

Numerical Methods and Utility Functions

This document describes a suite of numerical methods and utility functions that can be used to create and support your own MATLAB scripts. Routines are provided in the following areas of numerical analysis:

- Trigonometry and matrix routines
- Solution of first and second-order systems of differential equations
- Quadrature (numerical integration)
- One-dimensional root-finding and minimization
- Linear and cubic spline Interpolation functions
- Date and time routines
- Interactive data request routines

TRIGONOMETRY AND MATRIX ROUTINES

atan3.m – four quadrant inverse tangent

This function calculates the four quadrant inverse tangent such that the computed angle is between 0 and 2π .

The syntax of this MATLAB function is

```
function y = atan3 (a, b)  
  
% four quadrant inverse tangent  
  
% input  
  
% a = sine of angle  
% b = cosine of angle  
  
% output  
  
% y = angle (radians; 0 =< c <= 2 * pi)
```

rotmat.m – fundamental rotation matrix

This MATLAB function computes the elementary rotation matrix for a user-defined rotation angle about a user-defined rotation axis. All calculations follow the right-handed rule and counterclockwise rotation angles are positive.

The three rotation matrices for the x , y and z axes are defined by

Orbital Mechanics with MATLAB

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \quad R_z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where θ is the rotation angle.

The syntax of this MATLAB function is

```
function rmatrix = rotmat(iaxis, theta)

% rotation matrix

% input

% iaxis = 1 ==> x-axis
%       = 2 ==> y-axis
%       = 3 ==> z-axis
% theta = rotation angle (radians)

% output

% rmatrix = rotation matrix
```

matran.m – multiple sequence transformation matrix

This MATLAB function computes a transformation matrix for a given rotation sequence of up to four rotations about the x, y, or z axes.

The syntax of this MATLAB function is

```
function trans = matran (a1, nax1, a2, nax2, a3, nax3, a4, nax4)

% this function computes a transformation matrix for a
% given rotation sequence of up to four rotations about the
% x, y, or z axes. the first row of the matrix contains the
% direction cosines of the new x-axis with respect to the
% original xyz axes. the second row is the y-axis and the
% third row is the z-axis.

% in equation form

% x2      (1,1  1,2  1,3)  (x1)
% y2 = trans (2,1  2,2  2,3) * (y1)
% z2      (3,1  3,2  3,3)  (z1)

% input

% a1, a2, a3, a4 = rotation angle about the respective axis (radians)

% nax1, nax2, nax3, nax4 = axis of rotation
%                       (1 = x, 2 = y, 3 = z, 0 = no more rotations)
```

```
% output  
  
% trans = transformation matrix (3 rows by 3 columns)
```

DIFFERENTIAL EQUATIONS

rkf45.m – Runge-Kutta-Fehlberg 4(5) method for first-order systems

This function is a MATLAB implementation of the 4(5) version of the Runge-Kutta-Fehlberg algorithm. It can be used to solve systems of first order, nonlinear vector differential equations of the form $\dot{\mathbf{y}} = f(\mathbf{y}, t)$.

This function requires initialization the first time it is called. The following statement in the main MATLAB script will accomplish this:

```
rkcoef = 1;
```

This variable should also be placed in a `global` statement at the beginning of the main script which calls this function. After the first call, the function will set this value to 0.

The syntax of this MATLAB function is

```
function xout = rkf45 (deq, neq, ti, tf, h, tetol, y)  
  
% solution of first order system of differential equations  
  
% Runge-Kutta-Fehlberg 4(5)  
  
% fourth-order solution with fifth-order error control  
  
% input  
  
% neq = number of equations in system  
% ti = initial simulation time  
% tf = final simulation time  
% h = guess for step size  
% tetol = truncation error tolerance  
% y = initial integration vector  
  
% output  
  
% xout = final integration vector
```

rkf78.m – Runge-Kutta-Fehlberg 7(8) method for first-order systems

This function is a MATLAB implementation of the 7(8) version of the Runge-Kutta-Fehlberg algorithm. It can be used to solve systems of first order, nonlinear vector differential equations of the form $\dot{\mathbf{y}} = f(\mathbf{y}, t)$.

Orbital Mechanics with MATLAB

This function requires initialization the first time it is called. The following statement in the main MATLAB script will accomplish this:

```
rkcoef = 1;
```

This variable should also be placed in a `global` statement at the beginning of the main script which calls this function. After the first call, the function will set this value to 0.

Near the end of this function is the following code:

```
if (xerr > 1)
    % reject current step
    ti = twrk;
    x = xwrk;
else
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % accept current step
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % perform graphics, additional
    % calculations, etc. at this point
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
end
```

When the processing reaches this `else` statement, a successful integration step has been completed and additional processing such as graphics or other calculations can be performed here. For example, the following code performs the rectification required for Encke's method.

```
if (xerr > 1)
    % reject current step

    ti = twrk;
    x = xwrk;

    % compute new step size

    dt = 0.7 * dt * (1 / xerr) ^ (1 / 8);
else
    % rectify and re-initialize

    ytbf = twobody4(dt, ytbi);

    for i = 1:1:6
        ytrue(i) = ytbf(i) + x(i);

        ytbi(i) = ytrue(i);

        x(i) = 0;
    end

    tsaved = ti;
end
```

The syntax for this MATLAB function is

Orbital Mechanics with MATLAB

```
function xout = rkf78 (deq, neq, ti, tf, h, tetol, x)

% solve first order system of differential equations

% Runge-Kutta-Fehlberg 7(8) method

% input

% deq = name of function which defines the
%       system of differential equations
% neq = number of differential equations
% ti   = initial simulation time
% tf   = final simulation time
% h    = initial guess for integration step size
% tetol = truncation error tolerance (non-dimensional)
% x    = integration vector at time = ti

% output

% xout = integration vector at time = tf
```

This software suite includes a MATLAB script named `demo_rkf78` that demonstrates the proper user interaction with this function.

nym4.m – Nystrom fourth-order method for second-order systems

This function is a MATLAB implementation of a fourth-order Nystrom algorithm. It can be used to solve systems of second order differential equations of the form $\ddot{\mathbf{x}} = f(\mathbf{x}, \dot{\mathbf{x}}, t)$.

This function requires initialization the first time it is called. The following statement in the main MATLAB script will accomplish this:

```
nycoef = 1;
```

This variable should also be placed in a `global` statement at the beginning of the main script that calls this function.

The syntax of this MATLAB function is

```
function [rf, vf] = nym4(deq, n, tp, dt, rs, vs)

% solve a system of second order differential equations

% fourth order Nystrom method (fixed step size)

% input

% deq = function name of systems of
%       differential equations
% n   = number of equations in user-defined
%       system of differential equations
```

Orbital Mechanics with MATLAB

```
% tp = current simulation time
% dt = integration step size
% rs = position vector at initial time
% vs = velocity vector at initial time

% output

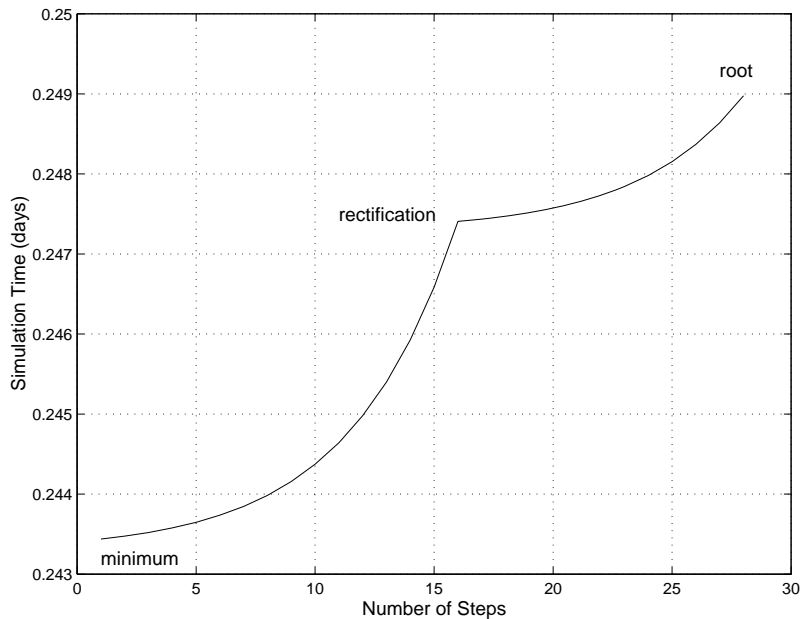
% rf = position vector at time = tp + dt
% vf = velocity vector at time = tp + dt
```

ROOT-FINDING AND MINIMIZATION

broot.m – bracket a single root of a nonlinear function

This function can be used to bracket a single root of a single nonlinear equation of the form $y = f(x)$. It uses a combination of *geometric acceleration* and *rectification* to bracket roots. The basic idea is to find the endpoints of an interval x_1 and x_2 such that $f(x_1)f(x_2) < 0$. The user must supply this function with an initial guess for x_1 and x_2 . Typically x_1 is the time of an objective function minimum or maximum and x_2 is a time 10 seconds after (forward root) or 10 seconds before (backward root) the value of x_1 .

The following diagram illustrates this process. The acceleration factor for this example is 0.25 and the rectification interval is 300 seconds. The acceleration factor increases the size of each step geometrically and the rectification interval monitors the length of the current bracketing interval. When necessary the rectification logic reinitializes the search and prevents the interval from becoming too large and skipping one or both ends of the bracketing interval completely. The process starts at the bottom left, labeled *minimum*, which is the time of the function minimum for this example. The x -axis is the number of steps taken in the search for the bracketing interval and the y -axis is the elapsed simulation time. The size of each step increases geometrically until the length of the current step reaches the value of the rectification interval. At that point, labeled *rectification* on the plot, the step size is reset and the process begins again. Eventually, the process will bracket the root, labeled *root* on this plot, and the function will return the endpoints x_1 and x_2 of this bracketing interval.



The syntax of this MATLAB function is

```
function [x1out, x2out] = broot (f, x1in, x2in, factor, dxmax)

% bracket a single root of a nonlinear equation

% input

% f      = objective function coded as y = f(x)
% x1in   = initial guess for first bracketing x value
% x2in   = initial guess for second bracketing x value
% factor = acceleration factor (non-dimensional)
% dxmax  = rectification interval

% output

% x1out = final value for first bracketing x value
% x2out = final value for second bracketing x value
```

brent.m – solve for a single root of a nonlinear function

This function uses Brent's method to find a single root of a single nonlinear equation of the form $y = f(x)$. Derivatives are not required for this algorithm. However, the user should ensure that a root is bracketed ($f(x_1)f(x_2) < 0$) before calling this routine.

The syntax of this MATLAB function is

```
function [xroot, froot] = brent (f, x1, x2, rtol)

% solve for a single real root of a nonlinear equation
```

Orbital Mechanics with MATLAB

```
% Brent's method

% input

% f    = objective function coded as y = f(x)
% x1   = lower bound of search interval
% x2   = upper bound of search interval
% rtol = algorithm convergence criterion

% output

% xroot = real root of f(x) = 0
% froot = function value at f(x) = 0
```

snle.m – solve for the roots of a system of nonlinear equations

This MATLAB function solves for the real roots of a user-defined system of non-linear equations using Broyden's method with finite-difference derivatives.

The syntax of this MATLAB function is

```
function [x, niter, icheck] = snle (usrfun, x, n, maxiter)

% solution of a system of non-linear equations function

% input

% usrfun = name of user-defined function that evaluates nle system
% x      = initial guess for solution vector
% n      = number of elements in solution vector
% maxiter = maximum number of algorithm iterations allowed

% output

% x      = solution vector
% niter  = number of algorithm iterations performed
% icheck = solution indicator
%        = 0 ==> normal return
%        = 1 ==> algorithm can make no further progress
```

The syntax of the MATLAB function that defines the user's system of non-linear equations is as follows:

```
function y = usrfun(x)

% user-defined system of nonlinear equations

% input

% x = array of current dependent variables

% output
```

```
% y = function value array evaluated at x
```

minima.m – one-dimensional minimization

This function uses Brent's method to find a single minimum or maximum of a user-defined scalar nonlinear objective function. Derivatives are not required for this algorithm. This MATLAB algorithm is based on the methods described in the book, *Algorithms for Minimization without Derivatives*.

The syntax of this MATLAB function is

```
function [xmin, fmin] = minima (f, a, b, tolm)

% one-dimensional minimization subroutine

% Brent's method

% input

% f    = objective function coded as y = f(x)
% a    = initial x search value
% b    = final x search value
% tolm = convergence criterion

% output

% xmin = minimum x value
% fmin = minimum function value
```

oevent1.m – find minimization/root finding orbital event - time argument in days

This MATLAB function is a combination of one-dimensional minimization and root-finding algorithms. It first finds a minimum or maximum within a user-defined search interval, and then calculates the corresponding “reverse” and “forward” root.

The syntax of this MATLAB function is

```
function oevent1 (objfunc, prtfunc, ti, tf, dt, dtsml)

% predict minimization/root-finding orbital events

% time argument in days

% input

% objfunc = objective function
% prtfunc = display results function
% ti      = initial simulation time
% tf      = final simulation time
% dt      = step size used for bounding minima
% dtsml   = small step size used to determine whether
%          the function is increasing or decreasing
```

oevent2.m – find minimization/root finding orbital event - time argument in seconds

This MATLAB function is a combination of one-dimensional minimization and root-finding algorithms. It first finds a minimum or maximum within a user-defined search interval, and then calculates the corresponding “reverse” and “forward” root.

The syntax of this MATLAB function is

```
function oevent2 (objfunc, prtfunc, ti, tf, dt, dtsml)

% predict minimization/root-finding orbital events

% time argument in seconds

% input

% objfunc = objective function
% prtfunc = display results function
% ti      = initial simulation time
% tf      = final simulation time
% dt      = step size used for bounding minima
% dtsml   = small step size used to determine whether
%          the function is increasing or decreasing
```

oevent3.m – find minimization orbital event - time argument in days

This function can be used to find a minimum or maximum of an objective function within a user-defined search interval. The time argument must be in days.

The syntax of this MATLAB function is

```
function oevent3 (objfunc, prtfunc, ti, tf, dt, dtsml)

% predict minimization type orbital events

% time argument in days

% input

% objfunc = objective function
% prtfunc = display results function
% ti      = initial simulation time
% tf      = final simulation time
% dt      = step size used for bounding minima
% dtsml   = small step size used to determine whether
%          the function is increasing or decreasing
```

oevent4.m – find minimization orbital event - time argument in seconds

This function can be used to find minimum or maximum of an objective function within a user-defined search interval. The time argument must be in seconds.

The syntax of this MATLAB function is

```
function oevent4 (objfunc, prtfunc, ti, tf, dt, dtsml)

% predict minimization type orbital events

% time argument in seconds

% input

% objfunc = objective function
% prtfunc = display results function
% ti      = initial simulation time
% tf      = final simulation time
% dt      = step size used for bounding minima
% dtsml   = small step size used to determine whether
%           the function is increasing or decreasing
```

INTERPOLATION

ointerp.m – interpolation of orbital state vectors

This function interpolates orbital state vectors. The input to this routine consists of two position, velocity and acceleration vectors and their corresponding epoch times.

The syntax of this MATLAB function is

```
function [ri, vi, ai] = ointerp (t1, t2, ti, r1, v1, a1, r2, v2, a2)

% state vector interpolation

% input

% t1 = first time
% t2 = second time
% ti = interpolation time
% r1 = position vector at time t1
% v1 = velocity vector at time t1
% a1 = acceleration vector at time t1
% r2 = position vector at time t2
% v2 = velocity vector at time t2
% a2 = acceleration vector at time t2

% output

% ri = position vector at time ti
% vi = velocity vector at time ti
% ai = acceleration vector at time ti
```

bilinear.m – bilinear interpolation function

This MATLAB function can be used to perform bilinear interpolation of tabular data of the form $z = f(x, y)$.

The syntax of this MATLAB function is

```
function zval = bilinear (x, y, z, xval, yval)

% bilinear interpolation

% z = f(x, y)

% input

% x    = column vector of x data (nx rows)
% y    = column vector of y data (ny rows)
% z    = matrix of z data (nx rows by ny columns)
% xval = x argument
% yval = y argument

% output

% zval = z function value at xval, yval
```

csint.m – cubic spline integration of tabular data

This MATLAB function uses a cubic spline to numerically integrate tabular data of the form $y = f(x)$. Please note that this algorithm requires at least two x - y pairs in the data.

The syntax of this MATLAB function is

```
function sum = csint(n, x, y)

% cubic spline integration of tabulated data

% input

% n    = number of x and y data points (n >= 2)
% x    = vector of x data (n rows)
% y    = vector of y data (n rows)

% output

% sum  = integral from x(1) to x(n)
```

csder.m – cubic spline derivative of tabular data

This MATLAB function uses a cubic spline to compute the derivative of tabular data of the form $y = f(x)$ for a user-defined value of x . Please note that this algorithm requires at least two x - y pairs in the data.

The syntax of this MATLAB function is

```
function dval = csder(n, x, y, xval)

% cubic spline derivative of tabulated data

% input

% n    = number of x and y data points (n >= 2)
% x    = vector of x data (n rows)
% y    = vector of y data (n rows)
% xval = x argument

% output

% dval = derivative of tabular data at x = xval
```

spcoef.m, spval.m and spdata.m – cubic spline interpolation of tabular data

A spline is a thin wooden or plastic instrument used by draftsmen to draw smooth curves through sets of x and y data points called “knots”. In the early years of aircraft design, “loftsmen” working in spacious lofts would use long splines to design the wing, fuselage, and other curved surfaces of an airplane.

Mathematically, a cubic spline is a third-order curve applied to subsets of user-defined pairs of x and y data points or knots. A cubic spline has minimum oscillatory behavior that results in smooth transitions between data points. This property makes the curve visually pleasing.

The general form of a third-order or cubic polynomial is given by:

$$f(x) = ax^3 + bx^2 + cx + d$$

where a , b , c and d are constant coefficients. These coefficients are determined from several equations that reflect the properties of the cubic spline. These conditions involve such things as function and derivative values. For example, the function or y values must be equal at the interior knots, and the first and last functions must pass through the endpoints. For smoothness the first derivatives at the interior knots must also be equal. Constraints on the endpoints also determine the type of cubic spline. The spline might be natural, clamped or even cyclic.

This software suite includes a MATLAB script called `demo_sfit` that demonstrates how to interpolate tabular data of the form $y = f(x)$ using cubic splines. The following is an outline of the major steps required to process a simple ASCII data file, create the cubic spline coefficient arrays and interpolate the data at user-defined x data values. This script reads and fits a data file called `f10nom.dat` which is a typical atmospheric data file.

The cubic spline process involves three individual functions. The MATLAB function that performs each operation is shown in courier font. The steps are as follows:

- (1) read the ASCII data file
- (2) find the total number of x and y data points in this file

Orbital Mechanics with MATLAB

- (3) sample and load data arrays for the spline operations (`spdata`)
- (4) generate the spline coefficient arrays (`spcoef`)
- (5) perform interpolation for one or more user-defined x data values (`spval`)

The following is part of the MATLAB code that performs the operations listed above:

```
% open and read data file
fid = fopen('f10nom.dat', 'r');
a = fscanf(fid, '%g %g', [inf]);
status = fclose(fid);

% find number of x and y data points
ndata = length(a) / 2;

% put data file into x and y arrays
j = 1;

for i = 1:1:ndata
    y(i) = a(j);
    x(i) = a(j + 1);
    j = j + 2;
end

% sample and load data arrays for spline operations
nsp = 5;

[xk, yk, npts] = spdata(x, y, ndata, nsp);

% generate spline coefficient arrays
[c1, c2, c3] = spcoef(xk, yk, npts);
```

Notice that the `f10nom.dat` data file is sampled at an interval defined by the value for `nsp`. Best results are usually obtained by setting this value to a number between 3 and 10. The interpolation of one or more x data points using the cubic spline coefficients is accomplished by a MATLAB function called `spval`.

The following source code demonstrates how to perform a cubic spline interpolation of every third point of the original data.

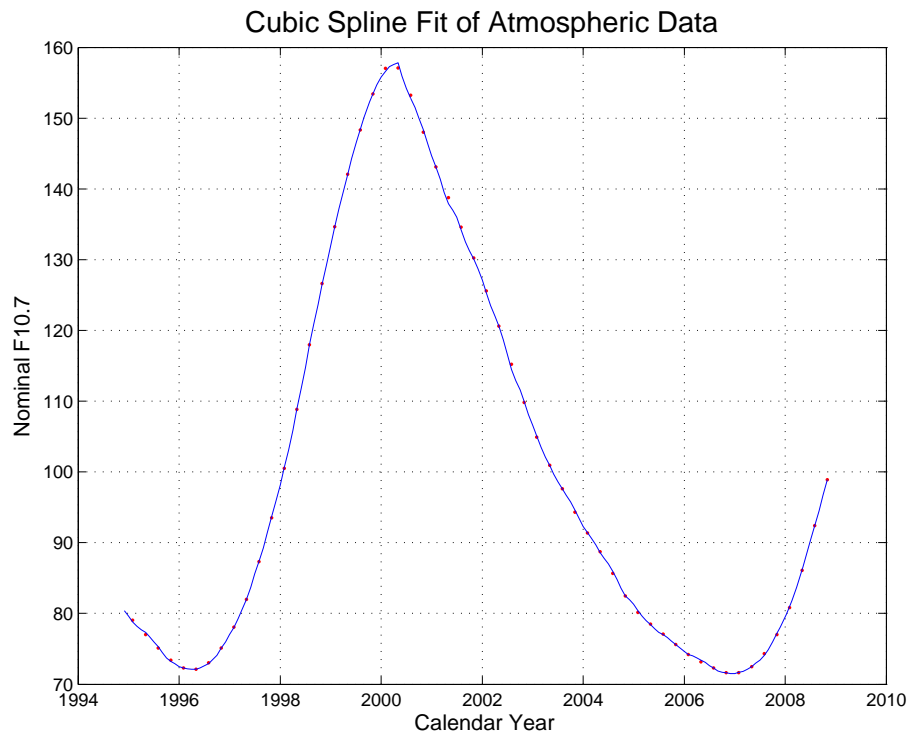
```
% evaluate spline fit at every third data point
nspj = 3;
i = 0;
```

Orbital Mechanics with MATLAB

```
for j = 1:1:ndata
    if (mod(j, nspj) ~= 0)
        % null
    else
        i = i + 1;
        xval = x(j);
        sx = spval(xval, xk, yk, npts, c1, c2, c3);
        xfit(i) = xval;
        yfit(i) = sx;
    end
end
```

The errors of the curve fit can be determined by examining the difference between the original y data and the $yfit$ array calculated by the `spval` function.

The following is a plot of the original data (solid blue line) and the cubic spline interpolation of the selected data points (red dots).



DATE AND TIME ROUTINES

This section describes MATLAB functions that can be used to convert between calendar and Julian dates and several useful time systems. A routine is also provided that converts a Julian date to its equivalent calendar date and time in character or “string” format.

julian.m – calendar date to Julian date

This MATLAB function converts a calendar date to its corresponding Julian date. Be sure to pass this routine all digits of the calendar year.

The syntax of this MATLAB function is

```
function jdate = julian (month, day, year)

% Julian date subroutine

% Input

% month = calendar month [1 - 12]
% day   = calendar day [1 - 31]
% year  = calendar year [yyyy]

% Output

% jdate = Julian date
```

gdate.m – julian date to calendar date

This MATLAB function converts a Julian date to its corresponding calendar date.

The syntax of this MATLAB function is

```
function [month, day, year] = gdate (jdate)

% convert Julian date to Gregorian (calendar) date

% input

% jdate = Julian date

% output

% month = calendar month [1 - 12]
% day   = calendar day [1 - 31]
% year  = calendar year [yyyy]
```

jd2str.m – julian date to calendar date and universal time strings

This function converts a Julian date to its equivalent calendar date and universal time strings.

The syntax of this MATLAB function is

```
function [cdstr, utstr] = jd2str(jdate)

% convert Julian date to string equivalent
% calendar date and universal time
```

```
% input  
  
% jdate = Julian date  
  
% output  
  
% cdstr = calendar date string  
% utstr = universal time string
```

utc2tdt.m – convert UTC Julian date to TDT Julian date

This function converts a Julian date on the coordinated universal time (UTC) time scale to a Julian date on the terrestrial dynamic time (TDT) scale.

The syntax of this MATLAB function is

```
function jdtdt = utc2tdt (jdutc, tai_utc)  
  
% convert UTC julian date to TDT julian date  
  
% input  
  
% jdutc   = UTC julian date  
% tai_utc = TAI-UTC (seconds)  
  
% output  
  
% jdtdt = TDT julian date
```

utc2tdb.m – convert UTC Julian date to TDB Julian date

This function converts a Julian date on the coordinated universal time (UTC) time scale to a Julian date on the barycentric dynamic time (TDB) scale.

The syntax of this MATLAB function is

```
function jdtdb = utc2tdb (jdutc, tai_utc)  
  
% convert UTC julian date to TDB julian date  
  
% input  
  
% jdutc   = UTC julian date  
% tai_utc = TAI-UTC (seconds)  
  
% output  
  
% jdtdb = TDB julian date
```

The difference between UTC and atomic time TAI, is always an integral number of seconds. The value of was 10 seconds in January 1972, and increases by one each time a leap second is

declared. The following is a table of TAI-UTC (column 1, in seconds) for calendar months and years between 1972 and 2008.

```
10, @1972-JAN-1
11, @1972-JUL-1
12, @1973-JAN-1
13, @1974-JAN-1
14, @1975-JAN-1
15, @1976-JAN-1
16, @1977-JAN-1
17, @1978-JAN-1
18, @1979-JAN-1
19, @1980-JAN-1
20, @1981-JUL-1
21, @1982-JUL-1
22, @1983-JUL-1
23, @1985-JUL-1
24, @1988-JAN-1
25, @1990-JAN-1
26, @1991-JAN-1
27, @1992-JUL-1
28, @1993-JUL-1
29, @1994-JUL-1
30, @1996-JAN-1
31, @1997-JUL-1
32, @1999-JAN-1
33, @2008-JAN-1
```

QUADRATURE

This section describes two MATLAB functions that can be used to integrate one-dimensional user-defined analytic functions.

asimpson.m – adaptive quadrature function

This MATLAB function uses an adaptive Simpson algorithm to calculate the definite integral of a user-defined function of the form $y = f(x)$.

```
function [sum, esterr, iflag] = asimpson (usrfunc, a, b, acc)

% adaptive simpson quadrature routine

% input

% usrfunc = name of user-defined function
% a       = lower integration limit
% b       = upper integration limit
% acc     = accuracy

% output

% sum     = integral from a to b
% esterr  = relative error
```

```
% iflag = error flag
%     1 => no error
%     2 => more than 30 levels
%     3 => subinterval too small
%     4 => more than 2000 function evaluations
```

kquad.m – Gauss-Kronrod quadrature function

This MATLAB function uses a Gauss-Kronrod quadrature algorithm due to T. Patterson with enhancements by F. Krogh and W. Snyder to calculate the definite integral of a user-defined function of the form $y = f(x)$.

The syntax of this MATLAB function is as follows:

```
function [result, k, icheck] = kquad(usrfunc, a, b, tol)

% one-dimensional quadrature

% method due to T. Patterson with modifications
% by F. Krogh and W. Snyder (ACM Algorithm 699)

% input

% usrfunc = name of user-defined function
% a       = lower integration limit
% b       = upper integration limit
% tol     = convergence tolerance

% output

% result = array of solution(s)
% k       = index of result array which best satisfies tol
% icheck = 0 ==> convergence
%         = 1 ==> non-convergence
```

UTILITY ROUTINES

This section describes several MATLAB functions that perform utility and conversion calculations.

sepang.m – separation angle function

This function computes the separation angle between two coplanar vectors. The user must provide a rotation direction unit vector. For example, an input unit vector $[1 \ 0 \ 0]^T$ will compute the separation angle from the first vector to the second vector.

The syntax of this MATLAB function is as follows:

```
function theta = sepang (r1, r2, rhat)

% separation angle between two coplanar vectors
```

```
% input

% r1 = first vector
% r2 = second vector
% rhat = rotation direction unit vector
%       = [+1 0 0] ==> from r1 to r2
%       = [-1 0 0] ==> from r2 to r1

% output

% theta = separation angle (radians)
```

deg2dms.m – convert degrees function

This function can be used to convert a floating point argument in degrees into its equivalent degrees, minutes, seconds, and string representation. For example, the command

```
[d,m,s,degstr]=deg2dms(22.12345)
```

produces the following screen output:

```
+22d 07m 24.42s
```

The syntax of this MATLAB function is as follows:

```
function [d, m, s, dmsstr] = deg2dms (dd)

% convert decimal degrees to degrees,
% minutes, seconds and equivalent string

% input

% dd = angle in decimal degrees

% output

% d      = integer degrees
% m      = integer minutes
% s      = seconds
% dmsstr = string equivalent of input
```

hrs2hms.m – convert hours function

This function can be used to convert a floating point argument in hours into its equivalent degrees, minutes, seconds, and string representation.

The syntax of this MATLAB function is as follows:

```
function [h, m, s, hmsstr] = hrs2hms (hrs)

% convert decimal hours to hours,
```

Orbital Mechanics with MATLAB

```
% minutes, seconds and equivalent string
% input
% dd = angle in hours
% output
% h      = integer hours
% m      = integer minutes
% s      = seconds
% hmsstr = string equivalent of input
```

keycheck.m – request key press function

This simple MATLAB function pauses a program and asks the user to press any key to continue. It will print the following text on the screen and wait for the user to press any key.

```
< please press any key to continue >
```

readoe1.m – read orbital elements data file – single satellite

This MATLAB function can be used to read a simple ASCII data file containing classical orbital elements for a single satellite. Please note that angular orbital elements are returned in radians.

The syntax of this MATLAB function is

```
function [fid, oev] = readoe1(filename)
% read orbital elements data file
% required by cowell11.m
% input
% filename = name of orbital element data file
% output
% fid = file id
% oev(1) = semimajor axis
% oev(2) = orbital eccentricity
% oev(3) = orbital inclination
% oev(4) = argument of perigee
% oev(5) = right ascension of the ascending node
% oev(6) = true anomaly
```

The following is a typical orbital elements data file. When creating this type of ASCII text file the user can change the numeric data and annotation, but do not change the number of lines in the file or the line location of the data. Please note the units and valid range for each data item.

```
semimajor axis (kilometers)
```

Orbital Mechanics with MATLAB

```
(semimajor axis > 0)
6878.14

orbital eccentricity (non-dimensional)
(0 <= eccentricity < 1)
.0125

orbital inclination (degrees)
(0 <= inclination <= 180)
28.5

argument of perigee (degrees)
(0 <= argument of perigee <= 360)
270

right ascension of the ascending node (degrees)
(0 <= RAAN <= 360)
45

true anomaly (degrees)
(0 <= true anomaly <= 360)
0
```

INTERACTIVE REQUEST OF PROGRAM INPUTS

This section describes several MATLAB functions that can be used to interactively request information required during the execution of main scripts.

getdate.m – request calendar date

This function interactively requests the calendar date. Please note that all digits of the calendar year are required.

The syntax of this MATLAB function is as follows:

```
function [m, d, y] = getdate

% interactive request and input of calendar date

% output

% m = calendar month
% d = calendar day
% y = calendar year
```

The following is the screen display for this function. Please note the valid range of inputs.

```
please input the calendar date
(1 <= month <= 12, 1 <= day <= 31, year = all digits!)
?
```

The calendar date should be input with individual elements separated by commas.

getobs.m – request observer coordinates

This function interactively requests the geodetic coordinates of a ground site. Please note that geodetic latitude and longitude are returned in the units of radians.

The syntax of this MATLAB function is as follows:

```
function [obslat, obslong, obsalt] = getobs

% interactive request of ground site coordinates

% output

% obslat = geographic latitude (radians)
% obslong = geographic longitude (radians)
% obsalt = geodetic altitude (kilometers)
```

The following is a typical user interaction with this function. Please note the sign conventions and range of valid inputs. North latitudes are positive and south latitudes are negative. East longitude is positive and west longitude is negative.

```
please input the geographic latitude of the ground site
(-90 <= degrees <= +90, 0 <= minutes <= 60, 0 <= seconds <= 60)
(north latitude is positive, south latitude is negative)
? 40,35,10

please input the geographic longitude of the ground site
(0 <= degrees <= 360, 0 <= minutes <= 60, 0 <= seconds <= 60)
(east longitude is positive, west longitude is negative)
? -105,30,45

please input the altitude of the ground site (meters)
(positive above sea level, negative below sea level)
? 100
```

getoe.m – request classical orbital elements

This function interactively requests six classical orbital elements. Please note that all angular elements are returned from this function in the units of radians. The user can control which orbital elements are requested by setting the corresponding value of the `ioev` vector. For example, the following statement will request all six orbital elements:

```
oev = getoe([1;1;1;1;1;1])
```

If the orbital eccentricity input by the user is 0, this function will bypass the prompt for argument of perigee (if requested via `ioev`) and set its value to 0.

The syntax of this MATLAB function is as follows:

```
function oev = getoe(ioev)
```

Orbital Mechanics with MATLAB

```
% interactive request of classical orbital elements
% NOTE: all angular elements are returned in radians
% input
%   ioev = request array (1 = yes, 0 = no)
% output
%   oev(1) = semimajor
%   oev(2) = orbital eccentricity
%   oev(3) = orbital inclination
%   oev(4) = argument of perigee
%   oev(5) = right ascension of the ascending node
%   oev(6) = true anomaly
```

getsv.m – request position and velocity vectors

This function interactively asks the user for the three rectangular components of both a position and velocity vector.

The syntax of this MATLAB function is as follows:

```
function [r, v] = getsv
% interactive request of state vector
% output
%   r = position vector (kilometers)
%   v = velocity vector (kilometers/second)
```

The following is a typical user interaction with this function.

```
please input the position vector x-component (kilometers)
? 8000

please input the position vector y-component (kilometers)
? 0

please input the position vector z-component (kilometers)
? 0

please input the velocity vector x-component (km/sec)
? 7.567

please input the velocity vector y-component (km/sec)
? 0

please input the position vector z-component (km/sec)
? 0
```

gettime.m – request universal time

This function interactively requests the universal time in hours, minutes and seconds.

The syntax of this MATLAB function is as follows:

```
function [uthr, utmin, utsec] = gettime  
  
% interactive request and input of universal time  
  
% output  
  
% uthr = universal time (hours)  
% utmin = universal time (minutes)  
% utsec = universal time (seconds)
```

The following is the screen prompt displayed by this function. Please note the range of valid inputs. If invalid data is input the function will redisplay this request.

```
please input the universal time  
(0 <= hours <= 24, 0 <= minutes <= 60, 0 <= seconds <= 60)
```