Appendix C

MatLab

This appendix is intended to help you understand numerical integration and to put it into practice using Matlab's `ode45` function.

C.1 Integration

We want to *integrate* ordinary differential equations (ODEs) of the form

$$\dot{x} = f(x,t), \quad \text{with initial conditions } \; x(0) = x_0 \tag{C.1}$$

where the *state* x may be a scalar or an $N \times 1$ matrix, and t is a scalar. This problem is known as an *initial value problem*, the variable t is the *independent variable* (usually time), and the state x is the *dependent variable*. The result of integration is to find the solution $x(t)$ that satisfies the differential equation and initial conditions given in Eq. (C.1).

The function $f(x,t)$ is frequently referred to as the *right-hand side* of the ODE, and most numerical algorithms involve repetitive calculation of the right-hand side for many values of x and t. Since $f(x,t)$ is the rate of change of the dependent variable x with respect to the independent variable t, we can think of the right-hand side as the *slope* of $x(t)$, even if x is a multi-dimensional state vector.

C.2 Some Example Problems of the Form of Eq. (C.1)

C.2.1 A simple first-order ODE

One of the simplest possible examples is

$$\dot{x} = a, \quad \text{with initial conditions } \; x(0) = b \tag{C.2}$$

where a and b are constants, and x is a scalar. This equation is easily integrated *analytically* to obtain

$$x(t) = at + b \tag{C.3}$$

which defines a straight line in the (t, x) plane, with slope a and intercept b. Notice how the right-hand side is in fact the slope of $x(t)$, as mentioned in the previous section.

C.2.2 A slightly less simple first-order ODE

Instead of a constant right-hand side, let the right-hand side have a t-dependence:

$$\dot{x} = at, \quad \text{with initial conditions} \;\; x(0) = b \tag{C.4}$$

where a and b are constants, and x is a scalar. This equation is also easily integrated *analytically* to obtain

$$x(t) = \frac{1}{2}at^2 + b \tag{C.5}$$

which defines a parabola in the (t, x) plane, with slope at any given point given by at and intercept b. Notice how the right-hand side is again the slope of $x(t)$, which is changing as a function of t.

C.2.3 A more interesting, but still simple first-order ODE

Instead of a t-dependence, let the right-hand side depend on the dependent variable:

$$\dot{x} = ax, \quad \text{with initial conditions} \;\; x(0) = b \tag{C.6}$$

where a and b are constants, and x is a scalar. This equation is also easily integrated *analytically* by using separation of variables (exercise!) to obtain

$$x(t) = be^{at} \tag{C.7}$$

which defines an exponential curve in the (t, x) plane, with slope at any given point given by $abe^{at} = ax(t)$ and intercept b. Notice how the right-hand side is still the slope of $x(t)$, which is again changing as a function of t.

Remark 1 *All of the preceding examples are of the class known as linear first-order ordinary differential equations. The two examples where $f(x, t)$ does not have any t-dependence are also known as constant-coefficient or time-invariant differential equations. The example where $f(x, t)$ does have time-dependence is known as a time-varying differential equation.*

C.2.4 Some nonlinear first-order ODEs

If the right-hand side has a nonlinear dependence on the dependent variable, then the equation is a nonlinear differential equation. Here are some examples:

$$\dot{x} = ax^2 \tag{C.8}$$

$$\dot{x} = a \sin x \tag{C.9}$$
$$\dot{x} = a e^{x} \tag{C.10}$$
$$\dot{x} = a \log x \tag{C.11}$$

All of these examples can be integrated *analytically* to obtain the solution $x(t)$, because the equations are all *separable.* As an exercise, you should separate the equations and carry out the integration.

C.2.5 Second-order ODEs

Many problems of interest to engineers arise in the form of second-order ODEs, because many problems are developed by using Newton's second law,

$$f = ma \quad e.g. \quad f = m\ddot{x} \tag{C.12}$$

where f is the force acting on an object of mass m, and a is the acceleration of the object, which might be for example the second derivative of a position described by the cartesian coordinate, x.

A familiar example is the mass-spring-damper system, where a mass m has position described by a scalar variable x and is acted on by a spring force $f_s = -kx$, by a damper force $f_d = -c\dot{x}$, and by a control force f_c. Application of $f = ma$ to this problem leads to the second-order ODE

$$m\ddot{x} + c\dot{x} + kx = f_c \tag{C.13}$$

Assuming that the *parameters* m, c, and k are all constant, then this equation is a second-order, constant-coefficient (time-invariant), linear ODE. Furthermore, if the control force f_c is zero, then the equation is called a homogeneous ODE.

As with the simple first-order examples already presented, this equation can be solved analytically. You should review the analytical solution to this equation in your vibrations and ODE textbooks. One can also put this second-order ODE into the first-order form of a *system* of first-order ODEs as follows. First define two *states* x_1 and x_2 as the position and velocity; *i.e.*

$$x_1 = x \tag{C.14}$$
$$x_2 = \dot{x} \tag{C.15}$$

Differentiating these two states leads to

$$\dot{x}_1 = \dot{x} = x_2 \tag{C.16}$$
$$\dot{x}_2 = \ddot{x} = \frac{f_c}{m} - \frac{k}{m}x - \frac{c}{m}\dot{x} = u - \hat{k}x_1 - \hat{c}x_2 \tag{C.17}$$

where we have introduced three new symbols: $u = f_c/m$ is the acceleration due to the control force, and $\hat{k}$ and $\hat{c}$ are the spring and damping coefficients divided by the mass. These equations may be written in matrix form as

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\hat{k} & -\hat{c} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \tag{C.18}$$

This form of the equation is so common throughout the dynamics and control literature that it has a standard notation. If we denote the 2×1 state vector by $\mathbf{x} = [x_1 \;\; x_2]^T$, the 2×2 matrix that appears in the equation by $\mathbf{A}$, and the 2×1 matrix that multiplies u by $\mathbf{B}$, then we can write the system as

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}u \tag{C.19}$$

In this form of linear, constant-coefficient ODEs, the variable matrix $\mathbf{x}$ is known as the *state vector*, and the variable u is known as the *control.* The matrices $\mathbf{A}$ and $\mathbf{B}$ are the *plant* and *input* matrices, respectively.

Since the input must either be constant, a function of time, a function of the state, or a function of both the time and the state, then this system of equations may be written in the even more general form of

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t) \tag{C.20}$$

Note the similarity between this ODE and Eq. (C.1). The only difference is that we have introduced the convention of using a bold-face font to denote that a particular variable or parameter is a matrix.

Also note that the right-hand side is the time-rate of change of the state, which can be interpreted as the slope of the state as a function of time.

In general, most ordinary differential equations can be written in the form of Eq. (C.20). In the next subsection we give some higher-order examples.

C.2.6 Higher-order ODEs

The rotational equations of motion for a rotating rigid body may be written as the combination of Euler's equations and a set of kinematics equations. A typical choice for kinematics variables is the quaternion. These equations are:

$$\dot{\omega} = -\mathbf{I}^{-1}\left[\omega^{\times}\mathbf{I}\omega + \mathbf{g}\right] \tag{C.21}$$

$$\dot{\bar{\mathbf{q}}} = \mathbf{Q}(\bar{\mathbf{q}})\omega \tag{C.22}$$

Note that the right-hand side of the $\dot{\omega}$ equation depends on ω and the torque $\mathbf{g}$, which (like the control in Eq. (C.19)) may depend on the angular velocity ω, on the attitude

$\bar{\mathbf{q}}$, and on the time t. The right-hand side of the $\dot{\bar{\mathbf{q}}}$ equation depends on $\bar{\mathbf{q}}$ and ω. Thus the equations are coupled. The combination of ω and $\bar{\mathbf{q}}$ forms the state; *i.e.*

$$\mathbf{x} = \begin{bmatrix} \omega \\ \bar{\mathbf{q}} \end{bmatrix} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} \tag{C.23}$$

Thus the equations can be combined into a single state vector differential equation (or system of equations):

$$\begin{bmatrix} \dot{\omega} \\ \dot{\bar{\mathbf{q}}} \end{bmatrix} = \begin{bmatrix} -\mathbf{I}^{-1}\left[\omega^{\times}\mathbf{I}\omega + \mathbf{g}\right] \\ \mathbf{Q}(\bar{\mathbf{q}})\omega \end{bmatrix} \tag{C.24}$$

which may be written in the more concise form of

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t) \tag{C.25}$$

As noted above, most dynamics and control problems can be written in this form, where the right-hand side corresponds to the rate of change of the state and depends on both the state and time.

C.3 Analytical *vs.*Numerical Solutions

We use the term *analytical* to refer to a solution that can be written in closed form in terms of known function such as trigonometric functions, hyperbolic functions, elliptic functions, and so forth. Most dynamics and control problems do not admit analytical solutions, and so the solution $\mathbf{x}(t)$ corresponding to a given set of initial conditions must be found numerically using a numerical integration algorithm. A numerical integration algorithm can be tested by using it to integrate a system of equations that has a known analytical solution and comparing the results.

C.4 Numerical Integration Algorithms

There are many popular numerical integration algorithms appropriate for solving the initial value problems of the form

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t) \quad \text{with initial conditions} \quad \mathbf{x}(0) = \mathbf{x}_0 \tag{C.26}$$

The simplest algorithm of any practical use is known as Euler integration. A slightly more complicated algorithm with substantially better performance is the Runge-Kutta algorithm. Higher-order (more accurate, but more complicated) algorithms are widely available, and usually one just selects an existing implementation of a suitable algorithm and uses it for the problem of interest. In most cases, the algorithm needs to know the initial and final times (t_0 is usually 0, and t_f), the initial conditions ($\mathbf{x}_0$), and how to calculate the right-hand side or slope $\mathbf{f}(\mathbf{x}, t)$ for given values of the state and time.

C.4.1 Euler integration

Euler integration is based directly on the notion that the right-hand side defines the slope of the function $\mathbf{x}(t)$. Since we know the initial conditions, $\mathbf{x}_0$, the initial slope is simply $\mathbf{f}(\mathbf{x}_0, t_0)$, and one might reasonably suppose that the slope is nearly constant for a short "enough" timespan following the initial time. Thus, one chooses a small "enough" time step, Δt, and estimates the state at time $t_0 + \Delta t$ as

$$\mathbf{x}_1 = \mathbf{x}(t_0 + \Delta t) \approx \mathbf{x}_0 + \mathbf{f}(\mathbf{x}_0, t_0)\Delta t \tag{C.27}$$

Continuing in this manner, one obtains a sequence of approximations at increasing times as

$$\begin{aligned}
\mathbf{x}_1 &= \mathbf{x}_0 + \mathbf{f}(\mathbf{x}_0, t_0)\Delta t && \text{(C.28)}\\
\mathbf{x}_2 &= \mathbf{x}_1 + \mathbf{f}(\mathbf{x}_1, t_1)\Delta t && \text{(C.29)}\\
\mathbf{x}_3 &= \mathbf{x}_2 + \mathbf{f}(\mathbf{x}_2, t_2)\Delta t && \text{(C.30)}\\
&\cdot && \text{(C.31)}\\
&\cdot && \text{(C.32)}\\
&\cdot && \text{(C.33)}\\
\mathbf{x}_n &= \mathbf{x}_{n-1} + \mathbf{f}(\mathbf{x}_{n-1}, t_{n-1})\Delta t && \text{(C.34)}
\end{aligned}$$

where $t_j = t_{j-1} + \Delta t$. This method is easy to understand, and easy to implement; however it is not accurate enough for integrating over any reasonably long time intervals. It is accurate enough to use in short time integrations though.

C.4.2 Runge-Kutta integration

C.5 Numerical Integration Using Matlab

Matlab has several numerical integration algorithms implemented as `.m` files. The most commonly used of these algorithms is `ode45`, and some examples using this function are provided below. In most implementations, the use of the integration function requires the development of two Matlab `.m` files: a *driver* file and a *right-hand side* file. For the examples here, we use the filenames `driver.m` and `rhs.m`. The driver file is a script file, meaning that it contains a sequence of Matlab statements that are executed in order. The rhs file is a function file, meaning that it defines a function with arguments (generally t and $\mathbf{x}$) that returns the value of the right-hand side (or slope) at the given values of time and state.

The driver file is responsible for defining parameters and initial conditions, calling the integration function, and then completing any post-processing of the results, such as making plots of the state variables as functions of time.

First we present a simple implementation of the Runge-Kutta algorithm.

C.5.1 An implementation of Runge-Kutta

Here is a sample integration algorithm that works similarly to the `ode45` algorithm, but is simpler. It implements the standard Runge-Kutta algorithm. It is a Matlab function, and has 3 arguments: `frhs` is a string containing the name of the `.m` file that computes the right-hand side of the differential equations; `tspan` is a $1 \times N$ matrix of the times at which the algorithm will sample the right-hand side $(t_0, t_0 + \Delta t, ...t_f)$; and `x0` is the initial state vector.

```
% oderk.m
% C. Hall
% [t, x] = oderk(frhs,tspan,x0)
% implements runge-kutta 4th order algorithm
% frhs = 'filename' where filename.m is m-file
% tspan = 1xN matrix of t values
% x0 = nx1 matrix of initial conditions
% returns t=tspan, x=nxN matrix
%                  x(:,i) = x(t(i))
function [tspan,x] = oderk(frhs,tspan,x0)
    N=length(tspan);
    n=length(x0);
    x0=reshape(x0,n,1);
    x=[x0 zeros(n,N-1)];
    w=x0;
    for i=1:N-1
        h=tspan(i+1)-tspan(i);
        t=tspan(i);
        K1=h*feval(frhs,t,w);
        K2=h*feval(frhs,t+h/2,w+K1/2);
        K3=h*feval(frhs,t+h/2,w+K2/2);
        K4=h*feval(frhs,t+h,w+K3);
        w=w+(K1+2*K2+2*K3+K4)/6;
        x(:,i+1)=w;
    end
    x=x';
```

As should be evident from the function, it divides each time interval into 2 segments and computes the right-hand side (slope) at the initial, final, and midpoint times of each interval. You should make an effort to know how these algorithms work, but you can use them as *black boxes* just as you might use an eigenvalue solver. That is, you can think of `oderk` as a function that takes its three arguments and returns the state vector calculated at each of the times in the `tspan` argument.

Here is an example illustrating the integration of a single first-order equation, $\dot{x} = ax$, $x(0) = x_0$, from $t = 0$ to $t = 10$ with a Δt of 0.1 s. The driver file is presented first.

```
% driver.m
% this file sets the parameters and initial conditions,
% calls oderk, and makes a plot of x(t)
global a                     % make the parameter a global
a     = 0.1;                 % set the parameter a
x0    = 1.0;                 % initial condition of state
t     = 0:0.1:10;            % [0 0.1 0.2 ... 10]
fname = 'rhs';               % the name of the rhs file (.m is assumed)
[t,x] = oderk(fname,t,x0);   % turn the work over to oderk
plot(t,x)                    % make the plot
xlabel('t');                 % add some labels
ylabel('x');
title('Sample Runge-Kutta Solution of First-Order ODE');
```

The right-hand side file is quite simple.

```
% rhs.m
% xdot = rhs(t,x)
function xdot = rhs(t,x)
global a
xdot = a*x;
% that's all!
```

The same integrator (`oderk`) can be used to integrate any system of ODEs. For example, suppose we have the system of three first-order equations:

$$\dot{x} = \sigma(y - x) \tag{C.35}$$

$$\dot{y} = -xz + rx - y \tag{C.36}$$

$$\dot{z} = xy - bz \tag{C.37}$$

These equations are called the Lorenz equations and are considered to be the first widely known system of equations admitting chaotic solutions*. The behavior of solutions to these equations is strongly dependent on the values of the three parameters b,

*For more information on chaotic systems, see Gleick's book *Chaos*, Moon's *Chaotic and Fractal Dynamics*, or Tabor's *Chaos and Integrability in Nonlinear Dynamics*.

σ, and r. An interesting chaotic attractor (also known as a *strange attractor*) occurs for the parameter values $b = 8/3$, $\sigma = 10$, $r = 28$.

Here are driver and rhs files to implement these equations and compute the strange attractor. Try it out.

```
% lorenzdriver.m
% this file sets the parameters and initial conditions,
% calls oderk, and makes plots
global b sig r              % make the parameters global
b     = 8/3;                % set the parameters
sig   = 10;
r     = 28;
x0    = [0; 1; 0];          % initial conditions of state vector
t     = 0:0.01:50;           % [0 0.01 0.02 ... 50]
fname = 'lorenzrhs';        % the name of the rhs file (.m is assumed)
[t,x] = oderk(fname,t,x0);  % turn the work over to oderk
plot3(x(:,1),x(:,2),x(:,3));% make a 3-d plot  (cool!)
xlabel('x');                % add some labels
ylabel('y');
zlabel('z');
title('Sample Runge-Kutta Solution of Lorenz System');
```

The right-hand side file is quite simple.

```
% lorenzrhs.m
% xdot = rhs(t,x)
function xdot = rhs(t,x)
global b sig r
xx = x(1);                   % the x component
y = x(2);                    % and y ...
z = x(3);
xdot = [ sig*(y-xx);  -xx*z+r*xx-y;  xx*y-b*z];
% that's all!
```

C.5.2 Using `ode45`

The calling format for using `ode45` is identical to that for calling `oderk`. However, there are numerous additional arguments that one can use in calling `ode45` that I have not included in my implementation of `oderk`. You can find out what the options are by typing `help ode45` at the Matlab prompt. Probably the most useful of these additional arguments are the `options` and `parameters` arguments. For example, instead of making the parameters in the Lorenz equations example `global` variables, we can pass them to the right-hand side file as parameters. And we can also set the desired precision we want from the integration. The following driver and rhs files implement these two options.

```
% lorenzdriverp.m
% this file sets the parameters and initial conditions,
% calls ode45, and makes plots

b       = 8/3;                % set the parameters
sig     = 10;
r       = 28;
x0      = [0; 1; 0];          % initial conditions of state vector
t       = 0:0.01:50;          % [0 0.01 0.02 ... 50]
fname   = 'lorenzrhs';        % the name of the rhs file (.m is assumed)
options = odeset('abstol',1e-9,'reltol',1e-9); % set tolerances
[t,x]   = ode45(fname,t,x0,options,b,sig,r);
plot3(x(:,1),x(:,2),x(:,3));  % make the plot
xlabel('x');                  % add some labels
ylabel('y');
zlabel('z');
title('Sample ode45 Solution of Lorenz System');

% lorenzrhsp.m
% xdot = rhs(t,x,b,sig,r)
function xdot = rhs(t,x,b,sig,r)
xx = x(1);                   % the x component
y  = x(2);                   % and y ...
z  = x(3);
xdot = [ sig*(y-xx);  -xx*z+r*xx-y;  xx*y-b*z];
% that's all!
```

C.6 References and further reading

Bibliography

C.7 Exercises

Get into MatLab by typing matlab at the DOS or Unix prompt, or by launching the MatLab application if you're using Windows or a Mac.

1. Compute the following quantities:

$\sin^2(\pi/4)+\cos^2(\pi/4)$	by typing `sin(pi/4)^2+cos(pi/4)^2`
$e^{i\pi}+1$	by typing `exp(sqrt(-1)*pi)+1`
$\ln(e^3), log_{10}(e^3), log_{10}(10^6)$	
$(1+3i)/(1-3i)$	can you check this result by hand?
$x=32\pi, y=\cosh^2 x-\sinh^2 x$	
`exp(pi/2*i)-exp(pi/2i)`	*why isn't this zero?*

2. Create some vectors and matrices:

```
u = [1 2 3 4 5 6 7 8 9]                 (a row vector)
v = 1:9                                 (a row vector)
w = rand(1,9)                           (a random row vector)
wp = w'                                 (a column vector)
M = [sin(u)' cos(v)' tan(w') atan(wp)]  (a 9-by-4 matrix)
M4 = M(1:4,1:4)                         (1st 4 rows and columns of M)
```

Can you multiply **M** and either of the vectors? Remember that to multiply two matrices **A** and **B**, where **A** has m rows and n columns, and **B** has p rows and q columns, then $n=p$ is required.

3. Make some simple plots

$y=\sin x, x\in[0,2\pi]$	`x = 0:2*pi; y = sin(x); plot(x,y)`
This might look a bit funny, so try making the "step" smaller	
	`x = 0:0.1: 2*pi; y = sin(x); plot(x,y)`
You can make it even smoother by using	
	`x = 0:0.01: 2*pi; y = sin(x); plot(x,y)`
Add some labels	`xlabel('This is the x label')`
	`ylabel('This is the y label')`
And a title	`title('This is the title')`
And a legend	`legend('This is the legend')`

Now, make some plots of your own.

4. Make a more complicated plot

$y=e^{-0.4x}\sin x, x\in[0,2\pi]$
```
x = linspace(0,2*pi,100);
y = exp(-0.4*x).*sin(x); plot(x,y)
```

Label the axes, create a nifty title and legend. Type `help plot` and use the information you get to make another plot of just the unconnected data points. Type `help linspace` to find out what that function does.

5. Make some `log` and `semilog` plots of $y=\ln x$, $y=\log_{10} x$, $y=x^2$, and $y=x^3$. The commands to use are `semilogx`, `semilogy`, and `loglog`. Use help to find out how to use them. To create a vector of x^3, type `y = x.^3`.

6. Make a polar plot of the spiral $r = e^{0.2\theta}, \theta \in [0, 5\pi]$. Use the command `polar(theta,r)`.

7. Make a 3-dimensional plot of the helix $x = \sin t$, $y = \cos t$, $z = t$, for $t \in [0, 6\pi]$. Create the vectors t, x, y, and z, then use the command `plot3(x,y,z)`. Add some labels, and check out the `print` command.

8. Create a menu that gives a choice between two options. Use the command `choice = menu('MenuName', 'ChoiceA', 'ChoiceB')`. If you click `ChoiceA`, then the variable choice will be assigned the value 1, and if you click `ChoiceB`, the variable choice will be assigned the value 2. Menus are most useful when used in `MatLab ".m"` files.

9. Create some more matrices

```
I  = eye(5)                  a 5-by-5 identity matrix
d  = diag(I)                 a 5-by-1 matrix of the diagonal elements
A  = rand(5,4)               a 5-by-4 matrix with random elements
d1 = diag(A,1)               the "super-diagonal" elements
d2 = diag(A,-1)              the "sub-diagonal" elements
u  = linspace(pi,8*pi,100)   a linearly spaced vector of length 100,
                             starting at pi and going to 8*pi
v  = logspace(pi,8*pi,100)   a logarithmically spaced vector of
                             length 100, from pi to 8*pi
```

10. Do some logical operations

```
x = 1; y = 2;
if ( x > y )
    disp('x > y')
else
    disp('y >= x')
end
```

Some vector-based examples:

```
x = 1:5;                 two vectors
y = [-1 2 -2 3 4];
m = x > y                0 for each false, 1 for each true
n = x > 4                ditto
p = m & n                logical AND
q = m | n                logical OR
```

11. Writing M-files

    For serious `MatLab` use, you need to write M-files, i.e., files with a `.m` extension, containing `MatLab` statements. To create and edit files, you need to use a text editor. Once you've written an M-file, you will want to run it. For example, to execute the `MatLab` code in `myprob.m`, you type `myprob` at the ¿¿ prompt. You can also write your own `MatLab` functions, which also live in M-files. The main difference between a function M-file and an ordinary M-file, is that a function M-file must begin with the statement `function result = fname(arguments)` where `result` is the variable(s) for the function to return, `fname` is the name of the function, and `arguments` is a list of comma-separated variables that the function is to use. If you put comment statements before the function statement, they will be printed if you type `help fname`. Here are some examples:

```
% this function returns sin(x)*sin(x)
% put this function in a file named
%   sin2.m
% to call this function use the statement
%   y = sin2(x)
%
function y = sin2(x)
    s = sin(x);
    y = s*s;
```

```
% This function has as its argument a number s.
% It first takes the absolute value of s,
% then creates a t vector with 101 elements ranging
% from 0 to s, and evenly spaced. It returns t and sin(t)
% Put this function in a file named
%   myf.m
% to call this function use the statement
%   [myx,myt] = myf(mys)
%
function [x,t] = myf(s)
    s = abs(s);
    t = 0:s/100:s;
    x = sin(t);
```

```
% This is a more advanced function using the feval feature.
% This function finds a root of a mathematical function
% using the Bisection Method.
% Put this function in a file named
%   bisect.m
% to call this function use the statement
%   [root, steps] = bisect('function',guess,tolerance)
%
function [root,steps] = bisect(f,x,tol)
    if nargin < 3, tol = eps; end
    trace = (nargout==2);
    if x ~= 0, dx = x/20; else, dx=1/20; end
    a = x-dx;  fa = feval(f,a);
    b = x+dx;  fb = feval(f,b);

    % find change of sign
    while (fa > 0) == (fb > 0)
        dx = 2.0*dx;
        a = x-dx; fa = feval(f,a);
        if (fa > 0) ~= (fb > 0), break, end
        b = x + dx; fb = feval(f,b);
    end
    if trace, steps = [a fa; b fb]; end

    % Main Loop
    while abs(b-a) > 2.0*tol*max(abs(b),1.0)
        c = a + 0.5*(b-a);  fc = feval(f,c);
        if trace, steps = [steps; [c fc]]; end
        if (fb > 0) == (fc > 0)
            b = c; fb = fc;
        else
            a = c; fa = fc;
        end
    end
```

C.8 Problems

1. Develop a simple "simulator" that implements the rotational equations of motion for a rigid body in a central gravitational field. Use quaternions for the kinematics equations. Your simulator should allow you to specify the initial conditions on ω and $\bar{\mathbf{q}}$, the principal moments of inertia, and starting and stopping times. You should be able to plot ω and $\bar{\mathbf{q}}$ as functions of time.

   Using your simulator, make two plots for a satellite in a circular, equatorial orbit with $a = 7000$ km: ω^{bo} *vs.t* and $\bar{\mathbf{q}}^{bo}$ *vs.t* for $t \in [0, 1000]$. Use principal moments of inertia $15, 20, 10$, and initial conditions

$$\omega^{bo} = (0.1, 0.2, 0.3)\text{rad/s and } \bar{\mathbf{q}}^{bo} = (0.0413, 0.0100, 0.0264, 0.9988)$$

You must turn in the following:

(a) A report describing the problem, the equations, your approach, your analysis, your simulator, and your results.

(b) The two requested plots.

(c) Similar plots for two different sets of initial conditions to be chosen by you. Give me the initial conditions so I can check them.

(d) A copy of the code that implements your simulator, with enough comments that I can tell how it works.