University of Alberta
ENGM 541: Modeling and Simulation of Engineering Systems
Laboratory #5

Integration Methods with Higher-Order Truncation Errors with MATLAB

MATLAB is capable of a number of integration techniques to simulate the behaviour of dynamical systems, once they are cast into first-order form.

Solving First-Order ODEs in MATLAB is straightforward. The first step is to define a function to represent the equation (or equations) of the propagation system. Or example, if we want to integrate numerically the equation

$$\frac{dy}{dt} = \cos(t),$$

we write a MATLAB function for the equation:

```
function ydot = eq1(t,y)
ydot = cos(t) ;
```

To solve this ODE, MATLAB calls the function `ode23`, which integrates the differential equation using second- and third-order Runge-Kutta[1] method, using the following syntax:

```
[t,y] = ode23('function_name', [start_time end_time], y(0))
```

If we are interested in the behaviour of the system in the time interval $0 \leq t \leq \pi$, and our system has the initial condition $y(0) = 2$, then the function call will be:

```
>> [t,y] = ode23(@eq1,[0 pi],2);
```

The notation `@eq1` is a function handle to the function `eq1` that we defined above, which contains the expressions for the derivatives. Note that this method does not use a defined (fixed) step size, but rather attempts to meet a truncation error specification by modifying the step size. You can create a function that contains multiple derivatives. In that case, you need to define a vector for the set of derivatives, and you need a row vector in the call to the solver for the set of initial conditions. The Mathworks web site has a lot more information about how to use these solvers; here is an excerpt:

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$\begin{aligned} y'_1 &= y_2 y_3 & y_1(0) &= 0 \\ y'_2 &= -y_1 y_3 & y_2(0) &= 1 \\ y'_3 &= -0.51 y_1 y_2 & y_3(0) &= 1 \end{aligned}$$

To simulate this system, create a function `rigid` containing the equations

```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

[1] These methods were developed about 100 years ago by C. Runge and M.W. Kutta, two German mathematicians. Runge-Kutta is pronounced run-guh kut-tuh, in case you were wondering.

Relative Error

While the numerical solution $y(t)$ "looks" right, it would be good to know how accurate the numerical result actually is. The relative error ε is

$$\varepsilon = \left| \frac{f(t) - y(t)}{f(t)} \right|,$$

where $f(t)$ is the analytical solution. In our example, we can easily find the analytical solution to be

$$f(t) = \sin(t) + 2.$$

Exercise 1

For most ordinary differential equations, `ode45` will have higher accuracy than `ode23`, because `ode45` uses fourth-and fifth-order Runge-Kutta solvers. The syntax is essentially the same.

a) Write an m-file with a `for` loop that calculates the relative error at each step for the example above solved using `ode23`, and verify that the maximum error is 6.907×10^{-4}.
b) Solve the example system using `ode45`, calculate the relative errors, and find the maximum error.

State Space Representations in SIMULINK

Previously we tried out SIMULINK with simple linear systems, using icons to show the admissibility equations (such as the sum of forces at a node equaling zero) and employing integrators to transform a variable to its integral form, and then using the constitutive relationships to turn the variable into the other variable type to connect back to the admissibility equation. We also added other features for inputs (such as forces) and outputs (such as measurements of displacement at different points in time).

SIMULINK can use other types of blocks to incorporate more complex dynamic behaviours.

Recall the equation of motion for a linear mass-spring-damper system:

$$m\ddot{x} + b\dot{x} + kx = f(t)$$

which can be simulated in SIMULINK using simple elements as shown in Figure 1.

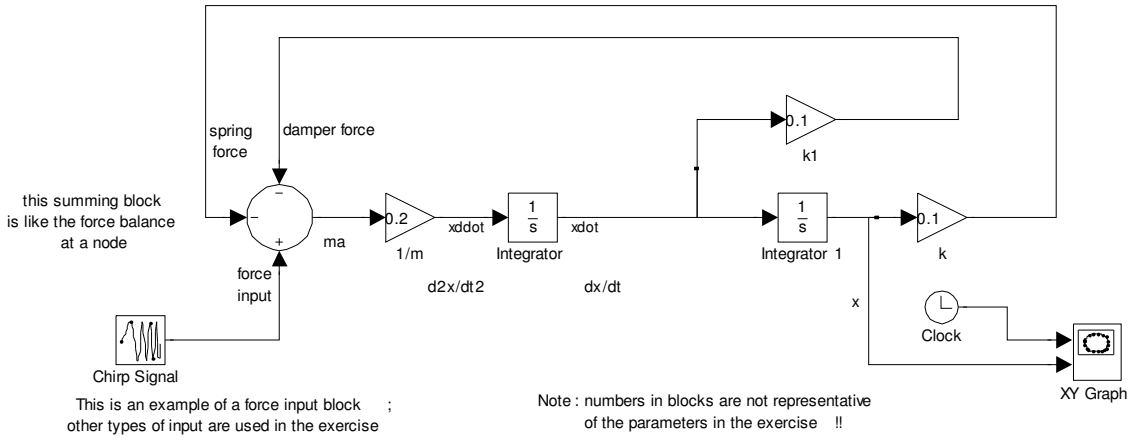

Figure 1

This linear system can also be expressed in state-space form as

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

where A is an n-by-n state matrix, with n as the number of state variables x, B is an n-by-l input matrix, where l is the number of inputs u; C is an p-by-n output matrix, where p is the number of outputs y; and D is a p-by-l direct-transmission matrix.

The state-space representation of the linear spring-mass-damper system can be written in first-order form as

$$\dot{x}_1 = x_2;$$

$$\dot{x}_2 = -\frac{b}{m}x_2 - \frac{k}{m}x_1 + \frac{f(t)}{m}$$

where $x_1 = x$. The input is $u = f(t)$. Let's assume that there is a linear potentiometer that measures the displacement of the mass, with a linear relationship for the output measurement y

$$y(t) = qx(t),$$

where q is a constant coefficient. With our definitions for x_1, x_{12}, u, and y, we can express this set of equations in state-space matrix form as

$$\begin{Bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{Bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{b}{m} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} + \begin{Bmatrix} 0 \\ \frac{1}{m} \end{Bmatrix} u;$$

$$y = \begin{bmatrix} q & 0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} + [0]u.$$

So we can build a SIMULINK block diagram of this system using a state space block, rather than having to put together a bunch of integrators and other elements.

The real advantage of a state-space representation is that it is not limited to linear systems. As we've seen when we write a set of governing equations of the form

$$\frac{dx_i}{dt} = f_i(x_1, x_2, ..., x_n, t), \quad i = 1, \ldots, n,$$

there is no restriction that the functions f_i all have to be linear functions of the state variables x_i and t. There are a number of different forms that state-space representations can take. The most common forms are described in the text. The matrix form is limited to linear functions.

Exercise 2

Build a SIMULINK model for the linear mass-spring-damper system using a state-space block and the following values for the parameters: $m = 1$, $k = 3$, $b = 4$, $q = 10$. Simulate the response of $x(t)$ for three different inputs $u = f(t)$:

- a unit step input,
- $f(t) = \sin(0.2t)$, and
- $f(t) = t$.

Use $x(0) = 0$ and $\dot{x}(0) = 0$ for the initial conditions, and run the simulation for a few seconds, displaying the output y.

Linearisation of a System

For the nonlinear mass-spring-damper system the equations of motion can be written

$$m\ddot{x} + b\dot{x} + f_K(x) = f(t)$$

where $f_K(x)$ represents the nonlinear spring force at displacement x. To linearise the system, we first have to find the operating point. We do this by replacing the forcing input $f(t)$ by its average value (or by an equilibrium value, if that has been specified), and x by $\bar{x}$:

$$m\ddot{\bar{x}} + b\dot{\bar{x}} + f_K(\bar{x}) = \bar{f}$$

Because $\bar{x}$ is a constant, that means that $\dot{\bar{x}} = \ddot{\bar{x}} = 0$. For this example, we assume that $\bar{f}$ has been specified to be zero, which means that the governing equation reduces to:

$$\bar{f} = f_K(0) = 0$$

The operating point for the spring is at $\bar{x}$=0, $\bar{f}_K = 0$. If the forcing input is nonzero at the operating point, then the element operating point will be different. The next step is to rewrite the linear terms in the governing equation in terms of the incremental variables for the state variables and input functions, which in this case are: $\hat{x} = x - \bar{x}$ and $\hat{f} = f - \bar{f}$, giving the governing equation in terms of the operating point and the incremental variables:

$$m(\ddot{\bar{x}} + \ddot{\hat{x}}) + b(\dot{\bar{x}} + \dot{\hat{x}}) + f_K(x) = \bar{f} + \hat{f}(t).$$

We'll deal with the nonlinear spring force term in a minute by linearising it, but first let's simplify the governing equation a bit by eliminating $\dot{\bar{x}}$ and $\ddot{\bar{x}}$, which are both zero terms:

$$m\ddot{\hat{x}} + b\dot{\hat{x}} + f_K(x) = \bar{f} + \hat{f}(t).$$

Now we use the Taylor series expansion of the nonlinear spring force

$$f_K(x) = f_K(0) + \left.\frac{df_K}{dt}\right|_{x=0} \hat{x} + \ldots$$
$$\approx f_K(0) + k(0)\hat{x}$$

where $k(0)$ is the slope at the operating point, and substitute the first two terms into the governing equation in terms of incremental variables to get the linearised form of the governing equation:

$$m\ddot{\hat{x}} + b\dot{\hat{x}} + f_K(0) + k(0)\hat{x} = \bar{f} + \hat{f}(t).$$

Since $\bar{f} = f_K(\bar{x}) = f_K(0) = 0$, the linearised model is finally written as

$$m\ddot{\hat{x}} + b\dot{\hat{x}} + f_K(0) + k(0)\hat{x} = \hat{f}(t),$$

which is a linear ODE in terms of the incremental variable $\hat{x}$ and the incremental input $\hat{f}(t)$. To solve this system, we need initial values for $\hat{x}(0)$ and $\dot{\hat{x}}(0)$, which we find from $x(0)$ and $\dot{x}(0)$:

$$\hat{x}(0) = x(0) - \bar{x}$$
$$\dot{\hat{x}}(0) = \dot{x}(0) - \dot{\bar{x}}$$

which for this example has $\bar{x} = \dot{\bar{x}} = 0$.

Once we have solved the linearised system model, we find the approximate solution for the nonlinear model by adding the operating point x to the incremental solution:

$$x(t) = \bar{x} + \hat{x}(t),$$

which is only an approximate solution of the nonlinear model.

To write this second-order system in state-variable form, we need an incremental velocity $\hat{v} = v - \bar{v}$, where $\bar{v} = 0$. With $\hat{v} = \dot{\hat{x}}$, we get the equations in first-order form as

$$\dot{\hat{x}} = \hat{v}$$
$$\dot{\hat{v}} = \frac{1}{m}\left[-k(0)\hat{x} - b\hat{v} + \hat{f}(t)\right]$$

And since $\bar{x} = \dot{\bar{x}} = 0$, the initial conditions are

$$\hat{x}(0) = 0$$
$$\hat{v}(0) = 0.$$

(This example came from Close & Frederick Chapter 9.)

References:

C.M. Close, C.E. Frederick. *Modeling and Analysis of Dynamic Systems, 2nd Ed.* Houghton-Mifflin 1993.
D. Etter, *Engineering Problem Solving with MATLAB.* Prentice-Hall 1997.
D. McMahon, *Matlab Demystified*. McGraw-Hill 2007.
www.mathworks.com