Other Algorithms for Ordinary Differential Equations

Larry Caretto
Mechanical Engineering 309

Numerical Analysis of Engineering Systems

April 28, 2014

Outline

- Schedule
- Review systems of ODEs
  - Spring-mass-damper problem with two masses as example
- Using ODE solvers in MATLAB
- Other approaches for solving the initial value problem
  - Multistep methods
  - Implicit methods
  - Extrapolation methods

2

Remaining Course Schedule

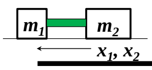
- April 28 (today) – More on ODEs; programming assignment six due
- April 30 – Last quiz (on ODEs). Final lecture on numerical solutions of ODEs
- May 5 – Review for final and programming exams; programming assignment seven due
- May 7 – Programming exam
- May 12 – Final exam, 8 – 10 pm

3

Review Systems of ODEs

- Can convert n[th] order ODE into n first-order ODEs
- Can apply algorithms for one first-order ODE to systems of first-order ODEs
  - Must have initial conditions on all variables
  - Converting an n[th] order ODE to n first-order ODEs gives n – 1 derivative ODEs whose initial values we need
  - Must apply each step of algorithms to all ODEs before going on to next step

4



Example

- Two masses joined by a spring/damper
- Original ODEs for each mass

$$m_1\frac{d^2x_1}{dt^2}+c\left(\frac{dx_1}{dt}-\frac{dx_2}{dt}\right)+k(x_1-x_2)=F_1$$

$$m_2\frac{d^2x_2}{dt^2}+c\left(\frac{dx_2}{dt}-\frac{dx_1}{dt}\right)+k(x_2-x_1)=F_2$$

- Define velocities $\frac{dx_1}{dt}=v_1$ $\frac{dx_2}{dt}=v_2$
- Rewrite original ODEs using velocities

$$\frac{dv_1}{dt}+\frac{c}{m_1}(v_1-v_2)+\frac{k}{m_1}(x_1-x_2)=\frac{F_1}{m_1}$$

$$\frac{dv_2}{dt}+\frac{c}{m_2}(v_2-v_1)+\frac{k}{m_2}(x_2-x_1)=\frac{F_2}{m_2}$$

k = spring constant (N/m)
c = damping coefficient (kg/s)

5

Example Continued

- Replace x_1, x_2, v_1, v_2 in equations below by y_1, y_2, y_3, y_4

$$\frac{dx_1}{dt}=v_1 \qquad \frac{dv_1}{dt}+\frac{c}{m_1}(v_1-v_2)+\frac{k}{m_1}(x_1-x_2)=\frac{F_1}{m_1}$$

$$\frac{dx_2}{dt}=v_2 \qquad \frac{dv_2}{dt}+\frac{c}{m_2}(v_2-v_1)+\frac{k}{m_2}(x_2-x_1)=\frac{F_2}{m_2}$$

- Result is standard-form system: $dy_k/dt = f_k$

$$\frac{dy_1}{dt}=f_1=y_3 \qquad \frac{dy_3}{dt}=f_3=\frac{F_1}{m_1}-\frac{c}{m_1}(y_3-y_4)-\frac{k}{m_1}(y_1-y_2)$$

$$\frac{dy_2}{dt}=f_2=y_4 \qquad \frac{dy_4}{dt}=f_4=\frac{F_2}{m_2}-\frac{c}{m_2}(y_4-y_3)-\frac{k}{m_2}(y_2-y_1)$$

6

MATLAB Derivative Function

```
function f = springMassDamper(t, y)
m1=1;  m2=2;  c = 0.5; k = 1;
f = zeros(4,1);
f(1) = y(3);
f(2) = y(4);
f(3) = (c*(y(4)-y(3))+k*(y(2)-y(1)))/m1;
f(4) = (c*(y(3)-y(4))+k*(y(1)-y(2)))/m2;
End
>>[t y] = ode45(@springMassDamper, [0 1], …
    [1 -1 0 0])
```

California State University Northridge 7

General System Form

- Have N ODEs with common form: $\frac{dy_m}{dt} = f_m$
- Each f_m may depend on t and all y_m
- Equations for f_m (in terms of t and all **y** values) depend on problem description
- Apply usual algorithms $y_{i+1} = y_i + hf_{avg}$ to each equation: $y_{m,i+1} = y_{m,i} + hf_{avg,m}$
  - $y_{m,i}$ is value of y_m at $t = t_i$ (or $x = x_i$)
- Must do each step/substep to all equations before taking next step/substep

California State University Northridge t_{i-2} t_{i-1} t_i t_{i+1} t_{i+2} t_{i+3} 8

How to Code This

- For any algorithm, each step must be done for all equations
- All equations have the form $dy_m/dt = f_m$
- User-defined function, f = fSub(t, y), computes all f values for input t, **y**
- Each step, in each algorithm, is a loop over all equations getting appropriate updates
  - Common time value for all y_k to compute f_k

California State University Northridge 9

Fourth-order Runge Kutta (RK4)

- Uses four derivative evaluations per step

$$y_{i+1} = y_i + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \qquad t_{i+1} = t_i + h$$

$$k_1 = hf(t_i, y_i)$$

$$k_2 = hf\left(t_i + \frac{h}{2}, y_i + \frac{k_1}{2}\right)$$

$$k_3 = hf\left(t_i + \frac{h}{2}, y_i + \frac{k_2}{2}\right)$$

$$k_4 = hf(t_i + h, y_i + k_3)$$

Look at code for this algorithm, then see changes to apply to a system of equations

California State University Northridge t_{i-2} t_{i-1} t_i t_{i+1} t_{i+2} t_{i+3} 10

RK4 Code, one ODE

```
h = (tEnd - tStart)/nSteps
For step = 1 To nSteps
    t = tStart + h * (step - 1)
    f = fFct(t, y)  'initial y values
    k1 = h * f
    f = fFct(t + h / 2, y + k1/2)
    k2 = h * f
    f = fFct(t + h / 2, y + k2/2)
    k3 = h * f
    f = fFct(t + h, y + k3)
    y = y + (k1 + 2*k2 + 2*k3 + h*f) / 6
Next step
```

t = independent variable at start of step = t_i

Use Application.Run("fFct",t,y) in VBA

k4 (h*f)

Use same program variable, y, for y_{i+1} and y_i

California State University Northridge 11

RK4 for N ODEs

- $y_{m,i}$ is value of m^{th} y variable at t_i

$$y_{m,i+1} = y_{m,i} + \frac{k_{1,m} + 2k_{2,m} + 2k_{3,m} + k_{4,m}}{6} \qquad t_{i+1} = t_i + h$$

$$k_{1,m} = hf_m(t_i, \mathbf{y}_i)$$

$$k_{2,m} = hf_m\left(t_i + \frac{h}{2}, \mathbf{y}_i + \frac{\mathbf{k}_1}{2}\right)$$

$$k_{3,m} = hf_m\left(t_i + \frac{h}{2}, \mathbf{y}_i + \frac{\mathbf{k}_2}{2}\right)$$

$$k_{4,m} = hf_m(t_i + h, \mathbf{y}_i + \mathbf{k}_3)$$

Vector notation for **y** and **k** shows that (1) f_m can depend on all y_m values and (2) each f_m calculation requires all y values to be updated

California State University Northridge t_{i-2} t_{i-1} t_i t_{i+1} t_{i+2} t_{i+3} 12

RK4 Code – Multiple ODEs

```
h = (xEnd - xStart) / nSteps
For step = 1 To nSteps
    x = xStart + h * (step - 1)
    f = fFct(x, y)  'initial y values
    For m = 1 To N
        k1(m) = h * f(m)
        yTemp(m) = y(m) + 0.5 * k1(m)
    Next m
    f = fFct(x + h / 2, yTemp)
    For m = 1 To N
        k2(m) = h * f(m)
        yTemp(m) = y(m) + 0.5 * k2(m)
```

Use Application. Run("fFct",t,y) in VBA

Example: 4th-order Runge-Kutta

```
    Call fFct(x + h / 2, yTemp, f)
    For m = 1 To N
        k2(m) = h * f(m)
        yTemp(m) = y(m) + 0.5 * k2(m)
    Next m
    f= fFct(x + h / 2, yTemp)
    For m = 1 To N
        k3(m) = h * f(m)
        yTemp(m) = y(m) + k3(m)
    Next m
    f = fFct(x + h, yTemp)
```


Example: 4th-order Runge-Kutta

```
    f = fFct(x + h, yTemp)
    For m = 1 To N
        k4(m) = h * f(m)
        y(m) = y(m) + (k1(m) + 2 * k2(m) _
             + 2 * k3(m) + h * f(m)) / 6
    Next m
Next step
```

Use same program variable, y(m), for $y_{m,i+1}$ and $y_{m,i}$

k4(m)

- These new y values are used at start of loop to begin next step
  - Same statements handle function input values for y(m)

ODE Solvers in MATLAB

- Several different solvers
- For initial value problems the general function call is [t, y] = solverName(derivativeF, tSpan, y0, options), where
  - t is a column vector of "time" points output by the calculation
  - y is the output matrix for the solution
    - Column k of y is the solution for variable y_k
    - Each row of y is the solution of all y_k for the "time" point in the same row of t

ODE Solvers in MATLAB II

- derivativeF is the handle for a function that evaluates the derivatives, f(t,y)
  - In derivativeF(t,y), t is a scalar time, and y is a column vector of the dependent variables
  - The function returns a column vector for f
  - The user has to write this function to define the problem being solved
- The tSpan argument is a row matrix that must give at least the initial and final time
  - MATLAB uses time is as the name of the independent variable, which can be any quantity

ODE Solvers in MATLAB III

- If there are only the minimum of two points (start and end) solvers will give output for each time (independent variable) used in calculation
  - Voluminous output good for smooth plots
- If three or more points are used in input, only these input times will appear in output
- The y_0 argument is a vector for initial conditions of the dependent y variables
  - Y0 = [1 5 12 -32] gives $y_1(0) = 1$, $y_2(0) = 5$, …
- The options argument allows the user to override normal defaults in the solver
  - See MATLAB help for more options information

ODE Solvers in MATLAB

- Solver names: ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb
  - ode45 should be first choice
    - This is a Runge-Kutta procedure that uses a fourth and fifth order expressions, called the Dormand-Prince pair, to adjust step size, h
  - ode113 is a multistep algorithm based on the Adams-Bashfort-Moulton approach
  - Application information for solvers from MATLAB help on next slide

MATLAB Solver Help

Solver	Problem Type	Order of Accuracy	When to Use
ode45	Nonstiff	Medium	Most of the time. This should be the first solver you try
ode23	Nonstiff	Low	Problems with crude error tolerances or for solving moderately stiff problems
ode113	Nonstiff	Low to high	Problems with stringent error tolerances or computationally intensive problems
ode15s	Stiff	Low to medium	If ode45 is slow because the problem is stiff
ode23s	Stiff	Low	With crude error tolerances to solve stiff systems (mass matrix is constant)
ode23t	Modera-tely Stiff	Low	Moderately stiff problems if you need a solution without numerical damping.
ode23tb	Stiff	Low	If using crude error tolerances to solve stiff systems.

MATLAB ode45 Example

```
>> type odeF.m
function f = odeF( t, y )
%odeF -- sample ode derivative routine
    f = zeros(3,1);
    f(1) = -y(2)*y(2)/y(3);    %Use semi-
    f(2) = -2*y(2)*y(3)/y(1)^3;   %colons
    f(3) = -3*y(1)*y(2); %to avoid prints
end
>> tS = [0 .1 .2 .4 .6 .8 1];  %Time data
>> y0 = [1 1 1 ]';    %Initial y values
>> [t y] = ode45(@odeF,tS,y0) %use solver
%Output time, t, and solution, y on next
%slide
```


MATLAB ode45 Example II

```
t =       0    y = 1.0000    1.0000    1.0000
    0.1000         0.9048    0.8187    0.7408
    0.2000         0.8187    0.6703    0.5488
    0.4000         0.6703    0.4493    0.3012
    0.6000         0.5488    0.3012    0.1653
    0.8000         0.4493    0.2019    0.0907
    1.0000         0.3679    0.1353    0.0498
    %Results shown only for specified times
    %If t array were entered as [0 1] results
    %  for all times would be displayed
    %If exact solution, yExact known, errors
    %in numerical solutions for all times are
    >> err = abs([y - yExact])
```


Numerical ODE Approaches

- Have seen explicit, single-step, methods, like Runge-Kutta, that solve for y_{n+1} using only values at step n
- Implicit methods use information about point n+1 in algorithm for y_{n+1}; some sort of approximation required
- Multistep methods use information from steps n – 1, n – 2, *etc*.
- Extrapolation methods

Implicit Methods

- Methods discussed previously are called explicit
  - Can find y_{n+1} in terms of values at n
  - Use predictors to estimate y values between y_n and y_{n+1}
- Implicit methods use f_{n+1} in algorithm
- Usually require approximate solution
- Can use larger h values with more work per step compared to explicit methods
- Trapezoid method is an example

Trapezoid Method I

- Basic implicit result for this method

$$y_{n+1} - y_n = \left(f_{n+1} + f_n\right)\frac{h}{2} + O(h^3)$$

- Need way to compute f_{n+1} when we do not know y_{n+1}
  - First approach: replace f_{n+1} by Taylor series

$$y_{n+1} - y_n = \frac{h}{2}\left[f_n + \boxed{f_n + \left.\frac{\partial f}{\partial x}\right|_n h + \left.\frac{\partial f}{\partial y}\right|_n \left(y_{n+1} - y_n\right)} \right]$$

- Have to compute f(x,y) partial derivatives

$$\left(y_{n+1} - y_n\right) = \frac{hf_n + \left.\frac{\partial f}{\partial x}\right|_n \frac{h^2}{2}}{1 - \frac{h}{2}\left.\frac{\partial f}{\partial y}\right|_n}$$

California State University Northridge 25

Trapezoid Method II

- Another approach to using f_{n+1} in algorithm to solve for the unknown y_{n+1}
  - Use an explicit approach to get an initial approximation for y_{n+1}
  - Iterate on implicit method
    - E.g.: Euler step for first approximation of y_{n+1}

$$y_{n+1}^{(0)} = y_n + hf_n$$

$$y_{n+1}^{(m+1)} = y_n + \frac{h\left[f_n + f\left(x_{n+1}, y_{n+1}^{(m)}\right)\right]}{2}$$

California State University Northridge 26

Trapezoid Method III

- Use Newton-Raphson iteration for y_{n+1}
  - Solve g(x) = 0 by iteration $x^{(m+1)} = x^{(m)} - g(x^{(m)}) / g'(x^{(m)})$
  - $g(y_{n+1}) = y_{n+1} - y_n - hf_n/2 - hf(x_{n+1},y_{n+1})/2$
  - $g'(y_{n+1}) = f_{n+1} - 0 - 0 - h(\partial f/\partial y)/2$

$$y_{n+1}^{(m+1)} = y_{n+1}^{(m)} - \frac{y_{n+1}^{(m)} - y_n - \frac{hf_n}{2} - \frac{hf\left(x_{n+1}, y_{n+1}^{(m)}\right)}{2}}{f\left(x_{n+1}, y_{n+1}^{(m)}\right) - \frac{h}{2}\left(\frac{\partial f}{\partial y}\right)_{n+1}^{(m)}}$$

California State University Northridge 27

Trapezoid Method Derivation

- Subtract series expansion for y_n about y_{n+1} from series for y_{n+1} about y_n

$$y_{n+1} = y_n + f_n h + \frac{h^2 y_n''}{2} + O(h^3)$$

$$y_n = y_{n+1} - f_{n+1} h + \frac{h^2 y_{n+1}''}{2} + O(h^3)$$

$$y_{n+1} - y_n = y_n - y_{n+1} + f_n h + f_{n+1} h + \frac{h^2\left(y_n'' - y_{n+1}''\right)}{2} + O(h^3)$$

California State University Northridge 28

Trapezoid Method Derivation II

- Collect terms is last equation and substitute $y''_{n+1} = y''_n + hy_n''' + O(h^2)$

$$y_{n+1} - y_n = \left(f_n + f_{n+1}\right)\frac{h}{2} + \frac{h^2\left(y_n'' - y_{n+1}''\right)}{4} + O(h^3)$$

$$y_{n+1} - y_n = \left(f_n + f_{n+1}\right)\frac{h}{2} + \frac{h^2\left[y_n'' - y_n'' - hy_n''' - O(h^2)\right]}{4} + O(h^3)$$

$$y_{n+1} - y_n = \left(f_{n+1} + f_n\right)\frac{h}{2} + O(h^3)$$

California State University Northridge 29

Trapezoid Method Example

- Look at sample equation dy/dx = f = -ay
- Here, $f_n = -ay_n$, $\partial f/\partial x = 0$ and $\partial f/\partial y = -a$

$$y_{n+1} = y_n + \frac{2hf_n + \left.\frac{\partial f}{\partial x}\right|_n h^2}{2 - h\left.\frac{\partial f}{\partial y}\right|_n} = y_n + \frac{-2hay_n + 0}{2 - h(-a)}$$

$$= \frac{y_n(2 + ha) - 2hay_n}{2 + ha} = \frac{(2 - ha)y_n}{2 + ha}$$

- So $y_{n+1} = G\, y_n$ with $G = (2 - ha)/(2 + ha)$
- Will use this later in stability discussion

California State University Northridge 30

Multistep Methods

- Previous methods used only information from most recent step (y_n and f_n)
- Took intermediate steps between x_n and x_{n+1} to improve accuracy
- Multistep methods use information from previous steps for improved accuracy with less work than single step methods
- Need starting procedure that is a single step method

California State University Northridge 31

Multistep Method Derivation

- Uses interpolation polynomial that passes through previous points
- Polynomial is integrated from x_n to x_{n+1}
- Resulting expression gives y_{n+1} in terms of values and derivatives of previous steps
- Leads to process known as predictor-corrector with two expressions for y_{n+1} and an error control expression

California State University Northridge 32

Adams-Bashforth-Moulton

- Predictor corrector method
- Predictor equation uses derivative values from four points

$$y_{n+1}^P = y_n + \frac{h}{24}\left(55 f_n - 59 f_{n-1} + 37 f_{n-2} - 9 f_{n-3}\right)$$

- Corrector equation uses four points including point n+1 with predicted y^P

$$y_{n+1}^C = y_n + \frac{h}{24}\left[9 f\left(x_{n+1}, y_{n+1}^P\right) + 19 f_n - 5 f_{n-1} + f_{n-2}\right]$$

California State University Northridge 33

Adams-Bashforth-Moulton II

- Use difference between predictor and corrector results to get error estimate

$$y_{n+1}^P = y_n + \frac{h}{24}\left(55 f_n - 59 f_{n-1} + 37 f_{n-2} - 9 f_{n-3}\right)$$

$$y_{n+1}^C = y_n + \frac{h}{24}\left[9 f\left(x_{n+1}, y_{n+1}^P\right) + 19 f_n - 5 f_{n-1} + f_{n-2}\right]$$

- Derivation result (next two slides) gives error estimate in terms of $(y^P - y^C)_{n+1}$

$$E_C = -\frac{19}{720} h^5 y^{(v)}(\xi_C) \approx \frac{19}{270}\left(y_{n+1}^P - y_{n+1}^C\right)$$

California State University Northridge 34

Derive Error Equation

- From an error analysis of the integrated interpolation polynomials we can find

$$y(x_{n+1}) = y_{n+1}^P + \frac{251}{720} h^5 y^{(v)}(\xi_P)$$

$$y(x_{n+1}) = y_{n+1}^C - \frac{19}{720} h^5 y^{(v)}(\xi_C)$$

1. Subtract equations
2. Subtract and add $y^{(v)}(\xi_C)$ term

$$0 = y_{n+1}^P - y_{n+1}^C + \left(\frac{251}{720} + \frac{19}{720}\right) h^5 y^{(v)}(\xi_C) + \frac{251}{720} h^5\left[y^{(v)}(\xi_P) - y^{(v)}(\xi_C)\right]$$

- Neglect $y^{(v)}(\xi_P) - y^{(v)}(\xi_C)$

$$y_{n+1}^C - y_{n+1}^P = \left(\frac{251}{720} + \frac{19}{720}\right) h^5 y^{(v)}(\xi_C)$$

California State University Northridge 35

Derive Error Equation

- Solve for E_C, the corrector error

$$E_C = y(x_{n+1}) - y_{n+1}^C = -\frac{19}{720} h^5 y^{(v)}(\xi_C)$$

$$y_{n+1}^C - y_{n+1}^P = \left(\frac{251}{720} + \frac{19}{720}\right) h^5 y^{(v)}(\xi_C) = \frac{270}{720} h^5 y^{(v)}(\xi_C)$$

$$E_C = -\frac{19}{720} h^5 y^{(v)}(\xi_C) = -\frac{19}{720}\frac{y_{n+1}^C - y_{n+1}^P}{\frac{270}{720}} = \frac{19}{270}\left(y_{n+1}^P - y_{n+1}^C\right)$$

- Error estimate gives step size control
- How to change h in multistep method?

California State University Northridge 36

Step Size Control

- Establish e_{min} and e_{max} to achieve desired problem accuracy
- If $e_{min} \le E_C \le e_{max}$, do not change h
- If $E_C < e_{min}$ double step size, h
- If $E_C > e_{max}$ half step size, h
- Carry extra steps to be ready for step-size doubling
- Interpolate data if h is cut in half

California State University Northridge 37

Grid halving if error too large

- Normal operation, no step size change

i-3 i-2 i-1 i i+1 (old step)

●---------●---------●---------●---------●---------●

(new) i-3 i-2 i-1 i i+1

- Error too large: Half grid size and repeat step

i-3 i-2 i-1 i i+1 (old step)

●----o----●----o----●----o----●----o----●

(repeated) i-3 i-2 i-1 i i+1

California State University Northridge (interpolated points) 38

Grid doubling for very small error

- Normal operation, no step size change

i-5 i-4 i-3 i-2 i-1 i i+1 (old step)

o-----o-----o-----o-----o-----o-----o-----o

i-5 i-4 i-3 i-2 i-1 i i+1 (new)

- Error very small: Double grid size

i-5 i-4 i-3 i-2 i-1 i i+1 (old step)

●-----o-----●-----o-----●-----o-----●-----------●

i-3 i-2 i-1 i i+1

California State University Northridge (Retained to use for doubling) 39

Grid Halving and Doubling

- Keep extra values f_{i-4} and f_{i-5} in memory to be ready for grid doubling
  - $f_{i-3,new} = f_{i-5}$; $f_{i-2,new} = f_{i-3}$; $f_{i-1,new} = f_{i-1}$; $f_{i,new} = f_{i+1}$
- Grid halving requires interpolation for missing values in old grid
  - $f_{i-2,new} = f_{i-1}$; $f_{i,new} = f_i$

$$f_{i-1,new} = \frac{1}{128}\left[-5 f_{i-4} + 28 f_{i-3} - 70 f_{i-2} + 140 f_{i-1} + 35 f_i\right]$$

$$f_{i-3,new} = \frac{1}{64}\left[3 f_{i-4} - 16 f_{i-3} + 54 f_{i-2} + 24 f_{i-1} - f_i\right]$$

California State University Northridge 40

Use of Multistep Methods

- Many different equations possible with different orders and errors
- Used for high accuracy computation requirements with less computer time
- Used in high-accuracy MATLAB solver ode113
- Runge-Kutta type methods easier to apply, and can have error control for lower accuracy requirements

California State University Northridge 41

Extrapolation Methods

- Use Richardson extrapolation for better estimate from results on two values of h
  - Construct large step, H, between two x values, x and x + H
    - Subdivide H into n smaller steps, h = H/n
    - Compute intermediate approximations to y, called z_m for the substep index, m
    - Use Richardson extrapolation for different m's
  - Bulirsch-Stoer method uses extrapolation and rational function approximation

California State University Northridge 42