MATLAB Handout: Integrators in MATLAB

MATLAB has several pre-built integrators for our use. This handout will give you a basic idea of how to work them to get the results that you need in ASEN 5070.

What are Integrators?

Integrators are algorithms that approximate a trajectory in a system based on initial conditions and knowledge of the equations of motion for that system. The basic idea is that a trajectory may be constructed by integrating its derivative, hence the name integrator.

Integrators may be as accurate as desired given perfect precision, arbitrarily large expansions, and arbitrarily small time-steps. But there is a trade-off between an integrator’s performance and how long it takes to perform the integration.

In general, integrators use a Taylor expansion to approximate the dynamics in the system. Euler’s Method invokes a 1st-order expansion, which works swiftly but with generally poor accuracy. The midpoint method invokes a 2nd-order expansion, which approximates the dynamics much better but still with noticeable errors (i.e., the energy in the system tends to change when it shouldn’t). On the other hand, a very high order expansion, e.g., 8th-order, works very well at approximating the dynamics for a very long amount of time, but takes a long time to operate. The most popular compromise is the Runge-Kutta integrator, which invokes a 4th-order expansion. It is the workhorse of many disciplines, including astrodynamics.

Variable Time-Step Integrators

The dynamics in astrodynamical systems tend to include very interesting regions near massive bodies and large slowly-changing regions between massive bodies. An integrator requires much smaller time-steps near the massive bodies than elsewhere in order to accurately approximate the dynamics. But it would be computationally wasteful to use such small time-steps far away from the planets. By implementing a variable time-step integrator, the integrator will naturally decrease its time-steps near the planets in order to retain the computational accuracy, but it will also increase its stride away from the planets and save large amounts of computational time. Thus, one only needs to specify an overall tolerance and the integrator will work to keep that tolerance despite the stride-lengths that it uses.

MATLAB’s Integrators

<table>
<thead>
<tr>
<th>Solver</th>
<th>Problem Type</th>
<th>Order of Accuracy</th>
<th>When to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>ode45</td>
<td>Nonstiff</td>
<td>Medium</td>
<td>Most of the time. This should be the first solver you try.</td>
</tr>
<tr>
<td>ode23</td>
<td>Nonstiff</td>
<td>Low</td>
<td>For problems with crude error tolerances or for solving moderately stiff problems.</td>
</tr>
<tr>
<td>ode113</td>
<td>Nonstiff</td>
<td>Low to high</td>
<td>For problems with stringent error tolerances or for solving computationally intensive problems.</td>
</tr>
<tr>
<td>ode15s</td>
<td>Stiff</td>
<td>Low to medium</td>
<td>If ode45 is slow because the problem is stiff.</td>
</tr>
<tr>
<td>ode23s</td>
<td>Stiff</td>
<td>Low</td>
<td>If using crude error tolerances to solve stiff systems and the mass matrix is constant.</td>
</tr>
</tbody>
</table>
ode23t

Moderately Stiff

Low

For moderately stiff problems if you need a solution without numerical damping.

ode23tb

Stiff

Low

If using crude error tolerances to solve stiff systems.

ode45.m

MATLAB’s pre-built ode45.m integrator is generally the best option for any astrodynamical application. If more accuracy is required, we suggest doing a google search for an ode78 routine or something similar.

The basic command to call the ode45 integrator looks like this:

\[
\begin{align*}
[t, \text{State}] &= \text{ode45}(\text{'dState'}, \text{time}, \text{ICs}, \text{options});
\end{align*}
\]

The integrator takes a vector of initial conditions (either a column or row vector) and integrates it using the dynamics given in the 'dstate' function. It outputs the approximate trajectory at every time-point given in the vector time. The outputs include t, the vector of time values that ode45 did integrate to (in general the same vector as time), and State, which is an array containing the state at every point in time. Summarized:

Inputs:
- 'dState': The name of the file that contains the derivative functions (discussed in the next section);
- time: The vector of time values to integrate;
- ICs: The vector of initial conditions for the system (either row or column);
- options: An optional specification for the options given to the integrator (tolerance, etc.).

Outputs:
- t: The vector of times that the integrator did produce (usually the same as time);
- State: An array containing the values of the state at every time.

The 'dState' Function

The derivative function for most astrodynamical systems looks like this:

```matlab
function dRV = dState(time,RV)

x = RV(1);
y = RV(2);
z = RV(3);
vx = RV(4);
vy = RV(5);
vz = RV(6);

constants = (declare what constants you are using)

ax = Some function of x,y,z,vx,vy,vz,time,constants
ay = Some function of x,y,z,vx,vy,vz,time,constants
az = Some function of x,y,z,vx,vy,vz,time,constants

% The function will return the following variables in a vector
dRV = [vx; vy; vz; ax; ay; az];
```

ode45.m

MATLAB’s pre-built ode45.m integrator is generally the best option for any astrodynamical application. If more accuracy is required, we suggest doing a google search for an ode78 routine or something similar.

The basic command to call the ode45 integrator looks like this:

\[
\begin{align*}
[t, \text{State}] &= \text{ode45}(\text{'dState'}, \text{time}, \text{ICs}, \text{options});
\end{align*}
\]

The integrator takes a vector of initial conditions (either a column or row vector) and integrates it using the dynamics given in the 'dstate' function. It outputs the approximate trajectory at every time-point given in the vector time. The outputs include t, the vector of time values that ode45 did integrate to (in general the same vector as time), and State, which is an array containing the state at every point in time. Summarized:

Inputs:
- 'dState': The name of the file that contains the derivative functions (discussed in the next section);
- time: The vector of time values to integrate;
- ICs: The vector of initial conditions for the system (either row or column);
- options: An optional specification for the options given to the integrator (tolerance, etc.).

Outputs:
- t: The vector of times that the integrator did produce (usually the same as time);
- State: An array containing the values of the state at every time.

The 'dState' Function

The derivative function for most astrodynamical systems looks like this:

```matlab
function dRV = dState(time,RV)

x = RV(1);
y = RV(2);
z = RV(3);
vx = RV(4);
vy = RV(5);
vz = RV(6);

constants = (declare what constants you are using)

ax = Some function of x,y,z,vx,vy,vz,time,constants
ay = Some function of x,y,z,vx,vy,vz,time,constants
az = Some function of x,y,z,vx,vy,vz,time,constants

% The function will return the following variables in a vector
dRV = [vx; vy; vz; ax; ay; az];
```
That’s all! The function must be named ‘dState.m’ (or whatever, so long as the first line of the function reflects its name and the ode45 call has the right function name). It’s probably easiest to declare all the constants in the function. If the constants change over the trajectory, you may need to declare the constants global in both the main program and the derivative function. That way, they will change as needed.

**The options object**

There are various options that may be set in ode45. Type “help odeset” to see all of the available options. In general we only care about setting the relative and absolute tolerance for the system. To do that, use the following snippet of code:

```matlab
% Tolerance
tolerance = 1e-12;

% Setting up ODE45
options = odeset('RelTol',tolerance,'AbsTol',tolerance);
```

**An Example**

```matlab
...
% Set up the time vector
t0 = 0;
tf = 86400;   % 1 day later
dt = 20;      % 20-second intervals
time = [t0:dt:tf];

% Set up initial conditions
ICs = [x0,y0,z0,vx0,vy0,vz0];

% Tolerance
tolerance = 1e-12;

% Setting up ODE45
options = odeset('RelTol',tolerance,'AbsTol',tolerance);

% Integrating with a simple 2-body integrator
[time RV] = ode45('dR_2body', time, ICs, options);

% Integrating with 2-body plus the J2 effect
[time RV] = ode45('dR_2body_J2', time, ICs, options);

% Integrating with 2-body plus the J2 effect and drag
[time RV] = ode45('dR_2body_J2_drag', time, ICs, options);

% Recording all of the corresponding vectors for Position and Velocity
x = RV(:,1);
y = RV(:,2);
z = RV(:,3);
vx = RV(:,4);
vy = RV(:,5);
vz = RV(:,6);
```

...