

4. NUMERICAL ANALYSIS

Topics:

- State variable form for differential equations
- Numerical integration with software and calculators
- Numerical integration theory: first-order, Taylor series and Runge-Kutta
- Using tabular data
- A design case

Objectives:

- To be able to solve systems of differential equations using numerical methods.

4.1 INTRODUCTION

For engineering analysis it is always preferable to develop explicit equations that include symbols, but this is not always practical. In cases where the equations are too costly to develop, numerical methods can be used. As their name suggests, numerical methods use numerical calculations (i.e., numbers not symbols) to develop a unique solution to a differential equation. The solution is often in the form of a set of numbers, or a graph. This can then be used to analyze a particular design case. The solution is often returned quickly so that trial and error design techniques may be used. But, without a symbolic equation the system can be harder to understand and manipulate.

This chapter focuses on techniques that can be used for numerically integrating systems of differential equations.

4.2 THE GENERAL METHOD

The general process of analyzing systems of differential equations involves first putting the equations into standard form, and then integrating these with one of a number of techniques. State variable equations are the most common standard equation form. In this form all of the equations are reduced to first-order differential equations. These first-order equations are then easily integrated to provide a solution for the system of equations.

4.2.1 State Variable Form

At any time a system can be said to have a state. Consider a car for example, the state of the car is described by its position and velocity. Factors that are useful when identifying state variables are:

- The variables should describe energy storing elements (potential or kinetic).
- The variables must be independent.
- They should fully describe the system elements.

After the state variables of a system have been identified, they can be used to write first-order state variable equations. The general form of state variable equations is shown in Figure 4.1. Notice that the state variable equation is linear, and the value of x is used to calculate the derivative. The output equation is not always required, but it can be used to calculate new output values.

$$\left(\frac{d}{dt}\right)x = Ax + Bu \quad \text{state variable equation}$$

$$y = Cx + Du \quad \text{output equation}$$

where,

x = state/output vector (variables such as position)

u = input vector (variables such as input forces)

A = transition matrix relating outputs/states

B = matrix relating inputs to outputs/states

y = non-state value that can be found directly (i.e. no integration)

C = transition matrix relating outputs/states

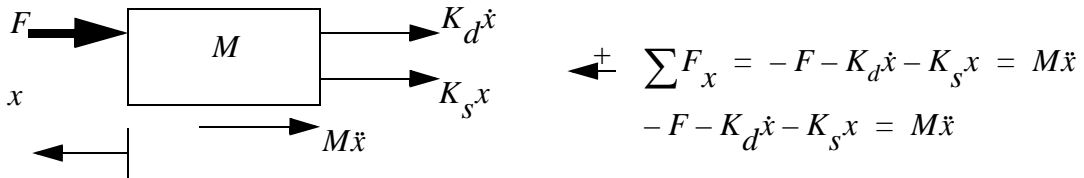
D = matrix relating inputs to outputs/states

Figure 4.1 The general state variable form

An example of a state variable equation is shown in Figure 4.2. As always, the FBD is used to develop the differential equation. The resulting differential equation is second-order, but this must be reduced to first-order. Using the velocity variable, 'v' the second-order differential equation can be reduced to a first-order equation. An equation is also required to define the velocity as the first derivative of the position, 'x'. In the example the two state equations are manipulated into a matrix form. This form can be useful, and may be required for determining a solution. For example, HP calculators require the matrix form, while TI calculators use the equation forms. Software such as Mathcad can use either form. The main disadvantage of the matrix form is that it will only work for lin-

ear differential equations.

Given the FBD shown below, the differential equation for the system is,



The equation is second-order, so two state variables will be needed. One obvious choice for a state variable in this equation is 'x'. The other choice can be the velocity, 'v'. Equation (1) defines the velocity variable. The velocity variable can then be substituted into the differential equation for the system to reduce it to first-order.

$$\dot{x} = v \tag{1}$$

$$M \ddot{x} = -F - K_d \dot{x} - K_s x$$

$$M \dot{v} = -F - K_d v - K_s x$$

$$\dot{v} = x \left(\frac{-K_s}{M} \right) + v \left(\frac{-K_d}{M} \right) + \left(\frac{-F}{M} \right) \tag{2}$$

Equations (1) and (2) can also be put into a matrix form similar to that given in Figure 4.1.

$$\frac{d}{dt} \begin{bmatrix} x \\ v \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{-K_s}{M} & \frac{-K_d}{M} \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{-F}{M} \end{bmatrix}$$

Note: To have a set of differential equations that is solvable, there must be the same number of state equations as variables. If there are too few equations, then an additional equation must be developed using an unexploited relationship. If there are too many equations, a redundancy or over constraint must be eliminated.

Figure 4.2 A state variable equation example

Put the equation into state variable and matrix form.

$$\sum F = F = M \left(\frac{d}{dt} \right)^2 x$$

$$\text{ans. } \dot{x} = v \quad \dot{v} = \frac{F}{M}$$

$$\frac{d}{dt} \begin{bmatrix} x \\ v \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{F}{M} \end{bmatrix}$$

Figure 4.3 Drill problem: Put the equation in state variable form

Consider the two cart problem in Figure 4.4. The carts are separated from each other and the wall by springs, and a force is applied to the left hand side. Free body diagrams are developed for each of the carts, and differential equations developed. For each cart a velocity state variable is created. The equations are then manipulated to convert the second-order differential equations to first-order state equations. The four resulting equations are then put into the state variable matrix form.

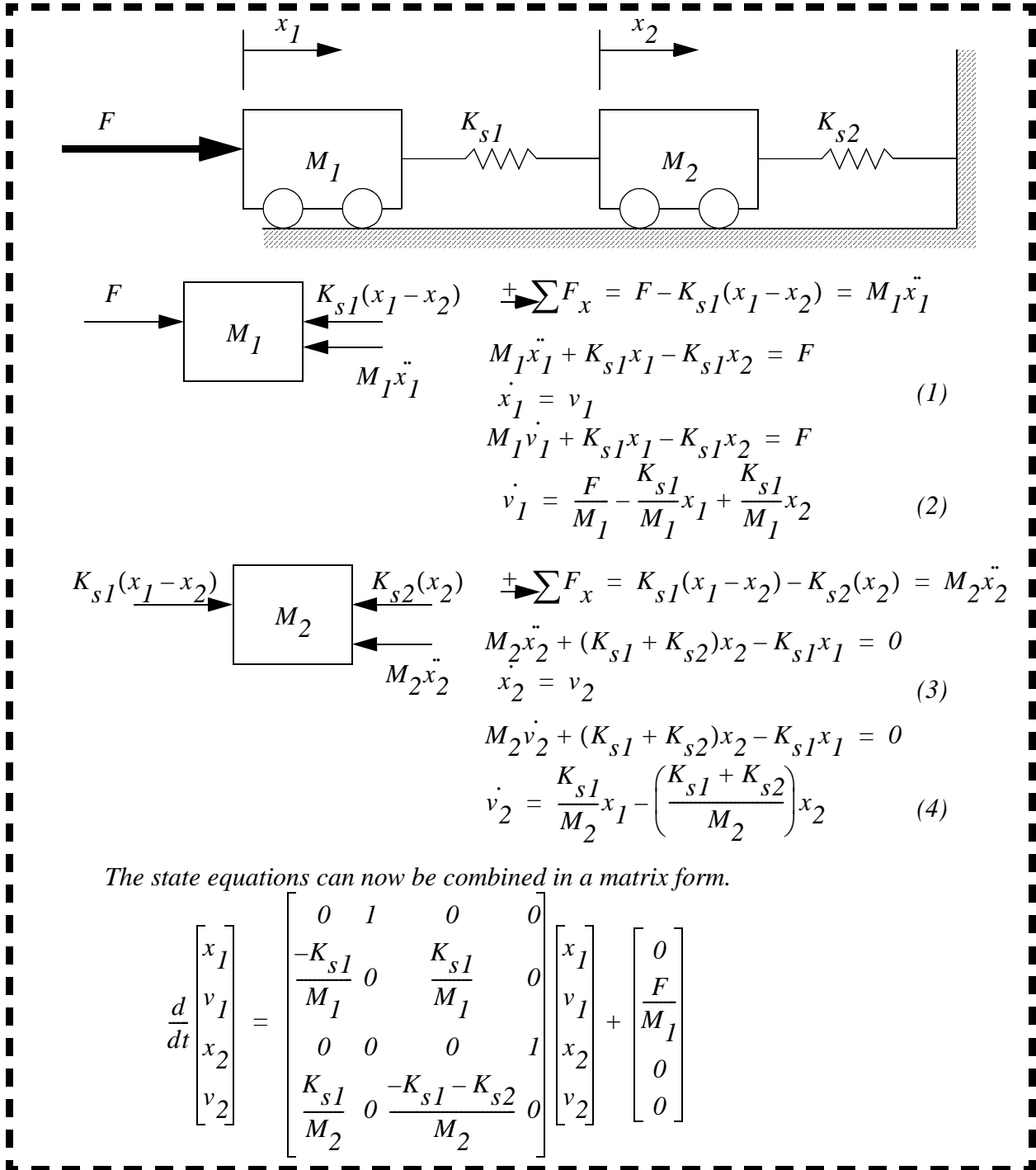


Figure 4.4 Two cart state equation example

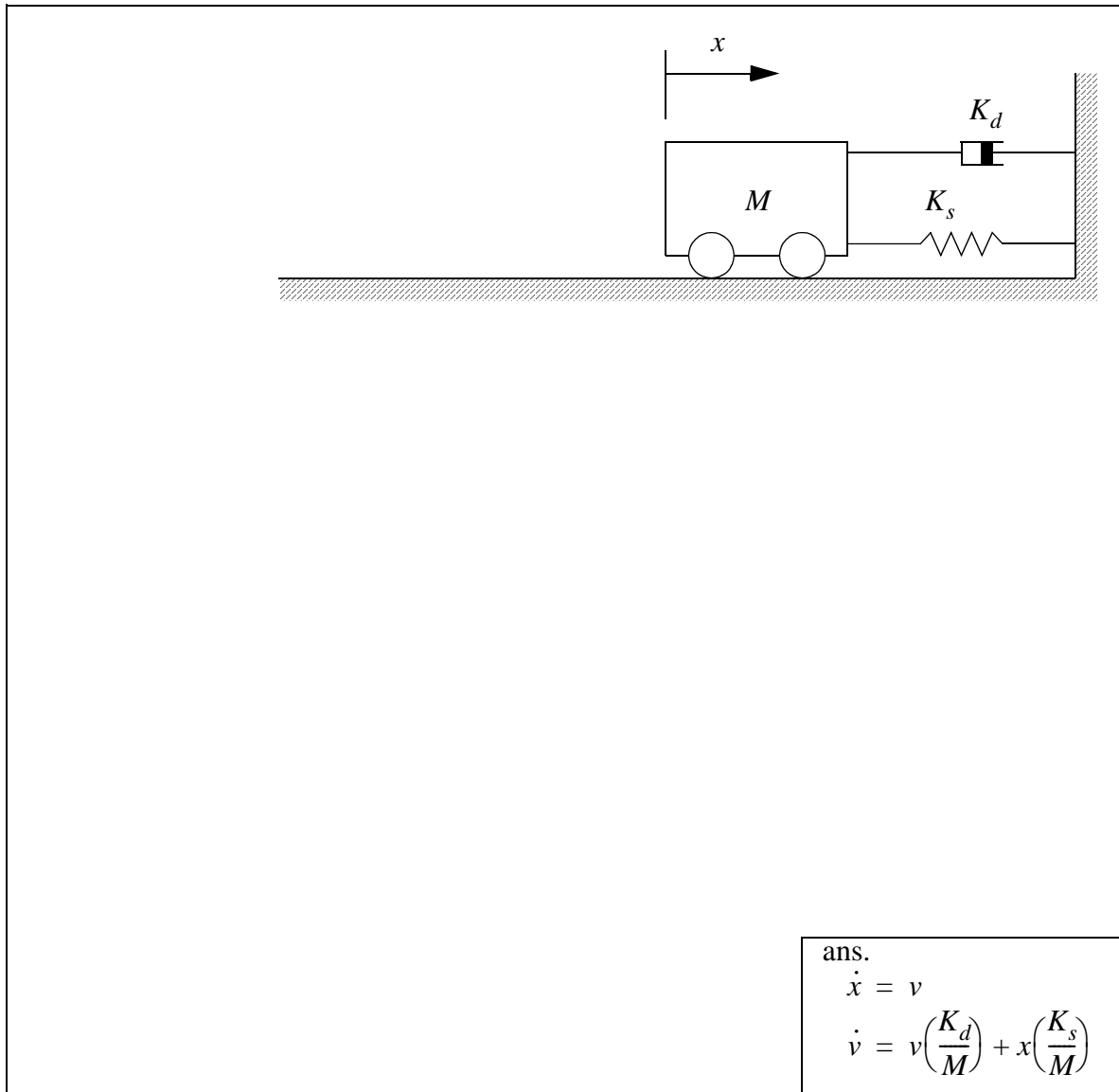
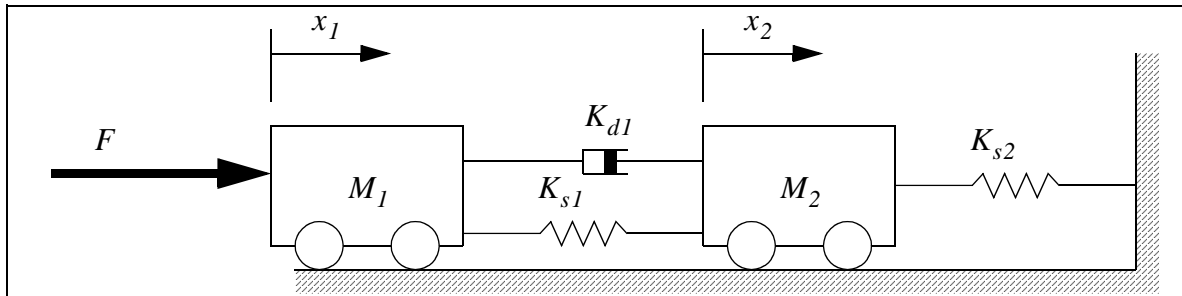


Figure 4.5 Drill problem: Develop the state equations in matrix form



ans. $\dot{x}_1 = v_1$
 $\dot{x}_2 = v_2$

$$\dot{v}_1 = v_1 \left(\frac{-K_{d1}}{M_1} \right) + x_1 \left(\frac{-K_{s1}}{M_1} \right) + v_2 \left(\frac{K_{d1}}{M_1} \right) + x_2 \left(\frac{K_{s1}}{M_1} \right) + \frac{F}{M_1}$$

$$\dot{v}_2 = v_2 \left(\frac{-K_{d1}}{M_2} \right) + x_2 \left(\frac{-K_{s1} - K_{s2}}{M_2} \right) + v_1 \left(\frac{K_{d1}}{M_2} \right) + x_1 \left(\frac{K_{s1}}{M_2} \right)$$

Figure 4.6 Drill problem: Convert the system to state equations

In some cases we will develop differential equations that cannot be directly reduced because they have more than one term at the highest order. For example, if a second-order differential equation has two second derivatives it cannot be converted to a state equation in the normal manner. In this case the two high order derivatives can be replaced with a dummy variable. In mechanical systems this often happens when masses are neglected. Consider the example problem in Figure 4.7, both 'y' and 'u' are first derivatives. To solve this problem, the highest order terms ('y' and 'u') are moved to the left of the equation. A dummy variable, 'q', is then created to replace these two variables with a single variable. This also creates an output equation as shown in Figure 4.1.

Given the equation,

$$3\dot{y} + 2y = 5\dot{u}$$

Step 1: put both the first-order derivatives on the left hand side,

$$3\dot{y} - 5\dot{u} = -2y$$

Step 2: replace the left hand side with a dummy variable,

$$q = 3y - 5u \quad \dot{q} = -2y$$

Step 3: solve the equation using the dummy variable, then solve for y as an output eqn.

$$\dot{q} = -2y \quad y = \frac{q + 5u}{3}$$

Figure 4.7 Using dummy variables for multiple high order terms

At other times it is possible to eliminate redundant terms through algebraic manipulation, as shown in Figure 4.8. In this case the force on both sides of the damper is the same, so it is substituted into the equation for the cart. But, the effects on the damper must also be integrated, so a dummy variable is created for the integration. An output equation was created to calculate the value for x_1 .

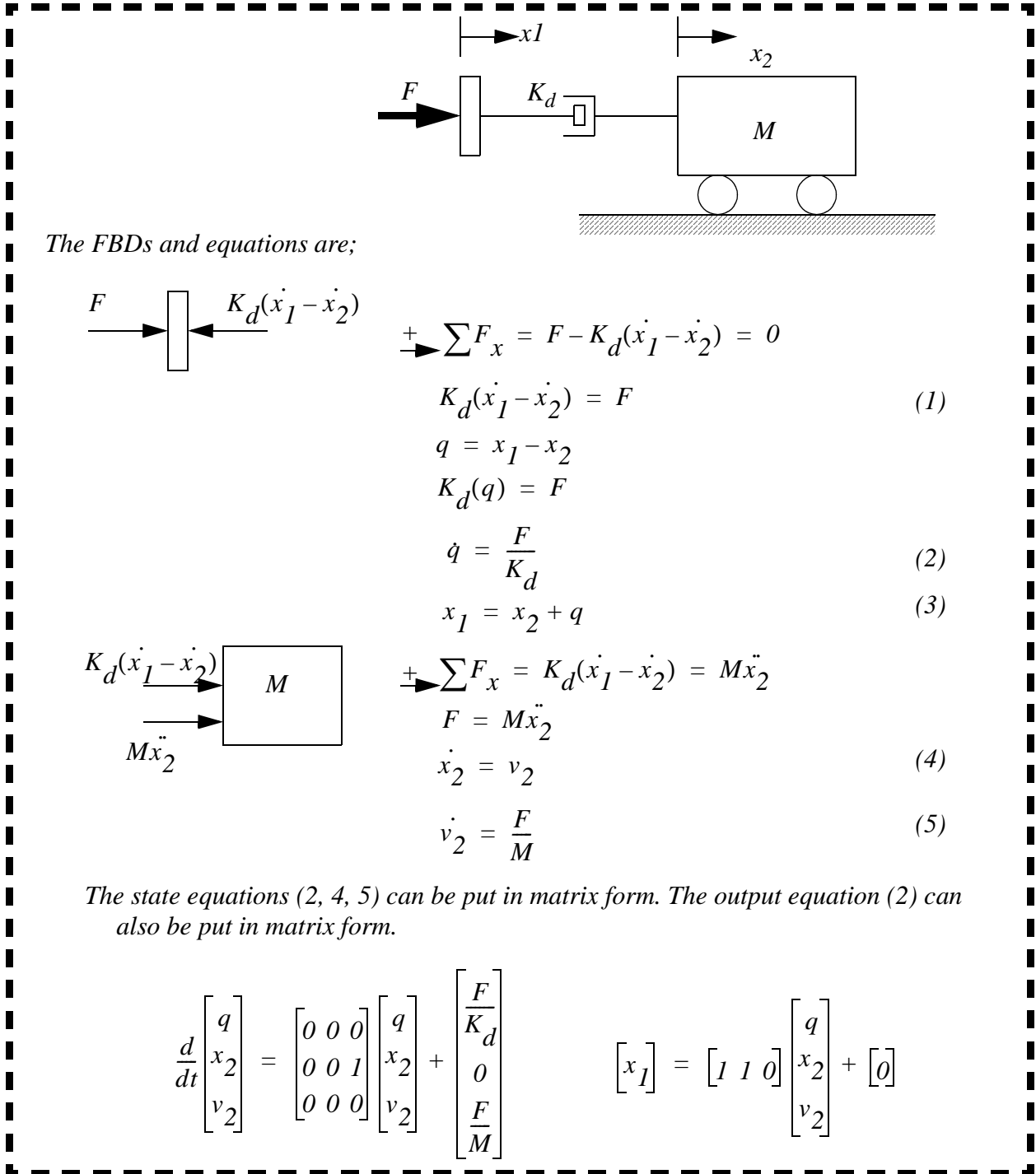


Figure 4.8 A dummy variable example

4.3 NUMERICAL INTEGRATION

Repetitive calculations can be used to develop an approximate solution to a set of differential equations. Starting from given initial conditions, the equation is solved with small time steps. Smaller time steps result in a higher level of accuracy, while larger time steps give a faster solution.

4.3.1 Numerical Integration With Tools

Numerical solutions can be developed with hand calculations, but this is a very time consuming task. In this section we will explore some common tools for solving state variable equations. The analysis process follows the basic steps listed below.

1. Generate the differential equations to model the process.
2. Select the state variables.
3. Rearrange the equations to state variable form.
4. Add additional equations as required.
5. Enter the equations into a computer or calculator and solve.

An example in Figure 4.9 shows the first four steps for a mass-spring-damper combination. The FBD is used to develop the differential equations for the system. The state variables are then selected, in this case the position, y , and velocity, v , of the block. The equations are then rearranged into state equations. The state equations are also put into matrix form, although this is not always necessary. At this point the equations are ready for solution.

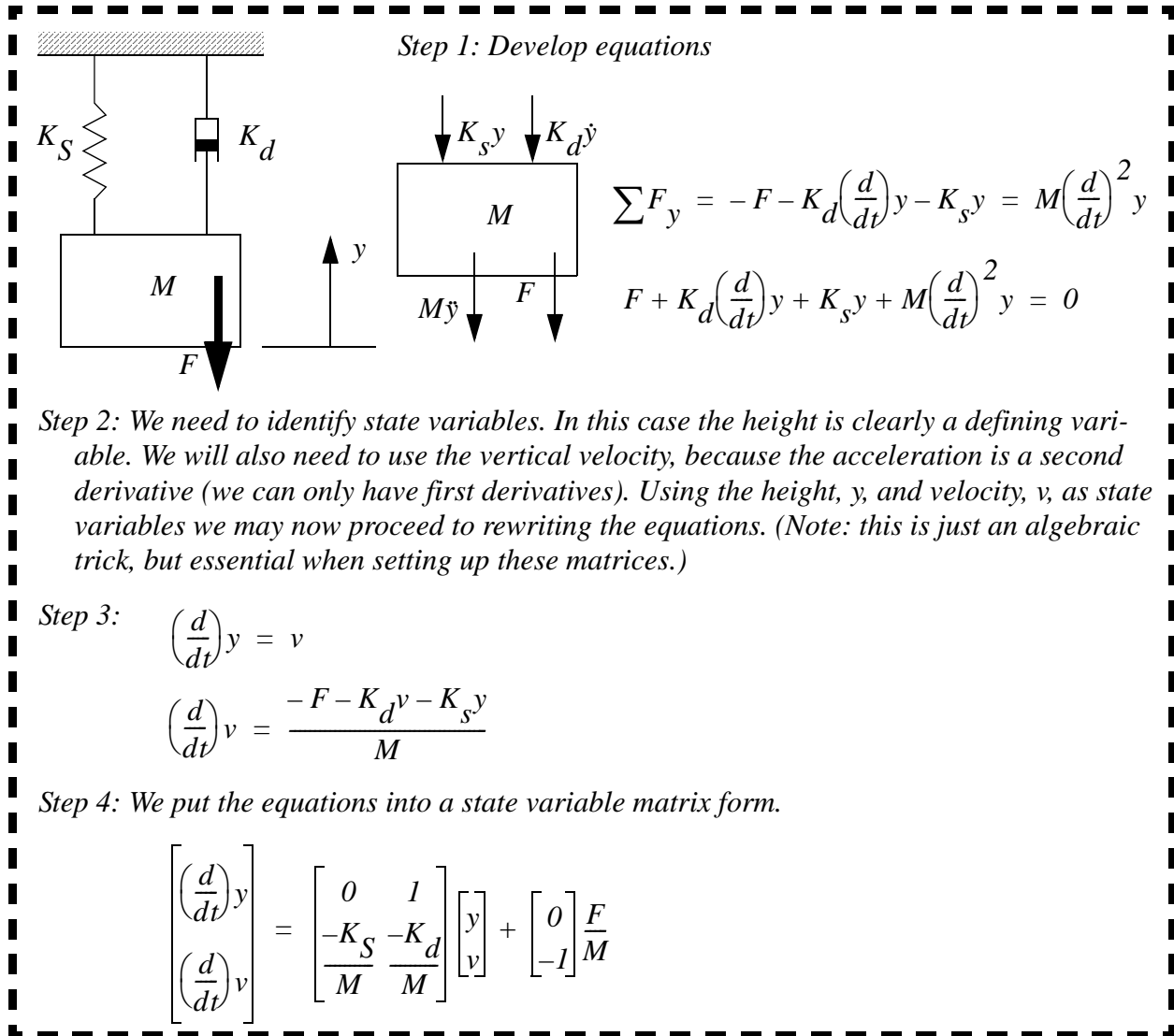


Figure 4.9 Dynamic system example

Figure 4.10 shows the method for solving state equations on a TI-86 graphing calculator. (Note: this also works on other TI-8x calculators with minor modifications.) In the example a sinusoidal input force, F , is used to make the solution more interesting. The next step is to put the equation in the form expected by the calculator. When solving with the TI calculator the state variables must be replaced with the predefined names Q1, Q2, etc. The steps that follow describe the button sequences required to enter and analyze the equations. The result is a graph that shows the solution of the equation. Points can then be taken from the graph using the cursors. (Note: large solutions can sometimes take a few minutes to solve.)

First, we select some parameter values for the equations of Figure 4.9. The input force will be a decaying sine wave.

$$\left(\frac{d}{dt}\right)y = v_y$$

$$\left(\frac{d}{dt}\right)v_y = \frac{-F - K_d v_y - K_s y}{M} = -4e^{-0.5t} \sin(t) - 2v_y - 5y$$

Next, the calculator requires that the state variables be $Q1, Q2, \dots, Q9$, so we replace y with $Q1$ and v with $Q2$.

$$Q1' = Q2$$

$$Q2' = -4e^{-0.5t} \sin(t) - 2Q2 - 5Q1$$

Now, we enter the equations into the calculator and solve. To do this roughly follow the steps below. Look at the calculator manual for additional details.

1. Put the calculator in differential equation mode
[2nd][MODE][DifEq][ENTER]
2. Go to graph mode and enter the equations above [GRAPH][F1]
3. Set up the axis for the graph [GRAPH][F2] so that time and the x-axis is from 0 to 10 with a time step of 0.5, and the y height is from +3 to -3.
4. Enter the initial conditions for the system [GRAPH][F3] as $Q1=0$, $Q2=0$
5. Set the axis [GRAPH][F4] as $x=t$ and $y=Q$
6. (TI-86 only) Set up the format [GRAPH][MORE][F1][Fld-Off][ENTER]
7. Draw the graph [GRAPH][F5]
8. Find points on the graph [GRAPH][MORE][F4]. Move the left/right cursor to move along the trace, use the up/down cursor to move between traces.

Figure 4.10 Solving state equations with a TI-85 calculator

First, we select some parameter values for the equations of Figure 4.9. The input force will be a decaying sine wave.

$$\left(\frac{d}{dt}\right)y = v_y$$

$$\left(\frac{d}{dt}\right)v_y = \frac{-F - K_d v_y - K_s y}{M} = -4e^{-0.5t} \sin(t) - 2v_y - 5y$$

Next, the calculator requires that the state variables be $Q1, Q2, \dots, Q9$, so we replace y with $Q1$ and v with $Q2$.

$$Q1' = Q2$$

$$Q2' = -4e^{-0.5t} \sin(t) - 2Q2 - 5Q1$$

Now, we enter the equations into the calculator and solve. To do this roughly follow the steps below. Look at the calculator manual for additional details.

1. Put the calculator in differential equation mode

[2nd][MODE][DifEq][ENTER]

2. Go to graph mode and enter the equations above [GRAPH][F1]

3. Set up the axis for the graph [GRAPH][F2] so that the x -axis is from 0 to 10 with a time step of 0.5 and the y -height is from +3 to -3.

4. Enter the initial conditions for the system [GRAPH][F3] as $Q1=0, Q2=0$

5. Set the axis [GRAPH][F4] as $x=t$ and $y=Q$

6. (TI-86 only) Set up the format [GRAPH][MORE][F1][Fld-Off][ENTER]

7. Draw the graph [GRAPH][F5]

8. Find points on the graph [GRAPH][MORE][F4]. Move the left/right cursor to move along the trace, use the up/down cursor to move between traces.

Figure 4.11 Solving state equations with a TI-89 calculator

State equations can also be solved in Mathcad using built-in functions, as shown in Figure 4.12. The first step is to enter the state equations as a function, 'D(t, Q)', where 't' is the time and 'Q' is the state variable vector. (Note: the equations are in a vector, but it is not the matrix form.) The state variables in the vector 'Q' replace the original state variables in the equations. The 'rkfixed' function is then used to obtain a solution. The arguments for the function, in sequence are; the state vector, the start time, the end time, the number of steps, and the state equation function. In this case the 10 second time interval is divided into 100 parts each 0.1s in duration. This time is chosen because of the general

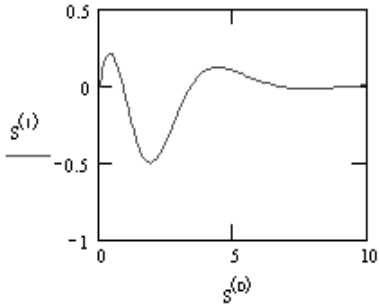
response time for the system. If the time step is too large the solution may become unstable and go to infinity. A time step that is too small will increase the computation time marginally. When in doubt, run the calculator again using a smaller time step.

$$D(t, Q) := \begin{bmatrix} Q_1 \\ (-4 e^{-0.5 \cdot t} \cdot \sin(t)) - 2 \cdot Q_1 - 5 \cdot Q_0 \end{bmatrix}$$

$QI := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ Define state variable equations and initial conditions. Note the origin is at 0' here.

$tMin := 0$ $tMax := 10$ $N := 100$ Define time range and number of steps for the integration.

$S := rkfixed(QI, tMin, tMax, N, D)$ A fixed time step runge-kutta integration



The first state variable S<1> = Q0 plotted against time S<0> = t.

A point in time is,

$i := 40$

$t := (S^{(0)})_i$ $Q_0 := (S^{(1)})_i$ $Q_1 := (S^{(2)})_i$

$t = 4$ $Q_0 = 0.103$ $Q_1 = 0.081$

Figure 4.12 Solving state variable equations with Mathcad

Note: Notice that for the TI calculators the variables start at Q1, while in Mathcad the arrays start at Q0. Many students encounter problems because they forget this.

4.3.2 Numerical Integration

The simplest form of numerical integration is Euler's first-order method. Given the current value of a function and the first derivative, we can estimate the function value a short time later, as shown in Figure 4.13. (Note: Recall that the state equations allow us to calculate first-order derivatives.) The equation shown is known as Euler's equation. Basically, using a known position and first derivative we can calculate an approximate value a short time, h , later. Notice that the function being integrated curves downward, creating an error between the actual and estimated values at time ' $t+h$ '. If the time step, h , were smaller, the error would decrease.

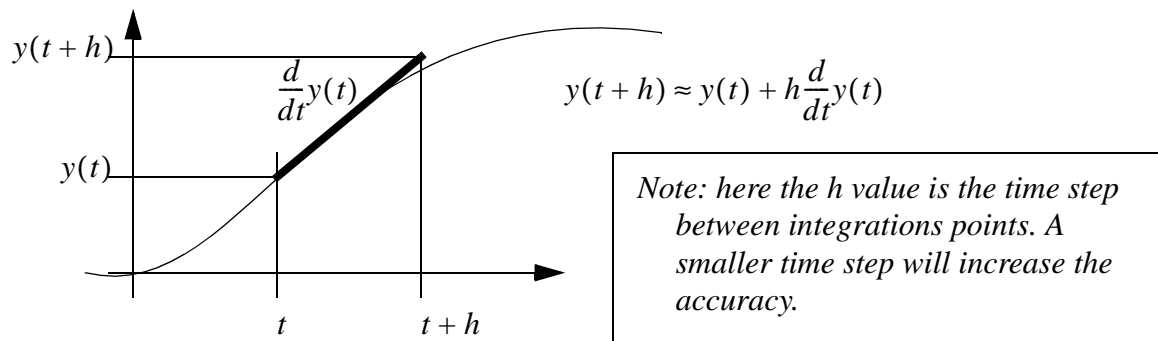


Figure 4.13 First-order numerical integration

The example in Figure 4.14 shows the solution of Newton's equation using Euler's method. In this example we are determining velocity by integrating the acceleration caused by a force. The acceleration is put directly into Euler's equation. This is then used to calculate values iteratively in the table. Notice that the values start before zero so that initial conditions can be used. If the system was second-order we would need two previous values for the calculations.

Given the differential equation,

$$F = M \left(\frac{d}{dt} \right) v$$

we can create difference equations using simple methods.

$$\left(\frac{d}{dt} \right) v = \frac{F}{M} \quad \text{first rearrange equation}$$

$$v(t+h) = v(t) + h \left(\frac{d}{dt} \right) v(t) \quad \text{put this in the Euler equation}$$

$$v(t+h) = v(t) + h \left(\frac{F(t)}{M} \right) \quad \text{finally substitute in known terms}$$

We can now use the equation to estimate the system response. We will assume that the system is initially at rest and that a force of 1N will be applied to the 1kg mass for 4 seconds. After this time the force will rise to 2N. A time step of 2 seconds will be used.

i	t (sec)	F (N)	$d/dt v_i$	v_i
-1	-2	0	0	0
0	0	1	1	0
1	2	1	1	2
2	4	2	2	4
3	6	2	2	8
4	8	2	2	12
5	10	2	2	16
6	12	2	2	20
7	14	2	etc	etc
8	16	2		

Figure 4.14 First order numerical integration example

Use first-order integration to solve the differential equation from 0 to 10 seconds with time steps of 1 second.

$$\dot{x} + 0.1x = 5$$

ans.

$$x(t+h) = x(t) + h(-0.1x(t) + 5)$$

$$x(10) =$$

Figure 4.15 Drill problem: Numerically integrate the differential equation

An example of solving the previous example with a traditional programming language is shown in Figure 4.16. In this example the results will be written to a text file 'out.txt'. The solution iteratively integrates from 0 to 10 seconds with time steps of 0.1s. The force value is varied over the time period with 'if' statements. The integration is done with a separate function.

```
double step(double, double, double);

int main(){
    double      h = 0.1,
               M = 1.0,
               F;

    FILE  *fp;
    double v,
           t;
    if( ( fp = fopen("out.txt", "w") ) != NULL){
        v = 0.0;
        for( t = 0.0; t < 10.0; t += h ){
            if((t >= 0.0) && (t < 4.0)) F = 1.0;
            if(t > 4.0) F = 2.0;
            v = step(v, h, F/M);
            fprintf(fp, "%f, %f, %f\n", t, v, F, M);
        }
        fclose(fp);
    }

    double step(double v, double h, double slope){
        double      v_new;
        v_new = v + h * slope;
        return v_new;
    }
}
```

Figure 4.16 Solving state variable equations with a C program

```

double step(double, double, double);

public class Integrate extends Object
    public void main() {
        double          h = 0.1,
                       M = 1.0,
                       F;

        FileOut fp = new FileOut("out.txt");
        if(fp.writeStatus != fp.IO_EXCEPTION){
            double v = 0.0;
            for( double t = 0.0; t < 10.0; t += h ){
                if((t >= 0.0) && (t < 4.0)) F = 1.0;
                if(t > 4.0) F = 2.0;
                v = step(v, h, F/M);
                fp.printf(fp, "%f, %f, %f\n", t, v, F, M);
            }
            fp.close();
        }
        fclose(fp);
    }

    public double step(double v, double h, double slope){
        double          v_new;
        v_new = v + h * slope;
        return v_new;
    }
}

```

Figure 4.17 Solving state variable equations with a Java program

The program below is for Scilab (a Matlab clone). The state variable function is defined first. This is followed by a definition of the parameters to be used for the numerical integration. Finally the function is integrated with rectangular, trapezoidal and Simpson's rule forms.

```
//  
// first_order.sce  
//  
// A first order integration of an accelerating mass  
//  
// To run this in Scilab use 'File' then 'Exec'.  
//  
// by: H. Jack Sept., 16, 2002  
//  
  
// System component values  
mass = 10;  
force = 100;  
  
x0 = 8;           // initial conditions  
v0 = 12;  
X=[x0, v0];  
  
// define the state matrix function  
// the values returned are [x, v]  
function foo=f(state,t)  
    foo = [ state($, 2), force/mass]; // d/dt x = v, d/dt v = F/M  
endfunction  
  
// Set the time length and step size for the integration  
steps = 100;  
t_start = 1;  
t_end = 100;  
h = (t_end - t_start) / steps;
```

Figure 4.18 First order integration with Scilab

```

//
// Loop for integration
//
for i=1:steps,
    X = [X ; X($,:) + h*f(X, i*h)];
end
printf("The value at the end of first order integration is (x, v) = (%f, %f)\n", ...
    X($,1), ...
    X($,2));

//
// Explicit equation
//
function x=position(x0, v0, a0, t)
    x = (0.5 * a0 * t^2) + (v0 * t) + x0;
endfunction

function v=velocity(v0, a0, t)
    v = (a0 * t) + v0;
endfunction

printf("The value with integration is (x, v) = (%f, %f)\n", ...
    position(x0, v0, force/mass, t_end), ...
    velocity(v0, force/mass, t_end));

//
// The results should be
//     first order integration = (49710, 1002)
//     explicit                 = (51208, 1012)
//
// The difference is 1498 for position and 10 for velocity. This is relatively small, but
// shows a clear case of the innacuracy of the numerical solutions.
//
// Note: increasing the number of steps increases the accuracy
//

```

Figure 4.19 First order integration with Scilab (continued)

4.3.3 Taylor Series

First-order integration works well with smooth functions. But, when a highly curved function is encountered we can use a higher order integration equation. The Taylor series equation shown in Figure 4.20 for approximating a function. Notice that the first part of the equation is identical to Euler's equation, but the higher order terms add accuracy.

$$x(t+h) = x(t) + h\left(\frac{d}{dt}\right)x(t) + \frac{1}{2!}h^2\left(\frac{d}{dt}\right)^2 x(t) + \frac{1}{3!}h^3\left(\frac{d}{dt}\right)^3 x(t) + \frac{1}{4!}h^4\left(\frac{d}{dt}\right)^4 x(t) + \dots$$

Figure 4.20 The Taylor series

An example of the application of the Taylor series is shown in Figure 4.21. Given the differential equation, we must first determine the derivatives and substitute these into Taylor's equation. The resulting equation is then used to iteratively calculate values.

Given $\dot{x} - x = 1 + e^{-20t} + t^3$

We can write, $\left(\frac{d}{dt}\right)x = 1 + e^{-20t} + t^3 + x$

$$\left(\frac{d}{dt}\right)^2 x = -e^{-20t} + 3t^2$$

$$\left(\frac{d}{dt}\right)^3 x = e^{-20t} + 6t$$

In the Taylor series this becomes,

$$x(t+h) = x(t) + h(1 + e^{-20t} + t^3 + x) + \frac{1}{2!}h^2(-e^{-t} + 3t^2) + \frac{1}{3!}h^3(e^{-t} + 6t)$$

Thus

$x_0 = 0$	$t (s)$	$x(t)$
$h = 0.1$	0	0
	0.1	0
	0.2	
	0.3	
	0.4	
	0.5	
	0.6	
	0.7	
	0.8	
	0.9	

e.g., for $t=0.1$

$$x(0 + 0.1) = 0 + 0.1(2) + \frac{1}{2!}(0.1)^2(-1) + \frac{1}{3!}(0.1)^3(1) =$$

Figure 4.21 Integration using the Taylor series method

Recall that the state variable equations are first-order equations. But, to obtain accuracy the Taylor method also requires higher order derivatives, thus making it unsuitable for use with state variable equations.

4.3.4 Runge-Kutta Integration

First-order integration provides reasonable solutions to differential equations. That accuracy can be improved by using higher order derivatives to compensate for function curvature. The Runge-Kutta technique uses first-order differential equations (such as state equations) to estimate the higher order derivatives, thus providing higher accuracy without requiring other than first-order differential equations.

The equations in Figure 4.22 are for fourth order Runge-Kutta integration. The function 'f()' is the state equation or state equation vector. For each time step the values 'F1' to 'F4' are calculated in sequence and then used in the final equation to find the next value. The 'F1' to 'F4' values are calculated at different time steps, and values from previous time steps are used to 'tweak' the estimates of the later states. The final summation equation has a remote similarity to the first order integration equation. Notice that the two central values in time are more heavily weighted.

$$F_1 = hf(t, x)$$

$$F_2 = hf\left(t + \frac{h}{2}, x + \frac{F_1}{2}\right)$$

$$F_3 = hf\left(t + \frac{h}{2}, x + \frac{F_2}{2}\right)$$

$$F_4 = hf(t + h, x + F_3)$$

Note: in this case the state equation function $f(t, x)$ includes the state variables, x , and time, t . However, in simpler systems the state equations may not include time and it could be replaced with $f(x)$.

$$x(t + h) = x(t) + \frac{1}{6}(F_1 + 2F_2 + 2F_3 + F_4)$$

where,

x = the state variables

f = the differential function or $(d/dt) x$

t = current point in time

h = the time step to the next integration point

Figure 4.22 Fourth order Runge-Kutta integration

An example of a Runge-Kutta integration calculation is shown in Figure 4.23. The solution begins by putting the state equations in matrix form and defining initial conditions. After this, the four integrating factors are calculated. Finally, these are combined to get the final value after one time step. The number of calculations for a single time step should make obvious the necessity of computers and calculators.

$$\begin{aligned} \frac{d}{dt}x &= v & y &= 2 \text{ (assumed input)} \\ \frac{d}{dt}v &= 3 + 4v + 5y & v_0 &= 1 \\ & & x_0 &= 3 \\ \frac{d}{dt} \begin{bmatrix} x \\ v \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 5 & 3 \end{bmatrix} \begin{bmatrix} y \\ 1 \end{bmatrix} & h &= 0.1 \end{aligned}$$

For the first time step,

$$F_1 = 0.1 \left(\begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 3 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 5 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} \right) = 0.1 \left(\begin{bmatrix} 1 \\ 4 \end{bmatrix} + \begin{bmatrix} 0 \\ 13 \end{bmatrix} \right) = \begin{bmatrix} 0.1 \\ 1.7 \end{bmatrix}$$

$$F_2 = 0.1 \left(\begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 3 + \frac{0.1}{2} \\ 1 + \frac{1.7}{2} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 5 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 0.185 \\ 2.04 \end{bmatrix}$$

$$F_3 = 0.1 \left(\begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 3 + \frac{0.185}{2} \\ 1 + \frac{2.04}{2} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 5 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 0.202 \\ 2.108 \end{bmatrix}$$

$$F_4 = 0.1 \left(\begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 3 + 0.202 \\ 1 + 2.108 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 5 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 0.3108 \\ 2.5432 \end{bmatrix}$$

$$\begin{bmatrix} x_{i+1} \\ v_{i+1} \end{bmatrix} = \begin{bmatrix} x_i \\ v_i \end{bmatrix} + \frac{1}{6}(F_1 + 2F_2 + 2F_3 + F_4)$$

$$\begin{bmatrix} x_{i+1} \\ v_{i+1} \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \end{bmatrix} + \frac{1}{6} \left(\begin{bmatrix} 0.1 \\ 1.7 \end{bmatrix} + 2 \begin{bmatrix} 0.185 \\ 2.04 \end{bmatrix} + 2 \begin{bmatrix} 0.202 \\ 2.108 \end{bmatrix} + \begin{bmatrix} 0.3108 \\ 2.5432 \end{bmatrix} \right) = \begin{bmatrix} 3.1974667 \\ 3.0898667 \end{bmatrix}$$

Figure 4.23 Runge-Kutta integration example

$$F = M \left(\frac{d}{dt} \right)^2 x$$

use,

$$x(0) = 1$$

$$v(0) = 2$$

$$h = 0.5 \text{ s}$$

$$F = 10$$

$$M = 1$$

Figure 4.24 Drill problem: Integrate the acceleration function

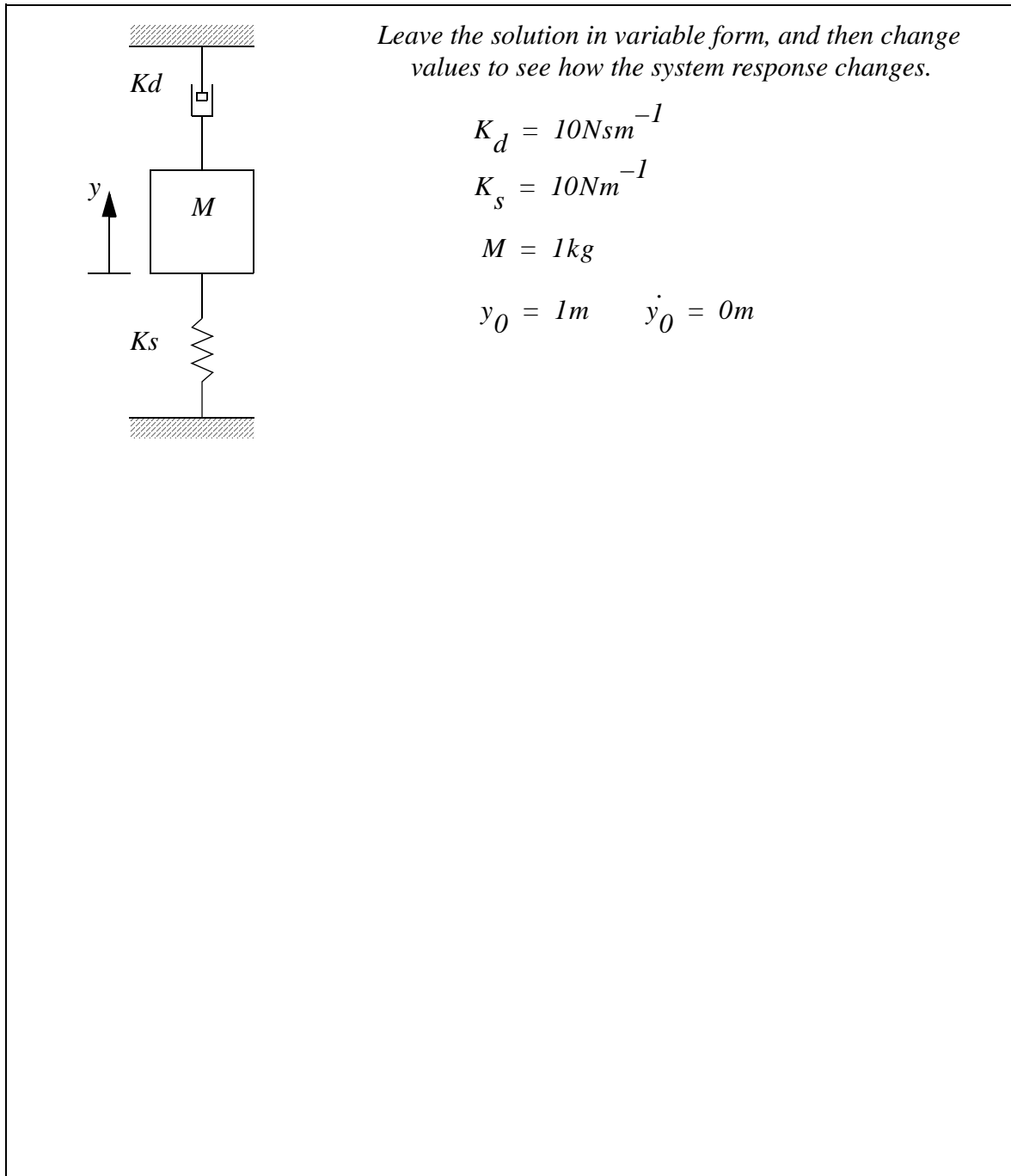


Figure 4.25 Drill problem: Integrate to find the system response

The program in Figure 4.26 and Figure 4.28 is used to perform a Runge-Kutta integration of a mass-spring-damper system. The 'main' program loops through the time steps and writes the value to a file. The 'step' function performs one timestep integration for a

second order Runge-Kutta integration. It uses the functions 'add' and 'multiply' to manipulate the state matrix. The 'derivative' function updates the state matrix with the new derivative values.

```

/* A program to do Runge Kutta integration of a mass spring damper system */
#include <stdio.h>

void multiply(double, double[], double[]);
void add(double[], double[], double[]);
void step(double, double, double[]);
void derivative(double, double[], double[]);

#define SIZE          2 /* the length of the state vector */
#define Ks            1000 /* the spring coefficient */
#define Kd            10000 /* the damping coefficient */
#define Mass          10 /* the mass coefficient */
#define Force         100 /* the applied force */

int main(){
    FILE *fp;

    double h = 0.001;
    double t;
    int j = 0;

    double X[SIZE]; // create state variable list
    X[0] = 0; // set initial condition for x
    X[1] = 0; // set initial condition for v

    if( ( fp = fopen("out.txt", "w") ) != NULL){
        fprintf(fp, " t(s) x v \n\n");
        for( t = 0.0; t < 50.0; t += h ){
            step(t, h, X);
            if(j == 0) fprintf(fp, "%9.5f %9.5f %9.5f\n", t, X[0], X[1]);
            j++; if(j >= 10) j = 0;
        }
    }
    fclose(fp);
}

```

Figure 4.26 Runge-Kutta integration C program

```

/* First order integration done here (could be replaced with runge kutta)*/
void step(double t, double h, double X[]){
    double tmp[SIZE],
           dX[SIZE],
           F1[SIZE],
           F2[SIZE],
           F3[SIZE],
           F4[SIZE];

    /* Calculate F1 */
    derivative(t, X, dX);
    multiply(h, dX, F1);

    /* Calculate F2 */
    multiply(0.5, F1, tmp);
    add(X, tmp, tmp);
    derivative(t+h/2.0, tmp, dX);
    multiply(h, dX, F2);

    /* Calculate F3 */
    multiply(0.5, F2, tmp);
    add(X, tmp, tmp);
    derivative(t+h/2.0, tmp, dX);
    multiply(h, dX, F3);

    /* Calculate F4 */
    add(X, F3, tmp);
    derivative(t+h, tmp, dX);
    multiply(h, dX, F4);

    /* Calculate the weighted sum */
    add(F2, F3, tmp);
    multiply(2.0, tmp, tmp);
    add(F1, tmp, tmp);
    add(F4, tmp, tmp);
    multiply(1.0/6.0, tmp, tmp);
    add(tmp, X, X);
}

/* State Equations Calculated Here */
void derivative(double t, double X[], double dX[]){
    dX[0] = X[1];
    dX[1] = (-Ks/Mass)*X[0] + (-Kd/Mass)*X[1] + (Force/Mass);
}

/* A subroutine to add vectors to simplify other equations */
void add(double X1[], double X2[], double R[]){
    for(int i = 0; i < SIZE; i++) R[i] = X1[i] + X2[i];
}

/* A subroutine to multiply a vector by a scalar to simplify other equations*/
void multiply(double X, double V[], double R[]){
    for(int i = 0; i < SIZE; i++) R[i] = X*V[i];
}

```

Figure 4.27 Runge-Kutta integration C program (cont'd)

A Scilab program is given in Figure 4.28 to perform a Runge Kutta integration.

```

// runge_kutta.sce
// A first order integration of an accelerating mass
// To run this in Scilab use 'File' then 'Exec'.
// by: H. Jack Sept., 15, 2003

// System component values
mass = 10;
force = 100;

x0 = 8;          // initial conditions
v0 = 12;
X=[x0, v0];

// define the state matrix function
// the values returned are [x, v]
function foo=f(state,t)
    foo = [ state($, 2), force/mass]; // d/dt x = v, d/dt v = F/M
endfunction

// Set the time length and step size for the integration
steps = 1000;
t_start = 0;
t_end = 100;
h = (t_end - t_start) / steps;
t = [t_start];

// Loop for integration
for i=1:steps,
    t = [t ; t($,:) + h];
    F1 = h * f(X($,:), t($,:));
    F2 = h * f(X($,:) + F1/2.0, t($,:) + h/2.0);
    F3 = h * f(X($,:) + F2/2.0, t($,:) + h/2.0);
    F4 = h * f(X($,:) + F3, t($,:) + h);
    X = [X ; X($,:) + (F1 + 2.0*F2 + 2.0*F3 + F4)/6.0];
end

// print some results to compare
printf("The value at the end of first order integration is (x, v) = (%f, %f)\n", ...
    X($,1), ...
    X($,2));
printf("The position (using an equation) should be %f\n", 0.5*force/mass*(t_end-
t_start)^2.0 + v0*(t_end - t_start) + x0);

// Graph the values
plot2d(t, X, [-2, -5], leg="position@velocity");
    // leg - the legend titles
    // style - draw lines with marks
    // nax - grid lines for the graph
xtitle('Time (s)');

```

Figure 4.28 Runge-Kutta integration Scilab program

4.4 SYSTEM RESPONSE

In most cases the result of numerical analysis is graphical or tabular. In both cases

details such as time constants and damped frequencies can be obtained by the same methods used for experimental analysis. In addition to these methods there is a technique that can determine the steady-state response of the system.

4.4.1 Steady-State Response

The state equations can be used to determine the steady-state response of a system by setting the derivatives to zero, and then solving the equations. Consider the example in Figure 4.29. The solution begins with a state variable matrix. (Note: this can also be done without the matrix also.) The derivatives on the left hand side are set to zero, and the equations are rearranged and solved with Cramer's rule.

Given the state variable form:

$$\frac{d}{dt} \begin{bmatrix} x \\ v \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{K_s}{M} & -\frac{K_d}{M} \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{F}{M} \end{bmatrix}$$

Set the derivatives to zero

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{K_s}{M} & -\frac{K_d}{M} \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{F}{M} \end{bmatrix}$$

Solve for x and v

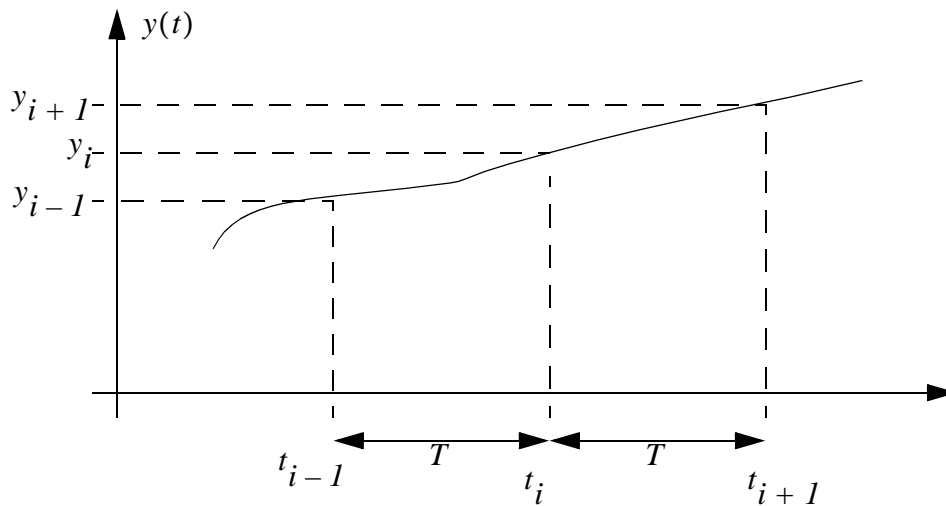
$$\begin{bmatrix} 0 & 1 \\ -\frac{K_s}{M} & -\frac{K_d}{M} \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} = \begin{bmatrix} 0 \\ -\frac{F}{M} \end{bmatrix}$$

$$x = \frac{\begin{vmatrix} 0 & 1 \\ -\frac{F}{M} & -\frac{K_d}{M} \end{vmatrix}}{\begin{vmatrix} 0 & 1 \\ -\frac{K_s}{M} & -\frac{K_d}{M} \end{vmatrix}} = \frac{\left(\frac{F}{M}\right)}{\left(\frac{K_s}{M}\right)} = \frac{F}{K_s} \quad v = \frac{\begin{vmatrix} 0 & 0 \\ -\frac{K_s}{M} & -\frac{F}{M} \end{vmatrix}}{\begin{vmatrix} 0 & 1 \\ -\frac{K_s}{M} & -\frac{K_d}{M} \end{vmatrix}} = \frac{0}{\left(\frac{K_s}{M}\right)} = 0$$

Figure 4.29 Finding the steady-state response

4.5 DIFFERENTIATION AND INTEGRATION OF EXPERIMENTAL DATA

When doing experiments, data is often collected in the form of individual data points (not as complete functions). It is often necessary to integrate or differentiate these values. The basic equations for integrating and differentiating are shown in Figure 4.30. Given data points, y , collected at given times, t , we can integrate and differentiate using the given equations. The integral is basically the average height of the two points multiplied by the width to give an area, or integral. The first derivative is basically the slope between two points. The second derivative is the change in slope values for three points. In a computer based system the time points are often equally spaced in time, so the difference in time can be replaced with a sample period, T . Ideally the time steps would be as small as possible to increase the accuracy of the estimates.



$$\int_{t_{i-1}}^{t_i} y(t) dt \approx \left(\frac{y_i + y_{i-1}}{2} \right) (t_i - t_{i-1}) = \frac{T}{2} (y_i + y_{i-1})$$

$$\frac{d}{dt} y(t_i) \approx \left(\frac{y_i - y_{i-1}}{t_i - t_{i-1}} \right) = \left(\frac{y_{i+1} - y_i}{t_{i+1} - t_i} \right) = \frac{1}{T} (y_i - y_{i-1}) = \frac{1}{T} (y_{i+1} - y_i)$$

$$\left(\frac{d}{dt} \right)^2 y(t_i) \approx \frac{\frac{1}{T} (y_{i+1} - y_i) - \frac{1}{T} (y_i - y_{i-1})}{T} = \frac{-2y_i + y_{i-1} + y_{i+1}}{T^2}$$

Figure 4.30 Integration and differentiation using data points

An example of numerical integration using Scilab is given in Figure 4.31 and Figure 4.32.

```
//  
// integrate.sce  
//  
// A simple program to integrate a function  
//  
// To run this in Scilab use 'File' then 'Exec'.  
//  
// by: H. Jack Sept., 9, 2002  
//  
  
// define the function  
function foo=f(x)  
    foo = 5 * x + 2 * log(sin(x) / x + 2);  
endfunction  
  
// Set the time length and step size  
steps = 10;  
x_start = 1;  
x_end = 10;  
x_delta = (x_end - x_start) / steps;  
  
//  
// Loop for rectangular integration  
//  
total = 0; // set the initial sum to zero  
for i=0:steps,  
    x = x_start + i * x_delta;  
    total = total + f(x);  
end  
total = total * x_delta;  
printf("Rectangular integration value %f\n", total);
```

Figure 4.31 Integration with a Scilab program

```

// Loop for trapezoidal integration
//
total = 0; // set the initial sum to zero
for i=0:steps,
    x = x_start + i * x_delta;
    if i == 0 then
        total = total + f(x);
    elseif i == steps then
        total = total + f(x);
    else
        total = total + 2 * f(x);
    end
end
total = total * x_delta / 2;
printf("Trapezoidal integration value %f\n", total);

//
// Loop for Simpson's rule integration
//
total = 0; // set the initial sum to zero
even = 0;
for i=0:steps,
    x = x_start + i * x_delta;
    if i == 0 then
        total = total + f(x);
    elseif i == steps then
        total = total + f(x);
    else
        even = even + 1;
        if even > 1 then
            total = total + 4 * f(x);
            even = 0;
        else
            total = total + 2 * f(x);
        end
    end
end
total = total * x_delta / 3;
printf("Simpsons rule integration value %f\n", total);

```

Figure 4.32 Integration with a Scilab Program (cont'd)

4.6 ADVANCED TOPICS

4.6.1 Switching Functions

When analyzing a system, we may need to choose an input that is more complex than inputs such as steps, ramps, sinusoids and parabolae. The easiest way to do this is to use switching functions. Switching functions turn on (have a value of 1) when their argu-

ments are greater than or equal to zero, or off (a value of 0) when the argument is negative. Examples of the use of switching functions are shown in Figure 4.33. By changing the values of the arguments we can change when a function turns on or off.

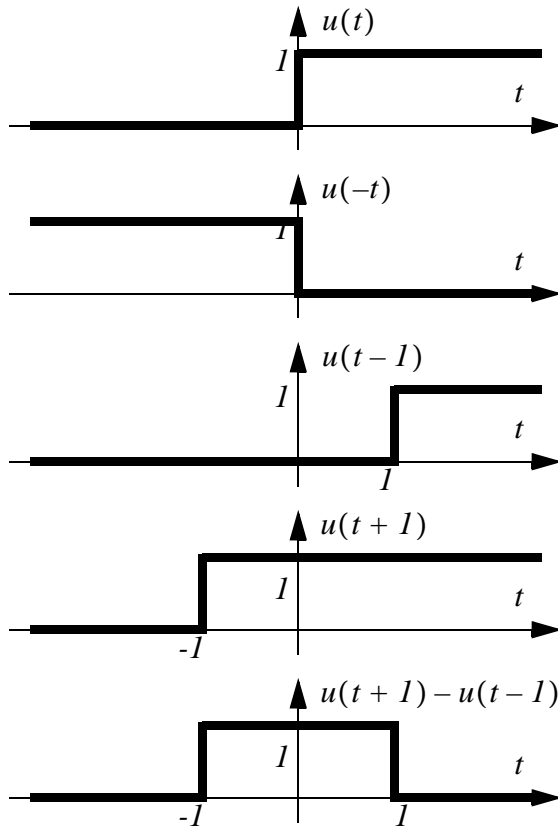


Figure 4.33 Switching function examples

These switching functions can be multiplied with other functions to create a complex function by turning parts of the function on or off. An example of a curve created with switching functions is shown in Figure 4.34.

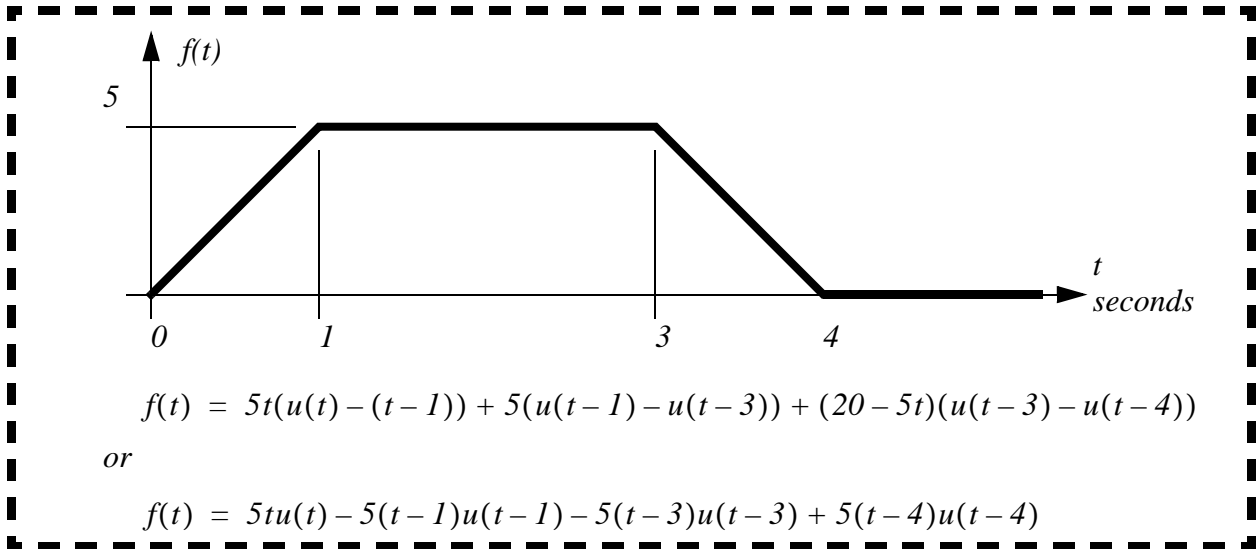


Figure 4.34 Switching functions to create a non-smooth function

The unit step switching function is available in Mathcad and makes creation of complex functions relatively trivial. Step functions are also easy to implement when writing computer programs, as shown in Figure 4.35.

For the function $f(t) = 5tu(t) - 5(t-1)u(t-1) - 5(t-3)u(t-3) + 5(t-4)u(t-4)$

```

double u(double t){
    if(t >= 0) return 1.0;
    return 0.0;
}

double function(double t){
    double f;

    f = 5.0 * t * u(t)
        - 5.0 * (t - 1.0) * u(t - 1.0)
        - 5.0 * (t - 3.0) * u(t - 3.0)
        + 5.0 * (t - 4.0) * u(t - 4.0);

    return f;
}

```

Figure 4.35 A subroutine implementing the example in Figure 4.34

4.6.2 Interpolating Tabular Data

In some cases we are given tables of numbers instead of equations for a system component. These can still be used to do numerical integration by calculating coefficient values as required, in place of an equation.

Tabular data consists of separate data points as seen in Figure 4.36. But, we may need values between the datapoints. A simple method for finding intermediate values is to interpolate with the "lever law". (Note: it is called this because of its' similarity to the equation for a lever.) The table in the example only gives flow rates for a valve at 10 degree intervals, but we want flow rates at 46 and 23 degrees. A simple application of the lever law gives approximate values for the flow rates.

<i>valve angle (deg.)</i>	<i>flow rate (gpm)</i>	
0	0	<i>Given a valve angle of 46 degrees the flow rate is,</i>
10	0.1	
20	0.4	$Q = 2.0 + (2.3 - 2.0)\left(\frac{46 - 40}{50 - 40}\right) = 2.18$
30	1.2	
40	2.0	<i>Given a valve angle of 23 degrees the flow rate is,</i>
50	2.3	
60	2.4	$Q = 0.4 + (1.2 - 0.4)\left(\frac{23 - 20}{30 - 20}\right) = 0.64$
70	2.4	
80	2.4	
90	2.4	

Figure 4.36 Using tables of values to interpolate numerical values using the lever law

The subroutine in Figure 4.37 was written to return the numerical value for the data table in Figure 4.36. In the subroutine the tabular data is examined to find the interval that the flow rate value falls inside. Once this is found the valve angle is calculated as the ratio between the two known values.

```

#define      SIZE      10;
double      data[SIZE][2] = {{0.0, 0.0},
                             {10.0, 0.1},
                             {20.0, 0.4},
                             {30.0, 1.2},
                             {40.0, 2.0},
                             {50.0, 2.3},
                             {60.0, 2.4},
                             {70.0, 2.4},
                             {80.0, 2.4},
                             {90.0, 2.4}};

double angle(double rate){
    int i;
    for(i = 0; i < SIZE-1; i++){
        if((rate >= data[i][0]) && (rate <= data[i+1][0])){
            return (data[i+1][1] - data[i][1])
                * (rate - data[i][0]) / (data[i+1][0] - data[i][0])
                + data[i][1];
        }
    }
    printf("ERROR: rate out of range\n");
    exit(1);
}

```

Figure 4.37 A tabular interpolation subroutine example

4.6.3 Modeling Functions with Splines

When greater accuracy is required, smooth curves can be fitted to interpolate points. These curves are known as splines. There are multiple methods for creating splines, but the simplest is to use a polynomial fitted to a set of points.

The example in Figure 4.38 shows a spline curve being fitted for three data points. In this case a second order polynomial is used. The three data points are written out as equations, and then put into matrix form, using the coefficients as the unknown values. The matrix is then solved to obtain the coefficient values for the final equation. This equation can then be used to build a mathematical model of the system.

The datapoints below might have been measured for the horsepower of an internal combustion engine on a dynamometer.

S (RPM)	P (HP)
1000	105
4000	205
6000	110

In this case there are three datapoints, so we can fit the curve with a second (3-1) order polynomial. The major task is to calculate the coefficients so that the curve passes through all of the given points.

$$P(S) = AS^2 + BS + C$$

Data values can be substituted into the equation,

$$P(1000) = A1000^2 + B1000 + C = 105$$

$$P(4000) = A4000^2 + B4000 + C = 205$$

$$P(6000) = A6000^2 + B6000 + C = 110$$

This can then be put in matrix form to find the coefficients,

$$\begin{bmatrix} 1000^2 & 1000 & 1 \\ 4000^2 & 4000 & 1 \\ 6000^2 & 6000 & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} 105 \\ 205 \\ 110 \end{bmatrix}$$

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} 1000000 & 1000 & 1 \\ 16000000 & 4000 & 1 \\ 36000000 & 6000 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 105 \\ 205 \\ 110 \end{bmatrix}$$

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} 6.667 \times 10^{-8} & -1.667 \times 10^{-7} & 1.000 \times 10^{-7} \\ -6.667 \times 10^{-4} & 1.167 \times 10^{-3} & -5.000 \times 10^{-4} \\ 1.600 & -1.000 & 0.400 \end{bmatrix} \begin{bmatrix} 105 \\ 205 \\ 110 \end{bmatrix}$$

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} -1.617 \times 10^{-5} \\ 0.114 \\ 7.000 \end{bmatrix}$$

$$P(S) = (-1.617 \times 10^{-5})S^2 + 0.114S + 7.000$$

Note: in this example the inverse matrix is used, but other methods for solving systems of equations are equally valid. If the equations were simpler, substitution might have been a better approach.

Figure 4.38 A spline fitting example

The order of the polynomial should match the number of points. Although, as the number of points increases, the shape of the curve will become less smooth. A common way for dealing with this problem is to fit the spline to a smaller number of points and then verify that it matches the remaining points, or use a least squares method to find the best approximation.

4.6.4 Non-Linear Elements

Despite our deepest wishes for simplicity, most systems contain non-linear components. In the last chapter we looked at dealing with non-linearities by linearizing them locally. Numerical techniques will handle non-linearities easily, but smaller time steps are required for accuracy.

Consider the mass and an applied force shown in Figure 4.39. As the mass moves an aerodynamic resistance force is generated that is proportional to the square of the velocity. This results in a non-linear differential equation. This equation can be numerically integrated using a technique such as Runge-Kutta. Note that the state equation matrix form cannot be used because it requires linear equations.

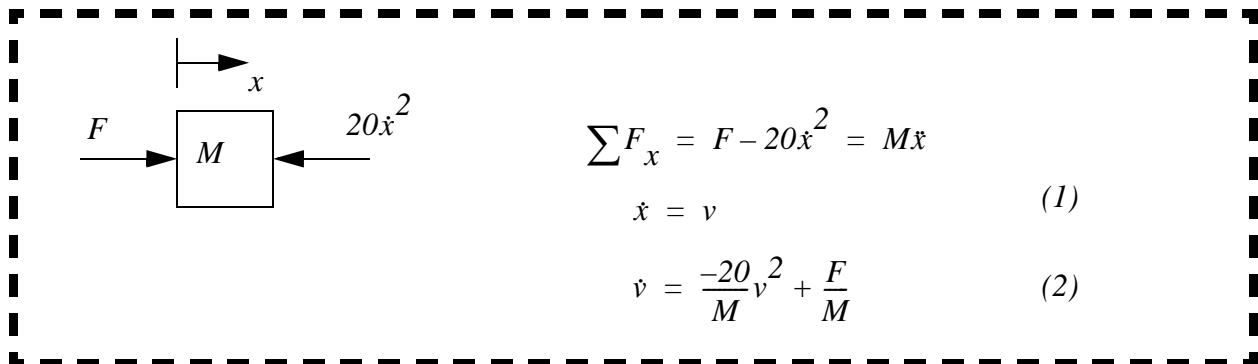


Figure 4.39 Developing state equations for a non-linear system

4.7 CASE STUDY

Consider the simplified model of a car suspension shown in Figure 4.40. The model distributes the vehicle weight over four tires with identical suspensions, so the mass of the vehicle is divided by four. In this model the height of the road will change and drive the tire up, or allow it to drop down. The tire acts as a stiff spring, with little deflection. The upper spring and damper are the vibration isolation units. The damper has been designed to stiffen as the damper is compressed. The given table shows how the damping

coefficient varies with the amount of compression.

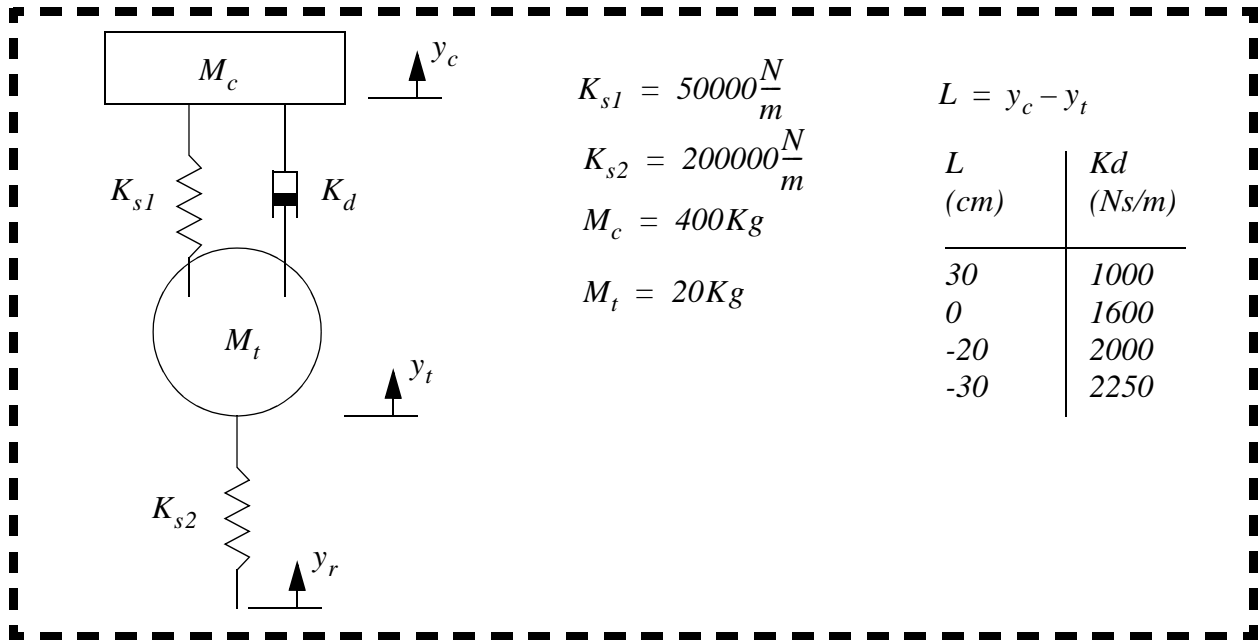


Figure 4.40 A model of a car suspension system

For our purposes we will focus only on the translation of the tire, and ignore its rotational motion. The differential equations describing the system are developed in Figure 4.41.

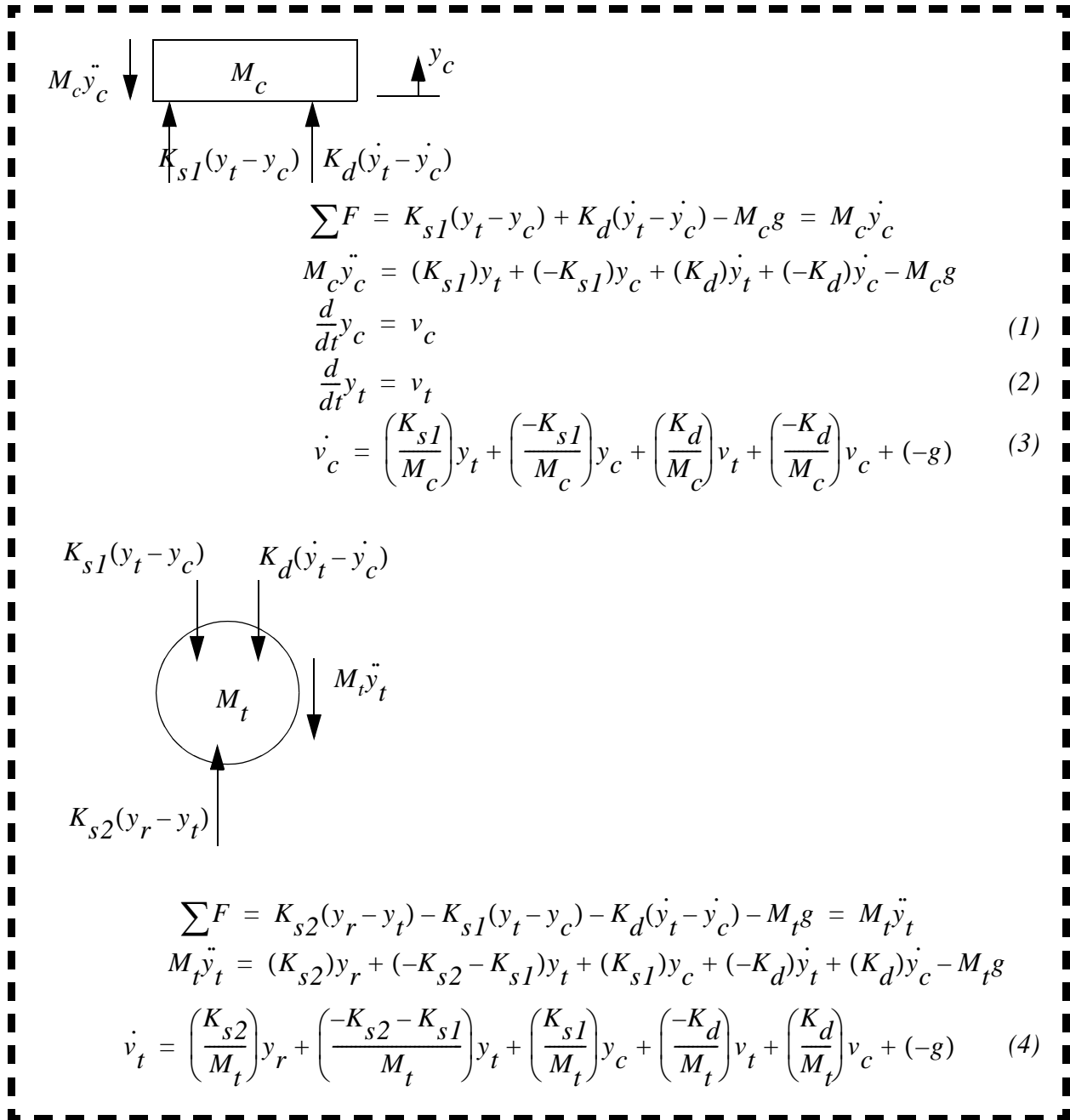


Figure 4.41 Differential and state equations for the car suspension system

The damping force must be converted from a tabular form to equation form. This is done in Figure 4.42.

There are four data points, so a third order polynomial is required.

$$K_d(L) = AL^3 + BL^2 + CL + D$$

L (cm)	Kd (Ns/m)
30	1000
0	1600
-20	2000
-30	2250

The four data points can now be written in equation form, and then put into matrix form.

$$\begin{aligned} 1000 &= A(0.3)^3 + B(0.3)^2 + C(0.3) + D \\ 1600 &= A(0)^3 + B(0)^2 + C(0) + D \\ 2000 &= A(-0.2)^3 + B(-0.2)^2 + C(-0.2) + D \\ 2250 &= A(-0.3)^3 + B(-0.3)^2 + C(-0.3) + D \end{aligned}$$

$$\begin{bmatrix} 0.027 & 0.09 & 0.3 & 1 \\ 0 & 0 & 0 & 1 \\ -0.008 & 0.04 & -0.2 & 1 \\ -0.027 & 0.09 & -0.3 & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = \begin{bmatrix} 1000 \\ 1600 \\ 2000 \\ 2250 \end{bmatrix}$$

The matrix can be solved to find the coefficients, and the final equation written.

$$\begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = \begin{bmatrix} -2778 \\ 277.8 \\ -1833 \\ 1600 \end{bmatrix}$$

$$K_d(L) = (-2778)L^3 + (277.8)L^2 + (-1833)L + 1600$$

Figure 4.42 Fitting a spline to the damping values

The system is to be tested for overall deflection when exposed to obstacles on the road. For the initial conditions we need to find the resting heights for the tire and car body. This can be done by setting the accelerations and velocities to zero, and finding the resulting heights.

The initial accelerations and velocities are set to zero, assuming the car has settled to a steady state height. This then yields equations that can be used to calculate the initial deflections. Assume the road height is also zero to begin with.

$$0 = \left(\frac{K_{s1}}{M_c}\right)y_t + \left(\frac{-K_{s1}}{M_c}\right)y_c + \left(\frac{K_d}{M_c}\right)0 + \left(\frac{-K_d}{M_c}\right)0 + (-g)$$

$$y_c = y_t - g \frac{M_c}{K_{s1}}$$

$$0 = \left(\frac{K_{s2}}{M_t}\right)0 + \left(\frac{-K_{s2} - K_{s1}}{M_t}\right)y_t + \left(\frac{K_{s1}}{M_t}\right)y_c + \left(\frac{-K_d}{M_t}\right)0 + \left(\frac{K_d}{M_t}\right)0 + (-g)$$

$$gM_t = (-K_{s2} - K_{s1})y_t + (K_{s1})y_c$$

$$gM_t = (-K_{s2} - K_{s1})y_t + (K_{s1})\left(y_t - g \frac{M_c}{K_{s1}}\right)$$

$$y_t = -g \frac{M_c + M_t}{K_{s2}}$$

$$y_c = -g \frac{M_c + M_t}{K_{s2}} - g \frac{M_c}{K_{s1}}$$

$$y_c = -g \left(\frac{M_c + M_t}{K_{s2}} + \frac{M_c}{K_{s1}} \right)$$

Figure 4.43 Calculation of initial deflections

The resulting calculations can then be written in a computer program for analysis, as shown in Figure 4.44.

```

#include <stdio.h>
#include <math.h>

#define SIZE 4 /* define state variables */
#define y_c 0
#define y_t 1
#define v_c 2
#define v_t 3

#define N_step 10000 // number of steps
#define h_step 0.001 // define step size

#define Ks1 50000.0 /* define component values */
#define Ks2 200000.0
#define Mc 400.0
#define Mt 20.0
#define grav 9.81

void integration_step(double h, double state[], double derivative[]){
    int i;
    for(i = 0; i < SIZE; i++) state[i] += h * derivative[i];
}

double damper(double L){
    return (-2778*L*L*L + 277.8*L*L - 1833*L + 1600);
}

double y_r(double t){
    /* return 0.0; /* a zero input to test the initial conditions */
    /* return 0.2 * sin(t);/* a sinusoidal oscillation */
    return 0.2; /* /* a step function */
    /* return 0.2 * t; /* /* a ramp function */
}

```

Figure 4.44 Program for numerical analysis of suspension system

```

void d_dt(double t, double state[], double derivative[]){
    double Kd;
    Kd = damper(state[y_c] - state[y_t]);
    derivative[y_c] = state[v_c];
    derivative[y_t] = state[v_t];
    derivative[v_c] = (Ks1/Mc)*state[y_t] - (Ks1/Mc)*state[y_c]
        + (Kd/Mc)*state[v_t] - (Kd/Mc)*state[v_c] - grav;
    derivative[v_t] = (Ks2/Mt)*y_r(t) - ((Ks2+Ks1)/Mt)*state[y_t]
        + (Ks1/Mt)*state[y_c] - (Kd/Mt)*state[v_t]
        + (Kd/Mt)*state[v_c] - grav;
}

main(){
    double state[SIZE];
    double derivative[SIZE];
    FILE *fp_out;
    double t;
    int i;

    state[y_c] = - grav * ( (Mc/Ks1) + (Mt + Mc)/Ks2 ); /* initial values */
    state[y_t] = - grav * (Mt + Mc) / Ks2;
    state[v_c] = 0.0;
    state[v_t] = 0.0;

    if((fp_out = fopen("out.txt", "w")) != NULL){ /* open the file */
        fprintf(fp_out, " t   Yc   Yt   Vc   Vt  \n");
        for(t = 0.0, i = 0; i < N_step; i++, t += h_step){
            if((i % 100) == 0) fprintf(fp_out, "%f   %f   %f   %f   %f \n",
                t, state[y_c], state[y_t], state[v_c], state[v_t]);
            d_dt(t, state, derivative);
            integration_step(h_step, state, derivative);
        }
    } else {
        printf("ERROR: Could not open file \n");
    }
    fclose(fp_out);
}

```

Figure 4.45 Program for numerical analysis of suspension system (continued)

This program was then used to test various design cases by selecting input types for changes in the road height, and then calculating how the tire and vehicle heights would change as a result. Some of these results are seen in Figure 4.46. These results were obtained by running the program, and then graphing the results in a spreadsheet program. The input of zero for the road height was used to test the program. As shown the height of the vehicle changes, indicating that the initial height calculations are correct, and the model is stable. The step function shows some oscillations that settle out to a stable final value. The oscillation is relatively slow, and is fully transmitted to the automobile. The ramp function shows that the car follows the rise of the slope with small transient effects at the start.

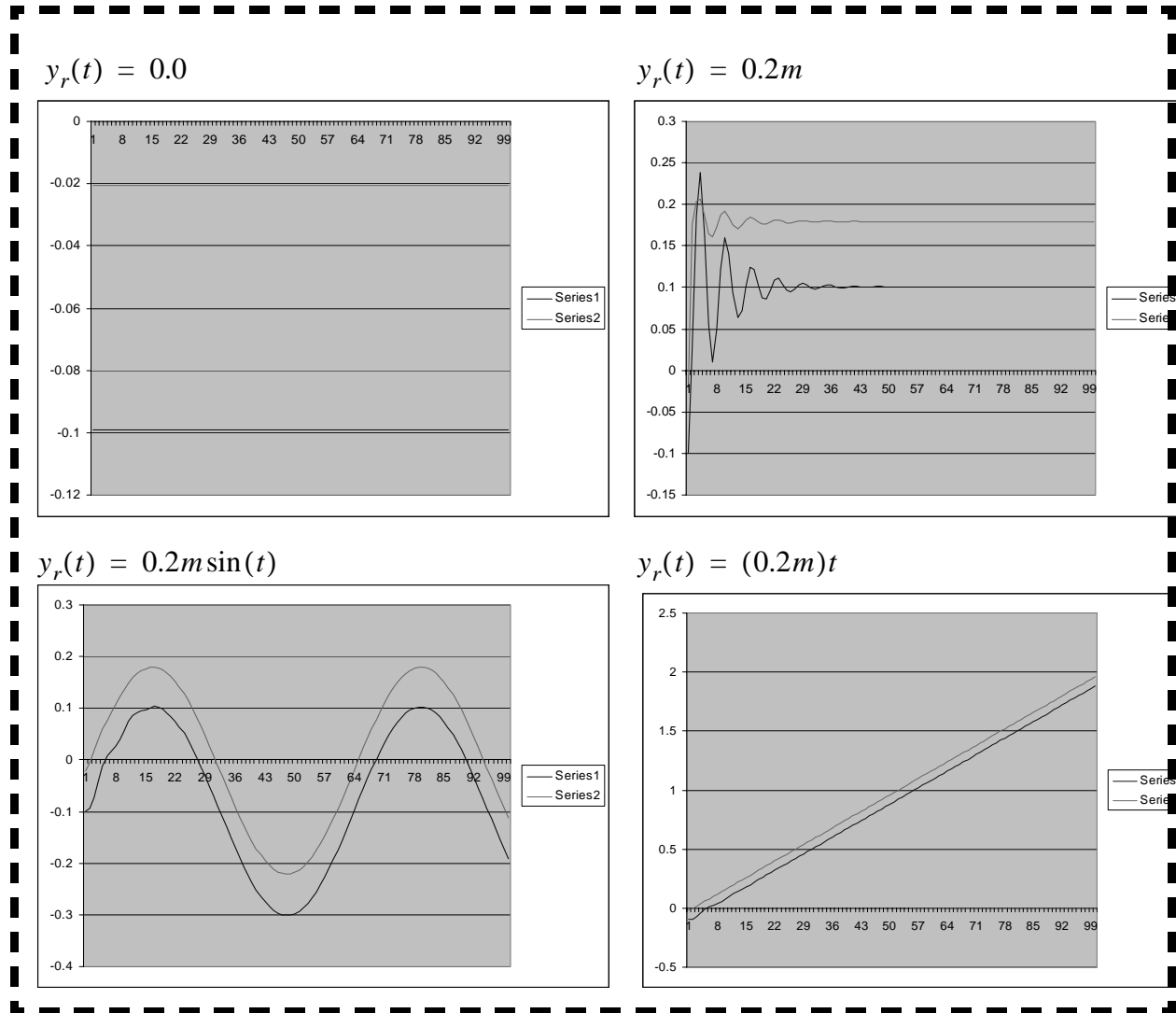


Figure 4.46 Graphs of simulation results

4.8 SUMMARY

- State variable equations are used to reduced to first order differential equations.
- First order equations can be integrated numerically.
- Higher order integration, such as Runge-Kutta increase the accuracy.
- Switching functions allow functions terms to be turned on and off to provide more complex function.
- Tabular data can be used to get numerical values.

4.9 PRACTICE PROBLEMS

1. Convert the following differential equations to state variable form.

$$\ddot{x} + 2\dot{x} + 3x + 5y = 3$$

$$\ddot{y} + \dot{y} + 6y + 9x = \sin(t)$$

2. a) Put the differential equations given below in state variable form.

b) Put the state equations in matrix form

$$\ddot{y}_1 + 2\dot{y}_1 + 3y_1 + 4\dot{y}_2 + 5y_2 + 6\dot{y}_3 + t + F = 0$$

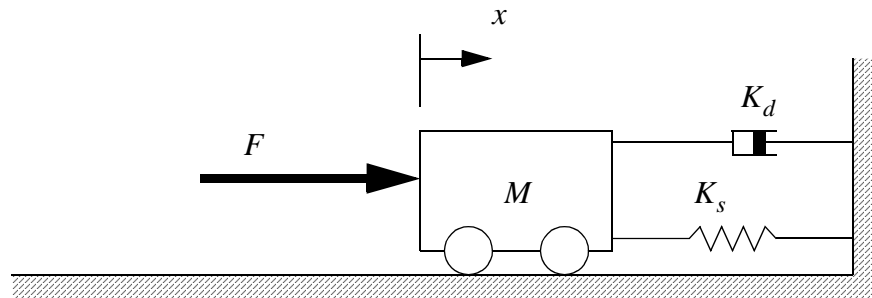
$$\dot{y}_1 + 7y_1 + 8\dot{y}_2 + 9y_2 + 10\ddot{y}_3 + 11y_3 + 5 \cos(5t) = 0$$

$$\dot{y}_1 + 12\ddot{y}_2 + 13\dot{y}_3 = 0$$

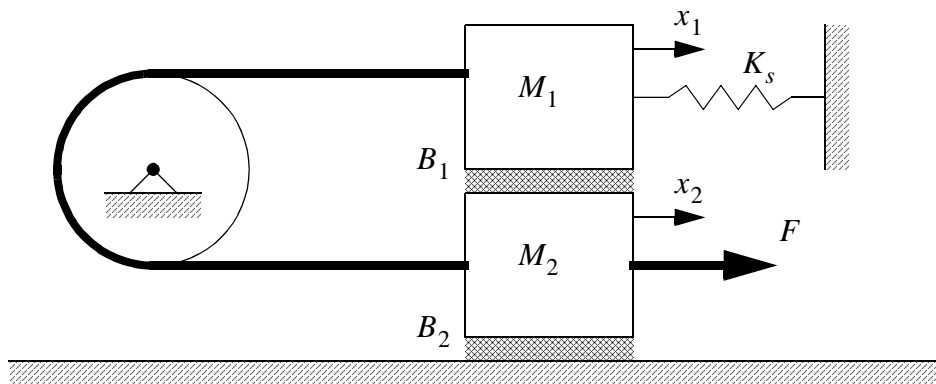
$y_1, y_2, y_3 =$ outputs

$F =$ input

3. Develop state equations for the mass-spring-damper system below.



4. The system below is comprised of two masses. There is viscous damping between the masses and between the bottom mass and the floor. The masses are also connected with a cable that is run over a massless and frictionless pulley. Write the differential equations for the system, and put them in state variable form.



5. Do a first order numerical integration of the derivative below from 0 to 10 seconds in one second steps. Assume the system starts undeflected. Write the equation for each time step.

$$\frac{d}{dt}x(t) = 5(t - 4)^2$$

6. Given the differential equation below integrate the values numerically for the first ten seconds with 1 second steps. Assume the initial value of x is 1. You may use first order or Runge Kutta integration.

$$\dot{x} + 0.25x = 3$$

7. Write a Scilab program to calculate the area under the function below using a numerical method, such as Simpson's rule. Test it by using the range from x=1 to x=1.2. Assume the sine function is in radians

$$f(x) = 5x + 2\ln\left(\frac{\sin x}{x}\right)$$

8. Write a computer program in a language of your choice (C/C++, Java, Scilab script, etc.) that will numerically integrate the differential equation below.

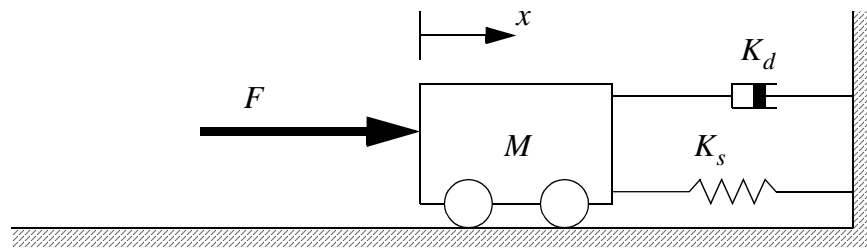
$$\ddot{\theta} + 3\dot{\theta} + 9\theta = 10$$

9. Given the following differential equation and initial conditions, draw a sketch of the first 5 seconds of the output response. The input is a step function that turns on at t=0. Use at least two different methods, and compare the results.

$$0.5\ddot{V}_o + 0.6\dot{V}_o + 2.1V_o = 3V_i + 2 \quad \text{initial conditions} \quad \begin{aligned} V_i(t \geq 0) &= 5V \\ V_o(0) &= 0V \\ \dot{V}_o(0) &= 1V \end{aligned}$$

10. a) For the mass-spring-damper system below solve the differential equation as a function of time. Assume the system starts at rest and undeflected. b) Also solve the problem using your calculator (and state equations) to verify your solution. Sketch the results.

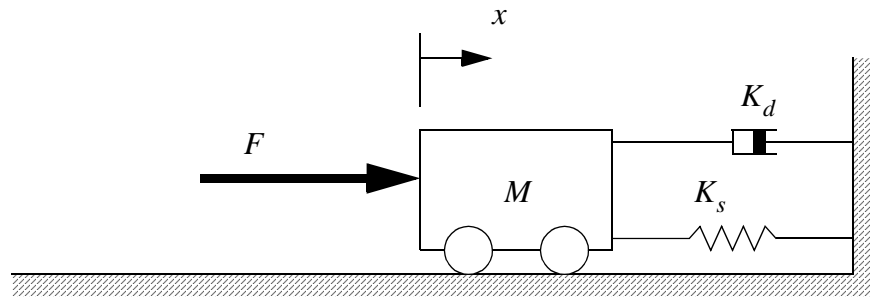
$$\begin{aligned} K_s &= 10 \\ K_d &= 10 \\ M &= 10 \\ F &= 10 \end{aligned}$$



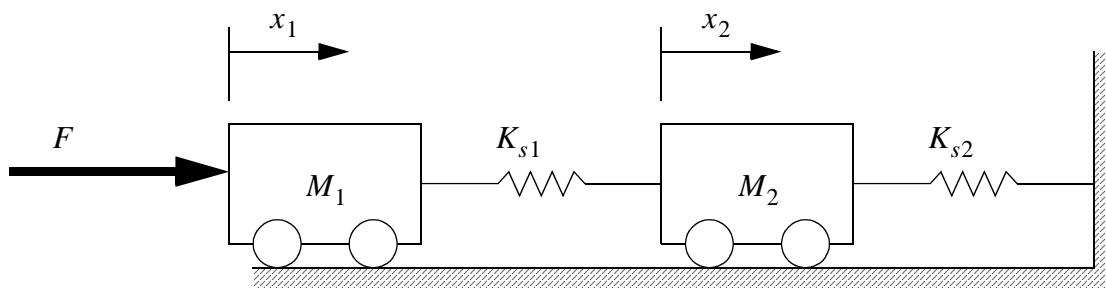
11. The mechanical system below is a mass-spring-damper system. A force 'F' of 100N is applied to the 10Kg cart at time t=0s. The motion is resisted by the spring and damper. The spring

coefficient is 1000N/m, and the damping coefficient is to be determined. Follow the steps below to develop a solution to the problem. Assume the system always starts undeflected and at rest.

- Develop the differential equation for the system.
- Solve the differential equation using damping coefficients of 100Ns/m and 10000Ns/m. Draw a graph of the results.
- Develop the state equations for the system.
- Solve the system with a first order numerical analysis using Scilab for damping coefficients of 100Ns/m and 10000Ns/m. Draw a graph of the results.
- Solve the system with a Runge Kutta numerical analysis using Scilab for damping coefficients of 100Ns/m and 10000Ns/m. Draw a graph of the results.
- Write a computer program (in C, Java or Fortran) to do the Runge Kutta numerical integration in step e). Draw a graph of the results.
- Compare all of the solutions found in the previous steps.
- Select a damper value to give an overall system damping coefficient of 1. Verify the results by numerically integrating.



12. For the mechanism illustrated in the figure below the values are $K_{s1}=K_{s2}=100\text{N/m}$, $M_1=M_2=1\text{kg}$, $F=1\text{N}$. Assume that the system starts at rest, and the springs are undeformed initially.



- Derive the differential equations for the system.
- Put the equations in state variable form.
- Put the equations in state variable matrices.
- Use a calculator to find values for x_1 and x_2 over the first 10 seconds. Provide the results in a table in 1 second intervals.
- Use Scilab to plot the values for the first 10 seconds.
- Use a Scilab program and the Runge-Kutta method to produce a graph of the first 10 seconds.

- g. Repeat step g. using the first-order approximation method.
 h. Use a C program to produce a graph of points for the first 10 seconds.

4.10 PRACTICE PROBLEM SOLUTIONS

1.

$$\begin{aligned} \dot{x} &= v \\ \dot{y} &= u \\ \dot{v} &= -2v - 3x - 5y + 3 \\ \dot{u} &= -u - 6y - 9x + \sin t \end{aligned}$$

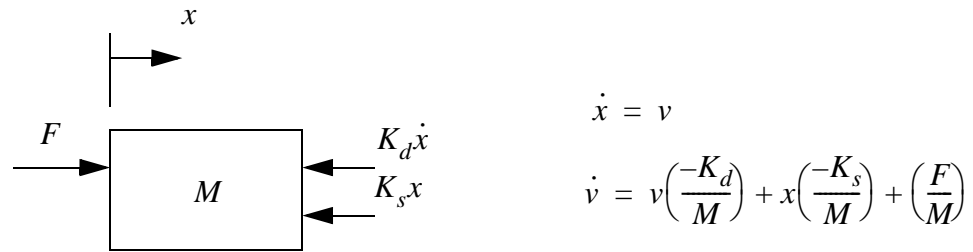
$$\frac{d}{dt} \begin{bmatrix} x \\ y \\ v \\ u \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & -5 & -2 & 0 \\ -9 & -6 & 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ v \\ u \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 3 \\ \sin t \end{bmatrix}$$

2.

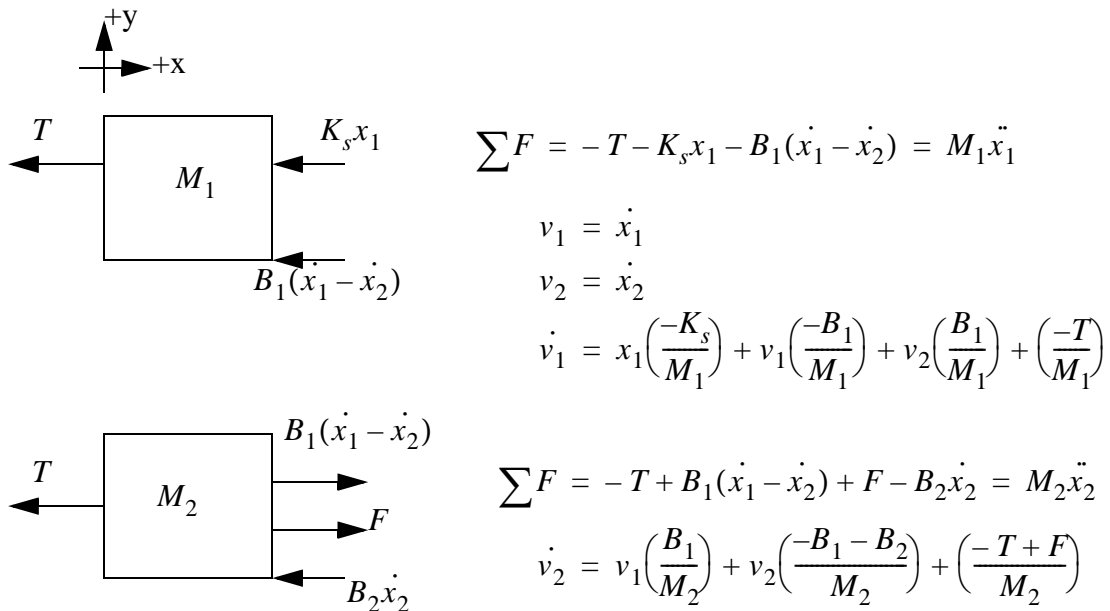
$$\begin{aligned} \dot{y}_1 &= v_1 \\ \dot{y}_2 &= v_2 \\ \dot{y}_3 &= v_3 \\ \dot{v}_1 &= -2v_1 - 3y_1 - 4v_2 - 5y_2 - 6v_3 - t - F \\ \dot{v}_2 &= \frac{-v_1}{12} - \frac{13}{12}v_3 \\ \dot{v}_3 &= \frac{-v_1}{10} - \frac{7}{10}y_1 - \frac{8}{10}v_2 - \frac{9}{10}y_2 - \frac{11}{10}y_3 - \frac{5}{10}\cos(5t) \end{aligned}$$

$$\frac{d}{dt} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -3 & -5 & 0 & -2 & -4 & -6 \\ 0 & 0 & 0 & -\frac{1}{12} & 0 & -\frac{13}{12} \\ -\frac{7}{10} & -\frac{9}{10} & -\frac{11}{10} & -\frac{1}{10} & -\frac{8}{10} & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ -t - F \\ 0 \\ -\frac{5}{10}\cos t \end{bmatrix}$$

3.



4.



5.

$h = 1$
 $x(t+h) = x(t) + h5(t-4)^2$

t	x
0	0
1	80
2	125
3	145
4	150
5	150
6	155
7	175
8	220
9	300
10	425

6.

$$x' = 3 - 0.25x$$

$$x(t+h) = x(t) + h\left(\frac{d}{dt}x(t)\right)$$

$$x(t+h) = x(t) + 1(3 - 0.25x(t))$$

$$x(t+h) = 0.75x(t) + 3$$

t	x	x'
0	1	2.75
1	3.75	2.06
2	5.81	1.55
3	7.36	1.16
4	8.52	0.870
5	9.39	0.652
6	10.0	0.489
7	10.5	0.367
8	10.9	0.275
9	11.2	0.206
10	11.4	

7.

```

// integrate.sce - A simple program to integrate a function
// To run this in Scilab use 'File' then 'Exec'.
// by: H. Jack Sept., 9, 2002
// define the function
function foo=f(x)
    foo = 5 * x + 2 * log(sin(x) / x);
endfunction
// Set the time length and step size
steps = 10;
x_start = 1;
x_end = 1.2;
x_delta = (x_end - x_start) / steps;
// Loop for rectangular integration
total = 0; // set the initial sum to zero
for i=0:steps,
    x = x_start + i * x_delta;
    total = total + f(x);
end
total = total * x_delta;
printf("Rectangular integration value %f\n", total);
// Loop for trapezoidal integration
total = 0; // set the initial sum to zero
for i=0:steps,
    x = x_start + i * x_delta;
    if i == 0 then
        total = total + f(x);
    elseif i == steps then
        total = total + f(x);
    else
        total = total + 2 * f(x);
    end
end
total = total * x_delta / 2;
printf("Trapezoidal integration value %f\n", total);
// Loop for Simpson's rule integration
total = 0; // set the initial sum to zero
even = 0;
for i=0:steps,
    x = x_start + i * x_delta;
    if i == 0 then
        total = total + f(x);
    elseif i == steps then
        total = total + f(x);
    else
        even = even + 1;
        if even > 1 then
            total = total + 4 * f(x);
            even = 0;
        else
            total = total + 2 * f(x);
        end
    end
end
total = total * x_delta / 3;
printf("Simpsons rule integration value %f\n", total);

```

8.

```
#include <stdio.h>

int main(){
    int    steps = 100,
          i;
    double theta,
           omega,
           step_t,
           theta_last,
           omega_last;

    theta = 0.0;
    omega = 0.0;
    step_t = 1.0;
    for(i = 0; i < steps; i++){
        theta_last = theta;
        omega_last = omega;
        theta = theta_last + step_t * omega_last;
        omega = omega_last + step_t*(-3 * omega_last - 9 * theta_last + 10);
        printf("%f  %f  %f \n", i+step_t, theta, omega);
    }
}
```

9.

$$\text{a) } V_o(t) = -8.331e^{-0.6t} \cos(1.960t - 0.238) + 8.095$$

$$\text{b) } \dot{V}_o = Y_o$$

$$\dot{Y}_o = -1.2Y_o - 4.2V_o + 34$$

```
// assign4_1.sce
// by: H. Jack Sept., 23, 2003

v0 = 0;           // initial conditions
y0 = 1;
X=[v0, y0];

// define the state matrix function
// the values returned are [x, v]
function foo=f(state,t)
    foo = [ state($, 2), -1.2*state($,2) - 4.2*state($,1) + 34];
endfunction

// Set the time length and step size for the integration
steps = 1000;
t_start = 0;
t_end = 10;
h = (t_end - t_start) / steps;
t = [t_start];

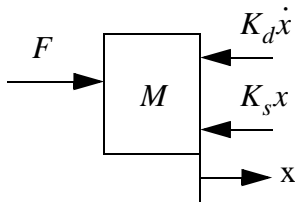
//
// Loop for integration
//
for i=1:steps,
    t = [t ; t($,:) + h];
    F1 = h * f(X($,:), t($,:));
    F2 = h * f(X($,:) + F1/2.0, t($,:) + h/2.0);
    F3 = h * f(X($,:) + F2/2.0, t($,:) + h/2.0);
    F4 = h * f(X($,:) + F3, t($,:) + h);
    X = [X ; X($,:) + (F1 + 2.0*F2 + 2.0*F3 + F4)/6.0];
end

//
// Graph the values
//
plot2d(t, X, [-2, -5], leg="position@velocity");
xlabel('Time (s)');

//
// Generate points from the given function
//
XX = [v0];
for i=1:steps,
    tt = i * h;
    XX = [XX ; -8.331 * exp(-0.6 * tt) * cos( 1.960 * tt - 0.238) + 8.095];
end
plot2d(t, XX, [-4], leg="explicit");
```

c) The two curves produced by the scilab program overlap, so the results agree.

10.



$$\sum F = F - K_d \dot{x} - K_s x = M \ddot{x}$$

$$M \ddot{x} + K_d \dot{x} + K_s x = F$$

$$10 \ddot{x} + 10 \dot{x} + 10x = 10$$

$$\ddot{x} + \dot{x} + x = 1$$

homogeneous: $\ddot{x} + \dot{x} + x = 0$

$$A^2 + A + 1 = 0$$

$$A = \frac{-1 \pm \sqrt{1 - 4(1)(1)}}{2(1)} = -0.5 \pm j \frac{\sqrt{3}}{2}$$

$$x_h = C_1 e^{-0.5t} \cos\left(\frac{\sqrt{3}}{2}t + C_2\right)$$

particular: $x_p = B$

$$0 + 0 + B = 1$$

$$x_p = 1$$

initial conditions:

$$x = x_h + x_p = C_1 e^{-0.5t} \cos\left(\frac{\sqrt{3}}{2}t + C_2\right) + 1$$

$$x' = -\frac{\sqrt{3}}{2} C_1 e^{-0.5t} \sin\left(\frac{\sqrt{3}}{2}t + C_2\right) - 0.5 C_1 e^{-0.5t} \cos\left(\frac{\sqrt{3}}{2}t + C_2\right)$$

$$x'(0) = -\frac{\sqrt{3}}{2} C_1 \sin(C_2) - 0.5 C_1 \cos(C_2) = 0$$

$$-\frac{\sqrt{3}}{2} \sin(C_2) = 0.5 \cos(C_2)$$

$$\tan(C_2) = \frac{-1}{\sqrt{3}} \quad C_2 = \text{atan}\left(\frac{-1}{\sqrt{3}}\right) = -0.5236$$

$$x(0) = C_1 \cos(-0.5236) + 1 = 0$$

$$C_1 = \frac{-1}{\cos(-0.5236)} = -1.155$$

$$x = -1.155 e^{-0.5t} \sin\left(\frac{\sqrt{3}}{2}t + 1.047\right) + 1$$

First peak is at $x=1.219$, $t=3.63s$ on graph

11.

$$\text{a) } \ddot{x} + \left(\frac{K_d}{M}\right)\dot{x} + \left(\frac{K_s}{M}\right)x = \frac{F}{M}$$

$$\text{b) } K_d = 100 \frac{N}{m} \quad x_1(t) = -0.115e^{-5t} \cos(5\sqrt{3}t - 0.524) + 0.10$$

$$K_d = 10000 \frac{N}{m} \quad x_1(t) = -0.1e^{-0.1t} + 10^{-5}e^{-999.9t} + 0.10$$

$$\text{c) } \frac{d}{dt} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{K_s}{M} & -\frac{K_d}{M} \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{F}{M} \end{bmatrix}$$

g) For 100N/m: all solutions are underdamped and overshoot at;

b) 0.1163 at $t = 0.363$ s

d) 0.1166 at $t = 0.361$ s

e) 0.1163 at $t = 0.363$ s

f) 0.1163 at $t = 0.360$ s

For 10000N/m: all solutions are overdamped. The time to reach the time constant (at 0.06321) is,

b) 0.06321 at $t = 10.001$ s

d) 0.06321 at $t = 10.000$ s

e) 0.06321 at $t = 10.000$ s

f) 0.06321 at $t = 10.0$ s

$$\text{h) } K_d = 200 \frac{Ns}{m} \quad (\text{verify with numerical integration also})$$

12.

$$\begin{aligned}
 \text{b)} \quad \dot{x}_1 &= v_1 \\
 \dot{x}_1 &= x_1 \left(\frac{-K_{s1}}{M_1} \right) + x_2 \left(\frac{K_{s1}}{M_1} \right) + \left(\frac{F}{M_1} \right) \\
 \dot{x}_2 &= v_2 \\
 \dot{x}_1 &= x_1 \left(\frac{K_{s1}}{M_2} \right) + x_2 \left(\frac{-K_{s1} - K_{s2}}{M_2} \right)
 \end{aligned}$$

$$\text{c)} \quad \frac{d}{dt} \begin{bmatrix} x_1 \\ v_1 \\ x_2 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{-K_{s1}}{M_1} & 0 & \frac{K_{s1}}{M_1} & 0 \\ 0 & 0 & 0 & 1 \\ \frac{K_{s1}}{M_2} & 0 & \frac{-K_{s1} - K_{s2}}{M_2} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ v_1 \\ x_2 \\ v_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{F}{M_1} \\ 0 \\ 0 \end{bmatrix}$$

```

// System component values
Ks1 = 100;
Ks2 = 100;
e), f), g) M1 = 1;
M2 = 1;
F = 1;

x0 = 0; // initial conditions
v0 = 0;
x1 = 0;
v1 = 0;
X=[x0, v0, x1, v1];

// define the state matrix function the values returned are [x, v]
function foo=f(state,t)
    foo = [ state($,2), -Ks1/M1*state($,1)+Ks1/M1*state($,3)+F/M1, state($,4),
Ks1/M2*state($,1)-(Ks1+Ks2)/M2*state($,3)];
endfunction

// Set the time length and step size for the integration
steps = 10000;
t_start = 0;
t_end = 10;
h = (t_end - t_start) / steps;
t = [t_start];

// Loop for integration
for i=1:steps,
    t = [t ; t($,:) + h];
    F1 = h * f(X($,:), t($,:));
    F2 = h * f(X($,:) + F1/2.0, t($,:) + h/2.0);
    F3 = h * f(X($,:) + F2/2.0, t($,:) + h/2.0);
    F4 = h * f(X($,:) + F3, t($,:) + h);
    X = [X ; X($,:) + (F1 + 2.0*F2 + 2.0*F3 + F4)/6.0];
end

// Graph the values for part e)
plot2d(t, X, [-2, -5, -7, -9], leg="position1@velocity1@position2@velocity2");
xlabel('Time (s)');

// printf the values for part f)
printf("\n\nPart e output\n\n");
for time_count=0:20,
    i = (time_count/2) / h + 1;
    printf("Point at t=%f x1=%f, v1=%f, x2=%f, v2=%f \n", time_count/2, X(i, 1),
X(i, 2), X(i, 3), X(i, 4));
end

// First order integration for part h)
X=[x0, v0, x1, v1];
t = [t_start];
for i=1:steps,
    t = [t ; t($,:) + h];
    F1 = h * f(X($,:), t($,:));
    X = [X ; X($,:) + F1 ];
end
printf("\n\nPart g output \n\n");
for time_count=0:20,
    i = (time_count/2) / h + 1;
    printf("Point at t=%f x1=%f, v1=%f, x2=%f, v2=%f \n", time_count/2, X(i, 1),
X(i, 2), X(i, 3), X(i, 4));
end

```

- h) The following subroutine is used in place of the subroutine in the program shown in Figure 4.26 and Figure 4.27.

```
//
// State Equations Calculated Here
//
void derivative(double t, double X[], double dX[]){
    dX[0] = X[1];
    dX[1] = -Ks1 / M1 * X[0] + Ks1 / M1 * X[2] + Force / M1;
    dX[2] = X[3];
    dX[3] = Ks1 / M2 * X[0] - (Ks1 + Ks2) / M2 * X[2];
}
```

4.11 ASSIGNMENT PROBLEMS

1. Write a Scilab program to implement the following equation to calculate the value of x .

$$x = \sum_{i=0}^{i < 100} 5iu(i-20)$$

where,

$$u(t) = 0 \quad \text{when} \quad t \leq 0$$

$$u(t) = 1 \quad \text{when} \quad t > 0$$

2. Write a Scilab program to implement the following equation to calculate the value of x .

$$v = \int_0^{10} \frac{F(t)}{M} dt \quad M = 10$$

where,

$$F(t) = 10 \quad \text{when} \quad t \leq 5$$

$$F(t) = 0 \quad \text{when} \quad t > 5$$

3. Write a Scilab program to implement the following equation to calculate the value of x .

$$x = \int_0^{10} f(t) dt \quad f(t) = 5 \ln(t)$$

4. Write a Scilab program to integrate the area under the function below using a numerical method, such as Simpson's rule. Find the area from 1 to 2.

$$f(x) = 5x + 2 \ln\left(\frac{\sin x}{x}\right)$$

5. Numerically integrate one time step of the differential equation below using a) first order integration and b) Runge Kutta integration.

$$\ddot{\theta} + 3\dot{\theta} + 9\theta = 10$$

6. Convert the third order differential equation below to state equation form. With a numerical method of your choice, find the state of the system 1 second later. Show all calculations.

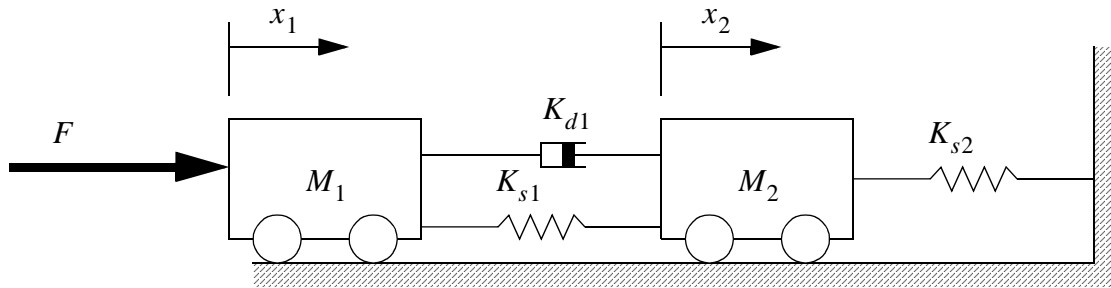
$$\ddot{x} + 4\dot{x} + 2x + 5x = 10$$

7. The differential equation below describes a first order system that starts with an initial value of $k=20$. Find the state at two milliseconds using a) explicit integration, b) first order numerical integration and c) Runge-Kutta integration. For the numerical methods use a timestep of $h=0.001$ s. ----> The final results must be put in a table for easy comparison.

$$\dot{k} + 10k = 5 \quad k(0) = 20$$

8. For the mechanism shown in the figure below the values are $K_{s1}=K_{s2}=100$ N/m, $K_{d1}=10$ Nm/s, $M_1=M_2=1$ kg, $F=1$ N. Assume that the system starts at rest, and the springs are undeformed ini-

tially.



- Derive the differential equations for the system.
- Put the equations in state variable form.
- Put the equations in state variable matrices.
- Use a calculator or Scilab using first order integration to find values for x_1 and x_2 over the first 10 seconds. Provide the results in a table in 1 second intervals.
- Use Scilab to plot the values for the first 10 seconds using the values obtained in part d.
- Use a Scilab program and the Runge-Kutta method to produce a graph of the first 10 seconds.
- Use a C program to produce a list of points for the first 10 seconds.
- Compare the results found in steps d, f and g in a table.

9. Explicitly solve the following differential equation. Verify the result numerically.

$$\dot{v} + 20v^2 = 200$$