)/abs(z + p)^3 + p*(z + p - 1)/abs(z + p - 1)^3 - z;
end %fun
```

```
% ~~~~~~~~~~~~~
function output
% ~~~~~~~~~~~~~
%{
  This function prints out the x-coordinates of L1, L2 and L3
  relative to the center of mass.
%}
%...Output to the command window:
fprintf('\n\n--------------------------------------------\n')
fprintf('\n For\n')
fprintf('\n   m1 = %g kg', m1)
fprintf('\n   m2 = %g kg', m2)
fprintf('\n  r12 = %g km\n', r12)
fprintf('\n the 3 colinear Lagrange points (the roots of\n')
fprintf(' Equation 2.204) are:\n')
fprintf('\n L3: x = %10g km    (f(x3) = %g)',x(1)*r12, fun(x(1)))
fprintf('\n L1: x = %10g km    (f(x1) = %g)',x(2)*r12, fun(x(2)))
fprintf('\n L2: x = %10g km    (f(x2) = %g)',x(3)*r12, fun(x(3)))
fprintf('\n\n--------------------------------------------\n')

end %output

end %Example_2_16
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

***Output from*** `Example_2_16.m`

```
--------------------------------------------

 For

   m1 = 5.974e+24 kg
   m2 = 7.348e+22 kg
  r12 = 384400 km

 the 3 colinear Lagrange points (the roots of
 Equation 2.204) are:

 L3: x =    -386346 km    (f(x3) = -1.55107e-06)
 L1: x =     321710 km    (f(x1) = 5.12967e-06)
 L2: x =     444244 km    (f(x2) = -4.92782e-06)

--------------------------------------------
```

## D.10 MATLAB solution of Example 2.18

### Function file: Example_2_18.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function Example_2_18
% ~~~~~~~~~~~~~~~~~~~~
%{
  This program uses the Runge-Kutta-Fehlberg 4(5) method to solve the
  earth-moon restricted three-body problem (Equations 2.192a and 2.192b)
  for the trajectory of a spacecraft having the initial conditions
  specified in Example 2.18.

  The numerical integration is done in the external function 'rkf45',
  which uses the subfunction 'rates' herein to compute the derivatives.

  days       - converts days to seconds
  G          - universal gravitational constant (km^3/kg/s^2)
  rmoon      - radius of the moon (km)
  rearth     - radius of the earth (km)
  r12        - distance from center of earth to center of moon (km)
  m1,m2      - masses of the earth and of the moon, respectively (kg)
  M          - total mass of the restricted 3-body system (kg)
  mu         - gravitational parameter of earth-moon system (km^3/s^2)
  mu1,mu2    - gravitational parameters of the earth and of the moon,
               respectively (km^3/s^2)
  pi_1,pi_2 - ratios of the earth mass and the moon mass, respectively,
               to the total earth-moon mass
  W          - angular velocity of moon around the earth (rad/s)
  x1,x2      - x-coordinates of the earth and of the moon, respectively,
               relative to the earth-moon barycenter (km)
  d0         - initial altitude of spacecraft (km)
  phi        - polar azimuth coordinate (degrees) of the spacecraft
               measured positive counterclockwise from the earth-moon line
  v0         - initial speed of spacecraft relative to rotating earth-moon
               system (km/s)
  gamma      - initial flight path angle (degrees)
  r0         - initial radial distance of spacecraft from the earth (km)
  x,y        - x and y coordinates of spacecraft in rotating earth-moon
               system (km)
  vx,vy      - x and y components of spacecraft velocity relative to
               rotating earth-moon system (km/s)
  f0         - column vector containing the initial values of x, y, vx and vy
  t0,tf      - initial time and final times (s)
  t          - column vector of times at which the solution was computed
  f          - a matrix whose columns are:
               column 1: solution for x  at the times in t
```

```
              column 2: solution for y  at the times in t
              column 3: solution for vx at the times in t
              column 4: solution for vy at the times in t
  xf,yf     - x and y coordinates of spacecraft in rotating earth-moon
              system at tf
  vxf, vyf  - x and y components of spacecraft velocity relative to
              rotating earth-moon system at tf
  df        - distance from surface of the moon at tf
  vf        - relative speed at tf


  User M-functions required:  rkf45
  User subfunctions required: rates, circle
%}
% ---------------------------------------------

clear all; close all; clc

days   =  24*3600;
G      =  6.6742e-20;
rmoon  =  1737;
rearth =  6378;
r12    =  384400;
m1     =  5974e21;
m2     =  7348e19;

M      =  m1 + m2;;
pi_1   =  m1/M;
pi_2   =  m2/M;

mu1    =  398600;
mu2    =  4903.02;
mu     =  mu1 + mu2;

W      =  sqrt(mu/r12^3);
x1     = -pi_2*r12;
x2     =  pi_1*r12;

%...Input data:
d0     =  200;
phi    =  -90;
v0     =  10.9148;
gamma  =  20;
t0     =  0;
tf     =  3.16689*days;

r0     =  rearth + d0;
x      =  r0*cosd(phi) + x1;
```

```
y      =  r0*sind(phi);

vx     =  v0*(sind(gamma)*cosd(phi) - cosd(gamma)*sind(phi));
vy     =  v0*(sind(gamma)*sind(phi) + cosd(gamma)*cosd(phi));
f0     =  [x; y; vx; vy];

%...Compute the trajectory:
[t,f]  = rkf45(@rates, [t0 tf], f0);
x      = f(:,1);
y      = f(:,2);
vx     = f(:,3);
vy     = f(:,4);

xf     = x(end);
yf     = y(end);

vxf    = vx(end);
vyf    = vy(end);

df     = norm([xf - x2, yf - 0]) - rmoon;
vf     = norm([vxf, vyf]);

%...Output the results:
output
return

% ~~~~~~~~~~~~~~~~~~~~~~~~
function dfdt = rates(t,f)
% ~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This subfunction calculates the components of the relative acceleration
  for the restricted 3-body problem, using Equations 2.192a and 2.192b.

  ax,ay - x and y components of relative acceleration (km/s^2)
  r1    - spacecraft distance from the earth (km)
  r2    - spacecraft  distance from the moon (km)
  f     - column vector containing x,  y,  vx and vy at time t
  dfdt  - column vector containing vx, vy, ax and ay at time t

  All other variables are defined above.

  User M-functions required: none
%}
% ------------------------
x      = f(1);
y      = f(2);
vx     = f(3);
vy     = f(4);
```

```
r1     = norm([x + pi_2*r12, y]);
r2     = norm([x - pi_1*r12, y]);

ax     =  2*W*vy + W^2*x - mu1*(x - x1)/r1^3 - mu2*(x - x2)/r2^3;
ay     = -2*W*vx + W^2*y - (mu1/r1^3 + mu2/r2^3)*y;

dfdt   = [vx; vy; ax; ay];
end %rates

% ~~~~~~~~~~~~~~
function output
% ~~~~~~~~~~~~~~
%{
  This subfunction echoes the input data and prints the results to the
  command window. It also plots the trajectory.

  User M-functions required: none
  User subfunction required: circle
%}
% -------------

fprintf('-----------------------------------------------------------')
fprintf('\n Example 2.18: Lunar trajectory using the restricted')
fprintf('\n three body equations.\n')
fprintf('\n Initial Earth altitude (km)         = %g', d0)
fprintf('\n Initial angle between radial')
fprintf('\n   and earth-moon line (degrees)     = %g', phi)
fprintf('\n Initial flight path angle (degrees) = %g', gamma)
fprintf('\n Flight time (days)                  = %g', tf/days)
fprintf('\n Final distance from the moon (km)   = %g', df)
fprintf('\n Final relative speed (km/s)         = %g', vf)
fprintf('\n-----------------------------------------------------------\n')

%...Plot the trajectory and place filled circles representing the earth
%   and moon on the plot:
plot(x, y)
%   Set plot display parameters
xmin = -20.e3;  xmax =  4.e5;
ymin = -20.e3;  ymax =  1.e5;
axis([xmin xmax ymin ymax])
axis equal
xlabel('x, km'); ylabel('y, km')
grid on
hold on

%...Plot the earth (blue) and moon (green) to scale
earth  = circle(x1, 0, rearth);
moon   = circle(x2, 0, rmoon);
```

```
fill(earth(:,1), earth(:,2),'b')
fill( moon(:,1),  moon(:,2),'g')

% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function xy = circle(xc, yc, radius)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This subfunction calculates the coordinates of points spaced
  0.1 degree apart around the circumference of a circle.

  x,y    - x and y coordinates of a point on the circumference
  xc,yc  - x and y coordinates of the center of the circle
  radius - radius of the circle
  xy     - an array containing the x coordinates in column 1 and the
           y coordinates in column 2

  User M-functions required: none
%}
% ---------------------------------
x      = xc + radius*cosd(0:0.1:360);
y      = yc + radius*sind(0:0.1:360);
xy     = [x', y'];

end %circle

end %output

end %Example_2_18
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

***Output from*** `Example_2_18.m`

```
------------------------------------------------------------
 Example 2.18: Lunar trajectory using the restricted
three body equations.

 Initial Earth altitude (km)         = 200
 Initial angle between radial
   and earth-moon line (degrees)     = -90
 Initial flight path angle (degrees) = 20
 Flight time (days)                  = 3.16689
 Final distance from the moon (km)   = 255.812
 Final relative speed (km/s)         = 2.41494
------------------------------------------------------------
```

# Chapter 3

## D.11 Algorithm 3.1: Solution of Kepler's equation by Newton's method

**Function file:** `kepler_E.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function E = kepler_E(e, M)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function uses Newton's method to solve Kepler's
  equation  E - e*sin(E) = M  for the eccentric anomaly,
  given the eccentricity and the mean anomaly.

  E  - eccentric anomaly (radians)
  e  - eccentricity, passed from the calling program
  M  - mean anomaly (radians), passed from the calling program
  pi - 3.1415926...

  User m-functions required: none
%}
% ----------------------------------------------

%...Set an error tolerance:
error = 1.e-8;

%...Select a starting value for E:
if M < pi
    E = M + e/2;
else
    E = M - e/2;
end

%...Iterate on Equation 3.17 until E is determined to within
%...the error tolerance:
ratio = 1;
while abs(ratio) > error
    ratio = (E - e*sin(E) - M)/(1 - e*cos(E));
    E = E - ratio;
end

end %kepler_E
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Script file: Example_3_02.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_3_02
% ~~~~~~~~~~~~~
%{
  This program uses Algorithm 3.1 and the data of Example 3.2 to solve
  Kepler's equation.

  e   - eccentricity
  M   - mean anomaly (rad)
  E   - eccentric anomaly (rad)

  User M-function required: kepler_E
%}
% ----------------------------------------------

clear all; clc

%...Data declaration for Example 3.2:
e = 0.37255;
M = 3.6029;
%...

%...Pass the input data to the function kepler_E, which returns E:
E = kepler_E(e, M);

%...Echo the input data and output to the command window:
fprintf('-----------------------------------------------------')
fprintf('\n Example 3.2\n')
fprintf('\n Eccentricity                = %g',e)
fprintf('\n Mean anomaly (radians)      = %g\n',M)
fprintf('\n Eccentric anomaly (radians) = %g',E)
fprintf('\n-----------------------------------------------------\n')

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

## *Output from* Example_3_02.m

```
-----------------------------------------------------
 Example 3.2

 Eccentricity                = 0.37255
 Mean anomaly (radians)      = 3.6029

 Eccentric anomaly (radians) = 3.47942
-----------------------------------------------------
```

## D.12 Algorithm 3.2: Solution of Kepler's equation for the hyperbola using Newton's method

### Function file: kepler_H.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function F = kepler_H(e, M)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function uses Newton's method to solve Kepler's equation
  for the hyperbola  e*sinh(F) - F = M  for the hyperbolic
  eccentric anomaly, given the eccentricity and the hyperbolic
  mean anomaly.

  F - hyperbolic eccentric anomaly (radians)
  e - eccentricity, passed from the calling program
  M - hyperbolic mean anomaly (radians), passed from the
      calling program

  User M-functions required: none
%}
% ---------------------------------------------

%...Set an error tolerance:
error = 1.e-8;

%...Starting value for F:
F = M;

%...Iterate on Equation 3.45 until F is determined to within
%...the error tolerance:
ratio = 1;
while abs(ratio) > error
    ratio = (e*sinh(F) - F - M)/(e*cosh(F) - 1);
    F = F - ratio;
end

end %kepler_H
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

### Script file: : Example_3_05.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_3_05
% ~~~~~~~~~~~~~
%{
  This program uses Algorithm 3.2 and the data of
```

```
  Example 3.5 to solve Kepler's equation for the hyperbola.

  e - eccentricity
  M - hyperbolic mean anomaly (dimensionless)
  F - hyperbolic eccentric anomaly (dimensionless)

  User M-function required: kepler_H
%}
% ---------------------------------------------
clear

%...Data declaration for Example 3.5:
e = 2.7696;
M = 40.69;
%...

%...Pass the input data to the function kepler_H, which returns F:
F = kepler_H(e, M);

%...Echo the input data and output to the command window:
fprintf('---------------------------------------------------')
fprintf('\n Example 3.5\n')
fprintf('\n Eccentricity                 = %g',e)
fprintf('\n Hyperbolic mean anomaly      = %g\n',M)
fprintf('\n Hyperbolic eccentric anomaly = %g',F)
fprintf('\n-----------------------------------------------------\n')

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

### *Output from* Example_3_05.m

```
-----------------------------------------------------
 Example 3.5

 Eccentricity                 = 2.7696
 Hyperbolic mean anomaly      = 40.69

 Hyperbolic eccentric anomaly = 3.46309
-----------------------------------------------------
```

## D.13 Calculation of the Stumpff functions S(z) and C(z)

The following scripts implement Equations 3.52 and 3.53 for use in other programs.

### Function file: stumpS.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function s = stumpS(z)
```

```
% ~~~~~~~~~~~~~~~~~~~~~~
%{
  This function evaluates the Stumpff function S(z) according
  to Equation 3.52.

  z - input argument
  s - value of S(z)

  User M-functions required: none
%}
% ----------------------------------------------

if z > 0
    s = (sqrt(z) - sin(sqrt(z)))/(sqrt(z))^3;
elseif z < 0
    s = (sinh(sqrt(-z)) - sqrt(-z))/(sqrt(-z))^3;
else
    s = 1/6;
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Function file: `stumpC.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function c = stumpC(z)
% ~~~~~~~~~~~~~~~~~~~~~~
%{
  This function evaluates the Stumpff function C(z) according
  to Equation 3.53.

  z - input argument
  c - value of C(z)

  User M-functions required: none
%}
% ----------------------------------------------

if z > 0
    c = (1 - cos(sqrt(z)))/z;
elseif z < 0
    c = (cosh(sqrt(-z)) - 1)/(-z);
else
    c = 1/2;
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## D.14 Algorithm 3.3: Solution of the universal Kepler's equation using Newton's method

### Function file: kepler_U.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function x = kepler_U(dt, ro, vro, a)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function uses Newton's method to solve the universal
  Kepler equation for the universal anomaly.

  mu   - gravitational parameter (km^3/s^2)
  x    - the universal anomaly (km^0.5)
  dt   - time since x = 0 (s)
  ro   - radial position (km) when x = 0
  vro  - radial velocity (km/s) when x = 0
  a    - reciprocal of the semimajor axis (1/km)
  z    - auxiliary variable (z = a*x^2)
  C    - value of Stumpff function C(z)
  S    - value of Stumpff function S(z)
  n    - number of iterations for convergence
  nMax - maximum allowable number of iterations

  User M-functions required: stumpC, stumpS
%}
% ----------------------------------------------
global mu

%...Set an error tolerance and a limit on the number of iterations:
error = 1.e-8;
nMax  = 1000;

%...Starting value for x:
x = sqrt(mu)*abs(a)*dt;

%...Iterate on Equation 3.65 until convergence occurs within
%...the error tolerance:
n     = 0;
ratio = 1;
while abs(ratio) > error && n <= nMax
    n     = n + 1;
    C     = stumpC(a*x^2);
    S     = stumpS(a*x^2);
    F     = ro*vro/sqrt(mu)*x^2*C + (1 - a*ro)*x^3*S + ro*x - sqrt(mu)*dt;
    dFdx  = ro*vro/sqrt(mu)*x*(1 - a*x^2*S) + (1 - a*ro)*x^2*C + ro;
    ratio = F/dFdx;
```

```
    x     = x - ratio;
end

%...Deliver a value for x, but report that nMax was reached:
if n > nMax
    fprintf('\n **No. iterations of Kepler's equation = %g', n)
    fprintf('\n   F/dFdx                             = %g\n', F/dFdx)
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## **Script file:** Example_3_06.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_3_06
% ~~~~~~~~~~~~
%{
  This program uses Algorithm 3.3 and the data of Example 3.6
  to solve the universal Kepler's equation.

  mu  - gravitational parameter (km^3/s^2)
  x   - the universal anomaly (km^0.5)
  dt  - time since x = 0 (s)
  ro  - radial position when x = 0 (km)
  vro - radial velocity when x = 0 (km/s)
  a   - semimajor axis (km)

  User M-function required: kepler_U
%}
% ---------------------------------------------

clear all; clc
global mu
mu = 398600;

%...Data declaration for Example 3.6:
ro  = 10000;
vro = 3.0752;
dt  = 3600;
a   = -19655;
%...

%...Pass the input data to the function kepler_U, which returns x
%...(Universal Kepler's requires the reciprocal of semimajor axis):
x   = kepler_U(dt, ro, vro, 1/a);

%...Echo the input data and output the results to the command window:
fprintf('-----------------------------------------------------')
```

```
fprintf('\n Example 3.6\n')
fprintf('\n Initial radial coordinate (km) = %g',ro)
fprintf('\n Initial radial velocity (km/s) = %g',vro)
fprintf('\n Elapsed time (seconds)         = %g',dt)
fprintf('\n Semimajor axis (km)            = %g\n',a)
fprintf('\n Universal anomaly (km^0.5)     = %g',x)
fprintf('\n-------------------------------------------------------\n')
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

***Output from*** `Example_3_06.m`

```
-----------------------------------------------------
  Example 3.6

 Initial radial coordinate (km) = 10000
 Initial radial velocity (km/s) = 3.0752
 Elapsed time (seconds)         = 3600
 Semimajor axis (km)            = -19655

 Universal anomaly (km^0.5)     = 128.511
-----------------------------------------------------
```

## D.15 Calculation of the Lagrange coefficients *f* and *g* and their time derivatives in terms of change in universal anomaly

The following scripts implement Equations 3.69 for use in other programs.

### Function file: `f_and_g.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function [f, g] = f_and_g(x, t, ro, a)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function calculates the Lagrange f and g coefficients.

  mu - the gravitational parameter (km^3/s^2)
  a  - reciprocal of the semimajor axis (1/km)
  ro - the radial position at time to (km)
  t  - the time elapsed since ro (s)
  x  - the universal anomaly after time t (km^0.5)
  f  - the Lagrange f coefficient (dimensionless)
  g  - the Lagrange g coefficient (s)

  User M-functions required:  stumpC, stumpS
%}
% ----------------------------------------------
```

```
global mu

z = a*x^2;

%...Equation 3.69a:
f = 1 - x^2/ro*stumpC(z);

%...Equation 3.69b:
g = t - 1/sqrt(mu)*x^3*stumpS(z);

end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Function file: fDot_and_gDot.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function [fdot, gdot] = fDot_and_gDot(x, r, ro, a)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function calculates the time derivatives of the
  Lagrange f and g coefficients.

  mu    - the gravitational parameter (km^3/s^2)
  a     - reciprocal of the semimajor axis (1/km)
  ro    - the radial position at time to (km)
  t     - the time elapsed since initial state vector (s)
  r     - the radial position after time t (km)
  x     - the universal anomaly after time t (km^0.5)
  fdot  - time derivative of the Lagrange f coefficient (1/s)
  gdot  - time derivative of the Lagrange g coefficient (dimensionless)

  User M-functions required:  stumpC, stumpS
%}
% --------------------------------------------------

global mu

z = a*x^2;

%...Equation 3.69c:
fdot = sqrt(mu)/r/ro*(z*stumpS(z) - 1)*x;

%...Equation 3.69d:
gdot = 1 - x^2/r*stumpC(z);
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## D.16 Algorithm 3.4: Calculation of the state vector given the initial state vector and the time lapse $\Delta t$

### Function file: rv_from_r0v0.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function [R,V] = rv_from_r0v0(R0, V0, t)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function computes the state vector (R,V) from the
  initial state vector (R0,V0) and the elapsed time.

  mu - gravitational parameter (km^3/s^2)
  R0 - initial position vector (km)
  V0 - initial velocity vector (km/s)
  t  - elapsed time (s)
  R  - final position vector (km)
  V  - final velocity vector (km/s)

% User M-functions required: kepler_U, f_and_g, fDot_and_gDot
%}
% ---------------------------------------------

global mu

%...Magnitudes of R0 and V0:
r0 = norm(R0);
v0 = norm(V0);

%...Initial radial velocity:
vr0 = dot(R0, V0)/r0;

%...Reciprocal of the semimajor axis (from the energy equation):
alpha = 2/r0 - v0^2/mu;

%...Compute the universal anomaly:
x = kepler_U(t, r0, vr0, alpha);

%...Compute the f and g functions:
[f, g] = f_and_g(x, t, r0, alpha);

%...Compute the final position vector:
R = f*R0 + g*V0;

%...Compute the magnitude of R:
r = norm(R);
```

```
%...Compute the derivatives of f and g:
[fdot, gdot] = fDot_and_gDot(x, r, r0, alpha);

%...Compute the final velocity:
V           = fdot*R0 + gdot*V0;
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Script file: Example_3_07.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_3_07
% ~~~~~~~~~~~~
%
% This program computes the state vector (R,V) from the initial
% state vector (R0,V0) and the elapsed time using the data in
% Example 3.7.
%
% mu - gravitational parameter (km^3/s^2)
% R0 - the initial position vector (km)
% V0 - the initial velocity vector (km/s)
% R  - the final position vector (km)
% V  - the final velocity vector (km/s)
% t  - elapsed time (s)
%
% User m-functions required: rv_from_r0v0
% ----------------------------------------------

clear all; clc
global mu
mu = 398600;

%...Data declaration for Example 3.7:
R0 = [  7000 -12124 0];
V0 = [2.6679 4.6210 0];
t  = 3600;
%...

%...Algorithm 3.4:
[R V] = rv_from_r0v0(R0, V0, t);

%...Echo the input data and output the results to the command window:
fprintf('-----------------------------------------------------')
fprintf('\n Example 3.7\n')
fprintf('\n Initial position vector (km):')
fprintf('\n   r0 = (%g, %g, %g)\n', R0(1), R0(2), R0(3))
fprintf('\n Initial velocity vector (km/s):')
fprintf('\n   v0 = (%g, %g, %g)', V0(1), V0(2), V0(3))
```

```
fprintf('\n\n Elapsed time = %g s\n',t)
fprintf('\n Final position vector (km):')
fprintf('\n   r = (%g, %g, %g)\n', R(1), R(2), R(3))
fprintf('\n Final velocity vector (km/s):')
fprintf('\n   v = (%g, %g, %g)', V(1), V(2), V(3))
fprintf('\n-----------------------------------------------------\n')
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

***Output from*** `Example_3_07`

```
-----------------------------------------------------
 Example 3.7

 Initial position vector (km):
   r0 = (7000, -12124, 0)

 Initial velocity vector (km/s):
   v0 = (2.6679, 4.621, 0)

 Elapsed time = 3600 s

 Final position vector (km):
   r = (-3297.77, 7413.4, 0)

 Final velocity vector (km/s):
   v = (-8.2976, -0.964045, -0)
-----------------------------------------------------
```

## Chapter 4

### D.17 Algorithm 4.1: Obtain the right ascension and declination from the position vector

**Function file:** `ra_and_dec_from_r.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function [ra dec] = ra_and_dec_from_r(r)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function calculates the right ascension and the
  declination from the geocentric equatorial position vector.

  r       - position vector
  l, m, n - direction cosines of r
  ra      - right ascension (degrees)
```

```
  dec     - declination (degrees)
%}
% --------------------------------------------
l = r(1)/norm(r);
m = r(2)/norm(r);
n = r(3)/norm(r);

dec = asind(n);

if m > 0
    ra = acosd(l/cosd(dec));
else
    ra = 360 - acosd(l/cosd(dec));
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Script file: Example_4_01.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example 4.1
% ~~~~~~~~~~~
%{
  This program calculates the right ascension and declination
  from the geocentric equatorial position vector using the data
  in Example 4.1.

  r   - position vector r (km)
  ra  - right ascension (deg)
  dec - declination (deg)

  User M-functions required: ra_and_dec_from_r

%}
% ---------------------------------------------

clear all; clc

r        = [-5368 -1784 3691];
[ra dec] = ra_and_dec_from_r(r);

fprintf('\n ---------------------------------------------------\n')
fprintf('\n Example 4.1\n')
fprintf('\n r               = [%g  %g  %g] (km)', r(1), r(2), r(3))
fprintf('\n right ascension = %g deg', ra)
fprintf('\n declination     = %g deg', dec)
fprintf('\n\n ---------------------------------------------------\n')

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

***Output from*** `Example_4_01.m`

```
-----------------------------------------------------
 Example 4.1

 r                = [-5368  -1784  3691] (km)
  right ascension = 198.384 deg
 declination      = 33.1245 deg

 -----------------------------------------------------
```

## D.18 Algorithm 4.2: Calculation of the orbital elements from the state vector

### Function file: `coe_from_sv.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function coe = coe_from_sv(R,V,mu)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
% This function computes the classical orbital elements (coe)
% from the state vector (R,V) using Algorithm 4.1.
%
  mu   - gravitational parameter (km^3/s^2)
  R    - position vector in the geocentric equatorial frame (km)
  V    - velocity vector in the geocentric equatorial frame (km)
  r, v - the magnitudes of R and V
  vr   - radial velocity component (km/s)
  H    - the angular momentum vector (km^2/s)
  h    - the magnitude of H (km^2/s)
  incl - inclination of the orbit (rad)
  N    - the node line vector (km^2/s)
  n    - the magnitude of N
  cp   - cross product of N and R
  RA   - right ascension of the ascending node (rad)
  E    - eccentricity vector
  e    - eccentricity (magnitude of E)
  eps  - a small number below which the eccentricity is considered
         to be zero
  w    - argument of perigee (rad)
  TA   - true anomaly (rad)
  a    - semimajor axis (km)
  pi   - 3.1415926...
  coe  - vector of orbital elements [h e RA incl w TA a]

  User M-functions required: None
%}
```

```
% -------------------------------------------

eps  = 1.e-10;

r    = norm(R);
v    = norm(V);

vr   = dot(R,V)/r;

H    = cross(R,V);
h    = norm(H);

%...Equation 4.7:
incl = acos(H(3)/h);

%...Equation 4.8:
N    = cross([0 0 1],H);
n    = norm(N);

%...Equation 4.9:
if n ~= 0
    RA = acos(N(1)/n);
    if N(2) < 0
        RA = 2*pi - RA;
    end
else
    RA = 0;
end

%...Equation 4.10:
E = 1/mu*((v^2 - mu/r)*R - r*vr*V);
e = norm(E);

%...Equation 4.12 (incorporating the case e = 0):
if n ~= 0
    if e > eps
        w = acos(dot(N,E)/n/e);
        if E(3) < 0
            w = 2*pi - w;
        end
    else
        w = 0;
    end
else
    w = 0;
end
```

```
%...Equation 4.13a (incorporating the case e = 0):
if e > eps
    TA = acos(dot(E,R)/e/r);
    if vr < 0
        TA = 2*pi - TA;
    end
else
    cp = cross(N,R);
    if cp(3) >= 0
        TA = acos(dot(N,R)/n/r);
    else
        TA = 2*pi - acos(dot(N,R)/n/r);
    end
end

%...Equation 4.62 (a < 0 for a hyperbola):
a = h^2/mu/(1 - e^2);

coe = [h e RA incl w TA a];

  end  %coe_from_sv
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

### **Script file:** `Example_4_03.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_4_03
% ~~~~~~~~~~~~~
%{
  This program uses Algorithm 4.2 to obtain the orbital
  elements from the state vector provided in Example 4.3.

  pi   - 3.1415926...
  deg  - factor for converting between degrees and radians
  mu   - gravitational parameter (km^3/s^2)
  r    - position vector (km) in the geocentric equatorial frame
  v    - velocity vector (km/s) in the geocentric equatorial frame
  coe  - orbital elements [h e RA incl w TA a]
         where h    = angular momentum (km^2/s)
               e    = eccentricity
               RA   = right ascension of the ascending node (rad)
               incl = orbit inclination (rad)
               w    = argument of perigee (rad)
               TA   = true anomaly (rad)
               a    = semimajor axis (km)
  T    - Period of an elliptic orbit (s)
```

```
  User M-function required: coe_from_sv
%}
% ---------------------------------------------
clear all; clc
deg = pi/180;
mu  = 398600;

%...Data declaration for Example 4.3:
r = [ -6045  -3490   2500];
v = [-3.457  6.618  2.533];
%...

%...Algorithm 4.2:
coe = coe_from_sv(r,v,mu);

%...Echo the input data and output results to the command window:
fprintf('-----------------------------------------------------')
fprintf('\n Example 4.3\n')
fprintf('\n Gravitational parameter (km^3/s^2) = %g\n', mu)
fprintf('\n State vector:\n')
fprintf('\n r (km)                             = [%g  %g  %g]', ...
                                              r(1), r(2), r(3))
fprintf('\n v (km/s)                           = [%g  %g  %g]', ...
                                              v(1), v(2), v(3))
disp(' ')
fprintf('\n Angular momentum (km^2/s)          = %g', coe(1))
fprintf('\n Eccentricity                       = %g', coe(2))
fprintf('\n Right ascension (deg)              = %g', coe(3)/deg)
fprintf('\n Inclination (deg)                  = %g', coe(4)/deg)
fprintf('\n Argument of perigee (deg)          = %g', coe(5)/deg)
fprintf('\n True anomaly (deg)                 = %g', coe(6)/deg)
fprintf('\n Semimajor axis (km):               = %g', coe(7))

%...if the orbit is an ellipse, output its period (Equation 2.73):
if coe(2)<1
    T = 2*pi/sqrt(mu)*coe(7)^1.5;
    fprintf('\n Period:')
    fprintf('\n   Seconds                          = %g', T)
    fprintf('\n   Minutes                          = %g', T/60)
    fprintf('\n   Hours                            = %g', T/3600)
    fprintf('\n   Days                             = %g', T/24/3600)
end
fprintf('\n-----------------------------------------------------\n')

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

***Output from*** `Example_4_03`

```
-----------------------------------------------------
 Example 4.3

 Gravitational parameter (km^3/s^2) = 398600

 State vector:

 r (km)                              = [-6045  -3490  2500]
 v (km/s)                            = [-3.457  6.618  2.533]

 Angular momentum (km^2/s)           = 58311.7
 Eccentricity                        = 0.171212
 Right ascension (deg)               = 255.279
 Inclination (deg)                   = 153.249
 Argument of perigee (deg)           = 20.0683
 True anomaly (deg)                  = 28.4456
 Semimajor axis (km):                = 8788.1
 Period:
   Seconds                           = 8198.86
   Minutes                           = 136.648
   Hours                             = 2.27746
   Days                              = 0.0948942
-----------------------------------------------------
```

## D.19 Calculation of $\tan^{-1}$ (*y/x*) to lie in the range 0 to 360°

**Function file:** `atan2d_360.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function t = atan2d_0_360(y,x)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function calculates the arc tangent of y/x in degrees
  and places the result in the range [0, 360].

  t - angle in degrees

%}
% ----------------------------------------------

if x == 0
    if y == 0
        t = 0;
    elseif y > 0
        t = 90;
```

```
    else
        t = 270;
    end
elseif x > 0
    if y >= 0
        t = atand(y/x);
    else
        t = atand(y/x) + 360;
    end
elseif x < 0
    if y == 0
        t = 180;
    else
        t = atand(y/x) + 180;
    end
end

end

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

## D.20 Algorithm 4.3: Obtain the classical Euler angle sequence from a direction cosine matrix

### Function file: dcm_to_euler.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function [alpha beta gamma] = dcm_to_euler(Q)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function finds the angles of the classical Euler sequence
  R3(gamma)*R1(beta)*R3(alpha) from the direction cosine matrix.

  Q     - direction cosine matrix
  alpha - first angle of the sequence (deg)
  beta  - second angle of the sequence (deg)
  gamma - third angle of the sequence (deg)

  User M-function required: atan2d_0_360
%}
% ----------------------------------------------

alpha = atan2d_0_360(Q(3,1), -Q(3,2));
beta  = acosd(Q(3,3));
gamma = atan2d_0_360(Q(1,3), Q(2,3));
```

```
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## D.21 Algorithm 4.4: Obtain the yaw, pitch, and roll angles from a direction cosine matrix

### Function file: dcm_to_ypr.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function [yaw pitch roll] = dcm_to_ypr(Q)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function finds the angles of the yaw-pitch-roll sequence
  R1(gamma)*R2(beta)*R3(alpha) from the direction cosine matrix.

  Q     - direction cosine matrix
  yaw   - yaw angle (deg)
  pitch - pitch angle (deg)
  roll  - roll angle (deg)

  User M-function required: atan2d_0_360
%}
% ---------------------------------------

yaw   = atan2d_0_360(Q(1,2), Q(1,1));
pitch = asind(-Q(1,3));
roll  = atan2d_0_360(Q(2,3), Q(3,3));
end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## D.22 Algorithm 4.5: Calculation of the state vector from the orbital elements

### Function file: sv_from_coe.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function [r, v] = sv_from_coe(coe,mu)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function computes the state vector (r,v) from the
  classical orbital elements (coe).

  mu   - gravitational parameter (km^3;s^2)
  coe  - orbital elements [h e RA incl w TA]
```

```
        where
            h    = angular momentum (km^2/s)
            e    = eccentricity
            RA   = right ascension of the ascending node (rad)
            incl = inclination of the orbit (rad)
            w    = argument of perigee (rad)
            TA   = true anomaly (rad)
  R3_w - Rotation matrix about the z-axis through the angle w
  R1_i - Rotation matrix about the x-axis through the angle i
  R3_W - Rotation matrix about the z-axis through the angle RA
  Q_pX - Matrix of the transformation from perifocal to geocentric
         equatorial frame
  rp   - position vector in the perifocal frame (km)
  vp   - velocity vector in the perifocal frame (km/s)
  r    - position vector in the geocentric equatorial frame (km)
  v    - velocity vector in the geocentric equatorial frame (km/s)

  User M-functions required: none
%}
% ----------------------------------------------

h    = coe(1);
e    = coe(2);
RA   = coe(3);
incl = coe(4);
w    = coe(5);
TA   = coe(6);

%...Equations 4.45 and 4.46 (rp and vp are column vectors):
rp = (h^2/mu) * (1/(1 + e*cos(TA))) * (cos(TA)*[1;0;0] + sin(TA)*[0;1;0]);
vp = (mu/h) * (-sin(TA)*[1;0;0] + (e + cos(TA))*[0;1;0]);

%...Equation 4.34:
R3_W = [ cos(RA)  sin(RA)  0
        -sin(RA)  cos(RA)  0
            0        0     1];

%...Equation 4.32:
R1_i = [1       0          0
        0   cos(incl)  sin(incl)
        0  -sin(incl)  cos(incl)];

%...Equation 4.34:
R3_w = [ cos(w)  sin(w)  0
        -sin(w)  cos(w)  0
           0       0     1];

%...Equation 4.49:
```

```
Q_pX = (R3_w*R1_i*R3_W)';

%...Equations 4.51 (r and v are column vectors):
r = Q_pX*rp;
v = Q_pX*vp;

%...Convert r and v into row vectors:
r = r';
v = v';

end
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Script file: Example_4_07.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_4_07
% ~~~~~~~~~~~~
%{
  This program uses Algorithm 4.5 to obtain the state vector from
  the orbital elements provided in Example 4.7.

  pi  - 3.1415926...
  deg - factor for converting between degrees and radians
  mu  - gravitational parameter (km^3/s^2)
  coe - orbital elements [h e RA incl w TA a]
        where h    = angular momentum (km^2/s)
              e    = eccentricity
              RA   = right ascension of the ascending node (rad)
              incl = orbit inclination (rad)
              w    = argument of perigee (rad)
              TA   = true anomaly (rad)
              a    = semimajor axis (km)
  r   - position vector (km) in geocentric equatorial frame
  v   - velocity vector (km) in geocentric equatorial frame

  User M-function required: sv_from_coe
%}
% ---------------------------------------------
clear all; clc
deg = pi/180;
mu  = 398600;

%...Data declaration for Example 4.5 (angles in degrees):
h    = 80000;
e    = 1.4;
RA   = 40;
incl = 30;
```

```
w    = 60;
TA   = 30;
%...

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

%...Algorithm 4.5 (requires angular elements be in radians):
[r, v] = sv_from_coe(coe, mu);

%...Echo the input data and output the results to the command window:
fprintf('---------------------------------------------------')
fprintf('\n Example 4.7\n')
fprintf('\n Gravitational parameter (km^3/s^2)  = %g\n', mu)
fprintf('\n Angular momentum (km^2/s)           = %g', h)
fprintf('\n Eccentricity                        = %g', e)
fprintf('\n Right ascension (deg)               = %g', RA)
fprintf('\n Argument of perigee (deg)           = %g', w)
fprintf('\n True anomaly (deg)                  = %g', TA)
fprintf('\n\n State vector:')
fprintf('\n   r (km)   = [%g  %g  %g]', r(1), r(2), r(3))
fprintf('\n   v (km/s) = [%g  %g  %g]', v(1), v(2), v(3))
fprintf('\n-----------------------------------------------\n')
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

### ***Output from*** Example_4_05

```
-----------------------------------------------------
 Example 4.7

 Gravitational parameter (km^3/s^2)  = 398600

 Angular momentum (km^2/s)           = 80000
 Eccentricity                        = 1.4
 Right ascension (deg)               = 40
 Argument of perigee (deg)           = 60
 True anomaly (deg)                   = 30

 State vector:
   r (km)   = [-4039.9  4814.56  3628.62]
   v (km/s) = [-10.386  -4.77192  1.74388]
-----------------------------------------------------
```

## D.23 Algorithm 4.6 Calculate the ground track of a satellite from its orbital elements

### Function file: ground_track.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function ground_track
% ~~~~~~~~~~~~~~~~~~~~~
%{
  This program plots the ground track of an earth satellite
  for which the orbital elements are specified.

  mu        - gravitational parameter (km^3/s^2)
  deg       - factor that converts degrees to radians
  J2        - second zonal harmonic
  Re        - earth's radius (km)
  we        - earth's angular velocity (rad/s)
  rP        - perigee of orbit (km)
  rA        - apogee of orbit (km)
  TA, TAo   - true anomaly, initial true anomaly of satellite (rad)
  RA, RAo   - right ascension, initial right ascension of the node (rad)
  incl      - orbit inclination (rad)
  wp, wpo   - argument of perigee, initial argument of perigee (rad)
  n_periods - number of periods for which ground track is to be plotted
  a         - semimajor axis of orbit (km)
  T         - period of orbit (s)
  e         - eccentricity of orbit
  h         - angular momentum of orbit (km^2/s)
  E, Eo     - eccentric anomaly, initial eccentric anomaly (rad)
  M, Mo     - mean anomaly, initial mean anomaly (rad)
  to, tf    - initial and final times for the ground track (s)
  fac       - common factor in Equations 4.53 and 4.53
  RAdot     - rate of regression of the node (rad/s)
  wpdot     - rate of advance of perigee (rad/s)
  times     - times at which ground track is plotted (s)
  ra        - vector of right ascensions of the spacecraft (deg)
  dec       - vector of declinations of the spacecraft (deg)
  TA        - true anomaly (rad)
  r         - perifocal position vector of satellite (km)
  R         - geocentric equatorial position vector (km)
  R1        - DCM for rotation about z through RA
  R2        - DCM for rotation about x through incl
  R3        - DCM for rotation about z through wp
  QxX       - DCM for rotation from perifocal to geocentric equatorial
  Q         - DCM for rotation from geocentric equatorial
              into earth-fixed frame
  r_rel     - position vector in earth-fixed frame (km)
```

```
  alpha     - satellite right ascension (deg)
  delta     - satellite declination (deg)
  n_curves  - number of curves comprising the ground track plot
  RA        - cell array containing the right ascensions for each of
              the curves comprising the ground track plot
  Dec       - cell array containing the declinations for each of
              the curves comprising the ground track plot

  User M-functions required: sv_from_coe, kepler_E, ra_and_dec_from_r
%}
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
clear all; close all; clc
global ra dec n_curves RA Dec

%...Constants
deg       = pi/180;
mu        = 398600;
J2        = 0.00108263;
Re        = 6378;
we        = (2*pi + 2*pi/365.26)/(24*3600);

%...Data declaration for Example 4.12:
rP        = 6700;
rA        = 10000;
TAo       = 230*deg;
Wo        = 270*deg;
incl      = 60*deg;
wpo       = 45*deg;
n_periods = 3.25;
%...End data declaration

%...Compute the initial time (since perigee) and
%   the rates of node regression and perigee advance
a         = (rA + rP)/2;
T         = 2*pi/sqrt(mu)*a^(3/2);
e         = (rA - rP)/(rA + rP);
h         = sqrt(mu*a*(1 - e^2));
Eo        = 2*atan(tan(TAo/2)*sqrt((1-e)/(1+e)));
Mo        = Eo - e*sin(Eo);
to        = Mo*(T/2/pi);
tf        = to + n_periods*T;
fac       = -3/2*sqrt(mu)*J2*Re^2/(1-e^2)^2/a^(7/2);
Wdot      = fac*cos(incl);
wpdot     = fac*(5/2*sin(incl)^2 - 2);

find_ra_and_dec
form_separate_curves
plot_ground_track
```

```
print_orbital_data

return

% ~~~~~~~~~~~~~~~~~~~~~~~~
function find_ra_and_dec
% ~~~~~~~~~~~~~~~~~~~~~~~~
% Propagates the orbit over the specified time interval, transforming
% the position vector into the earth-fixed frame and, from that,
% computing the right ascension and declination histories.
% ----------------------
%
times = linspace(to,tf,1000);
ra    = [];
dec   = [];
theta = 0;
for i = 1:length(times)
    t             = times(i);
    M             = 2*pi/T*t;
    E             = kepler_E(e, M);
    TA            = 2*atan(tan(E/2)*sqrt((1+e)/(1-e)));
    r             = h^2/mu/(1 + e*cos(TA))*[cos(TA) sin(TA) 0]';

    W             = Wo  + Wdot*t;
    wp            = wpo + wpdot*t;

    R1            = [ cos(W)  sin(W)  0
                     -sin(W)  cos(W)  0
                         0       0    1];

    R2            = [1      0          0
                     0   cos(incl)  sin(incl)
                     0  -sin(incl)  cos(incl)];

    R3            = [ cos(wp)  sin(wp)  0
                     -sin(wp)  cos(wp)  0
                         0        0     1];

    QxX           = (R3*R2*R1)';
    R             = QxX*r;

    theta         = we*(t - to);
    Q             = [ cos(theta)  sin(theta)  0
                     -sin(theta)  cos(theta)  0
                          0          0        1];
    r_rel         = Q*R;

    [alpha delta] = ra_and_dec_from_r(r_rel);
```

```
    ra            = [ra;  alpha];
    dec           = [dec; delta];
end

end %find_ra_and_dec

% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function form_separate_curves
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Breaks the ground track up into separate curves which start
% and terminate at right ascensions in the range [0,360 deg].
% --------------------------
tol = 100;
curve_no = 1;
n_curves = 1;
k        = 0;
ra_prev  = ra(1);
for i = 1:length(ra)
    if abs(ra(i) - ra_prev) > tol
        curve_no = curve_no + 1;
        n_curves = n_curves + 1;
        k = 0;
    end
    k                 = k + 1;
    RA{curve_no}(k)  = ra(i);
    Dec{curve_no}(k) = dec(i);
    ra_prev          = ra(i);
end
end %form_separate_curves

% ~~~~~~~~~~~~~~~~~~~~~~~~~~
function plot_ground_track
% ~~~~~~~~~~~~~~~~~~~~~~~~~~
hold on
xlabel('East longitude (degrees)')
ylabel('Latitude (degrees)')
axis equal
grid on
for i = 1:n_curves
    plot(RA{i}, Dec{i})
end

axis ([0 360 -90 90])
text(  ra(1),    dec(1), 'o Start')
text(ra(end), dec(end), 'o Finish')
line([min(ra) max(ra)],[0 0], 'Color','k') %the equator
end %plot_ground_track
```

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function print_orbital_data
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
coe      = [h e Wo incl wpo TAo];
[ro, vo]  = sv_from_coe(coe, mu);
fprintf('\n ---------------------------------------------------\n')
fprintf('\n Angular momentum     = %g km^2/s' , h)
fprintf('\n Eccentricity         = %g'        , e)
fprintf('\n Semimajor axis       = %g km'     , a)
fprintf('\n Perigee radius       = %g km'     , rP)
fprintf('\n Apogee radius        = %g km'     , rA)
fprintf('\n Period               = %g hours'  , T/3600)
fprintf('\n Inclination          = %g deg'    , incl/deg)
fprintf('\n Initial true anomaly = %g deg'    , TAo/deg)
fprintf('\n Time since perigee   = %g hours'  , to/3600)
fprintf('\n Initial RA           = %g deg'    , Wo/deg)
fprintf('\n RA_dot               = %g deg/period' , Wdot/deg*T)
fprintf('\n Initial wp           = %g deg'    , wpo/deg)
fprintf('\n wp_dot               = %g deg/period' , wpdot/deg*T)
fprintf('\n')
fprintf('\n r0 = [%12g, %12g, %12g] (km)', ro(1), ro(2), ro(3))
fprintf('\n magnitude = %g km\n', norm(ro))
fprintf('\n v0 = [%12g, %12g, %12g] (km)', vo(1), vo(2), vo(3))
fprintf('\n magnitude = %g km\n', norm(vo))
fprintf('\n ---------------------------------------------------\n')

end %print_orbital_data

end %ground_track
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Chapter 5

### D.24 Algorithm 5.1: Gibbs method of preliminary orbit determination

**Function file:** `gibbs.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function [V2, ierr] = gibbs(R1, R2, R3)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function uses the Gibbs method of orbit determination to
  to compute the velocity corresponding to the second of three
```

```
  supplied position vectors.

  mu            - gravitational parameter (km^3/s^2
  R1, R2, R3    - three coplanar geocentric position vectors (km)
  r1, r2, r3    - the magnitudes of R1, R2 and R3 (km)
  c12, c23, c31 - three independent cross products among
                  R1, R2 and R3
  N, D, S       - vectors formed from R1, R2 and R3 during
                  the Gibbs' procedure
  tol           - tolerance for determining if R1, R2 and R3
                  are coplanar
  ierr          - = 0 if R1, R2, R3 are found to be coplanar
                  = 1 otherwise
  V2            - the velocity corresponding to R2 (km/s)

  User M-functions required: none
%}
% ---------------------------------------

global mu
tol  = 1e-4;
ierr = 0;

%...Magnitudes of R1, R2 and R3:
r1 = norm(R1);
r2 = norm(R2);
r3 = norm(R3);

%...Cross products among R1, R2 and R3:
c12 = cross(R1,R2);
c23 = cross(R2,R3);
c31 = cross(R3,R1);

%...Check that R1, R2 and R3 are coplanar; if not set error flag:
if abs(dot(R1,c23)/r1/norm(c23)) > tol
    ierr = 1;
end

%...Equation 5.13:
N = r1*c23 + r2*c31 + r3*c12;

%...Equation 5.14:
D = c12 + c23 + c31;

%...Equation 5.21:
S = R1*(r2 - r3) + R2*(r3 - r1) + R3*(r1 - r2);

%...Equation 5.22:
```

```
V2 = sqrt(mu/norm(N)/norm(D))*(cross(D,R2)/r2 + S);
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  end %gibbs
```

## Script file: `Example_5_01.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_5_01
% ~~~~~~~~~~~~~
%{
  This program uses Algorithm 5.1 (Gibbs method) and Algorithm 4.2
  to obtain the orbital elements from the data provided in Example 5.1.

  deg        - factor for converting between degrees and radians
  pi         - 3.1415926...
  mu         - gravitational parameter (km^3/s^2)
  r1, r2, r3 - three coplanar geocentric position vectors (km)
  ierr       - 0 if r1, r2, r3 are found to be coplanar
               1 otherwise
  v2         - the velocity corresponding to r2 (km/s)
  coe        - orbital elements [h e RA incl w TA a]
               where h    = angular momentum (km^2/s)
                     e    = eccentricity
                     RA   = right ascension of the ascending node (rad)
                     incl = orbit inclination (rad)
                     w    = argument of perigee (rad)
                     TA   = true anomaly (rad)
                     a    = semimajor axis (km)
  T          - period of elliptic orbit (s)

  User M-functions required: gibbs, coe_from_sv
%}
% ----------------------------------------------

clear all; clc
deg = pi/180;
global mu

%...Data declaration for Example 5.1:
mu = 398600;
r1 = [-294.32 4265.1 5986.7];
r2 = [-1365.5 3637.6 6346.8];
r3 = [-2940.3 2473.7 6555.8];
%...

%...Echo the input data to the command window:
fprintf('-----------------------------------------------------')
```

```
fprintf('\n Example 5.1: Gibbs Method\n')
fprintf('\n\n Input data:\n')
fprintf('\n  Gravitational parameter (km^3/s^2)  = %g\n', mu)
fprintf('\n  r1 (km) = [%g  %g  %g]', r1(1), r1(2), r1(3))
fprintf('\n  r2 (km) = [%g  %g  %g]', r2(1), r2(2), r2(3))
fprintf('\n  r3 (km) = [%g  %g  %g]', r3(1), r3(2), r3(3))
fprintf('\n\n');

%...Algorithm 5.1:
[v2, ierr] = gibbs(r1, r2, r3);

%...If the vectors r1, r2, r3, are not coplanar, abort:
if ierr == 1
    fprintf('\n  These vectors are not coplanar.\n\n')
    return
end

%...Algorithm 4.2:
coe  = coe_from_sv(r2,v2,mu);

h    = coe(1);
e    = coe(2);
RA   = coe(3);
incl = coe(4);
w    = coe(5);
TA   = coe(6);
a    = coe(7);

%...Output the results to the command window:
fprintf(' Solution:')
fprintf('\n');
fprintf('\n  v2 (km/s) = [%g  %g  %g]', v2(1), v2(2), v2(3))
fprintf('\n\n  Orbital elements:');
fprintf('\n    Angular momentum (km^2/s)  = %g', h)
fprintf('\n    Eccentricity               = %g', e)
fprintf('\n    Inclination (deg)          = %g', incl/deg)
fprintf('\n    RA of ascending node (deg) = %g', RA/deg)
fprintf('\n    Argument of perigee (deg)  = %g', w/deg)
fprintf('\n    True anomaly (deg)         = %g', TA/deg)
fprintf('\n    Semimajor axis (km)        = %g', a)
%...If the orbit is an ellipse, output the period:
if e < 1
    T = 2*pi/sqrt(mu)*coe(7)^1.5;
    fprintf('\n    Period (s)                 = %g', T)
end
fprintf('\n-----------------------------------------------------\n')
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

***Output from*** `Example_5_01`

```
-----------------------------------------------------
 Example 5.1: Gibbs Method


 Input data:

  Gravitational parameter (km^3/s^2)  = 398600

  r1 (km) = [-294.32  4265.1  5986.7]
  r2 (km) = [-1365.4  3637.6  6346.8]
  r3 (km) = [-2940.3  2473.7  6555.8]

 Solution:

  v2 (km/s) = [-6.2176  -4.01237  1.59915]

  Orbital elements:
    Angular momentum (km^2/s)  = 56193
    Eccentricity               = 0.100159
    Inclination (deg)          = 60.001
    RA of ascending node (deg) = 40.0023
    Argument of perigee (deg)  = 30.1093
    True anomaly (deg)         = 49.8894
    Semimajor axis (km)        = 8002.14
    Period (s)                 = 7123.94
-----------------------------------------------------
```

## D.25 Algorithm 5.2: Solution of Lambert's problem

### Function file: `lambert.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function [V1, V2] = lambert(R1, R2, t, string)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function solves Lambert's problem.

  mu          - gravitational parameter (km^3/s^2)
  R1, R2      - initial and final position vectors (km)
  r1, r2      - magnitudes of R1 and R2
  t           - the time of flight from R1 to R2 (a constant) (s)
  V1, V2      - initial and final velocity vectors (km/s)
  c12         - cross product of R1 into R2
  theta       - angle between R1 and R2
  string      - 'pro'   if the orbit is prograde
```

```
                'retro' if the orbit is retrograde
  A           - a constant given by Equation 5.35
  z           - alpha*x^2, where alpha is the reciprocal of the
                semimajor axis and x is the universal anomaly
  y(z)        - a function of z given by Equation 5.38
  F(z,t)      - a function of the variable z and constant t,
              - given by Equation 5.40
  dFdz(z)     - the derivative of F(z,t), given by Equation 5.43
  ratio       - F/dFdz
  tol         - tolerance on precision of convergence
  nmax        - maximum number of iterations of Newton's procedure
  f, g        - Lagrange coefficients
  gdot        - time derivative of g
  C(z), S(z) - Stumpff functions
  dum         - a dummy variable

  User M-functions required: stumpC and stumpS
%}
% ----------------------------------------------

global mu

%...Magnitudes of R1 and R2:
r1 = norm(R1);
r2 = norm(R2);

c12   = cross(R1, R2);
theta = acos(dot(R1,R2)/r1/r2);

%...Determine whether the orbit is prograde or retrograde:
if nargin < 4 || (~strcmp(string,'retro') & (~strcmp(string,'pro')))
    string = 'pro';
    fprintf('\n ** Prograde trajectory assumed.\n')
end

if strcmp(string,'pro')
    if c12(3) <= 0
        theta = 2*pi - theta;
    end
elseif strcmp(string,'retro')
    if c12(3) >= 0
        theta = 2*pi - theta;
    end
end

%...Equation 5.35:
A = sin(theta)*sqrt(r1*r2/(1 - cos(theta)));
```

```
%...Determine approximately where F(z,t) changes sign, and
%...use that value of z as the starting value for Equation 5.45:
z = -100;
while F(z,t) < 0
    z = z + 0.1;
end

%...Set an error tolerance and a limit on the number of iterations:
tol   = 1.e-8;
nmax  = 5000;

%...Iterate on Equation 5.45 until z is determined to within the
%...error tolerance:
ratio = 1;
n     = 0;
while (abs(ratio) > tol) & (n <= nmax)
    n     = n + 1;
    ratio = F(z,t)/dFdz(z);
    z     = z - ratio;
end

%...Report if the maximum number of iterations is exceeded:
if n >= nmax
    fprintf('\n\n **Number of iterations exceeds %g \n\n ',nmax)
end

%...Equation 5.46a:
f    = 1 - y(z)/r1;

%...Equation 5.46b:
g    = A*sqrt(y(z)/mu);

%...Equation 5.46d:
gdot = 1 - y(z)/r2;

%...Equation 5.28:
V1   = 1/g*(R2 - f*R1);

%...Equation 5.29:
V2   = 1/g*(gdot*R2 - R1);

return

% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Subfunctions used in the main body:
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

%...Equation 5.38:
```

```
function dum = y(z)
    dum = r1 + r2 + A*(z*S(z) - 1)/sqrt(C(z));
end

%...Equation 5.40:
function dum = F(z,t)
    dum = (y(z)/C(z))^1.5*S(z) + A*sqrt(y(z)) - sqrt(mu)*t;
end

%...Equation 5.43:
function dum = dFdz(z)
    if z == 0
        dum = sqrt(2)/40*y(0)^1.5 + A/8*(sqrt(y(0)) + A*sqrt(1/2/y(0)));
    else
        dum = (y(z)/C(z))^1.5*(1/2/z*(C(z) - 3*S(z)/2/C(z)) ...
               + 3*S(z)^2/4/C(z)) + A/8*(3*S(z)/C(z)*sqrt(y(z)) ...
               + A*sqrt(C(z)/y(z)));
    end
end

%...Stumpff functions:
function dum = C(z)
    dum = stumpC(z);
end

function dum = S(z)
    dum = stumpS(z);
end

end %lambert

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

## **Script file:** Example_5_02.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_5_02
% ~~~~~~~~~~~~
%{
  This program uses Algorithm 5.2 to solve Lambert's problem for the
  data provided in Example 5.2.

  deg    - factor for converting between degrees and radians
  pi     - 3.1415926...
  mu     - gravitational parameter (km^3/s^2)
  r1, r2 - initial and final position vectors (km)
  dt     - time between r1 and r2 (s)
```

```
  string - = 'pro' if the orbit is prograde
           = 'retro if the orbit is retrograde
  v1, v2 - initial and final velocity vectors (km/s)
  coe    - orbital elements [h e RA incl w TA a]
           where h    = angular momentum (km^2/s)
                 e    = eccentricity
                 RA   = right ascension of the ascending node (rad)
                 incl = orbit inclination (rad)
                 w    = argument of perigee (rad)
                 TA   = true anomaly (rad)
                 a    = semimajor axis (km)
  TA1    - Initial true anomaly (rad)
  TA2    - Final true anomaly (rad)
  T      - period of an elliptic orbit (s)

  User M-functions required: lambert, coe_from_sv
%}
% --------------------------------------------

clear all; clc
global mu
deg = pi/180;

%...Data declaration for Example 5.2:
mu     = 398600;
r1     = [  5000  10000  2100];
r2     = [-14600   2500  7000];
dt     = 3600;
string = 'pro';
%...

%...Algorithm 5.2:
[v1, v2] = lambert(r1, r2, dt, string);

%...Algorithm 4.1 (using r1 and v1):
coe      = coe_from_sv(r1, v1, mu);
%...Save the initial true anomaly:
TA1      = coe(6);

%...Algorithm 4.1 (using r2 and v2):
coe      = coe_from_sv(r2, v2, mu);
%...Save the final true anomaly:
TA2      = coe(6);

%...Echo the input data and output the results to the command window:
fprintf('-----------------------------------------------------')
fprintf('\n Example 5.2: Lambert"s Problem\n')
fprintf('\n\n Input data:\n');
```

```
fprintf('\n   Gravitational parameter (km^3/s^2) = %g\n', mu);
fprintf('\n   r1 (km)                            = [%g  %g  %g]', ...
                                              r1(1), r1(2), r1(3))
fprintf('\n   r2 (km)                            = [%g  %g  %g]', ...
                                              r2(1), r2(2), r2(3))
fprintf('\n   Elapsed time (s)                   = %g', dt);
fprintf('\n\n Solution:\n')

fprintf('\n   v1 (km/s)                          = [%g  %g  %g]', ...
                                              v1(1), v1(2), v1(3))
fprintf('\n   v2 (km/s)                          = [%g  %g  %g]', ...
                                              v2(1), v2(2), v2(3))

fprintf('\n\n Orbital elements:')
fprintf('\n   Angular momentum (km^2/s)     = %g', coe(1))
fprintf('\n   Eccentricity                  = %g', coe(2))
fprintf('\n   Inclination (deg)             = %g', coe(4)/deg)
fprintf('\n   RA of ascending node (deg)    = %g', coe(3)/deg)
fprintf('\n   Argument of perigee (deg)     = %g', coe(5)/deg)
fprintf('\n   True anomaly initial (deg)    = %g', TA1/deg)
fprintf('\n   True anomaly final   (deg)    = %g', TA2/deg)
fprintf('\n   Semimajor axis (km)           = %g', coe(7))
fprintf('\n   Periapse radius (km)          = %g', coe(1)^2/mu/(1 + coe(2)))
%...If the orbit is an ellipse, output its period:
if coe(2)<1
    T = 2*pi/sqrt(mu)*coe(7)^1.5;
    fprintf('\n   Period:')
    fprintf('\n     Seconds                     = %g', T)
    fprintf('\n     Minutes                     = %g', T/60)
    fprintf('\n     Hours                       = %g', T/3600)
    fprintf('\n     Days                        = %g', T/24/3600)
end
fprintf('\n-----------------------------------------------------\n')
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

### ***Output from*** `Example_5_02`

```
-----------------------------------------------------
 Example 5.2: Lambert's Problem


 Input data:

   Gravitational parameter (km^3/s^2) = 398600

   r1 (km)                       = [5000  10000  2100]
   r2 (km)                       = [-14600  2500  7000]
   Elapsed time (s)              = 3600
```

```
 Solution:

   v1 (km/s)                    = [-5.99249  1.92536  3.24564]
   v2 (km/s)                    = [-3.31246  -4.19662  -0.385288]

 Orbital elements:
   Angular momentum (km^2/s)    = 80466.8
   Eccentricity                 = 0.433488
   Inclination (deg)            = 30.191
   RA of ascending node (deg)   = 44.6002
   Argument of perigee (deg)    = 30.7062
   True anomaly initial (deg)   = 350.83
   True anomaly final   (deg)   = 91.1223
   Semimajor axis (km)          = 20002.9
   Periapse radius (km)         = 11331.9
   Period:
     Seconds                    = 28154.7
     Minutes                    = 469.245
     Hours                      = 7.82075
     Days                       = 0.325865
-----------------------------------------------------
```

## D.26 Calculation of Julian day number at 0 hr UT

The following script implements Equation 5.48 for use in other programs.

### Function file: J0.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function j0 = J0(year, month, day)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function computes the Julian day number at 0 UT for any year
  between 1900 and 2100 using Equation 5.48.

  j0    - Julian day at 0 hr UT (Universal Time)
  year  - range: 1901 - 2099
  month - range: 1 - 12
  day   - range: 1 - 31

  User m-functions required: none
%}
% ----------------------------------

j0 = 367*year - fix(7*(year + fix((month + 9)/12))/4) ...
```

```
    + fix(275*month/9) + day + 1721013.5;

% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
end %J0
```

## Script file: Example_5_04.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_5_04
% ~~~~~~~~~~~~
%{
  This program computes J0 and the Julian day number using the data
  in Example 5.4.

  year   - range: 1901 - 2099
  month  - range: 1 - 12
  day    - range: 1 - 31
  hour   - range: 0 - 23 (Universal Time)
  minute - rage: 0 - 60
  second - range: 0 - 60
  ut     - universal time (hr)
  j0     - Julian day number at 0 hr UT
  jd     - Julian day number at specified UT

  User M-function required: J0
%}
% ---------------------------------------------

clear all; clc

%...Data declaration for Example 5.4:
year   = 2004;
month  = 5;
day    = 12;

hour   = 14;
minute = 45;
second = 30;
%...

ut = hour + minute/60 + second/3600;

%...Equation 5.46:
j0 = J0(year, month, day);

%...Equation 5.47:
jd = j0 + ut/24;
```

```
%...Echo the input data and output the results to the command window:
fprintf('----------------------------------------------------')
fprintf('\n Example 5.4: Julian day calculation\n')
fprintf('\n Input data:\n');
fprintf('\n   Year            = %g',   year)
fprintf('\n   Month           = %g',   month)
fprintf('\n   Day             = %g',   day)
fprintf('\n   Hour            = %g',   hour)
fprintf('\n   Minute          = %g',   minute)
fprintf('\n   Second          = %g\n', second)

fprintf('\n Julian day number = %11.3f', jd);
fprintf('\n----------------------------------------------------\n')
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

***Output from*** `Example_5_04`

```
-----------------------------------------------------
 Example 5.4: Julian day calculation

 Input data:

   Year           = 2004
   Month          = 5
   Day            = 12
   Hour           = 14
   Minute         = 45
   Second         = 30

 Julian day number = 2453138.115
-----------------------------------------------------
```

## D.27 Algorithm 5.3: Calculation of local sidereal time

### Function file: `LST.m`

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function lst = LST(y, m, d, ut, EL)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function calculates the local sidereal time.

  lst - local sidereal time (degrees)
  y   - year
  m   - month
  d   - day
  ut  - Universal Time (hours)
  EL  - east longitude (degrees)
```

```
  j0  - Julian day number at 0 hr UT
  j   - number of centuries since J2000
  g0  - Greenwich sidereal time (degrees) at 0 hr UT
  gst - Greenwich sidereal time (degrees) at the specified UT

  User M-function required:  J0
  User subfunction required: zeroTo360
%}
% ---------------------------------------------

%...Equation 5.48;
j0 = J0(y, m, d);

%...Equation 5.49:
j = (j0 - 2451545)/36525;

%...Equation 5.50:
g0 = 100.4606184 + 36000.77004*j + 0.000387933*j^2 - 2.583e-8*j^3;

%...Reduce g0 so it lies in the range 0 - 360 degrees
g0 = zeroTo360(g0);

%...Equation 5.51:
gst = g0 + 360.98564724*ut/24;

%...Equation 5.52:
lst = gst + EL;

%...Reduce lst to the range 0 - 360 degrees:
lst = lst - 360*fix(lst/360);

return

% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function y = zeroTo360(x)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This subfunction reduces an angle to the range 0 - 360 degrees.

  x - The angle (degrees) to be reduced
  y - The reduced value
%}
% ------------------------
if (x >= 360)
    x = x - fix(x/360)*360;
elseif (x < 0)
    x = x - (fix(x/360) - 1)*360;
end
```

```
y = x;
end %zeroTo360

end %LST
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Script file: Example_5_06.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_5_06
% ~~~~~~~~~~~~
%{
  This program uses Algorithm 5.3 to obtain the local sidereal
  time from the data provided in Example 5.6.

  lst   - local sidereal time (degrees)
  EL    - east longitude of the site (west longitude is negative):
             degrees (0 - 360)
             minutes (0 - 60)
             seconds (0 - 60)
  WL    - west longitude
  year  - range: 1901 - 2099
  month - range: 1 - 12
  day   - range: 1 - 31
  ut    - universal time
             hour (0 - 23)
             minute (0 - 60)
             second (0 - 60)

  User m-function required: LST
%}
% ----------------------------------------------

clear all; clc

%...Data declaration for Example 5.6:
%   East longitude:
degrees = 139;
minutes = 47;
seconds = 0;

%   Date:
year    = 2004;
month   = 3;
day     = 3;
```

```
%   Universal time:
hour    = 4;
minute  = 30;
second  = 0;
%...

%...Convert negative (west) longitude to east longitude:
if degrees < 0
    degrees = degrees + 360;
end

%...Express the longitudes as decimal numbers:
EL = degrees + minutes/60 + seconds/3600;
WL = 360 - EL;

%...Express universal time as a decimal number:
ut = hour + minute/60 + second/3600;

%...Algorithm 5.3:
lst = LST(year, month, day, ut, EL);

%...Echo the input data and output the results to the command window:
fprintf('-----------------------------------------------------')
fprintf('\n Example 5.6: Local sidereal time calculation\n')
fprintf('\n Input data:\n');
fprintf('\n   Year                        = %g', year)
fprintf('\n   Month                       = %g', month)
fprintf('\n   Day                         = %g', day)
fprintf('\n   UT (hr)                     = %g', ut)
fprintf('\n   West Longitude (deg)        = %g', WL)
fprintf('\n   East Longitude (deg)        = %g', EL)
fprintf('\n\n');

fprintf(' Solution:')

fprintf('\n');
fprintf('\n  Local Sidereal Time (deg) = %g', lst)
fprintf('\n  Local Sidereal Time (hr)  = %g', lst/15)

fprintf('\n-----------------------------------------------------\n')
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

***Output from*** `Example_5_06`

```
-----------------------------------------------------
 Example 5.6: Local sidereal time calculation

 Input data:
```

```
  Year                     = 2004
  Month                    = 3
  Day                      = 3
  UT (hr)                  = 4.5
  West Longitude (deg)     = 220.217
  East Longitude (deg)     = 139.783

 Solution:

  Local Sidereal Time (deg) = 8.57688
  Local Sidereal Time (hr)  = 0.571792
-----------------------------------------------------
```

## D.28 Algorithm 5.4: Calculation of the state vector from measurements of range, angular position, and their rates

### Function file: rv_from_observe.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function [r,v] = rv_from_observe(rho, rhodot, A, Adot, a, ...
                                   adot, theta, phi, H)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function calculates the geocentric equatorial position and
  velocity vectors of an object from radar observations of range,
  azimuth, elevation angle and their rates.

  deg    - conversion factor between degrees and radians
  pi     - 3.1415926...

  Re     - equatorial radius of the earth (km)
  f      - earth's flattening factor
  wE     - angular velocity of the earth (rad/s)
  omega  - earth's angular velocity vector (rad/s) in the
           geocentric equatorial frame

  theta  - local sidereal time (degrees) of tracking site
  phi    - geodetic latitude (degrees) of site
  H      - elevation of site (km)
  R      - geocentric equatorial position vector (km) of tracking site
  Rdot   - inertial velocity (km/s) of site

  rho    - slant range of object (km)
  rhodot - range rate (km/s)
  A      - azimuth (degrees) of object relative to observation site
  Adot   - time rate of change of azimuth (degrees/s)
```

```
  a      - elevation angle (degrees) of object relative to observation site
  adot   - time rate of change of elevation angle (degrees/s)
  dec    - topocentric equatorial declination of object (rad)
  decdot - declination rate (rad/s)
  h      - hour angle of object (rad)
  RA     - topocentric equatorial right ascension of object (rad)
  RAdot  - right ascension rate (rad/s)

  Rho    - unit vector from site to object
  Rhodot - time rate of change of Rho (1/s)
  r      - geocentric equatorial position vector of object (km)
  v      - geocentric equatorial velocity vector of object (km)

  User M-functions required: none
%}
% ----------------------------------------------------------------------

global f Re wE
deg   = pi/180;
omega = [0 0 wE];

%...Convert angular quantities from degrees to radians:
A     = A    *deg;
Adot  = Adot *deg;
a     = a    *deg;
adot  = adot *deg;
theta = theta*deg;
phi   = phi  *deg;

%...Equation 5.56:
R     = [(Re/sqrt(1-(2*f - f*f)*sin(phi)^2) + H)*cos(phi)*cos(theta), ...
         (Re/sqrt(1-(2*f - f*f)*sin(phi)^2) + H)*cos(phi)*sin(theta), ...
         (Re*(1 - f)^2/sqrt(1-(2*f - f*f)*sin(phi)^2) + H)*sin(phi)];

%...Equation 5.66:
Rdot  = cross(omega, R);

%...Equation 5.83a:
dec   = asin(cos(phi)*cos(A)*cos(a) + sin(phi)*sin(a));

%...Equation 5.83b:
h = acos((cos(phi)*sin(a) - sin(phi)*cos(A)*cos(a))/cos(dec));
if (A > 0) & (A < pi)
    h = 2*pi - h;
end

%...Equation 5.83c:
```

```
RA = theta - h;

%...Equations 5.57:
Rho = [cos(RA)*cos(dec)  sin(RA)*cos(dec)  sin(dec)];

%...Equation 5.63:
r   = R + rho*Rho;

%...Equation 5.84:
decdot = (-Adot*cos(phi)*sin(A)*cos(a) + adot*(sin(phi)*cos(a) ...
          - cos(phi)*cos(A)*sin(a)))/cos(dec);

%...Equation 5.85:
RAdot  = wE ...
         + (Adot*cos(A)*cos(a) - adot*sin(A)*sin(a) ...
         + decdot*sin(A)*cos(a)*tan(dec)) ...
          /(cos(phi)*sin(a) - sin(phi)*cos(A)*cos(a));

%...Equations 5.69 and 5.72:
Rhodot = [-RAdot*sin(RA)*cos(dec) - decdot*cos(RA)*sin(dec),...
           RAdot*cos(RA)*cos(dec) - decdot*sin(RA)*sin(dec),...
           decdot*cos(dec)];

%...Equation 5.64:
v = Rdot + rhodot*Rho + rho*Rhodot;

end %rv_from_observe
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Script file: Example_5_10.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_5_10
% ~~~~~~~~~~~~~
%
% This program uses Algorithms 5.4 and 4.2 to obtain the orbital
% elements from the observational data provided in Example 5.10.
%
% deg    - conversion factor between degrees and radians
% pi     - 3.1415926...
% mu     - gravitational parameter (km^3/s^2)

% Re     - equatorial radius of the earth (km)
% f      - earth's flattening factor
% wE     - angular velocity of the earth (rad/s)
% omega  - earth's angular velocity vector (rad/s) in the
%          geocentric equatorial frame
```

```
% rho    - slant range of object (km)
% rhodot - range rate (km/s)
% A      - azimuth (deg) of object relative to observation site
% Adot   - time rate of change of azimuth (deg/s)
% a      - elevation angle (deg) of object relative to observation site
% adot   - time rate of change of elevation angle (degrees/s)

% theta  - local sidereal time (deg) of tracking site
% phi    - geodetic latitude (deg) of site
% H      - elevation of site (km)

% r      - geocentric equatorial position vector of object (km)
% v      - geocentric equatorial velocity vector of object (km)

% coe    - orbital elements [h e RA incl w TA a]
%          where
%              h    = angular momentum (km^2/s)
%              e    = eccentricity
%              RA   = right ascension of the ascending node (rad)
%              incl = inclination of the orbit (rad)
%              w    = argument of perigee (rad)
%              TA   = true anomaly (rad)
%              a    = semimajor axis (km)
% rp    - perigee radius (km)
% T     - period of elliptical orbit (s)
%
% User M-functions required: rv_from_observe, coe_from_sv
% ------------------------------------------------------------------

clear all; clc
global  f Re wE

deg    = pi/180;
f      = 1/298.256421867;
Re     = 6378.13655;
wE     = 7.292115e-5;
mu     = 398600.4418;

%...Data declaration for Example 5.10:
rho    = 2551;
rhodot = 0;
A      = 90;
Adot   = 0.1130;
a      = 30;
adot   = 0.05651;
theta  = 300;
phi    = 60;
H      = 0;
```

```
%...

%...Algorithm 5.4:
[r,v] = rv_from_observe(rho, rhodot, A, Adot, a, adot, theta, phi, H);

%...Algorithm 4.2:
coe  = coe_from_sv(r,v,mu);

h    = coe(1);
e    = coe(2);
RA   = coe(3);
incl = coe(4);
w    = coe(5);
TA   = coe(6);
a    = coe(7);

%...Equation 2.40
rp   = h^2/mu/(1 + e);

%...Echo the input data and output the solution to
%   the command window:
fprintf('-----------------------------------------------------')
fprintf('\n Example 5.10')
fprintf('\n\n Input data:\n');
fprintf('\n Slant range (km)              = %g', rho);
fprintf('\n Slant range rate (km/s)       = %g', rhodot);
fprintf('\n Azimuth (deg)                 = %g', A);
fprintf('\n Azimuth rate (deg/s)          = %g', Adot);
fprintf('\n Elevation (deg)               = %g', a);
fprintf('\n Elevation rate (deg/s)        = %g', adot);
fprintf('\n Local sidereal time (deg)     = %g', theta);
fprintf('\n Latitude (deg)                = %g', phi);
fprintf('\n Altitude above sea level (km) = %g', H);
fprintf('\n\n');

fprintf(' Solution:')

fprintf('\n\n State vector:\n');
fprintf('\n r (km)                          = [%g, %g, %g]', ...
                                       r(1), r(2), r(3));
fprintf('\n v (km/s)                        = [%g, %g, %g]', ...
                                       v(1), v(2), v(3));

fprintf('\n\n Orbital elements:\n')
fprintf('\n   Angular momentum (km^2/s)   = %g', h)
fprintf('\n   Eccentricity                = %g', e)
fprintf('\n   Inclination (deg)           = %g', incl/deg)
```

```
fprintf('\n   RA of ascending node (deg)  = %g', RA/deg)
fprintf('\n   Argument of perigee (deg)   = %g', w/deg)
fprintf('\n   True anomaly (deg)          = %g\n', TA/deg)
fprintf('\n   Semimajor axis (km)         = %g', a)
fprintf('\n   Perigee radius (km)         = %g', rp)
%...If the orbit is an ellipse, output its period:
if e < 1
    T = 2*pi/sqrt(mu)*a^1.5;
    fprintf('\n   Period:')
    fprintf('\n     Seconds                   = %g', T)
    fprintf('\n     Minutes                   = %g', T/60)
    fprintf('\n     Hours                     = %g', T/3600)
    fprintf('\n     Days                      = %g', T/24/3600)
end
fprintf('\n-----------------------------------------------------\n')
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## ***Output from*** Example_5_10

```
-----------------------------------------------------
 Example 5.10

 Input data:

 Slant range (km)              = 2551
 Slant range rate (km/s)       = 0
 Azimuth (deg)                 = 90
 Azimuth rate (deg/s)          = 0.113
 Elevation (deg)               = 5168.62
 Elevation rate (deg/s)        = 0.05651
 Local sidereal time (deg)     = 300
 Latitude (deg)                = 60
 Altitude above sea level (km) = 0

 Solution:

 State vector:

 r (km)                        = [3830.68, -2216.47, 6605.09]
 v (km/s)                      = [1.50357, -4.56099, -0.291536]

 Orbital elements:

   Angular momentum (km^2/s)   = 35621.4
   Eccentricity                = 0.619758
   Inclination (deg)           = 113.386
   RA of ascending node (deg)  = 109.75
   Argument of perigee (deg)   = 309.81
```

```
   True anomaly (deg)          = 165.352

   Semimajor axis (km)         = 5168.62
   Perigee radius (km)         = 1965.32
   Period:
     Seconds                   = 3698.05
     Minutes                   = 61.6342
     Hours                     = 1.02724
     Days                      = 0.0428015
-----------------------------------------------------
```

## D.29 Algorithms 5.5 and 5.6: Gauss method of preliminary orbit determination with iterative improvement

### Function file: gauss.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 function [r, v, r_old, v_old] = ...
          gauss(Rho1, Rho2, Rho3, R1, R2, R3, t1, t2, t3)
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function uses the Gauss method with iterative improvement
  (Algorithms 5.5 and 5.6) to calculate the state vector of an
  orbiting body from angles-only observations at three
  closely-spaced times.

  mu                - the gravitational parameter (km^3/s^2)
  t1, t2, t3        - the times of the observations (s)
  tau, tau1, tau3   - time intervals between observations (s)
  R1, R2, R3        - the observation site position vectors
                      at t1, t2, t3 (km)
  Rho1, Rho2, Rho3  - the direction cosine vectors of the
                      satellite at t1, t2, t3
  p1, p2, p3        - cross products among the three direction
                      cosine vectors
  Do                - scalar triple product of Rho1, Rho2 and Rho3
  D                 - Matrix of the nine scalar triple products
                      of R1, R2 and R3 with p1, p2 and p3
  E                 - dot product of R2 and Rho2
  A, B              - constants in the expression relating slant range
                      to geocentric radius
  a,b,c             - coefficients of the 8th order polynomial
                      in the estimated geocentric radius x
  x                 - positive root of the 8th order polynomial
  rho1, rho2, rho3  - the slant ranges at t1, t2, t3
  r1, r2, r3        - the position vectors at t1, t2, t3 (km)
```

```
  r_old, v_old     - the estimated state vector at the end of
                     Algorithm 5.5 (km, km/s)
  rho1_old,
  rho2_old, and
  rho3_old         - the values of the slant ranges at t1, t2, t3
                     at the beginning of iterative improvement
                     (Algorithm 5.6) (km)
  diff1, diff2,
  and diff3        - the magnitudes of the differences between the
                     old and new slant ranges at the end of
                     each iteration
  tol              - the error tolerance determining
                    convergence
  n                - number of passes through the
                    iterative improvement loop
  nmax             - limit on the number of iterations
  ro, vo           - magnitude of the position and
                     velocity vectors (km, km/s)
  vro              - radial velocity component (km)
  a                - reciprocal of the semimajor axis (1/km)
  v2               - computed velocity at time t2 (km/s)
  r, v             - the state vector at the end of Algorithm 5.6
                     (km, km/s)

  User m-functions required:  kepler_U, f_and_g
  User subfunctions required: posroot
%}
% -------------------------------------------------------

global mu

%...Equations 5.98:
tau1 = t1 - t2;
tau3 = t3 - t2;

%...Equation 5.101:
tau  = tau3 - tau1;

%...Independent cross products among the direction cosine vectors:
p1 = cross(Rho2,Rho3);
p2 = cross(Rho1,Rho3);
p3 = cross(Rho1,Rho2);

%...Equation 5.108:
Do = dot(Rho1,p1);

%...Equations 5.109b, 5.110b and 5.111b:
D  = [[dot(R1,p1) dot(R1,p2) dot(R1,p3)]
```

```
      [dot(R2,p1) dot(R2,p2) dot(R2,p3)]
      [dot(R3,p1) dot(R3,p2) dot(R3,p3)]];

%...Equation 5.115b:
E = dot(R2,Rho2);

%...Equations 5.112b and 5.112c:
A = 1/Do*(-D(1,2)*tau3/tau + D(2,2) + D(3,2)*tau1/tau);
B = 1/6/Do*(D(1,2)*(tau3^2 - tau^2)*tau3/tau ...
            + D(3,2)*(tau^2 - tau1^2)*tau1/tau);

%...Equations 5.117:
a = -(A^2 + 2*A*E + norm(R2)^2);
b = -2*mu*B*(A + E);
c = -(mu*B)^2;

%...Calculate the roots of Equation 5.116 using MATLAB's
%   polynomial 'roots' solver:
Roots = roots([1 0 a 0 0 b 0 0 c]);

%...Find the positive real root:
x = posroot(Roots);

%...Equations 5.99a and 5.99b:
f1 =    1 - 1/2*mu*tau1^2/x^3;
f3 =    1 - 1/2*mu*tau3^2/x^3;

%...Equations 5.100a and 5.100b:
g1 = tau1 - 1/6*mu*(tau1/x)^3;
g3 = tau3 - 1/6*mu*(tau3/x)^3;

%...Equation 5.112a:
rho2 = A + mu*B/x^3;

%...Equation 5.113:
rho1 = 1/Do*((6*(D(3,1)*tau1/tau3 + D(2,1)*tau/tau3)*x^3 ...
               + mu*D(3,1)*(tau^2 - tau1^2)*tau1/tau3) ...
               /(6*x^3 + mu*(tau^2 - tau3^2)) - D(1,1));

%...Equation 5.114:
rho3 = 1/Do*((6*(D(1,3)*tau3/tau1 - D(2,3)*tau/tau1)*x^3 ...
               + mu*D(1,3)*(tau^2 - tau3^2)*tau3/tau1) ...
               /(6*x^3 + mu*(tau^2 - tau1^2)) - D(3,3));

%...Equations 5.86:
r1 = R1 + rho1*Rho1;
r2 = R2 + rho2*Rho2;
r3 = R3 + rho3*Rho3;
```

```
%...Equation 5.118:
v2 = (-f3*r1 + f1*r3)/(f1*g3 - f3*g1);

%...Save the initial estimates of r2 and v2:
r_old = r2;
v_old = v2;

%...End of Algorithm 5.5


%...Use Algorithm 5.6 to improve the accuracy of the initial estimates.

%...Initialize the iterative improvement loop and set error tolerance:
rho1_old = rho1;  rho2_old = rho2;  rho3_old = rho3;
diff1    = 1;     diff2    = 1;     diff3    = 1;
n    = 0;
nmax = 1000;
tol  = 1.e-8;

%...Iterative improvement loop:
while ((diff1 > tol) & (diff2 > tol) & (diff3 > tol)) & (n < nmax)
    n = n+1;

%...Compute quantities required by universal Kepler's equation:
    ro  = norm(r2);
    vo  = norm(v2);
    vro = dot(v2,r2)/ro;
    a   = 2/ro - vo^2/mu;

%...Solve universal Kepler's equation at times tau1 and tau3 for
%   universal anomalies x1 and x3:
    x1 = kepler_U(tau1, ro, vro, a);
    x3 = kepler_U(tau3, ro, vro, a);

%...Calculate the Lagrange f and g coefficients at times tau1
%   and tau3:
    [ff1, gg1] = f_and_g(x1, tau1, ro, a);
    [ff3, gg3] = f_and_g(x3, tau3, ro, a);

%...Update the f and g functions at times tau1 and tau3 by
%   averaging old and new:
    f1    = (f1 + ff1)/2;
    f3    = (f3 + ff3)/2;
    g1    = (g1 + gg1)/2;
    g3    = (g3 + gg3)/2;
```

```
%...Equations 5.96 and 5.97:
    c1    =  g3/(f1*g3 - f3*g1);
    c3    = -g1/(f1*g3 - f3*g1);

%...Equations 5.109a, 5.110a and 5.111a:
    rho1  = 1/Do*(       -D(1,1) + 1/c1*D(2,1) - c3/c1*D(3,1));
    rho2  = 1/Do*(   -c1*D(1,2) +      D(2,2) -    c3*D(3,2));
    rho3  = 1/Do*(-c1/c3*D(1,3) + 1/c3*D(2,3) -       D(3,3));

%...Equations 5.86:
    r1    = R1 + rho1*Rho1;
    r2    = R2 + rho2*Rho2;
    r3    = R3 + rho3*Rho3;

%...Equation 5.118:
    v2    = (-f3*r1 + f1*r3)/(f1*g3 - f3*g1);

%...Calculate differences upon which to base convergence:
    diff1 = abs(rho1 - rho1_old);
    diff2 = abs(rho2 - rho2_old);
    diff3 = abs(rho3 - rho3_old);

%...Update the slant ranges:
    rho1_old = rho1;  rho2_old = rho2;  rho3_old = rho3;
end
%...End iterative improvement loop

fprintf('\n( **Number of Gauss improvement iterations = %g)\n\n',n)

if n >= nmax
    fprintf('\n\n **Number of iterations exceeds %g \n\n ',nmax);
end

%...Return the state vector for the central observation:
r = r2;
v = v2;

return

% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  function x = posroot(Roots)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This subfunction extracts the positive real roots from
  those obtained in the call to MATLAB's 'roots' function.
  If there is more than one positive root, the user is
  prompted to select the one to use.
```

```
  x         - the determined or selected positive root
  Roots     - the vector of roots of a polynomial
  posroots  - vector of positive roots

  User M-functions required: none
%}
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~

%...Construct the vector of positive real roots:
posroots  = Roots(find(Roots>0 & ~imag(Roots)));
npositive = length(posroots);

%...Exit if no positive roots exist:
if npositive == 0
    fprintf('\n\n ** There are no positive roots.  \n\n')
    return
end

%...If there is more than one positive root, output the
%   roots to the command window and prompt the user to
%   select which one to use:
if npositive == 1
    x = posroots;
else
    fprintf('\n\n ** There are two or more positive roots.\n')
    for i = 1:npositive
        fprintf('\n root #%g = %g',i,posroots(i))
    end
    fprintf('\n\n Make a choice:\n')
    nchoice = 0;
    while nchoice < 1 | nchoice > npositive
        nchoice = input(' Use root #? ');
    end
    x = posroots(nchoice);
    fprintf('\n We will use %g .\n', x)
end

end %posroot

end %gauss
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## Script file: Example_5_11.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Example_5_11
% ~~~~~~~~~~~~~
%{
```

```
  This program uses Algorithms 5.5 and 5.6 (Gauss's method) to compute
  the state vector from the data provided in Example 5.11.

  deg            - factor for converting between degrees and radians
  pi             - 3.1415926...
  mu             - gravitational parameter (km^3/s^2)
  Re             - earth's radius (km)
  f              - earth's flattening factor
  H              - elevation of observation site (km)
  phi            - latitude of site (deg)
  t              - vector of observation times t1, t2, t3 (s)
  ra             - vector of topocentric equatorial right ascensions
                   at t1, t2, t3 (deg)
  dec            - vector of topocentric equatorial right declinations
                   at t1, t2, t3 (deg)
  theta          - vector of local sidereal times for t1, t2, t3 (deg)
  R              - matrix of site position vectors at t1, t2, t3 (km)
  rho            - matrix of direction cosine vectors at t1, t2, t3
  fac1, fac2     - common factors
  r_old, v_old   - the state vector without iterative improvement (km, km/s)
  r, v           - the state vector with iterative improvement (km, km/s)
  coe            - vector of orbital elements for r, v:
                   [h, e, RA, incl, w, TA, a]
                   where h    = angular momentum (km^2/s)
                         e    = eccentricity
                         incl = inclination (rad)
                         w    = argument of perigee (rad)
                         TA   = true anomaly (rad)
                         a    = semimajor axis (km)
  coe_old      - vector of orbital elements for r_old, v_old

  User M-functions required: gauss, coe_from_sv
%}
% --------------------------------------------

clear all; clc

global mu

deg = pi/180;
mu  = 398600;
Re  = 6378;
f   = 1/298.26;

%...Data declaration for Example 5.11:
H     = 1;
phi   = 40*deg;
t     = [      0   118.104   237.577];
```

```
ra    = [ 43.5365   54.4196   64.3178]*deg;
dec   = [-8.78334  -12.0739  -15.1054]*deg;
theta = [ 44.5065    45.000   45.4992]*deg;
%...

%...Equations 5.64, 5.76 and 5.79:
fac1 = Re/sqrt(1-(2*f - f*f)*sin(phi)^2);
fac2 = (Re*(1-f)^2/sqrt(1-(2*f - f*f)*sin(phi)^2) + H)*sin(phi);
for i = 1:3
    R(i,1) = (fac1 + H)*cos(phi)*cos(theta(i));
    R(i,2) = (fac1 + H)*cos(phi)*sin(theta(i));
    R(i,3) = fac2;
    rho(i,1) = cos(dec(i))*cos(ra(i));
    rho(i,2) = cos(dec(i))*sin(ra(i));
    rho(i,3) = sin(dec(i));
end

%...Algorithms 5.5 and 5.6:
[r, v, r_old, v_old] = gauss(rho(1,:), rho(2,:), rho(3,:), ...
                             R(1,:),    R(2,:),  R(3,:), ...
                             t(1),      t(2),    t(3));

%...Algorithm 4.2 for the initial estimate of the state vector
%   and for the iteratively improved one:
coe_old = coe_from_sv(r_old,v_old,mu);
coe     = coe_from_sv(r,v,mu);

%...Echo the input data and output the solution to
%   the command window:
fprintf('-----------------------------------------------------')
fprintf('\n Example 5.11: Orbit determination by the Gauss method\n')
fprintf('\n Radius of earth (km)               = %g', Re)
fprintf('\n Flattening factor                  = %g', f)
fprintf('\n Gravitational parameter (km^3/s^2) = %g', mu)
fprintf('\n\n Input data:\n');
fprintf('\n Latitude (deg)                = %g', phi/deg);
fprintf('\n Altitude above sea level (km) = %g', H);
fprintf('\n\n Observations:')
fprintf('\n               Right')
fprintf('                                    Local')
fprintf('\n   Time (s)   Ascension (deg)   Declination (deg)')
fprintf('   Sidereal time (deg)')
for i = 1:3
    fprintf('\n %9.4g %11.4f %19.4f %20.4f', ...
                 t(i), ra(i)/deg, dec(i)/deg, theta(i)/deg)
end

fprintf('\n\n Solution:\n')
```

```
fprintf('\n Without iterative improvement...\n')
fprintf('\n');
fprintf('\n r (km)                          = [%g, %g, %g]', ...
                                   r_old(1), r_old(2), r_old(3))
fprintf('\n v (km/s)                        = [%g, %g, %g]', ...
                                   v_old(1), v_old(2), v_old(3))
fprintf('\n');

fprintf('\n   Angular momentum (km^2/s)     = %g', coe_old(1))
fprintf('\n   Eccentricity                  = %g', coe_old(2))
fprintf('\n   RA of ascending node (deg)    = %g', coe_old(3)/deg)
fprintf('\n   Inclination (deg)             = %g', coe_old(4)/deg)
fprintf('\n   Argument of perigee (deg)     = %g', coe_old(5)/deg)
fprintf('\n   True anomaly (deg)            = %g', coe_old(6)/deg)
fprintf('\n   Semimajor axis (km)           = %g', coe_old(7))
fprintf('\n   Periapse radius (km)          = %g', coe_old(1)^2 ...
                                          /mu/(1 + coe_old(2)))
%...If the orbit is an ellipse, output the period:
if coe_old(2)<1
    T = 2*pi/sqrt(mu)*coe_old(7)^1.5;
    fprintf('\n   Period:')
    fprintf('\n     Seconds                     = %g', T)
    fprintf('\n     Minutes                     = %g', T/60)
    fprintf('\n     Hours                       = %g', T/3600)
    fprintf('\n     Days                        = %g', T/24/3600)
end

fprintf('\n\n With iterative improvement...\n')
fprintf('\n');
fprintf('\n r (km)                          = [%g, %g, %g]', ...
                                            r(1), r(2), r(3))
fprintf('\n v (km/s)                        = [%g, %g, %g]', ...
                                            v(1), v(2), v(3))
fprintf('\n');
fprintf('\n   Angular momentum (km^2/s)     = %g', coe(1))
fprintf('\n   Eccentricity                  = %g', coe(2))
fprintf('\n   RA of ascending node (deg)    = %g', coe(3)/deg)
fprintf('\n   Inclination (deg)             = %g', coe(4)/deg)
fprintf('\n   Argument of perigee (deg)     = %g', coe(5)/deg)
fprintf('\n   True anomaly (deg)            = %g', coe(6)/deg)
fprintf('\n   Semimajor axis (km)           = %g', coe(7))
fprintf('\n   Periapse radius (km)          = %g', coe(1)^2 ...
                                          /mu/(1 + coe(2)))
%...If the orbit is an ellipse, output the period:
if coe(2)<1
    T = 2*pi/sqrt(mu)*coe(7)^1.5;
    fprintf('\n   Period:')
    fprintf('\n     Seconds                     = %g', T)
```

```
    fprintf('\n     Minutes                    = %g', T/60)
    fprintf('\n     Hours                      = %g', T/3600)
    fprintf('\n     Days                       = %g', T/24/3600)
end
fprintf('\n-----------------------------------------------------\n')

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

***Output from*** `Example_5_11`

```
( **Number of Gauss improvement iterations = 14)

-----------------------------------------------------
 Example 5.11: Orbit determination by the Gauss method

 Radius of earth (km)               = 6378
 Flattening factor                  = 0.00335278
 Gravitational parameter (km^3/s^2) = 398600

 Input data:

 Latitude (deg)                = 40
 Altitude above sea level (km) = 1

 Observations:
              Right                                     Local
   Time (s)   Ascension (deg)   Declination (deg)   Sidereal time (deg)
         0     43.5365             -8.7833              44.5065
     118.1     54.4196            -12.0739              45.0000
     237.6     64.3178            -15.1054              45.4992

 Solution:

 Without iterative improvement...


 r (km)                          = [5659.03, 6533.74, 3270.15]
 v (km/s)                        = [-3.8797, 5.11565, -2.2397]

   Angular momentum (km^2/s)     = 62705.3
   Eccentricity                  = 0.097562
   RA of ascending node (deg)    = 270.023
   Inclination (deg)             = 30.0105
   Argument of perigee (deg)     = 88.654
   True anomaly (deg)            = 46.3163
   Semimajor axis (km)           = 9959.2
   Periapse radius (km)          = 8987.56
```

```
  Period:
    Seconds                     = 9891.17
    Minutes                     = 164.853
    Hours                       = 2.74755
    Days                        = 0.114481

 With iterative improvement...


 r (km)                         = [5662.04, 6537.95, 3269.05]
 v (km/s)                       = [-3.88542, 5.12141, -2.2434]

  Angular momentum (km^2/s)     = 62816.7
  Eccentricity                  = 0.0999909
  RA of ascending node (deg)    = 269.999
  Inclination (deg)             = 30.001
  Argument of perigee (deg)     = 89.9723
  True anomaly (deg)            = 45.0284
  Semimajor axis (km)           = 9999.48
  Periapse radius (km)          = 8999.62
  Period:
    Seconds                     = 9951.24
    Minutes                     = 165.854
    Hours                       = 2.76423
    Days                        = 0.115176
-----------------------------------------------------
```

# Chapter 6

## D.30 Calculate the state vector after a finite-time, constant thrust delta-v maneuver

### Function file: integrate_thrust.m

```
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function integrate_thrust
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function uses rkf45 to numerically integrate Equation 6.26 during
  the delta-v burn and then find the apogee of the post-burn orbit.

  The input data are for the first part of Example 6.15.

  mu        - gravitational parameter (km^3/s^2)
  RE        - earth radius (km)
  g0        - sea-level acceleration of gravity (m/s^2)
```

```
  T         - rated thrust of rocket engine (kN)
  Isp       - specific impulse of rocket engine (s)
  m0        - initial spacecraft mass (kg)
  r0        - initial position vector (km)
  v0        - initial velocity vector (km/s)
  t0        - initial time (s)
  t_burn    - rocket motor burn time (s)
  y0        - column vector containing r0, v0 and m0
  t         - column vector of the times at which the solution is found (s)
  y         - a matrix whose elements are:
                 columns 1, 2 and 3:
                    The solution for the x, y and z components of the
                    position vector r at the times t
                 columns 4, 5 and 6:
                    The solution for the x, y and z components of the
                    velocity vector v at the times t
                 column 7:
                    The spacecraft mass m at the times t
  r1        - position vector after the burn (km)
  v1        - velocity vector after the burn (km/s)
  m1        - mass after the burn (kg)
  coe       - orbital elements of the post-burn trajectory
              (h e RA incl w TA a)
  ra        - position vector vector at apogee (km)
  va        - velocity vector at apogee (km)
  rmax      - apogee radius (km)

  User M-functions required:  rkf45, coe_from_sv, rv_from_r0v0_ta
  User subfunctions required: rates, output
%}
% ---------------------------------------------

%...Preliminaries:
clear all; close all; clc
global mu
deg     = pi/180;
mu      = 398600;
RE      = 6378;
g0      = 9.807;

%...Input data:
r0      = [RE+480    0   0];
v0      = [  0   7.7102 0];
t0      = 0;
t_burn  = 261.1127;

m0      = 2000;
T       = 10;
```

```
Isp     = 300;
%...end Input data

%...Integrate the equations of motion over the burn time:
y0    = [r0 v0 m0]';
[t,y] = rkf45(@rates, [t0 t_burn], y0, 1.e-16);

%...Compute the state vector and mass after the burn:
r1  = [y(end,1) y(end,2) y(end,3)];
v1  = [y(end,4) y(end,5) y(end,6)];
m1  = y(end,7);
coe = coe_from_sv(r1,v1,mu);
e   = coe(2);  %eccentricity
TA  = coe(6);  %true anomaly (radians)
a   = coe(7);  %semimajor axis

%...Find the state vector at apogee of the post-burn trajectory:
if TA <= pi
    dtheta = pi - TA;
else
    dtheta = 3*pi - TA;
end
[ra,va] = rv_from_r0v0_ta(r1, v1, dtheta/deg, mu);
rmax    = norm(ra);

output

%...Subfunctions:

%~~~~~~~~~~~~~~~~~~~~~~~~~~
function dfdt = rates(t,f)
%~~~~~~~~~~~~~~~~~~~~~~~~~~
%{
  This function calculates the acceleration vector using Equation 6.26.

  t          - time (s)
  f          - column vector containing the position vector, velocity
               vector and the mass at time t
  x, y, z    - components of the position vector (km)
  vx, vy, vz - components of the velocity vector (km/s)
  m          - mass (kg)
  r          - magnitude of the position vector (km)
  v          - magnitude of the velocity vector (km/s)
  ax, ay, az - components of the acceleration vector (km/s^2)
  mdot       - rate of change of mass (kg/s)
  dfdt       - column vector containing the velocity and acceleration
               components and the mass rate
%}
```