

# Ordinary Differential Equations

W. E. Schiesser  
Iacocca D307  
111 Research Drive  
Lehigh University  
Bethlehem, PA 18015

(610) 758-4264 (office)  
(610) 758-5057 (fax)

wes1@lehigh.edu

[http://www.lehigh.edu/~ wes1/wes1.html](http://www.lehigh.edu/~wes1/wes1.html)

# Ordinary Differential Equations

[http://www.lehigh.edu/~ wes1/apci/25apr00.tex](http://www.lehigh.edu/~wes1/apci/25apr00.tex)  
<http://www.lehigh.edu/~ wes1/apci/25apr00.ps>  
<http://www.lehigh.edu/~ wes1/apci/25apr00.pdf>  
<http://www.lehigh.edu/~ wes1/apci/euler1.for>  
to  
<http://www.lehigh.edu/~ wes1/apci/euler7.for>  
[files.txt](#)

1. Euler and modified Euler methods
2. Runge-Kutta methods
3. Accuracy and stability
4. BDF methods
5. References and software

### **Why differential equations?**

Differential equations are possibly the most widely used form of mathematics in science and engineering.

### **What are differential equations?**

Differential equations are equations with one or more derivatives.

### **What are ordinary differential equations (ODEs)?**

ODEs are differential equations with one independent variable, typically time.

### **Why solve differential equations using computers?**

Essentially all realistic problems in differential equations are:

- High order (of order  $n \times n$ )
  - $10^3 \times 10^3$  systems are now routine
  - $10^4 \times 10^4$  to  $10^5 \times 10^5$  systems are now rather commonplace
  - $10^6 \times 10^6$  systems are at the forefront of applications
- Nonlinear

We can solve differential equations (ODEs) analytically only if  $n \leq 4$ , and they are linear.

Representative analytical methods include: integrating factor, separation of variables, Laplace transforms

### **What is the origin of such high order, nonlinear problems?**

Examples: Solid state device simulation; molecular dynamics

- ODEs - (one independent variable,  $t$ )

$$\frac{dy}{dt} = \lambda y, y(t_0) = y_0; \text{ solution: } y(t) = y_0 e^{\lambda(t-t_0)}$$

A solution is the *dependent variable(s)* as a function of the *independent variable* (which satisfies the ODE(s) and all of its auxiliary conditions).

- DAEs - differential algebraic equations (one independent variable,  $t$ ), e.g.,

$$\frac{dy}{dt} = f(x, y), g(x, y) = 0; \text{ solution: } x(t), y(t)$$

- PDEs - partial differential equations (more than one independent variable, e.g.,  $x, t$ )

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}; \text{ solution: } u(x, t)$$

In general, we will obtain the solution in numerical form.

- Order of the ODE/DAE/PDE system

$$\frac{dy_1}{dt} = a_{11}y_1 + a_{12}y_2 \quad y_1(0) = y_{10}$$

$$\frac{dy_2}{dt} = a_{21}y_1 + a_{22}y_2 \quad y_2(0) = y_{20}$$

This is a 2 x 2 system, or a second order system (2 equations in 2 unknowns). It is also a *linear* or first degree system since all of the dependent variables appear to the first power (including the derivatives).

- Degree of the ODE/DAE/PDE system

$$\frac{dy_1}{dt} = a_{11}y_1^3 + a_{12}y_2^3 \quad y_1(0) = y_{10}$$

$$\frac{dy_2}{dt} = a_{21}y_1^3 + a_{22}y_2^3 \quad y_2(0) = y_{20}$$

This is a second order (2 x 2, with two dependent variables), third degree system. More generally, it is a *nonlinear* system, i.e., it is not first degree. As other examples

(a):

$$\left(\frac{dy_1}{dt}\right)^2 = a_{11}y_1 + a_{12}y_2 \quad y_1(0) = y_{10}$$

$$\left(\frac{dy_2}{dt}\right)^{1/3} = a_{21}y_1 + a_{22}y_2 \quad y_2(0) = y_{20}$$

(b) (The famous Bernoulli pendulum equation):

$$\frac{d^2\theta}{dt^2} + (g/L) \sin \theta = 0 \tag{1}$$

with the two initial conditions

$$\theta(0) = \theta_0 \tag{2}$$

$$\frac{d\theta(0)}{dt} = 0 \tag{3}$$

Note that eq. (1) is:

- Second order (it's highest order derivative is second order, but it can also be considered to have two dependent variables if it is written as two first order ODEs)
- Nonlinear (the dependent variable  $\theta$  is not to the first power, e.g., consider expanding  $\sin \theta$  in a Taylor series with increasing powers of  $\theta$ ; because of this nonlinearity, eq. (1) has no known analytical solution)
- Requires two auxiliary conditions since it is second order. The auxiliary conditions in this case are initial conditions.

For eq. (1), we can define two new variables

$$y_1 = \theta, y_2 = \frac{d\theta}{dt}$$

then write eq. (1) as a system of two first order ODEs

$$\frac{dy_2}{dt} + (g/L) \sin y_1 = 0$$

$$\frac{dy_1}{dt} = y_2$$

which is a 2 x 2 (second order) system. Note also that we can conveniently write the initial conditions, eqs. (2) and (3), in terms of these new variables

$$y_1(0) = \theta_0$$

$$y_2 = 0$$

In general, we can write an nth order ODE as a system of n first order ODEs, i.e., as a n x n system. Thus, numerical methods and software for first order ODEs are general.

Finally, if we use the approximation

$$\sin \theta \approx \theta$$

which is valid only for small  $\theta$ , eq. (1) becomes the linear equation for a harmonic oscillator. This is an example of *linearization*.

$$\frac{d^2\theta}{dt^2} + (g/L)\theta = 0 \tag{4}$$

We can also apply initial conditions (2) and (3) to eq. (4). Analytical integration of eq. (4) gives

$$\theta(t) = \theta_0 \cos \sqrt{\frac{g}{L}}t \quad (5)$$

Proof:

$$\begin{aligned} \frac{d^2\theta}{dt^2} & -\frac{g}{L}\theta_0 \cos \sqrt{\frac{g}{L}}t \\ +\frac{g}{L}\theta & \frac{g}{L}\theta_0 \cos \sqrt{\frac{g}{L}}t \\ 0 & \quad 0 \end{aligned}$$

and eq. (5) clearly satisfies the initial conditions.

As another application leading to ODEs, consider a batch reactor for  $A \xrightarrow{k_1} B \xrightarrow{k_2} C$

$$V \frac{dc_a}{dt} = -V k_1 c_a \quad c_a(0) = c_{a0} \quad (1)(2)$$

$$V \frac{dc_b}{dt} = V k_1 c_a - V k_2 c_b \quad c_b(0) = 0 \quad (3)(4)$$

$$V \frac{dc_c}{dt} = V k_2 c_b \quad c_c(0) = 0 \quad (5)(6)$$

This 3 x 3 system is classified as:

- Third order (three first order ODEs)
- First degree (linear)
- Initial value

- Constant coefficient

It is one of the (rare) instances when we can derive an analytical solution. We can approach an analytical solution in two ways:

- (1) Linear, constant coefficient ODEs have exponential solutions. Thus, eqs. (1) and (2) have the solution

$$c_a(t) = c_{a0}e^{-k_1t} \quad (7)$$

Substitution in the eq. (3) gives

$$\frac{dc_b}{dt} + k_2c_b = k_1c_a = k_1C_{a0}e^{-k_1t} \quad (8)$$

The solution to eq. (8) will consist of the sum of a homogeneous solution,  $c_{bh}$ , and a particular solution,  $c_{bp}$ . If  $k_1 \neq k_2$ ,

$$c_{bh} = C_1e^{-k_2t}$$

$$c_{bp} = C_2e^{-k_1t}$$

Substitution of  $c_{bp}$  in eq. (8) gives

$$C_2(-k_1)e^{-k_1t} + k_2C_2e^{-k_1t} = k_1C_{a0}e^{-k_1t}$$

or

$$C_2 = \frac{k_1C_{a0}}{k_2 - k_1}$$

The solution of eq. (3) to this point is

$$c_b = c_{bp} + c_{bh} = C_1e^{-k_2t} + \frac{k_1C_{a0}}{k_2 - k_1}e^{-k_1t}$$

Application of initial condition (4) gives

$$0 = C_1 + \frac{k_1 C_{a0}}{k_2 - k_1}$$

or

$$C_1 = -\frac{k_1 c_{a0}}{k_2 - k_1}$$

and the solution to eqs. (3) and (4) is

$$c_b = \frac{k_1 c_{a0}}{k_2 - k_1} (e^{-k_1 t} - e^{-k_2 t}) \quad (9)$$

Finally, eq. (5) becomes

$$\frac{dc_c}{dt} = k_2 c_b = k_2 \frac{k_1 c_{a0}}{k_2 - k_1} (e^{-k_1 t} - e^{-k_2 t})$$

which integrates to

$$c_c = \int k_2 \frac{k_1 c_{a0}}{k_2 - k_1} (e^{-k_1 t} - e^{-k_2 t}) dt = k_2 \frac{k_1 c_{a0}}{k_2 - k_1} \left( \frac{1}{-k_1} e^{-k_1 t} - \frac{1}{-k_2} e^{-k_2 t} \right) + C_3$$

Application of initial condition (6) gives

$$0 = k_2 \frac{k_1 c_{a0}}{k_2 - k_1} \left( \frac{1}{-k_1} + \frac{1}{k_2} \right) + C_3$$

or

$$C_3 = c_{a0}$$

and therefore the solutions to eqs. (5) and (6) is

$$c_c = k_2 \frac{k_1 c_{a0}}{k_2 - k_1} \left( \frac{1}{k_2} e^{-k_2 t} - \frac{1}{k_1} e^{-k_1 t} \right) + c_{a0} \quad (10)$$

Proof:

Eqs. (1), (2), (7):

$$\begin{array}{cc} \frac{dc_a}{dt} & -k_1 c_{a0} e^{-k_1 t} \\ +k_1 c_a & k_1 c_{a0} e^{-k_1 t} \\ 0 & 0 \end{array}$$

$$c_a(0) = c_{a0}$$

Eqs. (3), (4), (9):

$$\begin{array}{cc} \frac{dc_b}{dt} & \frac{k_1 c_{a0}}{k_2 - k_1} ((-k_1)e^{-k_1 t} - (-k_2)e^{-k_2 t}) \\ -(k_1 c_a - k_2 c_b) & -(k_1 c_{a0} e^{-k_1 t} - k_2 \frac{k_1 c_{a0}}{k_2 - k_1} (e^{-k_1 t} - e^{-k_2 t})) \\ 0 & = \left[ \frac{-k_1 c_{a0} (k_2 - k_1) + k_2 k_1 c_{a0}}{k_2 - k_1} \right] e^{-k_1 t} - k_2 \frac{k_1 c_{a0}}{k_2 - k_1} e^{-k_2 t} \end{array}$$

$$c_b(0) = \frac{k_1 c_{a0}}{k_2 - k_1} (e^{-k_1(0)} - e^{-k_2(0)}) = 0$$

Eqs. (5), (6), (10):

$$\begin{array}{cc} \frac{dc_c}{dt} & k_2 \frac{k_1 c_{a0}}{k_2 - k_1} (-e^{-k_2 t} + e^{-k_1 t}) \\ -k_2 c_b & -k_2 \frac{k_1 c_{a0}}{k_2 - k_1} (e^{-k_1 t} - e^{-k_2 t}) \\ 0 & 0 \end{array}$$

$$c_c(0) = k_2 \frac{k_1 c_{a0}}{k_2 - k_1} \left( \frac{1}{k_2} e^{-k_2(0)} - \frac{1}{k_1} e^{-k_1(0)} \right) + c_{a0} = 0$$

(2) If the Laplace transforms of  $c_a(t)$ ,  $c_b(t)$  and  $c_c(t)$  are defined as

$$C_a(s) = L \{c_a(t)\}, C_b(s) = L \{c_b(t)\}, C_c(s) = L \{c_c(t)\}$$

eqs. (1) to (6) transform to

$$sC_a - c_{a0} = -k_1C_a \quad (11)$$

$$sC_b - 0 = k_1C_a - k_2C_b \quad (12)$$

$$sC_c - 0 = k_2C_b \quad (13)$$

Thus, ODEs (1) to (6) have been transformed to algebraic eqs. (11) to (13), which is one of the major advantages of the Laplace transform. If eq. (11) is rearranged

$$C_a = \frac{c_{a0}}{s + k_1}$$

which inverts to

$$c_a(t) = L^{-1}\{C_a\} = c_{a0}e^{-k_1t}$$

as expected, i.e., this is eq. (7).

If  $C_a$  is eliminated between eqs. (11) and (12),

$$C_b = k_1 \frac{c_{a0}}{(s + k_1)(s + k_2)}$$

We can now expand the RHS in partial fractions

$$\frac{1}{(s + k_1)(s + k_2)} = \frac{A}{s + k_1} + \frac{B}{s + k_2}$$

Then

$$A = \frac{1}{s + k_2} \Big|_{s=-k_1} = \frac{1}{k_2 - k_1}$$

$$B = \frac{1}{s + k_1} \Big|_{s=-k_2} = -\frac{1}{k_2 - k_1}$$

Thus

$$C_b = k_1 C_{a0} \frac{1}{k_2 - k_1} \left( \frac{1}{s + k_1} - \frac{1}{s + k_2} \right)$$

which inverts to

$$c_b(t) = L^{-1}\{C_b\} = k_1 C_{a0} \frac{1}{k_2 - k_1} (e^{-k_1 t} - e^{-k_2 t})$$

This is eq. (9) as expected.

Finally, we have eq. (13)

$$C_c = (k_2/s) k_1 C_{a0} \frac{1}{k_2 - k_1} \left( \frac{1}{s + k_1} - \frac{1}{s + k_2} \right)$$

which inverts to

$$\begin{aligned} c_c(t) &= L^{-1}\{C_c\} = C_{a0} \frac{k_1 k_2}{k_2 - k_1} \int_0^t (e^{-k_1 \lambda} - e^{-k_2 \lambda}) d\lambda \\ &= C_{a0} \frac{k_1 k_2}{k_2 - k_1} (1/(-k_1) e^{-k_1 \lambda} - 1/(-k_2) e^{-k_2 \lambda}) \Big|_0^t \\ &= C_{a0} \frac{k_1 k_2}{k_2 - k_1} \{1/k_1 (1 - e^{-k_1 t}) - 1/k_2 (1 - e^{-k_2 t})\} \end{aligned}$$

$$= C_{a0} \left\{ 1 + \frac{k_1 k_2}{k_2 - k_1} \left[ (1/k_2)e^{-k_2 t} - (1/k_1)e^{-k_1 t} \right] \right\}$$

This is eq. (10) as expected.

Note that these solutions have two exponentials, e.g.,

$$c_b(t) = k_1 C_{a0} \frac{1}{k_2 - k_1} (e^{-k_1 t} - e^{-k_2 t})$$

We can note the following points about this 3 x 3 system:

- If one reaction is much faster than the other ( $k_2 \gg k_1$ ), one of the exponentials will decay much faster than the other ( $e^{-k_2 t} \ll e^{-k_1 t}$ ); this is the basis of *stiffness* in ODEs, and requires special integration methods to compute a stable solution (chemical reactions are a common source of stiff ODEs). In other words, we can make the problem arbitrarily stiff (e.g.,  $k_2 = 10^6 k_1$ ), and therefore it could be used as a stringent test problem for a stiff (implicit) integrator
- If the reactions are not first order, the ODEs are nonlinear and generally we cannot integrate them analytically. For example, if the reactions are second order, eq. (3) becomes

$$V \frac{dc_b}{dt} = V k_1 c_a^2 - V k_2 c_b^2$$

However, in general, nonlinear ODEs are no more difficult to integrate numerically than linear ODEs.

The preceding analytical solutions demonstrate that even for problems of very modest size and complexity (i.e.,  $n = 3$ , linear), the analysis is relatively complicated (and clearly, solving higher order, nonlinear systems is out of the question). Actually, the preceding 3 x 3 system is relatively easy to solve analytically because eq. (1) can be solved without knowing the solution to eqs. (3) and (5), that is, the ODEs are lightly coupled.

More generally, all three ODEs would have to be solved simultaneously, and this procedure would be more complicated than the two analytical procedures considered above (e.g., in general, it would be necessary to factor a third order polynomial, the so-called *characteristic equation* for the ODE system to obtain the three *eigenvalues*).

However, analytical solutions serve as absolute standards in testing numerical methods and their associated software. This is typically done in two ways:

- To check the performance of a new integration algorithm
- To check for coding errors

### **Euler and Modified Euler Methods**

How do we compute a numerical solution to an ODE? Consider the model problem

$$\frac{dy}{dt} = f(y, t), y(t_0) = y_0$$

If we expand the solution in a Taylor series

$$y(t) = y(t_0) + \frac{dy(t_0)}{dt}h + \frac{d^2y(t_0)}{dt^2}h^2/2! + \dots$$

where  $h = t - t_0$ . We can truncate this series after the linear term in  $h$

$$y(t) \approx y(t_0) + \frac{dy(t_0)}{dt}h$$

and use this approximation to step along the solution from  $y_0$  to  $y$ . Then we can use the computed  $y$  as the initial value  $y_0$ , and repeat the step to the next point along the numerical solution, etc. This is the famous *Euler's method*.

Since the Euler method has limited accuracy, we now consider the application of the modified (or extended) Euler method to the solution of the pendulum equation to produce a solution with improved accuracy

$$\frac{d^2\theta}{dt^2} + (g/L) \sin \theta = 0 \quad (1)$$

with the two initial conditions

$$\theta(0) = \theta_0 \quad (2)$$

$$\frac{d\theta(0)}{dt} = 0 \quad (3)$$

Note from eq. (3) that we will take the initial velocity as zero (not required).

We will also consider the approximation

$$\sin \theta \approx \theta$$

Eq. (1) becomes the linear equation for a harmonic oscillator

$$\frac{d^2\theta}{dt^2} + (g/L)\theta = 0 \quad (4)$$

We will also apply initial conditions (2) and (3) to eq. (4); as noted previously, we have an analytical solution that can be used to evaluate the numerical solution.

Since Euler's method, and most other numerical methods for ODEs, can only be applied to first-order ODEs (this is not really a significant limitation since we can almost always express higher-order ODEs as systems of first-order ODEs), we will write eqs. (1) and (4) as a system of first-order ODEs. Thus, if we define for eq. (1) two new variables,

$$y_1 = \theta, y_2 = \frac{d\theta}{dt}$$

and for eq. (4) two new variables ( $\theta$  will be different for eqs. (1) and (4) since these are different problems)

$$y_3 = \theta, y_4 = \frac{d\theta}{dt}$$

then the system of first-order ODEs to be integrated numerically is

$$\frac{dy_2}{dt} + (g/L) \sin y_1 = 0, \frac{dy_1}{dt} = y_2, y_1(0) = y_{10}, y_2(0) = 0 \quad (5)$$

$$\frac{dy_4}{dt} + (g/L)y_3 = 0, \frac{dy_3}{dt} = y_4, y_3(0) = y_{30}, y_4(0) = 0 \quad (6)$$

where  $y_{10}$  and  $y_{30}$  are given initial values.

To reiterate, eqs.(5) and (6) may seem essentially the same, but they are in fact, different for the following reasons:

- Eqs. (5) are nonlinear due to the  $\sin\theta$  term while eqs. (6) are linear.
- As a consequence of the difference in linearity of eqs. (5) and (6), eqs. (6) can be solved by the methods of calculus (e.g., assuming exponential solutions, Laplace transforms), while eqs. (5) cannot. The solution to eqs. (6) is (note: the exponentials in this case are complex, which can be expressed in terms of sines and cosines):

$$y_3(t) = y_{30} \cos\left(\sqrt{\frac{g}{L}}t\right) \quad (7)$$

Application of Euler's method to eqs. (5) and (6) at point  $i$  along a grid in  $t$  gives

$$y_{1,i+1} = y_{1,i} + \frac{dy_{1,i}}{dt}h = y_{1,i} + y_{2,i}h \quad (8)$$

$$y_{2,i+1} = y_{2,i} + \frac{dy_{2,i}}{dt}h = y_{2,i} + (-g/L) \sin y_{1,i}h \quad (9)$$

$$y_{1,0} = y_{10}, y_{2,0} = 0 \quad (10)$$

$$y_{3,i+1} = y_{3,i} + \frac{dy_{3,i}}{dt}h = y_{3,i} + y_{4,i}h \quad (11)$$

$$y_{4,i+1} = y_{4,i} + \frac{dy_{4,i}}{dt}h = y_{4,i} + (-g/L)y_{3,i}h \quad (12)$$

$$y_{3,0} = y_{30}, y_{4,0} = 0 \quad (13)$$

A program to implement eqns. (11) to (13) (the linear problem, along with the analytical solution, eq. (7)) follows:

```

PROGRAM EULER1
*
* Double precision coding is used
  IMPLICIT DOUBLE PRECISION (A-H,O-Z)
*
* Open a file for output
  OPEN(1,FILE='pend1.out')
*
* Problem parameters
*
*   Pendulum length (m)
     XL=30.0D0
*
*   Acceleration of gravity (m/s**2)
     G=9.8D0
*
* Step through a series of Euler integrations
* (with varying step size, h)

```

```

DO IH=1,4
  IF(IH.EQ.1)H=1.0D0
  IF(IH.EQ.2)H=0.1D0
  IF(IH.EQ.3)H=0.01D0
  IF(IH.EQ.4)H=0.001D0
*
*   Begin integration (set initial conditions)
*
*   Initial time (sec)
  T=0.0D0
*
*   Initial angle (radians)
*   Linear problem
  Y3=1.0D0
  Y30=Y3
*
*   Initial velocity (radians/sec)
*   Linear problem
  Y4=0.0D0
*
*   Final t (sec)
  TF=5.0D0
*
*   Number of steps for 0 le t le TF
  NSTEPS=INT(TF/H)
*
*   Perform integration
  DO I=0,NSTEPS
*
*   Write numerical solutions
*
*   t=0
  IF(T.LT.0.00001D0)THEN
  WRITE(*,2)
  WRITE(1,2)
2  FORMAT(//,7X,'h',7X,'t',8X,'y3',7X,'y3e',6X,'error')
  ERROR=0.0D0
  WRITE(*,1)H,T,Y3,Y30,ERROR

```

```

        WRITE(1,1)H,T,Y3,Y30,ERROR
1      FORMAT(F8.4,F8.2,2F10.4,F11.6)
*
*      t = 1, 2, 3, 4, 5
        ELSE IF (
+      ((T.GT.0.99999D0).AND.(T.LT.1.00001D0)).OR.
+      ((T.GT.1.99999D0).AND.(T.LT.2.00001D0)).OR.
+      ((T.GT.2.99999D0).AND.(T.LT.3.00001D0)).OR.
+      ((T.GT.3.99999D0).AND.(T.LT.4.00001D0)).OR.
+      ((T.GT.4.99999D0).AND.(T.LT.5.00001D0)))THEN
*
*      Exact solution and error
        Y3E=Y3*DCOS(DSQRT(G/XL)*T)
        ERROR=Y3-Y3E
*
*      Write numerical and exact solutions, error
        WRITE(*,1)H,T,Y3,Y3E,ERROR
        WRITE(1,1)H,T,Y3,Y3E,ERROR
        END IF
*
*      Derivative at base point
        Linear problem
        DY3DT=Y4
        DY4DT=-(G/XL)*Y3
*
*      Euler step
        T=DFLOAT(I+1)*H
*
*      Linear problem
        Y3=Y3+DY3DT*H
        Y4=Y4+DY4DT*H
*
*      Continue Euler integration
        END DO
*
*      Integration complete; next integration step size
        WRITE(*,3)NSTEPS
        WRITE(1,3)NSTEPS

```

```

3   FORMAT(/,3X,'NSTEPS = ',I6)
   PAUSE
   END DO
*
*   Calculation for all integration steps completed
   WRITE(*,4)
   WRITE(1,4)
4   FORMAT(///,'Conclusion:                                ',/,3X,
+   'Euler''s method is first order correct,',/,3X,
+   'i.e., O(h)')
*
*   End of calculations
   STOP
   END

   DOUBLE PRECISION FUNCTION DFLOAT(I)
*
*   Function DFLOAT converts a single precision integer to a
*   double precision float
*
   DFLOAT=DBLE(REAL(I))
   RETURN
   END

```

We can note the following points about this program:

(1) The integration step size is set to one of four values,  $h = 1, 0.1, 0.01, 0.001$  (during each of four runs of the program)

```

DO IH=1,4
   IF(IH.EQ.1)H=1.0D0
   IF(IH.EQ.2)H=0.1D0
   IF(IH.EQ.3)H=0.01D0
   IF(IH.EQ.4)H=0.001D0

```

In this way, the effect of varying the integration step,  $h$  can be observed.

(2) Initial conditions (13) are set at the beginning of each run, where, in this case,  $y_{30} = 1$ . Also, the final time for each solution is set

```
*      Initial time (sec)
      T=0.0D0
*
*      Initial angle (radians)
*      Linear problem
*      Y3=1.0D0
*      Y30=Y3
*
*      Initial velocity (radians/sec)
*      Linear problem
*      Y4=0.0D0
*
*      Final t (sec)
*      TF=5.0D0
```

(3) The solution is computed (stepped) from one  $t$  to the next using a DO loop

```
*      Number of steps for  $0 \leq t \leq TF$ 
*      NSTEPS=INT(TF/H)
*
*      Perform integration
*      DO I=0,NSTEPS
```

(4) The numerical solution is displayed at  $t = 0, 1, 2, 3, 4, 5$  sec along with the analytical solution, eq. (7), and the difference between the numerical and analytical solutions

```
*      Write numerical solutions
*
*      t=0
*      IF(T.LT.0.00001D0)THEN
```

```

WRITE(*,2)
WRITE(1,2)
2  FORMAT(/,7X,'h',7X,'t',8X,'y3',7X,'y3e',6X,'error')
   ERROR=0.0D0
   WRITE(*,1)H,T,Y3,Y30,ERROR
   WRITE(1,1)H,T,Y3,Y30,ERROR
1  FORMAT(F8.4,F8.2,2F10.4,F11.6)
*
*      t = 1, 2, 3, 4, 5
   ELSE IF (
+      ((T.GT.0.99999D0).AND.(T.LT.1.00001D0)).OR.
+      ((T.GT.1.99999D0).AND.(T.LT.2.00001D0)).OR.
+      ((T.GT.2.99999D0).AND.(T.LT.3.00001D0)).OR.
+      ((T.GT.3.99999D0).AND.(T.LT.4.00001D0)).OR.
+      ((T.GT.4.99999D0).AND.(T.LT.5.00001D0)))THEN
*
*      Exact solution and error
   Y3E=Y30*DCOS(DSQRT(G/XL)*T)
   ERROR=Y3-Y3E
*
*      Write numerical and exact solutions, error
   WRITE(*,1)H,T,Y3,Y3E,ERROR
   WRITE(1,1)H,T,Y3,Y3E,ERROR
   END IF

```

(5) The Euler integration of eqs. (11) and (12) is programmed as

```

*      Derivative at base point
*      Linear problem
   DY3DT=Y4
   DY4DT=-(G/XL)*Y3
*
*      Euler step
   T=DFLOAT(I+1)*H
*
*      Linear problem
   Y3=Y3+DY3DT*H

```

```

          Y4=Y4+DY4DT*H
*
*      Continue Euler integration
      END DO

```

Note that the Euler integration is done in two steps: (1) all of the derivatives are evaluated at the base point (grid point  $i$ ), and then (2) all of the dependent variables are moved forward together to the new  $t$  (at grid point  $i + 1$ ). The procedure of moving all of the variables together should always be followed in a numerical integration to ensure that the variables remain synchronized in  $t$ . This is termed a *central integration*.

The program concludes with some output to indicate the main conclusions that can be drawn from execution of the program.

The output from the preceding program is listed below

h	t	y3	y3e	error
1.0000	0.00	1.0000	1.0000	0.000000
1.0000	1.00	1.0000	0.8411	0.158935
1.0000	2.00	0.6733	0.4148	0.258553
1.0000	3.00	0.0200	-0.1434	0.163351
1.0000	4.00	-0.8533	-0.6559	-0.197374
1.0000	5.00	-1.7331	-0.9600	-0.080602

NSTEPS = 5

h	t	y3	y3e	error
0.1000	0.00	1.0000	1.0000	0.000000
0.1000	1.00	0.8552	0.8411	0.014169
0.1000	2.00	0.4297	0.4148	0.014918
0.1000	3.00	-0.1486	-0.1434	-0.005250
0.1000	4.00	-0.6981	-0.6559	-0.042205
0.1000	5.00	-1.0406	-0.9600	-0.080602

NSTEPS = 50

h	t	y3	y3e	error
0.0100	0.00	1.0000	1.0000	0.000000
0.0100	1.00	0.8424	0.8411	0.001378
0.0100	2.00	0.4161	0.4148	0.001369
0.0100	3.00	-0.1440	-0.1434	-0.000686
0.0100	4.00	-0.6602	-0.6559	-0.004280
0.0100	5.00	-0.9678	-0.9600	-0.007863

NSTEPS = 500

h	t	y3	y3e	error
0.0010	0.00	1.0000	1.0000	0.000000
0.0010	1.00	0.8412	0.8411	0.000137
0.0010	2.00	0.4149	0.4148	0.000136
0.0010	3.00	-0.1434	-0.1434	-0.000070
0.0010	4.00	-0.6563	-0.6559	-0.000428
0.0010	5.00	-0.9608	-0.9600	-0.000784

NSTEPS = 5000

Conclusion:

Euler's method is first order correct,  
i.e.,  $O(h)$

Note how the error in the solution is proportional to  $h$ , i.e., Euler's method is first order correct. This is evident from the following results at  $t = 5.0$  excerpted from the previous output:

$h$	error
1.0	-0.080602
0.1	-0.080602
0.01	-0.007863
0.001	-0.000784

Thus, if  $h$  is reduced by a factor of  $1/10$ , the error is also reduced by a factor of  $1/10$  if  $h$  is sufficiently small (e.g.,  $h \leq 0.1$ ).

We can analyze this first order characteristic ( $O(h^1)$ ) of the Euler method in an approximate way. We have considered the Euler method as a truncated Taylor series

$$y(t) = y(t_0) + \frac{dy(t_0)}{dt}h + \frac{d^2y(t_0)}{dt^2}h^2/2!$$

so that the principal error term for the Euler method is

$$\frac{d^2y(t_0)}{dt^2}h^2/2!$$

Note that this is a *one step error* since it is the error that is committed by moving along the solution numerically for one step of size  $h$ .

However, in computing a complete solution to an ODE system, we typically will require many steps (e.g., 100s or 1000s of steps are not uncommon), and each step incurs a one step error. Thus, the question of interest is what is the error after many steps? This is termed the *total or global error*, and is the error of primary interest since we would like to know the actual error in a numerical solution after taking a series of integration steps.

We can estimate the global error approximately in the following way. If we integrate over the interval  $t_0 \leq t \leq t_f$  using a fixed (constant) step of size  $h$ , then the total number of integration steps taken will be

$$\text{number of steps} = (t_f - t_0)/h$$

If we take as the error at each of these steps (the one step error)  $\frac{d^2y(t_0)}{dt^2}h^2/2!$ , then the total or global error can be estimated as

$$\text{total error} \cong (\text{one step error})(\text{number of steps})$$

$$\cong \left( \frac{d^2y(t_0)}{dt^2}h^2/2! \right) (t_f - t_0)/h = (t_f - t_0) \left( \frac{d^2y(t_0)}{dt^2}h/2! \right) = O(h^1)$$

so that the global error is first order, as we observed in the preceding calculation for the pendulum equation. This analysis is general in the sense that we have not assumed any particular form for the ODE, but it is also approximate in the sense that the second derivative,  $\frac{d^2y(t)}{dt^2}$ , is not constant along the solution (we could consider using an average value over the interval  $t_0 \leq t \leq t_f$ ). In any case, we observe that the one step error and the global error differ by one order in  $h$ , and this conclusion follows for any of the ODE integration algorithms we will consider that are based on the Taylor series. For example, the classical fourth order Runge Kutta method to be discussed in the next section is fifth order in the one step error, and fourth order in the global error. Since we are interested primarily in the global error, this method is referred to as the classical *fourth order* Runge Kutta method.

As an additional point, estimates of the global error that could be used to adjust  $h$  to achieve a prescribed accuracy in the numerical solution are generally not available (we will discuss step size adjustment in the next section). Rather, one step estimates are used to adjust the step size  $h$  (since they are readily available), and the expectation (or hope) is that by controlling the one step error through the adjustment of the step size, we will also control the global error. This might seem like too much to expect, but in reality, this actually works quite well. Thus, virtually all library ODE integrators attempt to control the global integration error by computing the one step error, and then using the one step error to control the global error.

Finally, the preceding program for the pendulum problem is in

<http://www.lehigh.edu/~wes1/apci/euler1.for>

and can be downloaded for execution with any Fortran compiler.

The preceding programming can easily be extended to include the nonlinear problem, eqs. (1) to (3). The essential programming (for eqs. (8) to (10)) is

```
*           Derivative at base point
*
*           Nonlinear problem
*           DY1DT=Y2
```

```

          DY2DT=-(G/XL)*DSIN(Y1)
*
*      Linear problem
          DY3DT=Y4
          DY4DT=-(G/XL)*Y3
*
*      Euler step
          T=DFLOAT(I+1)*H
*
*      Nonlinear problem
          Y1=Y1+DY1DT*H
          Y2=Y2+DY2DT*H
*
*      Linear problem
          Y3=Y3+DY3DT*H
          Y4=Y4+DY4DT*H

```

Note how easily the nonlinear problem is programmed (all that is required is to use  $\sin \theta$  in place of  $\theta$ ). This example demonstrates the power of numerical methods, i.e., the programming for a nonlinear problem (which in this case cannot be solved analytically) is generally no more difficult than for a linear problem.

The output from the modified program (to include the nonlinear problem of eqs. (1) to (3)) is

h	t	y1	y3
1.0000	0.00	1.0000	1.0000
1.0000	1.00	1.0000	1.0000
1.0000	2.00	0.7251	0.6733
1.0000	3.00	0.1754	0.0200
1.0000	4.00	-0.5911	-0.8533
1.0000	5.00	-1.4145	-1.7331

NSTEPS = 5

h	t	y1	y3
---	---	----	----

0.1000	0.00	1.0000	1.0000
0.1000	1.00	0.8774	0.8552
0.1000	2.00	0.5040	0.4297
0.1000	3.00	-0.0335	-0.1486
0.1000	4.00	-0.5778	-0.6981
0.1000	5.00	-0.9641	-1.0406

NSTEPS = 50

h	t	y1	y3
0.0100	0.00	1.0000	1.0000
0.0100	1.00	0.8659	0.8424
0.0100	2.00	0.4882	0.4161
0.0100	3.00	-0.0377	-0.1440
0.0100	4.00	-0.5536	-0.6602
0.0100	5.00	-0.9056	-0.9678

NSTEPS = 500

h	t	y1	y3
0.0010	0.00	1.0000	1.0000
0.0010	1.00	0.8648	0.8412
0.0010	2.00	0.4868	0.4149
0.0010	3.00	-0.0380	-0.1434
0.0010	4.00	-0.5511	-0.6563
0.0010	5.00	-0.8998	-0.9608

NSTEPS = 5000

#### Conclusion:

The difference between the linear and nonlinear solutions is a function of the initial condition, and more generally, the magnitude of the dependent variables.

We can note in this output the difference in the solutions for the linear and

nonlinear problems (when  $\sin \theta$  is approximated by  $\theta$ ). In general, this difference will become larger as  $\theta$  is increased (this might be done, for example, by using a larger value of the initial condition  $y_1(0) = y_{10}$ ); in the preceding programming  $y_{10} = 1$ .

Also, if the solutions were plotted, we would observe that the solution of the linear problem is the analytical solution of eq. (7) (i.e., a cosine), while the solution of the nonlinear problem would have a significant departure from a pure harmonic solution (consisting of just a linear combination of a single sine and a cosine). This point could be investigated further by doing a Fourier analysis on the nonlinear solution.

The program that generated the preceding output is in

<http://www.lehigh.edu/~wes1/apci/euler2.for>

We can also consider the modified (extended) Euler method applied to eqs. (1) to (6)

$$y_{1,i+1}^p = y_{1,i} + \frac{dy_{1,i}}{dt} h, y_{1,i+1}^c = y_{1,i} + (1/2) \left( \frac{dy_{1,i}}{dt} + \frac{dy_{1,i+1}^p}{dt} \right) h \quad (14)$$

$$y_{2,i+1}^p = y_{2,i} + \frac{dy_{2,i}}{dt} h, y_{2,i+1}^c = y_{2,i} + (1/2) \left( \frac{dy_{2,i}}{dt} + \frac{dy_{2,i+1}^p}{dt} \right) h \quad (15)$$

$$y_{1,0} = y_{10}, y_{2,0} = 0 \quad (16)$$

$$y_{3,i+1}^p = y_{3,i} + \frac{dy_{3,i}}{dt} h, y_{3,i+1}^c = y_{3,i} + (1/2) \left( \frac{dy_{3,i}}{dt} + \frac{dy_{3,i+1}^p}{dt} \right) h \quad (17)$$

$$y_{4,i+1}^p = y_{4,i} + \frac{dy_{4,i}}{dt} h, y_{4,i+1}^c = y_{4,i} + (1/2) \left( \frac{dy_{4,i}}{dt} + \frac{dy_{4,i+1}^p}{dt} \right) h \quad (18)$$

$$y_{3,0} = y_{30}, y_{4,0} = 0 \quad (19)$$

Note that the modified Euler method is based on the averaging of the derivatives at the base and advanced points (note the averaging in the second equations of eqs. (14), (15), (17) and (18)). The derivatives at the advanced point are evaluated from the ODEs using the *predicted* solutions from the usual Euler method,  $y_{1,i+1}^p, y_{2,i+1}^p, y_{3,i+1}^p, y_{4,i+1}^p$ . The *corrected* solutions,  $y_{1,i+1}^c, y_{2,i+1}^c, y_{3,i+1}^c, y_{4,i+1}^c$ , are then computed by the derivative averaging. Thus, the modified Euler method is an example of a *predictor-corrector* method.

Although the modified Euler method is somewhat more complicated than the Euler method, and does require more calculations, the extra effort in the programming and computation is usually well worthwhile. First, the extra programming is minimal. For example, the programming of eqs. (14),(15), (17) and (18) is illustrated with the following coding

```

*      Derivative at base point
*      Linear problem
*      Y3B=Y3
*      Y4B=Y4
*      DY3DTB=Y4B
*      DY4DTB=-(G/XL)*Y3B
*
*      Euler step
*      T=DFLOAT(I+1)*H
*      Y3=Y3B+DY3DTB*H
*      Y4=Y4B+DY4DTB*H
*
*      Derivative at advanced point
*      DY3DT=Y4
*      DY4DT=-(G/XL)*Y3
*
*      Modified Euler step
*      Y3=Y3B+(DY3DTB+DY3DT)/2.0D0*H
*      Y4=Y4B+(DY4DTB+DY4DT)/2.0D0*H

```

Note the modest additional programming required for the modified Euler method (specifically, one additional derivative evaluation is required at the advanced point  $i+1$ , plus some additional arithmetic; this added computation does not produce any perceptible increase in the computer run time for the pendulum problem). The output of the program that contains this coding is listed below

h	t	y3	y3e	error
1.0000	0.00	1.0000	1.0000	0.0000000000
1.0000	1.00	0.8367	0.8411	-0.0043981625
1.0000	2.00	0.3733	0.4148	-0.0414356492
1.0000	3.00	-0.2343	-0.1434	-0.0909064384
1.0000	4.00	-0.7753	-0.6559	-0.1193801631
1.0000	5.00	-1.0568	-0.9600	-0.0968372608

NSTEPS = 5

h	t	y3	y3e	error
0.1000	0.00	1.0000	1.0000	0.0000000000
0.1000	1.00	0.8409	0.8411	-0.0001569845
0.1000	2.00	0.4142	0.4148	-0.0005547655
0.1000	3.00	-0.1443	-0.1434	-0.0009286907
0.1000	4.00	-0.6569	-0.6559	-0.0009731641
0.1000	5.00	-0.9605	-0.9600	-0.0004982059

NSTEPS = 50

h	t	y3	y3e	error
0.0100	0.00	1.0000	1.0000	0.0000000000
0.0100	1.00	0.8411	0.8411	-0.0000016720
0.0100	2.00	0.4148	0.4148	-0.0000056518
0.0100	3.00	-0.1434	-0.1434	-0.0000092445
0.0100	4.00	-0.6559	-0.6559	-0.0000094303
0.0100	5.00	-0.9600	-0.9600	-0.0000044212

NSTEPS = 500

h	t	y3	y3e	error
0.0010	0.00	1.0000	1.0000	0.0000000000
0.0010	1.00	0.8411	0.8411	-0.0000000168
0.0010	2.00	0.4148	0.4148	-0.0000000566
0.0010	3.00	-0.1434	-0.1434	-0.0000000924
0.0010	4.00	-0.6559	-0.6559	-0.0000000940
0.0010	5.00	-0.9600	-0.9600	-0.0000000436

NSTEPS = 5000

Conclusion:

The modified Euler method is second order correct, i.e.,  $O(h^2)$ ; the cost of this enhanced accuracy is one additional derivative evaluation in each step.

Note how the error in the solution is proportional to  $h^2$ , i.e., Euler's method is second-order correct. This is evident from the following results at  $t = 5.0$  excerpted from the previous output:

$h$	error
1.0	-0.0968372608
0.1	-0.0004982059
0.01	-0.0000044212
0.001	-0.0000000436

Thus, if  $h$  is reduced by a factor of 1/10, the error is reduced by a factor of  $1/10^2$  if  $h$  is sufficiently small (e.g.,  $h \leq 0.1$ ). Clearly this second-order convergence of the modified Euler method gives substantially better solutions than the first-order Euler method for a given number of steps (for a given integration step,  $h$ ).

The program that produced the preceding numerical output is in

<http://www.lehigh.edu/~wes1/apci/euler3.for>

We can combine the programming of the modified Euler method for the

nonlinear and linear pendulum equations, eqs. (1) to (6). The principal programming follows

```
*      Derivative at base point
*
*      Nonlinear problem
*      Y1B=Y1
*      Y2B=Y2
*      DY1DTB=Y2B
*      DY2DTB=-(G/XL)*DSIN(Y1B)
*
*      Linear problem
*      Y3B=Y3
*      Y4B=Y4
*      DY3DTB=Y4B
*      DY4DTB=-(G/XL)*Y3B
*
*      Euler step
*      T=DFLOAT(I+1)*H
*
*      Nonlinear problem
*      Y1=Y1B+DY1DTB*H
*      Y2=Y2B+DY2DTB*H
*
*      Linear problem
*      Y3=Y3B+DY3DTB*H
*      Y4=Y4B+DY4DTB*H
*
*      Derivative at advanced points
*
*      Nonlinear problem
*      DY1DT=Y2
*      DY2DT=-(G/XL)*DSIN(Y1)
*
*      Linear problem
*      DY3DT=Y4
*      DY4DT=-(G/XL)*Y3
```

```

*
*      Modified Euler step
*
*      Nonlinear problem
*      Y1=Y1B+(DY1DTB+DY1DT)/2.0D0*H
*      Y2=Y2B+(DY2DTB+DY2DT)/2.0D0*H
*
*      Linear problem
*      Y3=Y3B+(DY3DTB+DY3DT)/2.0D0*H
*      Y4=Y4B+(DY4DTB+DY4DT)/2.0D0*H

```

The output of the program that included the preceding coding is listed below

h	t	y1	y3
1.0000	0.00	1.0000	1.0000
1.0000	1.00	0.8367	0.8626
1.0000	2.00	0.3733	0.4636
1.0000	3.00	-0.2343	-0.0989
1.0000	4.00	-0.7753	-0.6411
1.0000	5.00	-1.0568	-0.9858

NSTEPS = 5

h	t	y1	y3
0.1000	0.00	1.0000	1.0000
0.1000	1.00	0.8409	0.8646
0.1000	2.00	0.4142	0.4863
0.1000	3.00	-0.1443	-0.0386
0.1000	4.00	-0.6569	-0.5516
0.1000	5.00	-0.9605	-0.8997

NSTEPS = 50

h	t	y1	y3
0.0100	0.00	1.0000	1.0000
0.0100	1.00	0.8411	0.8647
0.0100	2.00	0.4148	0.4866

```
0.0100    3.00   -0.1434   -0.0380
0.0100    4.00   -0.6559   -0.5508
0.0100    5.00   -0.9600   -0.8992
```

```
NSTEPS =    500
```

```
      h      t      y1      y3
0.0010    0.00    1.0000    1.0000
0.0010    1.00    0.8411    0.8647
0.0010    2.00    0.4148    0.4866
0.0010    3.00   -0.1434   -0.0380
0.0010    4.00   -0.6559   -0.5508
0.0010    5.00   -0.9600   -0.8992
```

```
NSTEPS =    5000
```

**Conclusion:**

The modified Euler method can easily be programmed for systems of nonlinear ODEs for which analytical solutions and associated theorems are unavailable; thus, for realistic engineering problems, we must use numerical methods.

Plotting the solutions for the linear and nonlinear problems would demonstrate the differences in the solutions, e.g., the nonlinear problem would not have pure linear harmonics, i.e., sine and cosine solutions.

The program that produced the preceding numerical output is in

<http://www.lehigh.edu/~wes1/apci/euler4.for>

The numerical solutions of both the nonlinear and linear problems appear to have converged to four figures. However, we cannot determine the actual error in the nonlinear solution as we did in the linear case since we do not have an exact (analytical) solution to the nonlinear problem.

However, we can compute a solution to the nonlinear problem using successively smaller  $h$  until there is apparent convergence to a specified accuracy (as illustrated in the preceding output). This technique for establishing the accuracy of a numerical solution is usually termed *h refinement*. To illustrate how this might be done to estimate the order of the numerical method, we extend the preceding programming (in euler4.for) to include five values of  $h$

```

DO IH=1,5
  IF(IH.EQ.1)H=1.0D0
  IF(IH.EQ.2)H=0.1D0
  IF(IH.EQ.3)H=0.01D0
  IF(IH.EQ.4)H=0.001D0
  IF(IH.EQ.5)H=0.0001D0

```

The numerical solution is also printed with more figures

```

WRITE(*,1)H,T,Y1,Y3
WRITE(1,1)H,T,Y1,Y3
1 FORMAT(F10.4,F8.2,2F15.9)

```

The output from this program in

[http://www.lehigh.edu/~ wes1/apci/euler5.for](http://www.lehigh.edu/~wes1/apci/euler5.for)

is listed below (note that with  $h = 0.0001$ , 50,000 integration steps are required for the interval  $0 \leq t \leq 5$ , but the computer time for these 50,000 steps was imperceptible on a workstation)

h	t	y1	y3
1.0000	0.00	1.000000000	1.000000000
1.0000	1.00	0.862559739	0.836666667
1.0000	2.00	0.463625893	0.373344444
1.0000	3.00	-0.098906746	-0.234257370
1.0000	4.00	-0.641086792	-0.775295111

1.0000 5.00 -0.985799887 -1.056820316

NSTEPS = 5

h	t	y1	y3
0.1000	0.00	1.000000000	1.000000000
0.1000	1.00	0.864576104	0.840907845
0.1000	2.00	0.486267506	0.414225328
0.1000	3.00	-0.038646593	-0.144279623
0.1000	4.00	-0.551555448	-0.656888112
0.1000	5.00	-0.899703801	-0.960481261

NSTEPS = 50

h	t	y1	y3
0.0100	0.00	1.000000000	1.000000000
0.0100	1.00	0.864652085	0.841063157
0.0100	2.00	0.486586709	0.414774442
0.0100	3.00	-0.038013799	-0.143360176
0.0100	4.00	-0.550806265	-0.655924378
0.0100	5.00	-0.899198127	-0.959987476

NSTEPS = 500

h	t	y1	y3
0.0010	0.00	1.000000000	1.000000000
0.0010	1.00	0.864652895	0.841064812
0.0010	2.00	0.486589963	0.414780037
0.0010	3.00	-0.038007499	-0.143351024
0.0010	4.00	-0.550798998	-0.655915042
0.0010	5.00	-0.899193511	-0.959983099

NSTEPS = 5000

h	t	y1	y3
0.0001	0.00	1.000000000	1.000000000
0.0001	1.00	0.864652904	0.841064829
0.0001	2.00	0.486589996	0.414780093



```

*
*      t = 1, 2, 3, 4, 5
      ELSE IF (
+      ((T.GT.0.99999D0).AND.(T.LT.1.00001D0)))THEN
          EXACT=E1
          ERROR=Y1-E1
          WRITE(*,1)H,T,Y1,EXACT,ERROR
          WRITE(1,1)H,T,Y1,EXACT,ERROR
      ELSE IF (
+      ((T.GT.1.99999D0).AND.(T.LT.2.00001D0)))THEN
          EXACT=E2
          ERROR=Y1-E2
          WRITE(*,1)H,T,Y1,EXACT,ERROR
          WRITE(1,1)H,T,Y1,EXACT,ERROR
      ELSE IF (
+      ((T.GT.2.99999D0).AND.(T.LT.3.00001D0)))THEN
          EXACT=E3
          ERROR=Y1-E3
          WRITE(*,1)H,T,Y1,EXACT,ERROR
          WRITE(1,1)H,T,Y1,EXACT,ERROR
      ELSE IF (
+      ((T.GT.3.99999D0).AND.(T.LT.4.00001D0)))THEN
          EXACT=E4
          ERROR=Y1-E4
          WRITE(*,1)H,T,Y1,EXACT,ERROR
          WRITE(1,1)H,T,Y1,EXACT,ERROR
      ELSE IF (
+      ((T.GT.4.99999D0).AND.(T.LT.5.00001D0)))THEN
          EXACT=E4
          ERROR=Y1-E5
          WRITE(*,1)H,T,Y1,EXACT,ERROR
          WRITE(1,1)H,T,Y1,EXACT,ERROR
      END IF

```

The output from this program in

<http://www.lehigh.edu/~wes1/apci/euler6.for>

is listed below

h	t	y1	exact	error
1.0000	0.00	1.000000000	1.000000000	0.000000000
1.0000	1.00	1.000000000	0.864652904	0.135347096
1.0000	2.00	0.725119478	0.486589996	0.238529482
1.0000	3.00	0.175358435	-0.038007436	0.213365871
1.0000	4.00	-0.591056065	-0.550798926	-0.040257139
1.0000	5.00	-1.414461187	-0.550798926	-0.515267722

NSTEPS = 5

h	t	y1	exact	error
0.1000	0.00	1.000000000	1.000000000	0.000000000
0.1000	1.00	0.877351245	0.864652904	0.012698341
0.1000	2.00	0.503982421	0.486589996	0.017392425
0.1000	3.00	-0.033466921	-0.038007436	0.004540515
0.1000	4.00	-0.577768525	-0.550798926	-0.026969599
0.1000	5.00	-0.964060516	-0.550798926	-0.064867051

NSTEPS = 50

h	t	y1	exact	error
0.0100	0.00	1.000000000	1.000000000	0.000000000
0.0100	1.00	0.865902800	0.864652904	0.001249896
0.0100	2.00	0.488242486	0.486589996	0.001652490
0.0100	3.00	-0.037711126	-0.038007436	0.000296310
0.0100	4.00	-0.553615959	-0.550798926	-0.002817033
0.0100	5.00	-0.905625119	-0.550798926	-0.006431654

NSTEPS = 500

h	t	y1	exact	error
0.0010	0.00	1.000000000	1.000000000	0.000000000
0.0010	1.00	0.864777683	0.864652904	0.000124779
0.0010	2.00	0.486754367	0.486589996	0.000164371
0.0010	3.00	-0.037979335	-0.038007436	0.000028101

0.0010	4.00	-0.551081704	-0.550798926	-0.000282778
0.0010	5.00	-0.899835931	-0.550798926	-0.000642466

NSTEPS = 5000

Conclusion:

The Euler method is first-order correct  
for the nonlinear pendulum problem.

Thus, the Euler method is first-order correct for both the linear problem (established previously using the exact analytical solution) and the nonlinear problem (established by the preceding output using the high accuracy numerical solution at  $h = 0.0001$ ). This first-order behavior for the nonlinear problem is summarized in the following table

$h$	error
1.0	-0.515267722
0.1	-0.064867051
0.01	-0.006431654
0.001	-0.000642466

Note that the error is proportional to  $h$ , e.g., when  $h$  is reduced by 1/10, the error is reduced by 1/10.

If the same computations are performed for the modified Euler method with the program in

<http://www.lehigh.edu/~wes1/apci/euler7.for>

the output is

$h$	$t$	$y_1$	exact	error
1.0000	0.00	1.000000000	1.000000000	0.000000000
1.0000	1.00	0.862559739	0.864652904	-0.002093165
1.0000	2.00	0.463625893	0.486589996	-0.022964103
1.0000	3.00	-0.098906746	-0.038007436	-0.060899310

1.0000	4.00	-0.641086792	-0.550798926	-0.090287866
1.0000	5.00	-0.985799887	-0.550798926	-0.086606422

NSTEPS = 5

h	t	y1	exact	error
0.1000	0.00	1.000000000	1.000000000	0.000000000
0.1000	1.00	0.864576104	0.864652904	-0.000076800
0.1000	2.00	0.486267506	0.486589996	-0.000322490
0.1000	3.00	-0.038646593	-0.038007436	-0.000639157
0.1000	4.00	-0.551555448	-0.550798926	-0.000756522
0.1000	5.00	-0.899703801	-0.550798926	-0.000510336

NSTEPS = 50

h	t	y1	exact	error
0.0100	0.00	1.000000000	1.000000000	0.000000000
0.0100	1.00	0.864652085	0.864652904	-0.000000819
0.0100	2.00	0.486586709	0.486589996	-0.000003287
0.0100	3.00	-0.038013799	-0.038007436	-0.000006363
0.0100	4.00	-0.550806265	-0.550798926	-0.000007339
0.0100	5.00	-0.899198127	-0.550798926	-0.000004662

NSTEPS = 500

h	t	y1	exact	error
0.0010	0.00	1.000000000	1.000000000	0.000000000
0.0010	1.00	0.864652895	0.864652904	-0.000000009
0.0010	2.00	0.486589963	0.486589996	-0.000000033
0.0010	3.00	-0.038007499	-0.038007436	-0.000000063
0.0010	4.00	-0.550798998	-0.550798926	-0.000000072
0.0010	5.00	-0.899193511	-0.550798926	-0.000000046

NSTEPS = 5000

Conclusion:

The modified Euler method is second-order

correct for the nonlinear pendulum problem.

Thus, the modified Euler method is second-order correct for the linear problem (established previously using the analytical solution) and the nonlinear problem (from the preceding output based on the high accuracy solution of euler5.for). This second-order behavior for the modified Euler method is summarized in the following table

$h$	error
1.0	-0.086606422
0.1	-0.000510336
0.01	-0.000004662
0.001	-0.000000046

Note that the error is proportional to  $h^2$ , e.g., when  $h$  is reduced by 1/10, the error is reduced by  $1/10^2$ .

Generally, we can expect that the numerical solution from the modified Euler method will be substantially better than from the Euler method for a given number of integration steps, and this enhanced accuracy can be achieved with only a modest increase in the computational effort.

### Summary:

We can numerically integrate  $n \times n$  systems of linear and nonlinear ODEs, where  $n$  is essentially unlimited

- Nonlinear ODEs are generally no more difficult to integrate numerically than linear ODEs (this is certainly not true for analytical integration)
- Some form of *error analysis* or *error estimation* is required to assess the accuracy of the numerical solution
  - An error analysis can be done by changing the integration step and observing the apparent convergence of the numerical solution (*h refinement*)

- An error analysis can be done by comparing the numerical solutions computed with two algorithms of different orders, e.g., the Euler method ( $O(h)$ ) and the modified Euler method ( $O(h^2)$ ) (*p refinement*)

The preceding discussion is intended as an introduction to the computation and analysis of errors in the numerical solution of systems of ODEs. To conclude, an error analysis of a numerical solution is essential to assess the accuracy of the solution.

In the next section, we will consider how this error analysis can be automated, and thus used to automatically adjust the integration step  $h$  to achieve a numerical solution of prescribed accuracy.

The small Fortran programs are summarized below

File name	Algorithm Specifics	Problem Conclusions
euler1.for	Euler h=1,0.1,0.01,0.001	Linear pendulum First order correct, $O(h)$
euler2.for	Euler h=1,0.1,0.01,0.001	Nonlinear pendulum Three-figure convergence
euler3.for	Modified Euler h=1,0.1,0.01,0.001	Linear pendulum Second order correct, $O(h^2)$
euler4.for	Modified Euler h=1,0.1,0.01,0.001	Nonlinear pendulum Five-figure convergence
euler5.for	Modified Euler h=1,0.1,0.01,0.001,0.0001	Nonlinear pendulum Seven-figure convergence
euler6.for	Euler h=1,0.1,0.01,0.001	Nonlinear pendulum First order correct, $O(h)$
euler7.for	Modified Euler h=1,...,0.001	Nonlinear pendulum Second order correct, $O(h^2)$

## Runge Kutta Methods

One approach to an explicit error estimate (for step size adjustment) is to calculate an Euler solution, and a modified Euler solution, then take the difference of the two solutions as the error estimate. This approach is an example of an *embedded algorithm* (i.e., the Euler method is embedded in the modified Euler method). Since we are basically varying the order of the method, i.e., the Euler method is first-order correct, while the modified Euler method is second-order, and since in the numerical analysis literature, the order of an algorithm is often designated with the letter “p”, the variation in the algorithm order (e.g., from first- to second-order) is generally termed *p refinement*.

To illustrate p refinement, consider again the Euler and modified Euler methods applied to the model ODE

$$\frac{dy}{dt} = f(y, t), y(t_0) = y_0 \quad (1)(2)$$

Euler method:

$$y_{i+1} = y_i + \frac{dy_i}{dt} h \quad (3)$$

Modified Euler method:

$$y_{i+1}^p = y_i + \frac{dy_i}{dt} h \quad (4)$$

$$y_{i+1}^c = y_i + \frac{1}{2} \left( \frac{dy_i}{dt} + \frac{dy_{i+1}^p}{dt} \right) h \quad (5)$$

Note again that the Euler method is *embedded* in the modified Euler method. Thus, if we subtract eq. (4) from eq. (5),

$$\begin{aligned} \epsilon_{i+1} &= y_{i+1}^c - y_{i+1}^p = y_i + \frac{1}{2} \left( \frac{dy_i}{dt} + \frac{dy_{i+1}^p}{dt} \right) h - y_i - \frac{dy_i}{dt} h \\ &= \frac{1}{2} \left( \frac{dy_i}{dt} + \frac{dy_{i+1}^p}{dt} \right) h - \frac{dy_i}{dt} h = \left( \frac{dy_{i+1}^p}{dt} - \frac{dy_i}{dt} \right) \frac{h}{2} \end{aligned} \quad (6)$$

Eq. (6) gives an *explicit error estimate* that can be used to adjust  $h$  to meet a user-specified error criterion; once  $h$  is small enough to satisfy the error criterion, the solution can proceed on to the next point. Note that this error estimate has the important property that it requires only the first derivative, and therefore can be computed directly from the ODE.

This method for *automatic or adaptive h refinement* can be stated in words as:

- 1 Set initial condition, initial  $h$
- 2  $t = t_{final}$ ?
- 2.1 Yes - terminate execution
- 2.2 No - continue integration
- 3  $t = t_{out}$ ?
- 3.1 Yes - output solution and continue integration
- 3.2 No - continue integration
- 4 Compute the predicted solution,  $y_{i+1}^p$ , using an Euler step, eq. (4)
- 5 Compute the estimated error,  $\epsilon_{i+1}$ , from eq. (6)
- 6 Estimated error  $\leq$  specified error?
- 6.1 Yes - add estimated error to result of 4,  $y_{i+1}^c = y_{i+1}^p + \epsilon_{i+1}$ , increment  $t$ , go to 2
- 6.2 No - take  $h = h/2$ , go to 2

Step 6 is usually based on the use of an absolute and a relative error tolerance (library ODE integrators will accept both types of error tolerances).

Step (6.1) then gives a solution that matches the Taylor series up to and including the second derivative (second order  $h^2$ ) term (even though we used only first derivatives computed from the ODE). To demonstrate this we consider again the error estimate

$$\epsilon_{i+1} = \left( \frac{dy_{i+1}^p}{dt} - \frac{dy_i}{dt} \right) \frac{h}{2} \quad (1)$$

Another way to look at the estimated error is to consider the Taylor series expansion of the solution at the point  $i$

$$y_{i+1} = y_i + \frac{dy_i}{dt}h + \frac{d^2y_i}{dt^2} \frac{h^2}{2!} + \dots \quad (2)$$

Note that the first two RHS terms,  $y_i + \frac{dy_i}{dt}h$ , are just the Euler method. If we consider the next term,  $\frac{d^2y_i}{dt^2} \frac{h^2}{2!}$ , as the principal source of error (the *truncation error*), in order to evaluate this error, we need the second derivative  $\frac{d^2y_i}{dt^2}$ . We can get a computationally useful expression for this second derivative through the Taylor series expansion

$$\frac{dy_{i+1}}{dt} = \frac{dy_i}{dt} + \frac{d^2y_i}{dt^2}h + \dots$$

Truncation of this series after the  $h$  term, gives

$$\frac{d^2y_i}{dt^2} = \frac{\frac{dy_{i+1}}{dt} - \frac{dy_i}{dt}}{h} \quad (3)$$

Thus, the estimated error is given by

$$\epsilon_{i+1} = \frac{d^2y_i}{dt^2} \frac{h^2}{2!} = \frac{\frac{dy_{i+1}}{dt} - \frac{dy_i}{dt}}{h} \frac{h^2}{2!} = \left( \frac{dy_{i+1}}{dt} - \frac{dy_i}{dt} \right) \frac{h}{2}$$

which is the same result as in eq. (1). Note that the error estimate of eq. (1) involves only first derivatives that are immediately available from the ODEs, i.e., the ODEs do not have to be differentiated with respect to  $t$  to get the higher order derivatives in the Taylor series. This is an important property of an error estimator, i.e., it uses only the ODEs with no required differentiation (this idea is the basis of the Runge Kutta method, discussed below).

Finally, if we correct the Euler solution by adding on the estimated error (to get a more accurate solution),

$$y_{i+1} = y_i + \frac{dy_i}{dt}h + \epsilon_{i+1} = y_i + \frac{dy_i}{dt}h + \left( \frac{dy_{i+1}}{dt} - \frac{dy_i}{dt} \right) \frac{h}{2} = y_i + \left( \frac{dy_{i+1}}{dt} + \frac{dy_i}{dt} \right) \frac{h}{2} \quad (4)$$

which is just the modified Euler method (and also, the Taylor series up to and including  $\frac{d^2y_i}{dt^2}h$ ).

However, using eq. (1) rather than eq. (4) is a much better procedure for programming the modified Euler method since the error can be estimated by eq. (1) as the solution proceeds to adjust the step  $h$  to achieve a specified

accuracy. Once  $h$  is reduced to the level required to achieve the required accuracy, this error can then be added to the first order solution (from the Euler method) to get a second order solution (via the modified Euler method), as demonstrated by eq. (4) (and illustrated by the previous six-step algorithm).

Can we generalize this procedure, using algorithms that are higher order than the modified Euler method? Yes - this is the basic idea of the Runge Kutta method.

We consider first the Runge Kutta notation:

$$k_1 = f(y_i, t_i)h \quad (5a)$$

$$k_2 = f(y_i + k_1, t_i + h)h \quad (5b)$$

The Euler method is then

$$y_{i+1} = y_i + k_1 \quad (5c)$$

The modified Euler method is

$$y_{i+1} = y_i + \frac{k_1 + k_2}{2} \quad (5d)$$

The error estimation formula is

$$\epsilon_{i+1} = \frac{k_2 - k_1}{2} \quad (5e)$$

Advantages: Second order accuracy; automatic adjustment of  $h$ ; compact, extensible formulation (notation).

*The Runge Kutta method is a procedure for fitting the Taylor series up to a prescribed number of terms using only first derivatives (from the ODEs). This is accomplished by computing the first derivative at selected points along*

the solution. In other words, the calculation of higher order derivatives in the Taylor series is avoided by calculating the first derivative (from the ODEs) at intermediate points in the interval  $t_i \leq t \leq t_{i+1}$ .

The classical fourth order Runge Kutta method (first reported in 1895), again for the model problem  $dy/dt = f(y, t)$ , is

$$k_1 = f(y_i, t_i)h \tag{6a}$$

$$k_2 = f(y_i + k_1/2, t_i + h/2)h \tag{6b}$$

$$k_3 = f(y_i + k_2/2, t_i + h/2)h \tag{6c}$$

$$k_4 = f(y_i + k_3, t_i + h)h \tag{6d}$$

These are the so-called Runge Kutta constants (which actually aren't constant, but rather, change along the solution). Note that they each involve a derivative evaluation, so that the RK method involves multiple evaluation of the first derivatives of the solution (the RHSs of the ODEs). The stepping formula for this method is then

$$y_{i+1} = y_i + (1/6)(k_1 + 2k_2 + 2k_3 + k_4) \tag{6e}$$

Eq. (6e) fits the Taylor series exactly up to and including the fourth order derivative term,  $\frac{d^4y}{dt^4} \frac{h^4}{4!}$ , but without having to compute the second, third and fourth derivatives (this is the essential idea behind the Runge Kutta method). In analogy with the modified Euler method, eq. (6e) provides a solution of  $O(h^4)$ , but it doesn't provide a way to estimate the integration error to adjust  $h$ . To do these, we first recognize the second order RK

$$k_1 = f(y_i, t_i)h \tag{7a}$$

$$k_2 = f(y_i + k_1/2, t_i + h/2)h \tag{7b}$$

$$y_{i+1} = y_i + k_2 \quad (7c)$$

is *embedded* in the fourth order method of eqs. (6a) to (6e), i.e.,  $k_1$  and  $k_2$  are the same for both methods. Thus, if eq. (7c) is subtracted from eq. (6e) to produce an error estimate,

$$\begin{aligned} \epsilon_{i+1} &= y_{i+1,4} - y_{i+1,2} = y_i + (1/6)(k_1 + 2k_2 + 2k_3 + k_4) - (y_i + k_2) \\ &= (1/6)(k_1 - 4k_2 + 2k_3 + k_4) \end{aligned} \quad (7d)$$

Note how the  $k_1$  and  $k_2$  terms combine in arriving at eq. (7d) since they are the same for both algorithms, i.e., eqs. (6a) and (7a), and (6b) and (7b), are the same.  $\epsilon_{i+1}$  of eq. (7d) can now be used to automatically adjust the integration step,  $h$ . Note also that since this error estimate was achieved by subtracting the stepping formula for a second order method (eq. (7c)), from the stepping formula for a fourth order method (eq. (6e)), the error estimate actually represents two terms in the Taylor series, i.e.,  $\frac{d^3 y}{dt^3} \frac{h^3}{3!}$  and  $\frac{d^4 y}{dt^4} \frac{h^4}{4!}$ , i.e.  $\epsilon_{i+1}$  from eq. (7d) is a *two term error estimate*.

As an additional point, eqs. (5a), (5b) and (5d), and eqs. (7a), (7b) and (7c) are both second RK methods. This demonstrates that there is not a single second order RK method (in fact, there are an infinite number of such methods, depending on how the RK constants and the stepping formula are defined).

Finally, another widely used RK method, Runge Kutta Fehlberg (RKF), is given by

$$k_1 = f(y_i, t_i)h \quad (8a)$$

$$k_2 = f(y_i + k_1/4, t_i + h/4)h \quad (8b)$$

$$k_3 = f(y_i + (3/32)k_1 + (9/32)k_2, t_i + (3/8)h)h \quad (8c)$$

$$k_4 = f(y_i + (1932/2197)k_1 - (7200/2197)k_2 + (7296/2197)k_3, t_i + (12/13)h)h \quad (8d)$$

$$k_5 = f(y_i + (439/216)k_1 - 8k_2 + (3680/513)k_3 - (845/4104)k_4, t_i + h)h \quad (8e)$$

$$k_6 = f(y_i - (8/27)k_1 + 2k_2 - (3544/2565)k_3 + (1859/4104)k_4 - (11/40)k_5, t_i + (1/2)h)h \quad (8f)$$

Note that six derivative evaluations are required, even though the final result will be fifth order (the number of derivative evaluations will, in general, be equal to or greater than the order of the final stepping formula). A fourth order stepping formula is then

$$y_{i+1,4} = y_i + (25/216)k_1 + (1408/2565)k_3 + (2197/4104)k_4 - (1/5)k_5 \quad (8g)$$

and a fifth order stepping formula is (with the  $k$ 's)

$$y_{i+1,5} = y_i + (16/315)k_1 + (6656/12825)k_3 + (28561/56430)k_4 - (9/50)k_5 + (2/55)k_6 \quad (8h)$$

An error estimate can then be obtained by subtracting eq. (8g) from eq. (8h)

$$\epsilon_{i+1} = y_{i+1,5} - y_{i+1,4} \quad (8i)$$

Note that the error estimate of eq. (8i) is a one term estimate (compared to the two term estimate of eq. (7d)). The result from stepping formula (8h) is fifth order correct, i.e., it matches the Taylor series up to and including the term  $\frac{d^5 y_i}{dt^5} \frac{h^5}{5!}$ . Eqs. (8a) to (8i) are programmed in a widely used library subroutine

Name: RKF45

- Features/characteristics: Explicit (nonstiff) RK with error monitoring and step size adjustment; fourth order method embedded in a fifth order method (eqs. (8a) to (8i))
- References: Forsythe, G. E., M. A. Malcolm and C. B. Moler (1977), *Computer Methods for Mathematical Computations*, Prentice-Hall, Englewood Cliffs, NJ; a main program to call RKF45 is given in et al and C. A. Silebi (1997), *Computational Transport Phenomena*, Cambridge University Press, Cambridge, UK

All three algorithms

2nd order RK (modified Euler) Eqs. (5a) to (5e)

4th order RK (classical) Eqs. (6a) to (7d)

5th order (RK Fehlberg) Eqs. (8a) to (8i)

have been coded in library routines. The Fortran code for the previous batch reactor equations that is called by the library integrators is listed below:

```

SUBROUTINE INITAL
C...
C... Dynamic response of a batch reactor
C...
C... The differential equations for a perfectly mixed, isothermal
C... batch reactor (with no inlet/outlet streams) for the first
C... order reaction system
C...
C...      k1      k2
C...      A ----> B ----> C
C...
C... are
C...

```

C...  $V \frac{dca}{dt} = -V k_1 ca, ca(0) = ca_0$  (1)(2)  
C...  
C...  $V \frac{dcb}{dt} = V k_1 ca - V k_2 cb, cb(0) = cb_0$  (3)(4)  
C...  
C...  $V \frac{dcc}{dt} = V k_2 cb, cc(0) = cc_0$  (5)(6)  
C...  
C... where  
C...  
C... ca, cb, cc concentrations of A, B and C, respectively,  
C... (kg-mols/m<sup>3</sup>)  
C...  
C... t time (s)  
C...  
C... V reaction volume (m<sup>3</sup>)  
C...  
C... k1, k2 forward reaction rate constants as indicated  
C... above (1/s)  
C...  
C... ca0, cb0, initial concentrations of A, B, C, respectively,  
C... cc0 (at t = 0), (kg-mols/m<sup>3</sup>)  
C...  
C... Since eqs. (1), (3) and (5) are linear, analytical solutions  
C... can easily be derived, subject to initial conditions (2), (4)  
C... (6) (to simplify the analysis, we take  $cb_0 - cc_0 = 0$ )  
C...  
C...  $ca(t) = ca_0 \exp(-k_1 t)$  (7)  
C...  
C...  $cb(t) = \frac{k_1 ca_0}{k_2 - k_1} (\exp(-k_1 t) - \exp(-k_2 t))$  (8)  
C...  
C...  $cc(t) = ca_0 + \frac{k_1 k_2 ca_0}{k_2 - k_1} \left( \frac{1}{k_2} \exp(-k_2 t) \right.$   
C...  $\left. - \frac{1}{k_1} \exp(-k_1 t) \right)$  (9)  
C...  
C... Eqs. (7), (8) and (9) can be used to evaluate the numerical  
C... solution computed by the Euler method.  
C...

```

        IMPLICIT DOUBLE PRECISION (A-H,K,0-Z)
        COMMON/T/          T
+       /Y/   C(3)
+       /F/   CT(3)
+       /C/   CA0,      K1,      K2
C...
C...  Problem parameters
C...
C...  Rate constants (1/S)
        K1=1.0D0
        K2=2.0D0
C...
C...  Initial concentration of A (kg mols/m**3)
        CA0=1.0D0
C...
C...  Initial conditions (2), (4), (6)
        C(1)=CA0
        C(2)=0.0D0
        C(3)=0.0D0
        RETURN
        END

        SUBROUTINE DERV
        IMPLICIT DOUBLE PRECISION (A-H,K,0-Z)
        COMMON/T/          T
+       /Y/   C(3)
+       /F/   CT(3)
+       /C/   CA0,      K1,      K2
C...
C...  ODEs, eqs. (1), (3), (5)
        CT(1)=-K1*C(1)
        CT(2)= K1*C(1)-K2*C(2)
        CT(3)= K2*C(2)
        RETURN
        END

        SUBROUTINE PRINT(NI,NO)
        IMPLICIT DOUBLE PRECISION (A-H,K,0-Z)

```

```

COMMON/T/      T
+      /Y/    C(3)
+      /F/    CT(3)
+      /C/    CA0,      K1,      K2
C...
C...  t=0 (for heading)
      IF(T.LT.0.00001D0)THEN
          WRITE(NO,2)
2      FORMAT(7X,'t',8X,'cb',7X,'cbe',6X,'error')
      END IF
C...
C...  t = 0, 1, 2, 3, 4, 5
C...
C...  Exact solution and error
      CBE=K1*CA0/(K2-K1)*(DEXP(-K1*T)-DEXP(-K2*T))
      ERROR=C(2)-CBE
C...
C...  Write numerical and exact solutions, error
      WRITE(NO,1)T,C(2),CBE,ERROR
1      FORMAT(F8.2,2F10.4,F11.6)
      RETURN
      END

```

These routines were executed, along with the three library integrators with an error tolerance of 0.0001 absolute. The output is:

2nd order RK:

t	cb	cbe	error
0.00	0.0000	0.0000	0.000000
1.00	0.2325	0.2325	-0.000018
2.00	0.1170	0.1170	-0.000001
3.00	0.0473	0.0473	0.000007
4.00	0.0180	0.0180	0.000009
5.00	0.0067	0.0067	0.000008

4th order RK (classical)

t	cb	cbe	error
0.00	0.0000	0.0000	0.000000
1.00	0.2325	0.2325	0.000000
2.00	0.1170	0.1170	0.000000
3.00	0.0473	0.0473	0.000000
4.00	0.0180	0.0180	0.000000
5.00	0.0067	0.0067	0.000000

5th order RK (RK Fehlberg)

t	cb	cbe	error
0.00	0.0000	0.0000	0.000000
1.00	0.2326	0.2325	0.000011
2.00	0.1170	0.1170	0.000003
3.00	0.0473	0.0473	0.000008
4.00	0.0180	0.0180	0.000001
5.00	0.0067	0.0067	0.000000

Most library ODE integrators provide the option of specifying either an absolute error tolerance, a relative error tolerance, or both. The selection of an error tolerance is an important consideration, and must be done with care (the poor choice of an error tolerance is possibly the single most frequent cause of failures in ODE numerical integration, and in using mathematical software in general).

For example, if we are integrating two ODEs that have as dependent variables, concentration and temperature, and if typical values for the concentration and temperature are  $y = 0.01$ ,  $T = 300$ , we can consider the following possible choices for error tolerances:

absolute	0.0001	reasonable for $y$ too stringent for $T$
absolute	0.1	too loose for $y$ reasonable for $T$

0.0001 is too stringent for  $T$  and therefore will result in excessively small integration steps. 0.1 is too loose for  $y$  and will therefore result in excessive error in computing  $y$ .

We can therefore consider a relative error tolerance, e.g., 0.001, which will be appropriate for both  $y$  and  $T$ . However, if either variable starts out at zero, or passes through zero, a relative error tolerance is not possible (e.g., for example, if  $y$  starts at a zero initial condition).

Thus, we may want to specify both absolute and relative error tolerances. To get around the problem of the absolute error tolerance indicated above, some ODE integrators accept absolute and relative error tolerances for each dependent variable, so that, for example, absolute error tolerances for  $y$  and  $T$  of 0.0001 and 0.1, respectively, could be specified, and a relative error tolerance of 0.001 could be specified for both variables. To reiterate, error tolerances must be chosen with care.

In conclusion, library ODE integrators are available (from WES, from NAG) that automatically adjust the integration step,  $h$ , to achieve numerical solutions with a user-prescribed accuracy.

### **Accuracy and Stability**

So far, we have considered only the *accuracy* of numerical methods for ODEs, e.g., the classical RT method is fourth order correct. There is, however, another important consideration, the *stability* of numerical methods.

For the problems considered so far, accuracy has determined (limited) the step size. But there is an important class of problems for which stability limits the step size, so-called *stiff* problems. We considered these briefly with the batch reactor for which one rate constant was much larger than the other, e.g.,  $k_2 = 10^6 k_1$

We start the discussion of stability by observing what happens when the Euler method is applied to the model problem

$$\frac{dy}{dt} = \lambda y, y(0) = y_0, \lambda < 0 \quad (1)(2)$$

where we have chosen  $\lambda < 0$  so that the solution  $y(t) = y_0 e^{\lambda t}$  is stable (decays exponentially with  $t$ ). If we apply the Euler method to this system,

$$y_1 = y_0 + \frac{dy_0}{dt}h = y_0 + (\lambda y_0)h = y_0(1 + \lambda h)$$

For the next step

$$y_2 = y_1 + \frac{dy_1}{dt}h = y_1 + (\lambda y_1)h = y_1(1 + \lambda h) = y_0(1 + \lambda h)(1 + \lambda h) = y_0(1 + \lambda h)^2$$

In general, after  $n$  steps

$$y_n = y_0(1 + \lambda h)^n \tag{3}$$

How does this Euler solution (eq. (3)) compare with the exact solution to eqs. (1) and (2)

$$y = y_0 e^{\lambda t}, \lambda < 0$$

$\lambda h = -2$	$n$	$y_n$
	1	$y_1 = y_0(1 - 2) = -y_0$
	2	$y_2 = y_1(1 - 2) = -y_0(-1) = y_0$
	3	$y_3 = y_2(1 - 2) = y_0(-1) = -y_0$ , etc.
$\lambda h = -3$	$n$	$y_n$
	1	$y_1 = y_0(1 - 3) = -2y_0$
	2	$y_2 = y_1(1 - 3) = -2y_0(-2) = 4y_0$
	3	$y_3 = y_2(1 - 3) = 4y_0(-2) = -8y_0$ , etc.
$\lambda h = -0.5$	$n$	$y_n$
	1	$y_1 = y_0(1 - 0.5) = 0.5y_0$
	2	$y_2 = y_1(1 - 0.5) = 0.5y_0(0.5) = 0.25y_0$
	3	$y_3 = y_2(1 - 0.5) = 0.25y_0(0.5) = 0.125y_0$ , etc.

Thus,  $|\lambda h| = 2$  is the stability limit of the Euler method when applied to this model problem (eqs. (1), (2)). To confirm this, run hw2s.for with  $h = 1$ .

Also, since the eigenvalues of ODEs can, in general, be complex, the stability criterion  $|\lambda h| = 2$  defines a circle in the complex plane with center at  $(-1,0)$  and unit radius.

Usually, the accuracy requirement will set the step  $h$  to a value much smaller than for  $\lambda h = 2$  (again, refer to the previous examples). However, there is an exception to this conclusion. Consider the system

$$\frac{dy_1}{dt} = -ay_1 + by_2, y_1(0) = 0 \tag{4a}$$

$$\frac{dy_2}{dt} = by_1 - ay_2, y_2(0) = 2 \tag{4b}$$

The solution to system (4) is

$$y_1(t) = -e^{\lambda_1 t} + e^{\lambda_2 t} \tag{5a}$$

$$y_2(t) = e^{\lambda_1 t} + e^{\lambda_2 t} \tag{5b}$$

for which the eigenvalues are

$$\lambda_1 = -(a + b), \lambda_2 = -(a - b) \tag{6}$$

Consider some particular values of  $a$  and  $b$

Case 1		
$a = 50.5$	$\lambda_1 = -100$	$\lambda_1/\lambda_2 = 100$
$b = 49.5$	$\lambda_2 = -1$	nonstiff
Case 2		
$a = 500.5$	$\lambda_1 = -1000$	$\lambda_1/\lambda_2 = 1000$
$b = 499.5$	$\lambda_2 = -1$	moderately stiff
Case 3		
$a = 500,000.5$	$\lambda_1 = -1,000,000$	$\lambda_1/\lambda_2 = 1,000,000$
$b = 499,999.5$	$\lambda_2 = -1$	stiff

ODE systems with widely separated eigenvalues are termed *stiff*. Consider the maximum Euler step for the stiff case. If  $\lambda_1 = -1,000,000$ , the maximum stable step is given by  $|\lambda h| = 2$  or  $h = 2/1,000,000 = 0.000002$ . However, to compute a complete solution, we require a final  $t$  given approximately by  $\lambda_2 t \approx -10$  (or  $t = 10$  so that  $\exp(\lambda_2 t) = \exp(-10)$  has decayed to insignificance compared to the initial condition (4b)). Thus, we must take  $10/0.000002 = 5 \times 10^6$  steps! If this doesn't seem like a large number of steps, consider  $a = 5,000,000.5, b = 4,999,999.5$  for which the ratio  $|\text{largest eigenvalue}/\text{smallest eigenvalue}| = 10^9$  and  $5 \times 10^9$  steps would be required to compute a complete solution (physical problems in which the stiffness ratio  $= |\text{largest eigenvalue}/\text{smallest eigenvalue}| = \lambda_1/\lambda_2 = 10^{12}$  to  $10^{15}$  are not unusual).

As an incidental point, note that the calculation of  $\lambda_2$  requires a subtraction,  $\lambda_2 = -(a - b)$ . If  $a$  and  $b$  are nearly equal, e.g.,  $a = 500,000.5, b = 499,999.5$ , then this subtraction might be done with substantial error. For example, if the machine epsilon is  $10^{-7}$  (one part in  $10^7$ ) corresponding to 32-bit arithmetic, this stiff ODE system could not be integrated numerically since the calculation of  $(a - b)$  requires a precision better more than one part in  $10^7$ . Although this is a heuristic argument, generally the conclusion is correct, i.e., stiff systems require a precision that is substantially better than set by  $\lambda_{\max}/\lambda_{\min}$  (1,000,000 in the preceding example).

Thus, if we require the solution to a system of stiff ODEs, we should not use Euler's method (because of the stability limit  $|\lambda h| = 2$ ). We might consider a higher order method, e.g., the modified Euler method, the classical fourth order RK method, but if we did a similar stability analysis, we would arrive at a stability limit which is not much greater than for the Euler method, e.g., for the classical fourth order RK, the stability limit along the negative real axis is  $|\lambda h| = 2.785$  (and the stability region of this method, is only slightly larger than the unit circle stability region of the Euler method).

In general, any explicit method will have a stability limit similar to the Euler method. Thus, to integrate stiff ODEs, we must use an implicit algorithm (which generally will have a much larger stability region). Therefore, we now consider the *implicit Euler method* applied to the model problem

$$\frac{dy}{dt} = \lambda y, y(0) = y_0, \lambda < 0 \quad (1)(2)$$

where we have chosen  $\lambda < 0$  so that the solution  $y(t) = y_0 e^{\lambda t}$  is stable (decays exponentially with  $t$ ). If we apply the implicit Euler method to this system,

$$y_1 = y_0 + \frac{dy_1}{dt} h = y_0 + (\lambda y_1) h \text{ or } y_1/y_0 = \frac{1}{1 - \lambda h}$$

For the next step

$$y_2 = y_1 + \frac{dy_2}{dt} h = y_1 + (\lambda y_2) h \text{ or } \frac{y_2}{y_1} = \frac{1}{1 - \lambda h}$$

Then for two steps

$$\frac{y_2}{y_0} = \frac{y_2}{y_1} \frac{y_1}{y_0} = \left( \frac{1}{1 - \lambda h} \right) \left( \frac{1}{1 - \lambda h} \right) = \left( \frac{1}{1 - \lambda h} \right)^2$$

In general, after  $n$  steps

$$y_n = y_0 \left( \frac{1}{1 - \lambda h} \right)^n \quad (3)$$

How does this Euler solution (eq. (3)) compare with the exact solution to eqs. (1) and (2)

$$y = y_0 e^{\lambda t}$$

$$\begin{array}{l} \lambda h = -2 \quad n \quad y_n \\ 1 \quad y_1 = y_0/(1 + 2) = y_0/3 \\ 2 \quad y_2 = y_1/(1 + 2) = y_0(1/3)^2 \\ 3 \quad y_3 = y_2/(1 + 2) = y_0(1/3)^3 \text{ etc.} \end{array}$$

and the numerical solution decays in accordance with the exact solution.

Note that in contrast with the explicit Euler method, the implicit Euler method has no limit on  $\lambda h$  with respect to stability, i.e., *the implicit Euler method is unconditionally stable. The step  $h$  is therefore only limited by accuracy.*

However, there is generally a price to be paid for the enhanced stability of implicit methods. If the model equation is nonlinear, i.e.,

$$\frac{dy}{dt} = f(y_i, t_i)$$

and  $f(y_i, t_i)$  is nonlinear, then application of the implicit Euler method gives

$$y_1 = y_0 + \frac{dy_1}{dt}h = y_0 + f(y_1, t_1)h$$

Note that the solution at the advanced point,  $y_1$ , now appears on both sides of the stepping formula, and we therefore must solve a nonlinear equation to compute  $y_1$ . Thus, the price we pay in general when using implicit methods is the *solution of systems of nonlinear (algebraic) equations*. For stiff ODEs, the additional effort of solving systems of nonlinear equations is usually well worthwhile since much larger integration steps are possible because of the improved stability characteristics of implicit methods. Conversely, using a stiff (implicit) integrator on a nonstiff problem will waste computer time since the solution of nonlinear equations is unnecessary.

We have established that the implicit Euler method circumvents the stability problem of stiff ODEs. However, it has limited accuracy (it's only first order correct). Therefore, it would be desirable to have an ODE integration algorithm that is both stable and accurate. This is the intention of the BDF methods.

## **BDF Methods**

As we observed, the implicit Euler method is unconditionally stable, but it also has low accuracy (it is first order). Thus, we seek integration algorithms that have good stability and good accuracy. A widely used implicit method for stiff ODEs that has good stability and accuracy is based on the Backward

Differentiation Formulas (BDFs), which have the general form [Shampine (1994), pp 183-184]

$$\alpha_0 y_{j+1} + \alpha_1 y_j + \cdots + \alpha_\nu y_{j-\nu+1} = hf(x_{j+1}, y_{j+1})$$

and  $dy_i/dt = f(y_i, t_i)$ . The BDF coefficients are

$\nu$	$\alpha_0$	$\alpha_1$	$\alpha_2$	$\alpha_3$	$\alpha_4$	$\alpha_5$	$\alpha_6$
1	1	-1					
2	3/2	-2	1/2				
3	11/6	-3	3/2	-1/3			
4	25/12	-4	3	-4/3	1/4		
5	137/60	-5	-10/3	5/4	-1/5		
6	147/60	-6	15/2	-20/3	15/4	-6/5	1/6

Note that the solution at the advanced point,  $y_{i+1}$ , appears on both sides of the stepping equation so that in general the calculation of  $y_{i+1}$  requires the solution of nonlinear equations (if  $f(y, t)$  from the ODE is nonlinear). Also, for  $\alpha = 1$ , the BDF first order method ( $\nu = 1$ ) is just the implicit Euler method. State of the art implementations of the BDFs are available in the routines LSODE, LSODES and DASSL that vary both  $h$  and  $\nu$  automatically, i.e., they are variable step and variable order implementations (they perform simultaneous  $h$  and  $p$  refinement).

The stability properties of the BDFs are summarized by their stability diagrams. In particular, for  $\nu = 1$ , the unstable region is within a circle centered at  $(1, i0)$  with unit radius (this follows from eq. (3)). Also, the entire LHS of the complex plane is stable for  $\nu \leq 2$ , but a portion of the LHS near the imaginary axis is unstable for  $\nu \geq 3$ , and this unstable region (in the complex plane LHS) increases with increasing  $\nu$ . Thus, stiff ODEs with complex eigenvalues that fall close to the imaginary axis may cause long computer runs due to the limited stability along the imaginary axis (for this case, limiting  $\nu$  to 2 can often increase the computational efficiency substantially, even though the order is relatively low). All of the BDFs for  $\nu \leq 6$  are unconditionally stable along the negative real axis.

As a numerical example, the preceding  $2 \times 2$  ODE problem with  $\lambda_1/\lambda_2 = 1,000,000$  was integrated using LSODES with only a few hundred integration steps. As we observed, an explicit integrator would have required on the order

of  $5 \times 10^6$  steps. The details of the calculation, including a Fortran program consisting of the calling program for LSODES and the function defining the two ODEs, is given in Silebi, et al, *Dynamic Modeling of Transport Process Systems*, Academic Press, San Diego, 1992.

Thus, we require the solution of systems of nonlinear equations to use the stiff (implicit) ODE integration algorithms. In most cases, this is done by Newton's method or a quasi-Newton method. As we concluded previously, we generally have to exploit the structure of the nonlinear system Jacobian matrix for large ODE systems. Thus, stiff ODE integrators are often characterized by the way they process the ODE Jacobian matrix, e.g., full, banded, sparse. Here are some representative, (but high quality) library integrators for stiff ODE (and DAE) systems.

Name: LSODE - (Livermore Solver for Ordinary Differential Equations)

- Features/characteristics: Nonstiff Adams method; stiff BDF method; full and banded Jacobian matrix
- Reference: Hindmarsh, A. C. (1983), *ODEPACK, A Systematized Collection of ODE Solvers*, in Scientific Computing, R. S. Stepleman et al. (eds.), North-Holland, Amsterdam, pp 55-64

Name: LSODES

- Features/characteristics: Nonstiff Adams method; stiff BDF method; sparse Jacobian matrix
- Reference: Hindmarsh, A. C. (1983), *ODEPACK, A Systematized Collection of ODE Solvers*, in Scientific Computing, R. S. Stepleman et al. (eds.), North-Holland, Amsterdam, pp 55-64

Name: DASSL

- Features/characteristics: Stiff BDF method for DAE systems; full and banded Jacobian matrix

- Reference: Brenan, K. E., S. L. Campbell and L. R. Petzold (1989), *Numerical Solution of Initial-value Problems in Differential-algebraic Equations* North-Holland, New York; LSODE, LSODES and DASL are based on the **B**ackward **D**ifferentiation **F**ormulas (BDF) developed by Gear, C. W. (1971), *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ

Name: RADAU5

- Features/characteristics: Stiff Runge Kutta methods for DAE systems; full and banded Jacobian matrix
- Reference: Hairer, E., and G. Wanner (1991), *Solving Ordinary Differential Equations II: Stiff and Differential-algebraic Problems*, Springer-Verlag, Berlin; <http://www.cwi.nl/cwi/projects/IVPtestset/>

Sources of mathematical software include:

Name: Transactions on Mathematical Software (TOMS), Association for Computing Machinery (ACM)

- Features: An extensive library of high quality, public domain mathematical software
- Reference: <http://www.acm.org/dl/search.html>

Name: Netlib (**I**nternet **L**ibrary) - accessed 74,280,550 times by 22APR00

- Features: An extensive library of high quality, public domain mathematical software (including LSODE, LSODES, DASL)
- Reference:  
<http://www.netlib.org/liblist.html>  
Select ode, odepack

[netlib@ornl.gov](mailto:netlib@ornl.gov) (whois ....., send index, send index from toms)

Name: NA-Digest (**N**umerical **A**nalysis Digest; an electronic newsletter)

- Features: A free, weekly newsletter edited by Dr. Cleve Moler, the author of Matlab; to subscribe and get information:  
na.digest@na-net.ornl.gov  
na.whois@na-net.ornl.gov  
na.help@na-net.ornl.gov
- Reference: All past issues of NA-Digest are in Netlib (and they can be searched by subject and author)
- Example:

NA Digest Sunday, April 9, 2000 Volume 00 : Issue 15

Today's Editor:

Cleve Moler The MathWorks, Inc. moler@mathworks.com

Today's Topics:

Five New Codes for Stiff ODEs and DAEs

From: J. Cash [jj.cash@ic.ac.uk](mailto:jj.cash@ic.ac.uk); Date: Thu, 6 Apr 2000 16:59:35 +0100

Subject: Five New Codes for Stiff ODEs and DAEs

This is to announce some new codes and modifications of existing codes for the solution of stiff initial value problems and differential algebraic equations of index less than 4. All of these codes are based on the well known approach defined by Modified Extended Backward Differentiation Formulae. The codes are MEBDFDAE.f which is already in existence and has been slightly modified. This solves linearly implicit DAEs of the form

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

In particular it solves stiff ODEs when  $M = I$ . In addition we have a code MEBDFV.f which solves

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

Also there is a code MEBDFI.f which solves general problems of the form

$$G(x, y, y') = 0$$

In addition there are two sparse solvers. These are MEBDFSO.f which solves sparse systems of initial value ODEs and MEBDFSD.f which solves sparse systems of DAEs. These last two codes have particular applications in the method of lines solution of time dependent partial differential equations. Extensive results comparing these new codes with a whole variety of existing codes together with double precision FORTRAN versions of all the new codes are on the Imperial College web page. This can be accessed at

[http://www.ma.ic.ac.uk/jcash/IVP\\_software/readme.html](http://www.ma.ic.ac.uk/jcash/IVP_software/readme.html)

Any comments will be gratefully received by

[j.cash@ma.ic.ac.uk](mailto:j.cash@ma.ic.ac.uk)

[t.abdulla@ma.ic.ac.uk](mailto:t.abdulla@ma.ic.ac.uk).

The structures of the ODEs defined above are, briefly

$$My' = f(x, y) \text{ (linearly implicit)}$$

$$\begin{aligned} a_{11} \frac{dy_1}{dt} + a_{12} \frac{dy_2}{dt} + \cdots + a_{1n} \frac{dy_n}{dt} &= f_1(y_1, y_2, \cdots, y_n, t) \\ a_{21} \frac{dy_1}{dt} + a_{22} \frac{dy_2}{dt} + \cdots + a_{2n} \frac{dy_n}{dt} &= f_2(y_1, y_2, \cdots, y_n, t) \\ &\vdots \\ a_{n1} \frac{dy_1}{dt} + a_{n2} \frac{dy_2}{dt} + \cdots + a_{nn} \frac{dy_n}{dt} &= f_n(y_1, y_2, \cdots, y_n, t) \end{aligned}$$

$M$  is a coupling matrix

$$M = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & & \ddots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

ODEs of this form typically arise in finite element and weighted residual analysis of PDEs (to be considered subsequently). `MEBDFSD.f` is a sparse routine for this problem (a stiff BDF integrator with sparse Jacobian).

If  $M$  is the identity matrix, the ODEs are uncoupled (there is only one derivative in each ODE, or the ODEs are *explicit* - not to be confused with explicit integration algorithms), i.e.,

$$\begin{aligned}\frac{dy_1}{dt} &= f_1(y_1, y_2, \dots, y_n, t) \\ \frac{dy_2}{dt} &= f_2(y_1, y_2, \dots, y_n, t) \\ &\vdots \\ \frac{dy_n}{dt} &= f_n(y_1, y_2, \dots, y_n, t)\end{aligned}$$

which is the form we have considered in the preceding examples. `MEBDFSO.f` is a sparse routine for this (explicit ODE) problem (a stiff BDF integrator with sparse Jacobian).

If  $M$  has at least one row of zeros, the equation corresponding to that row is algebraic

$$\begin{aligned}a_{11}\frac{dy_1}{dt} + a_{12}\frac{dy_2}{dt} + \dots + a_{1n}\frac{dy_n}{dt} &= f_1(y_1, y_2, \dots, y_n, t) \\ 0 &= f_2(y_1, y_2, \dots, y_n, t) \\ &\vdots \\ a_{n1}\frac{dy_1}{dt} + a_{n2}\frac{dy_2}{dt} + \dots + a_{nn}\frac{dy_n}{dt} &= f_n(y_1, y_2, \dots, y_n, t)\end{aligned}$$

Thus, we have a linearly implicit DAE system. Note that the algebraic equation(s) acts as a constraint on the ODE dependent variables. Also,

the constraint does not have to be algebraic, i.e., a polynomial, (despite the term “Algebraic” in DAE). These problems can be accommodated by MEBDFDAE.f

If the elements of the coupling matrix are functions of the dependent variables,

$$\begin{aligned}
 a_{11}(y_1, y_2, \dots, y_n) \frac{dy_1}{dt} + a_{12}(y_1, y_2, \dots, y_n) \frac{dy_2}{dt} + \dots + a_{1n}(y_1, y_2, \dots, y_n) \frac{dy_n}{dt} \\
 &= f_1(y_1, y_2, \dots, y_n, t) \\
 a_{21}(y_1, y_2, \dots, y_n) \frac{dy_1}{dt} + a_{22}(y_1, y_2, \dots, y_n) \frac{dy_2}{dt} + \dots + a_{2n}(y_1, y_2, \dots, y_n) \frac{dy_n}{dt} \\
 &= f_2(y_1, y_2, \dots, y_n, t) \\
 &\vdots \\
 a_{n1}(y_1, y_2, \dots, y_n) \frac{dy_1}{dt} + a_{n2}(y_1, y_2, \dots, y_n) \frac{dy_2}{dt} + \dots + a_{nn}(y_1, y_2, \dots, y_n) \frac{dy_n}{dt} \\
 &= f_n(y_1, y_2, \dots, y_n, t)
 \end{aligned}$$

this problem can be accommodated by MEBDFV.f

If the equations are fully implicit DAEs (this encompasses all of the preceding cases),

$$\begin{aligned}
 g_1(y_1, y_2, \dots, y_n, \frac{dy_1}{dt}, \frac{dy_2}{dt}, \dots, \frac{dy_n}{dt}, t) &= 0 \\
 g_2(y_1, y_2, \dots, y_n, \frac{dy_1}{dt}, \frac{dy_2}{dt}, \dots, \frac{dy_n}{dt}, t) &= 0 \\
 &\vdots \\
 g_n(y_1, y_2, \dots, y_n, \frac{dy_1}{dt}, \frac{dy_2}{dt}, \dots, \frac{dy_n}{dt}, t) &= 0
 \end{aligned}$$

MEBDFI.f can be used for this fully implicit system. Equations of this form typically arise in various applications in chemical engineering.

Finally, all of the discussion of the various forms of ODEs and DAEs has been in terms of IVPs. A corresponding discussion could be developed for BVPs, including software for this class of problems (which are generally more difficult than IVPs). Software for IVPs and BVPs is evident in the following list of routines from NAG.

## References and Software

In addition to the routines listed in the previous reference, e.g., Netlib, TOMS, IC, an extensive set of routines is available in NAG:

### Chapter D02 - Ordinary Differential Equations

D02AGF - ODEs, boundary value problem, shooting and matching technique, allowing interior matching point, general parameters to be determined

D02BGF - ODEs, IVP, Runge-Kutta-Merson method, until a component attains given value (simple driver)

D02BHF - ODEs, IVP, Runge-Kutta-Merson method, until function of solution is zero (simple driver)

D02BJF - ODEs, IVP, Runge-Kutta method, until function of solution is zero, integration over range with intermediate output (simple driver)

D02CJF - ODEs, IVP, Adams method, until function of solution is zero, intermediate output (simple driver)

D02EJF - ODEs, stiff IVP, BDF method, until function of solution is zero, intermediate output (simple driver)

D02GAF - ODEs, boundary value problem, finite difference technique with deferred correction, simple nonlinear problem

D02GBF - ODEs, boundary value problem, finite difference technique with deferred correction, general linear problem

D02HAF - ODEs, boundary value problem, shooting and matching, boundary values to be determined

D02HBF - ODEs, boundary value problem, shooting and matching, general parameters to be determined

D02JAF - ODEs, boundary value problem, collocation and least-squares, single N- th order linear equation

D02JBF - ODEs, boundary value problem, collocation and least-squares, system of 1st order linear equations

D02KAF - 2nd order Sturm-Liouville problem, regular system, finite range, eigenvalue only

D02KDF - 2nd order Sturm-Liouville problem, regular/singular system, finite/infinite range, eigenvalue only, user-specified break-points  
D02KEF - 2nd order Sturm-Liouville problem, regular/singular system, finite/infinite range, eigenvalue and eigenfunction, user-specified break-points  
D02LAF - 2nd order ODEs, IVP, Runge-Kutta-Nystrom method  
D02LXF - 2nd order ODEs, IVP, set-up for D02LAF  
D02LYF - 2nd order ODEs, IVP, diagnostics for D02LAF  
D02LZF - 2nd order ODEs, IVP, interpolation for D02LAF  
D02MVF - ODEs, IVP, DASSL method, set-up for D02M-N routines  
D02MZF - ODEs, IVP, interpolation for D02M-N routines, natural interpolant  
D02NBF - Explicit ODEs, stiff IVP, full Jacobian (comprehensive)  
D02NCF - Explicit ODEs, stiff IVP, banded Jacobian (comprehensive)  
D02NDF - Explicit ODEs, stiff IVP, sparse Jacobian (comprehensive)  
D02NGF - Implicit/algebraic ODEs, stiff IVP, full Jacobian (comprehensive)  
D02NHF - Implicit/algebraic ODEs, stiff IVP, banded Jacobian (comprehensive)  
D02NJF - Implicit/algebraic ODEs, stiff IVP, sparse Jacobian (comprehensive)  
D02NMF - Explicit ODEs, stiff IVP (reverse communication, comprehensive)  
D02NNF - Implicit/algebraic ODEs, stiff IVP (reverse communication, comprehensive)  
D02NRF - ODEs, IVP, for use with D02M-N routines, sparse Jacobian, enquiry routine  
D02NSF - ODEs, IVP, for use with D02M-N routines, full Jacobian, linear algebra set-up  
D02NTF - ODEs, IVP, for use with D02M-N routines, banded Jacobian, linear algebra set-up  
D02NUF - ODEs, IVP, for use with D02M-N routines, sparse Jacobian, linear algebra set-up  
D02NVF - ODEs, IVP, BDF method, set-up for D02M-N routines  
D02NWF - ODEs, IVP, Blend method, set-up for D02M-N routines  
D02NXF - ODEs, IVP, sparse Jacobian, linear algebra diagnostics, for use with D02M-N routines  
D02NYF - ODEs, IVP, integrator diagnostics, for use with D02M-N routines  
D02NZF - ODEs, IVP, set-up for continuation calls to integrator, for use with D02M-N routines,  
D02PCF - ODEs, IVP, Runge-Kutta method, integration over range with

output

D02PDF - ODEs, IVP, Runge-Kutta method, integration over one step  
D02PVF - ODEs, IVP, set-up for D02PCF and D02PDF  
D02PWF - ODEs, IVP, resets end of range for D02PDF  
D02PXF - ODEs, IVP, interpolation for D02PDF  
D02PYF - ODEs, IVP, integration diagnostics for D02PCF and D02PDF  
D02PZF - ODEs, IVP, error assessment diagnostics for D02PCF and D02PDF  
D02QFF - ODEs, IVP, Adams method with root-finding (forward communication, comprehensive)  
D02QGF - ODEs, IVP, Adams method with root-finding (reverse communication, comprehensive)  
D02QWF - ODEs, IVP, set-up for D02QFF and D02QGF  
D02QXF - ODEs, IVP, diagnostics for D02QFF and D02QGF  
D02QYF - ODEs, IVP, root-finding diagnostics for D02QFF and D02QGF  
D02QZF - ODEs, IVP, interpolation for D02QFF or D02QGF  
D02RAF - ODEs, general nonlinear boundary value problem, finite difference technique with deferred correction, continuation facility  
D02SAF - ODEs, boundary value problem, shooting and matching technique, subject to extra algebraic equations, general parameters to be determined  
D02TGF - N-th order linear ODEs, boundary value problem, collocation and least-squares  
D02TKF - ODEs, boundary value problems, collocation technique, general nonlinear problem solver  
D02TVF - ODEs, boundary value problems, collocation technique, set-up routine  
D02TXF - ODEs, boundary value problems, collocation technique, continuation routine  
D02TYF - ODEs, boundary value problems, collocation technique, interpolation routine  
D02TZF - ODEs, boundary value problems, collocation technique, diagnostic routine  
D02XJF - ODEs, IVP, interpolation for D02M-N routines, natural interpolant  
D02XKF - ODEs, IVP, interpolation for D02M-N routines, C1 interpolant  
D02ZAF - ODEs, IVP, weighted norm of local error estimate for D02M-N routines

Thus, the starting point in selecting a routine(s) from NAG (Netlib, TOMS) is the classification of the problem, particularly the degree of coupling be-

tween the ODEs, and whether they are an initial value problem (IVP) or a boundary value problem (BVP).