Leveraging the power of Java and Matlab to solve ODE's

Dr. Mohammad Rafiq Muqri, DeVry University, Pomona

Hasan Muqri, UCLA

Prof. Shih Ek Chng, DeVry University, Pomona

Professor College of Engineering and Information Science

Leveraging the power of Java and Matlab to solve ODEs

Abstract

Ordinary Differential Equations (ODE) are used to model a wide range of physical processes. An ODE is an equation containing a function of one independent variable and its ordinary derivatives. This paper presents the development and application of a practical teaching module introducing java programming techniques to electronics, computer, and bioengineering students before they encounter digital signal processing and its applications in junior and senior level courses. This paper will focus primarily on how to solve ODEs using Java and Matlab programming tools.

There are two basic types of boundary condition categories for ODEs – initial value problems and two-point boundary value problems. Initial value problems are simpler to solve because you only have to integrate the ODE one time. The solution of a two- point boundary value problem usually involves iterating between the values at the beginning and end of the range of integration. Runge-Kutta schemes are among the most commonly used techniques to solve initial-value problem ODEs.

Matlab also presents several tools for modeling linear systems. These tools can be used to solve differential equations arising in such models, and to visualize the input-output relations. This paper attempts to describe how to use Java programming tool to solve initial value problems of ordinary differential equations (ODEs) using the Runge-Kutta scheme. It will also discuss how to represent initial value problems and demonstrate how to apply Matlab's ODE solvers to such problems. It will also explain how to select a solver and how to specify solver options for efficient, customized execution.

This paper provides an introduction to the Ordinary Differential Equations(ODEs). After a quick overview of selected numerical methods for solving differential equations using Matlab, we will briefly give an account of Euler and modified Euler methods for solving first order differential equations. This will be followed by numerical method for systems specially Runge-Kutta schemes and applications of second order differential equations in mechanical vibrations and electric circuits by leveraging the power of Java and Matlab.

This paper will explain how this learning and teaching module is instrumental for progressive learning of students; the paper will also demonstrate how the numerical and integral algorithms are derived and computed through leverage of the java data structures. As a result, there will be a discussion concerning the comparison of Java and Matlab programming as well as students' feedback. The result of this new approach is expected to strengthen the capacity and quality of our undergraduate degree programs and enhance overall student learning and satisfaction.

Introduction to ODEs

Differential equations are used to model a wide range of physical processes; technology students will use them in chemistry, biophysics, mechanics, thermodynamics, electronics, and

almost every other scientific and engineering discipline. An ODE is used to express the rate of change of one quantity with respect to another. One defining characteristic of an ODE is that its derivatives are a function of one independent variable. The order of a differential equation is defined as the order of the highest derivative appearing in the equation and ODE can be of any order. A general form of a first-order ODE can be written in the form

$$dy/dx + p(x)y + q(x) + r = 0$$

where $p(x)$ and $q(x)$ are functions of x . This equation can be rewritten as shown below

$$d/dx(y) + y p(x) = -q(x) - r$$

where r is zero. A classical integrating factor method can be used for solving this linear differential equation of first order. The integrating factor is $(\exp)^{\int p \, dx}$.

Euler Method

Graphical methods produce plots of solutions to first order differential equations of the form $y' = f(x,y)$, where the derivative appears on the left side of the equation. If an initial condition of the form $y(x_0) = y_0$ is also specified, then the only solution curve of interest is

$$y' = f(x,y)$$

the one that passes through the initial point (x_0, y_0) . For the first-order initial-value problem the popular graphical method also known as Euler method can be used that satisfies the formula given below

$$y_{n+1} = y_n + hf(x_n, y_n)$$

which can also be written as $y_{n+1} = y_n + h(y'_n)$, where the approximate solution at x_n is designated by $y(x_n)$, or simply y_n . The true solution at x_n will be denoted by either $Y(x_n)$ or Y_n . Note that once y_n is known, equation $y' = f(x,y)$ can be used to obtain y'_n as

$$y'_n = f(x_n, y_n) \quad (1.0)$$

Modified Euler's Method:

This is a simple predictor-corrector method that uses Euler's method as the predictor and then uses the average value of y' at both the left and right end points of the interval $[x_n, x_{n+1}]$ ($n = 0, 1, 2, \dots$) as the slope of the line element approximation to the solution over that interval. The resulting equations are:

$$\text{predictor: } y_{n+1} = y_n + h(y'_n)$$

$$\text{corrector: } y_{n+1} = y_n + h/2 \dots \dots$$

For notational convenience, we designate the predicted value of y_{n+1} by p_{n+1} .

Since $y'_n = f(x_n, y_n)$, it then follows that

$$p_{y'_{n+1}} = f(x_{n+1}, y_{n+1}) \quad (1.1)$$

and the modified Euler method becomes

$$\text{predictor: } p_{y_{n+1}} = y_n + h(y'_n)$$

$$\text{corrector: } y_{n+1} = y_n + h/2(p_{y'_{n+1}} + y'_n) \quad (1.2)$$

Example 1

Use the modified Euler's method to solve

$y' - y + x = 0$; $y(0) = 2$ on the interval $[0,1]$ with $h = 0.1$

Using Matlab dsolve function we get

```
>> dsolve('Dy = y - x', 'y(0)= 2','x')
```

the solution $y = x + \exp(x) + 1$

Euler's modified Numerical Method

Here $f(x,y) = y - x$, and $y_0 = 2$. From equation (1.0), we have $y_0' = f(0,2) = 2 - 0 = 2$

Using equations (1.1) and (1.2), we compute

For $n = 0$, $x_1 = 0.1$

$$py_1' = y_0' + h y_0' = 2 + 0.1(2) = 2.2$$

$$py_1' = f(x_1, py_1) = f(0.1, 2.2) = 2.2 - 0.1 = 2.1$$

$$y_1 = y_0 + h/2(py_1' + y_0') = 2 + 0.05(2.1 + 2) =$$

$$2.205 \quad y_1' = f(x_1, y_1) = f(0.1, 2.205) = 2.205 - 0.1$$

$$= 2.105$$

It can be shown that

For $n = 1$, $x_2 = 0.2$

$$py_2' = y_1' + h y_1' = 2.4155$$

$$py_2' = f(x_2, py_2) = f(0.2, 2.4155) =$$

$$2.2155 \quad y_2 = y_1 + h/2(py_2' + y_1') =$$

$$2.421025$$

$$y_2' = f(x_2, y_2) = f(0.2, 2.421025) = 2.221025 \text{ and so on}$$

Instead of computing x_n , y_n , and true solution etc. at different points, the following Matlab script can be used to obtain the Modified Euler solution as depicted in Figure below.

```
% nsteps = 150
```

```
Final_Time=10.0;
```

```
stepsize=Final_Time /
```

```
nsteps; clear x y
```

```
exactSolution y(1)=2.205;
```

```
x(1)=0.1;
```

```
exactSolution(1)=4.718281
```

```
8; for k=1:nsteps
```

```
    x(k+1)=x(k) +
```

```
    stepsize;
```

```
    y(k+1)=y(k) + stepsize*((y(k) - x(k)));
```

```
    exactSolution(k+1) = exp(x(k+1)) + x(k+1) + 1;
```

```
end
```

```
plot(x,y); % default line color is blue
```

```
title('Modified Euler Method solution of yprime = y - x, y(0) = 2')
```

```
hold on
```

```
plot(x,exactSolution,'g'); % g for green
```

```
line legend('Euler solution','Exact
```

```

solution')
hold off
error=norm(y-exactSolution)/norm(exactSolution);

```

A general form of a second-order ODE is shown as follows:

$$d^2y/dx^2 + p(x)dy/dx + q(x)y + r(x) + s = 0 \quad (1.2)$$

Any high order ODE can be expressed as a coupled set of first-order differential equations. For example the second-order ODE given in equation (1.2) can be reduced to a coupled set of two first-order differential equations.

$$\begin{aligned} d/dx(dy/dx) &= -p(x)dy/dx - q(x)y - r(x) - s \\ d/dx(y) &= dy/dx \end{aligned} \quad (1.3)$$

Java's ODE Class

We will use and demonstrate a class named ODESolver that will define a number of methods² used to solve ODEs and also subclasses that can be used to represent modeling real world applications.

The ODE class will declare a field to store the number of first order equations. The other declared field will store the number of free variables (those that are not specified by boundary

conditions at the beginning of the integration range). For initial value problems, the number of free variables will be zero. Two-point boundary problems will have one or more free variables. The ODE class will declare methods to set the conditions at the start of the integration range and to compute the error at the end. These methods are ODE-specific, but since the ODE class represents a generic ODE, they will be implemented as stubs, and the required ODE subclasses will override those methods according to their needs.

The Runge-Kutta family of methods are step-wise integration algorithms that are expected to give good results as long as very high accuracy is not required. Starting from an initial condition, the ODE is solved at discrete steps over the desired integration range. Consider a simple first order differential equation which can be integrated in a step-wise manner to solve for the dependent variable y .

$$\begin{aligned} dy/dx &= f(x,y) \\ dy &= dx \, f(x,y) \end{aligned} \tag{1.4}$$

Replacing the derivative by its delta form, we can write

$$\Delta y = y_{n+1} - y_n = \Delta x \, f(x,y) \tag{1.5}$$

Equation (1.5) is the general form of the equation that is solved. Using Taylor series expansion on the Euler's method, we find that it is first order accurate in Δx and is only useful for linear or nearly linear functions. If the slope of the $f(x,y)$ curve changes between changes x_n and x_{n+1} , the Euler's method will compute an incorrect value for y_{n+1} . This is why we choose Runge-Kutta method since it performs a successive approximation of y_{n+1} by evaluating the $f(x,y)$ function at different locations between x_n and x_{n+1} .

There are numerous Runge-Kutta schemes of various orders of accuracy, we will use one of the popular fourth-order algorithm accurate in Δx . This algorithm consists of five steps, four successive approximations of Δy and a fifth step that computes y_{n+1} based on a linear combination of the successive approximations.

The fourth order Runge-Kutta method solution process steps are as follows:

1. Compute Δy_1 using Euler's method
 $\Delta y_1 = \Delta x \, f(x_n, y_n)$
2. Compute Δy_2 by evaluating $f(x,y)$ at $(x_n + \frac{1}{2} \Delta x, y_n + \frac{1}{2} \Delta y_1)$
 $\Delta y_2 = \Delta x \, f(x_n + \frac{1}{2} \Delta x, y_n + \frac{1}{2} \Delta y_1)$
3. Compute Δy_3 by evaluating $f(x,y)$ at $(x_n + \frac{1}{2} \Delta x, y_n + \frac{1}{2} \Delta y_2)$
 $\Delta y_3 = \Delta x \, f(x_n + \frac{1}{2} \Delta x, y_n + \frac{1}{2} \Delta y_2)$
4. Compute Δy_4 by evaluating $f(x,y)$ at $f(x_n + \Delta x, y_n + \Delta y_3)$

$$\Delta y_4 = \Delta x f(x_n + \Delta x, y_n + \Delta y_3)$$

5. Compute y_{n+1} using a linear combination of Δy_1 thru Δy_4
 such as $\Delta y_1/6 + \Delta y_2/3 + \Delta y_3/3 + \Delta y_4/6$
 $y_{n+1} = y_n + 1/6(\Delta y_1 + 2\Delta y_2 + 2\Delta y_3 + \Delta y_4)$

Let us define the class named ODESolver and create the rungekutta4() method in it. This method takes three arguments. The first argument is an ODE object (or an ODE subclass object). The other two input arguments are the range over which the integration will take place and the increment to the independent variables which will be held constant throughout the entire integration. The number of steps here is not a user-specified value but is computed based on the range and dx arguments. The integration will stop if the step number reaches the MAX_STEPS parameters defined in the ODE class. Given below is an example of an instructor lead program which the student edited, compiled and displayed the output.

The purpose of this program is to show how a general Java workhorse ODE class code and other Runge-Kutta methods can be introduced at an early stage to engineering technology students for the purpose of solving initial value problems. These methods and tools can then be used further in future control system and signal processing courses.

The complete class definition and methods for the classes ODE and ODESolver are included in Appendix A for reference. The examples illustrating the application of second-order ODEs are included in Appendix B.

Matlab Methods

Electrical Circuit Application

Now that we know how to solve second-order linear equations using Runge-Kutta methods for Spring Vibration, we are in a position to analyze the RLC circuit containing a resistor R , an inductor L , and a capacitor C , in series with an electromotive force E supplied by a battery or generator. If the charge on the capacitor at time t is $Q(t)$, then the current is the rate of change of charge with respect to t . According to Kirchhoff's voltage law the sum of the voltage drops across these components is equal to the supplied voltage $E(t)$:

$$L \frac{dI}{dt} + R I + Q / C = E(t) \quad (1.7)$$

Since $I = dQ/dt$, this equation becomes

$$L \frac{d^2Q}{dt^2} + R \frac{dQ}{dt} + Q/C = E(t) \quad (1.8)$$

This is a second-order ODE with time as its independent variable. A differential equation for the current can be obtained by differentiating Equation (1.7) with respect to t .

$$L \frac{d^2 I}{dt^2} + R \frac{dI}{dt} + \frac{I}{C} = E'(t) \quad (1.9)$$

Circuit Application Example 1

Find the charge and current at time in the RLC series circuit if resistance $R = 40$ ohms, inductance $L = 1$ H, capacitance $C = 16 \times 10^{-4}$ Farads, supply voltage $E(t) = 100 \cos 10t$, and the initial charge and current are both 0. Substituting R , L , C , and E in Equation (1.8) we get

$$\frac{d^2 Q}{dt^2} + 40 \frac{dQ}{dt} + 625Q = 100 \cos 10t \quad (1.10)$$

Equation (1.10) is a second-order linear differential equation and its auxiliary equation is with roots

$$r_1 = -20 + 15i$$

$$r_2 = -20 - 15i$$

$$r^2 + 40r + 625 = 0$$

so the solution of the complimentary solution is $Q_c(t) = e^{-20t} (c_1 \cos 15t + c_2 \sin 15t)$ (1.11)

For the method of undetermined coefficients we try the particular solution

$$Q_p(t) = (A \cos 10t + B \sin 10t)$$

$$Q_p'(t) = -10 A \sin 10t + 10B$$

Substituting in Equation (1.10) and comparing coefficients we get $A = 84/697$ and $B = 64/697$

$$\text{So a particular solution is } Q_p(t) = 1/697(84 \cos 10t + 64 \sin 10t) \quad (1.12)$$

$$\text{So the general solution is } Q(t) = Q_c(t) + Q_p(t)$$

$$(1.13) \quad Q(t) = e^{-20t} (c_1 \cos 15t + c_2 \sin 15t) + 1/697(84 \cos 10t + 64 \sin 10t)$$

$$(1.14) \text{ Since } I = dQ/dt,$$

Differentiating Equation (1.14) and substituting the given the initial condition $Q(0) = 0$, we determine $c_1 = -84/697$, $c_2 = -464/2091$

Thus the formula for the charge is

$$Q(t) = e^{-20t} (-84/697 \cos 15t - 464/2091 \sin 15t) + 1/697(84 \cos 10t + 64$$

$\sin 10t)$ And the expression for the current is

$$I = dQ/dt = 1/2091 [e^{-20t} (-1920 \cos 15t + 13060 \sin 15t) + 120(16 \cos 10t - 21 \sin 10t)]$$

The total response³ of an Electrical circuit consists of two parts namely, forced (or steady state) and transient.

Transient Analysis Example 2 using the Matlab Symbolic ToolBox

```
>> % The differential integral equation for the electric circuit is given as follows:
```

```
>> %CdV/dt + V/R = 1/L * integral V(t)dt = iL(0+) = 0.2exp(-1000t)
```

```
>> %Second Order Equation with initial conditions
```

```
>> %V(0+) =0, V'(0+)=dV(0+)/dt = 0.2* 1000000 V/s
```

```
>> clear
```

```
>> syms IL V t
```

```
>> V = dsolve('(1e-6)*D2V + (4e-3)*DV + 250*V = -200*exp(-1000*t)', 'DV(0)=0.2e6', 'V(0)=0');
```

```
>> IL=(250)*int(V,t,0,t);
```

```
>> pretty(vpa(IL,4))
```

```
-0.00082288272275476970207819249480963 exp(-2000.0 t)
```

```
(246.0 cos(15684.38714134693145751953125 t) - 246.0 exp(1000.0 t) +
```

```
15.684387141343904659152030944824 sin(15684.38714134693145751953125 t))
```

```
>> ezplot(IL,[0 2e-3])
```

MATLAB has quite a few different ODE solvers, including ode23 which uses simultaneously second and third order Runge-Kutta formulas to make estimates of the error, and calculate the time step size. Since the second and third order Runge-Kutta methods require less steps, ode23 is considered less expensive in terms of computation demands than ode45, but it is also lower order.

These few examples demonstrate how students can be introduced not only to classic differential equations and numerical methods for solving differential equations, but also to basic concepts of object based programming with Java. Given time, it can be proved that this group of students who are currently taking the junior-level signal and systems course have a far better understanding than the group of students who were never exposed to this teaching module.

Student Feedback

We administered an anonymous questionnaire to obtain feedback from the students in regards to the choice of using Java and Matlab for solving ODEs. We have provided some selected questions from the questionnaire and their respective responses that apply particularly to the choice of programming language and lessons learned by early exposure to this module. Note that all the students' comments were encouraging and positive; there were almost no negative responses obtained.

- Do you feel the learning module reinforced class material in a good, bad, or indifferent way?

Sample Responses:

- Excellent. The java programs have greatly enhanced the overall class information.
 - In an indifferent way. I like the computing with Java but it was slightly cumbersome.
 - Why java, why not C++? I was able to retrofit them using C++, although I like java's GUI development.
 - These lab modules really set the pace for the class and enhance the experience.
- Has the teaching module been straight forward and easy to understand? Do you have suggestions for improvement?

Sample Responses:

- Pretty straight forward.
 - Add a section on Transient Analysis using the Matlab Symbolic Toolbox
- Did the Java software in conjunction with Matlab aid in understanding of the lab exercises?

Sample Responses:

- MatLab was very easy to use.
 - Yes it did. The first lab was probably difficult because I had to brush up on my Java basics.
 - Yes. The use of java to solve ODEs was like a different outlook, but probably not the best.
- What other aspects do you feel would have been useful in this teaching module?

Sample Responses:

- When to use Runge-Kutta and when not to use it.
- Few more examples explaining use of ODE45 and ODE23

Summary and Conclusions

We can see from the above responses that students enjoyed this teaching module overall. Their responses to the choice of programming language were also fairly positive, but more mixed as expected. The incorporation of many demonstrations and interactive lab experiences with Java and Matlab into this teaching module proved to be effective.

I was fortunate to monitor and discuss the experiences of these students who took the senior project capstone class with me last session. As expected they were very positive with the outcome and obtained an enhanced understanding of the application of second order differential equations which they attributed to their early exposure to Java Object Oriented Programming as well as the reinforcement of the usage of Matlab functions to solve ODEs.

In conclusion, it can be stated that with proper guidance, monitoring, and diligent care, engineering technology students can be exposed earlier to Java data structures and the basics of Matlab, and Simulink. This will go a long way in motivating them, eliminating their fear, improving their understanding, and enhancing their quality of education.

Bibliography

1. Muqri, M., Shakib, J., *A Taste of Java-Discrete and Fast Fourier Transforms*, American Society for Engineering Education, AC 2011-451.
2. Shakib, J., Muqri, M., *Leveraging the Power of Java in the Enterprise*, American Society for Engineering Education, AC 2010-1701.
3. Hambley, Alan R., *Electrical Engineering Principles and Applications*, Prentice Hall, 2011.
4. Blinowska, K., Durka, P., *The application of wavelet transform and matching pursuit to the time-varying EEG signals*, in *Intelligent Engineering Systems Through Artificial Neural Networks*, Editors, Dagli & Fernandez, volume 4, pp. 535-540, ASME Press, New York, 1994.
5. Deitel, H.M., Deitel, P.J., *Java How to program*, Prentice Hall, 2003.
6. Palmer G., *Technical Java - Developing Scientific and Engineering Applications*, Prentice Hall, 2003.

Appendix A: Java Classes ODE and ODE Solver

```
public class ODE {
//This is used to allocate memory to the
//x[] and y[][] arrays

    public static int MAX_STEPS = 999;

//numEqns = number of 1st order ODEs to be solved
//numFreeVariables = number of free variables at domain boundaries
//x[] = array of independent variables
//y[][] = array of dependent variables

    private int numEqns, numFreeVariables;
    private double x[];
    private double y[][];

    public ODE(int numEqns, int numFreeVariables)
    {
        this.numEqns = numEqns;
        this.numFreeVariables = numFreeVariables;
        x = new double[MAX_STEPS];
        y = new double[MAX_STEPS][numEqns];
    }

//these methods return the value of some of the fields public

    int getNumEqns( )

    {
        return numEqns;
    }

    public int getNumFreeVariables()
    {
        return numFreeVariables;
    }

    public double[] getX()
    {
```

```

        return x;
    }

    public double[][] getY()
    {
        return y;
    }
    public double getOneX(int step)
    {
        return x[step];
    }
    public double getOneY(int step, int equation)
    {
        return y[step][equation];
    }

//This method lets you change one of the dependent or independent variables public
void setOneX(int step, double value)
{
    x[step] = value;
}

public void setOneY(int step, int equation, double value)
{
    y[step][equation] = value;
}

//These methods are implemented as stubs.
//Subclasses of ODE will override them

    public void getFunction(double x, double dy[], double ytmp[]) {}

    public void getError(double E[], double endY[]) {}

    public void setInitialConditions(double V[]) {}
}

public class ODESolver {
    public static int rungeKutta4(ODE ode, double range, double dx) {
        //Define some convenience variables to make the code more readable

```

```

int numEqns = ode.getNumEqns();
double x[] = ode.getX();
double y[][] = ode.getY();

//Define some local variabls and arrays int i, j, k;
double scale[] = { 1.0, .5, .5, 1.0};
double dy[][] = new double[4][numEqns];
double ytmp[] = new double[numEqns];

//Integrate the ODE over the desired range
//Stop if you are going to overflow the matrices

i = 1;
while(x[i-1] < range && i < ODE.MAX_STEPS - 1)
{
//Increment independent variable. Make sure it
//doesn't exceed the range.

x[i] = x[i-1] + dx;
if(x[i] > range)
{
x[i] = range;
dx = x[i] - x[i-1];
}

//First Runge-Kutta step
ode.getFunction(x[i-1], dy[0], y[i-1]);

//Runge-Kutta steps 2-4

for(k = 1; k < 4; ++k)
{
for(j = 0; j < numEqns; ++j)
ytmp[j] = y[i-1][j] + scale[k]*dx*dy[k-1][j];
ode.getFunction(x[i-1] + scale[k]*dx, dy[k], ytmp);
}
//Update the dependent variables

for(j = 0; j < numEqns; ++j)
{
y[i][j] = y[i-1][j] + dx*(dy[0][j] + 2.0*dy[1][j] + 2.0*dy[2][j] + dy[3][j])/6.0;

```

```
    }  
    //Increment i  
    ++i;  
  
} //end of while loop  
//return the number of steps  
//computed  
return i;  
}  
}
```


Appendix B: Application Examples of second-order ODEs

We will begin with the first basic mathematical model application of mechanical spring vibration resulting in second-order linear differential equation. According to Hooke's Law if the spring is stretched (or compressed) x units from its natural length, then it exerts a restoring force F that is proportional to x :

$$F = -kx$$

where k is a positive constant (called the **spring constant**). If we ignore any external resisting forces (due to air resistance or friction) then, by Newton's Second Law (force equals mass times acceleration), we have

$$F = ma = m \, d^2x/dt^2$$

Combining these two equations we

$$\begin{aligned} &\text{get } m \, d^2x/dt^2 + kx \\ &= 0 \end{aligned}$$

as the general equation for the motion of an un-damped spring. Let us next consider the motion of a spring that is subject to a frictional force or a damping force. An example is the damping force supplied by a shock absorber in a car or a bicycle. We assume that the damping force is proportional to the velocity of the mass and acts in the direction opposite to the motion.

damping Force = $\mu dx/dt$ where μ is a positive constant, called the **damping constant**. Thus, in this case, Newton's

Second Law gives the general equation of motion for a spring

$$m \, d^2x/dt^2 + \mu dx/dt + kx = 0 \quad (1.)$$

6) This is a second-order ODE with time as its independent variable.

Equation (1.6) is a second-order linear differential equation and its auxiliary equation is

$$mr^2 + \mu r + k = 0$$

Case 1: $\mu^2 - 4mk > 0$
(overdamping)

This case corresponds to r_1 and r_2 are distinct real roots. Since μ , m , and k are all positive, notice that oscillations do not occur. (It's possible for the mass to pass through the equilibrium position once, but only once.) This is because means that there is a strong damping force (high-viscosity oil or grease) compared with a weak spring or small mass.

Case 2: $\mu^2 - 4mk = 0$ (critical damping) This case corresponds to equal roots.

$$r_1 = r_2 = -\mu/2m$$

Case 3: $\mu^2 - 4mk < 0$ (under damping) This case corresponds to complex roots:

The solution is $e^{(-\mu/2m)t} (c_1 \cos \omega t + c_2 \sin \omega t)$

$$r_1 = -\mu/2m + i \omega$$

$$r_2 = -\mu/2m - i \omega$$

$$\omega^2 = (4mk - \mu^2)/4m^2$$

We see that there are oscillations which are damped by the factor $e^{(-\mu/2m)t}$ that is the motion decays to zero as time increases.

Forced Vibrations

Suppose that, in addition to the restoring force and the damping force, the motion of the spring is affected by an external force. Then Newton's Second Law gives

$$m \frac{d^2x}{dt^2} + \mu \frac{dx}{dt} + kx = F(t) \quad (1.6A)$$

For equation (1.6A), the motion of the spring can be determined by the methods of Nonhomogeneous Linear Equations. A commonly occurring type of external force is a periodic force function

$$F(t) = F_0 \cos \omega t$$

Where ω_0 is not equal to ω , and $\omega_0^2 = k/m$, we can use the method of undetermined coefficients to solve for $x(t)$.

If $\omega_0 = \omega$, then the applied frequency reinforces the natural frequency and the result is vibrations of large amplitude. This is the phenomenon of resonance.

Solving the Mechanical Spring Vibration ODE using Java

In order to solve mechanical spring vibration initial value problem, we use a SpringODE class that is a subclass of the ODE class. The SpringODE class represents the equations of motion for

a damped spring. In addition to the members it inherits from the ODE class, the SpringODE class defines three new fields representing the spring constant k , damping constant μ (μ) and mass m . The SpringODE class source code is shown below.

```
public class SpringODE extends
ODE
{
    double k, mu, mass;

    public SpringODE(double k, double mu, double mass)
    {
        super(2,0);
        this.k = k;
        this.mu =
        mu;
        this.mass = mass;
    }

    //The getFunctions() method returns the right hand sides
    //of the two first-order damped spring ODEs

    public void getFunction(double x, double dy[], double ytmp[])
```

```

    {   dy[0] = -k*ytmp[1]/mass - mu*ytmp[0]/mass;

        dy[1] = ytmp[0];

    }

    //This method initializes the dependent variables at the start
    //of the integration range

    public void setInitialConditions(double V[])
    {

        setOneY(0, 0, 0.0);

        setOneY(0,1,
            V[0]); setOneX(0,
            0.0);

    }

}

```

Now let us apply the fourth-order Runge-Kutta solver to compute the motion of a mechanical vibration damped spring. A driver program named RK4Spring.java is listed below which creates a SpringODE object and calls the rungeKutta4() method on that object. The initial conditions are set such that the spring is extended 0.3 meters from its equilibrium position. The ODE is integrated from $t = 0$ to $t = 5.0$ seconds using a step size of 0.1 seconds.

The RK4Spring class source code is shown below:

```

public class RK4Spring {

    public static void main(String[] args) {

        //create a springODE object

        double mass = 1.0;
    }
}

```

```

double mu = 1.5;

double k = 20.0;

SpringODE ode = new SpringODE(k, mu, mass);

//The string is initially stretched .3 meters from its
//equilibrium position

    double V[] = {-.2};

    ode.setInitialConditions(V);


//solve the ODE over the desired range using a constant step size


double dx = .1;

double range = 5.0;


int numSteps = ODESolver.rungeKutta4(ode, range, dx);


//print out the results


System.out.println("i   t       dxdt       x");

for(int i = 0; i < numSteps; ++i) {

    System.out.println(

        "" + i + " " + ode.getOneX(i) +

        " " + ode.getOneY(i, 0) + " " + ode.getOneY(i, 1));

    }

}

}

```

Here are the sample partial Runge-Kutta based Spring Vibration results:

i	t	dxdt	x
0	0.0	0.0	-0.3
1	0.1	0.538665625	-0.27194375
2	0.2	0.9040381698347981	-0.19815123462076822
3	0.3	0.30000000000000004	1.0535390575206194
4	0.4	0.9899206101842039	0.005334462813926105
5	0.5	0.7544550529206122	0.09370828010372466
6	0.6	0.4140399394391337	0.1526777707608392
7	0.7	0.04542041921295331	0.17557073219996705
8	0.7999999999999999	-0.2801903348419779	0.1632289474328672
9	0.8999999999999999	-0.509340552399849	0.12280882327205371
10	0.9999999999999999	-0.6136251762177285	0.0655962652795802
11	1.0999999999999999	-0.5913852518994047	0.004371849709903133
12	1.2	-0.4642885675928221	-0.049130160373204314
13	1.3	-0.27012807917110837	-0.08621818207375898
14	1.4000000000000001	-0.05367906200735442	-0.10240643760386174
15	1.5000000000000002	0.1424459073618853	-0.0976484798013848
16	1.6000000000000003	0.2852745003127429	-0.07572786031032912
17	1.7000000000000004	0.3561518169094504	-0.04303446653963582
18	1.8000000000000005	0.35215314298667594	-0.007035390525057854
19	1.9000000000000006	0.28442869949228267	0.025238030536776354