Other Algorithms for Ordinary Differential Equations

Larry Caretto
Mechanical Engineering 309
Numerical Analysis of Engineering Systems
April 28, 2014

Remaining Course Schedule

- April 28 (today) – More on ODEs; programming assignment six due
- April 30 – Final programming exam
- May 5 – Review systems of ODEs
- May 7 – Programming exam
- May 12 – Final exam, 8 – 10 pm

Example

- Two masses joined by a spring/damper
- Original ODEs for each mass
- Define velocities
- Rewrite original ODEs using velocities

Example Continued

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

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

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

ME 309 – Numerical Analysis of Engineering Systems

April 28, 2014

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

Outline
- 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

Example
- Two masses joined by a spring/damper
- Original ODEs for each mass
- Define velocities
- Rewrite original ODEs using velocities

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

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

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

Outline
- April 28 (today) – More on ODEs; programming assignment six due
- April 30 – Final programming exam
- May 5 – Review systems of ODEs
- May 7 – Programming exam
- May 12 – Final exam, 8 – 10 pm

Example
- Two masses joined by a spring/damper
- Original ODEs for each mass
- Define velocities
- Rewrite original ODEs using velocities

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

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

Outline
- April 28 (today) – More on ODEs; programming assignment six due
- April 30 – Final programming exam
- May 5 – Review systems of ODEs
- May 7 – Programming exam
- May 12 – Final exam, 8 – 10 pm

Example
- Two masses joined by a spring/damper
- Original ODEs for each mass
- Define velocities
- Rewrite original ODEs using velocities

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

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

Outline
- April 28 (today) – More on ODEs; programming assignment six due
- April 30 – Final programming exam
- May 5 – Review systems of ODEs
- May 7 – Programming exam
- May 12 – Final exam, 8 – 10 pm

Example
- Two masses joined by a spring/damper
- Original ODEs for each mass
- Define velocities
- Rewrite original ODEs using velocities

Example Continued
- Replace $x_1, x_2, y_1, v_1, v_2$ in equations below by $y_1, y_2, y_3, y_4$
**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])
```

**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 \( y_{m,i+1} = y_i + hf_{avg,m} \) to each equation: \( y_{m,i+1} = y_{m,i} + hf_{avg,m} \)
- \( y_{m,i} \) is value of \( y_m \) at \( t_i \) (or \( x = x_i \))
- Must do each step/substep to all equations before taking next step/substep

**How to Code This**

- For any algorithm, each step must be done for all equations
- All equations have the form \( \frac{dy_m}{dt} = f_m \)
- User-defined function, \( f = f\text{Sub}(t, y) \), 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 \( f_k \)

**Fourth-order Runge Kutta (RK4)**

- Uses four derivative evaluations per step
  - \( y_{i+1} = y_i + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \)
  - \( t_{i+1} = t_i + h \)
  - \( k_1 = hf(t_i, y_i) \)
  - \( k_2 = hf(t_i + h/2, y_i + k_1/2) \)
  - \( k_3 = hf(t_i + h/2, y_i + k_2/2) \)
  - \( k_4 = hf(t_i + h, y_i + k_3) \)
  - Use same program variable, \( y \) for \( y_{i+1} \) and \( y_i \)

**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 Use Application.Run("Fct",t,y) in VBA
    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 + 2k2 + 2k3 + h4)/6
Use same program variable, y for y_{i+1} and y_i
```

**RK4 for N ODEs**

- \( y_{m,i+1} \) is value of \( m^{th} \) \( y \) variable at \( t_i \)
  - \( y_{m,i+1} = y_{m,i} + \frac{k_{1,m} + 2k_{2,m} + 2k_{3,m} + k_{4,m}}{6} \)
  - \( t_{i+1} = t_i + h \)
  - \( k_{1,m} = hf_m(t_i, y_i) \)
  - \( k_{2,m} = hf_m(t_i + h/2, y_i + k_{1,m}/2) \)
  - \( k_{3,m} = hf_m(t_i + h/2, y_i + k_{2,m}/2) \)
  - \( k_{4,m} = hf_m(t_i + h, y_i + k_{3,m}) \)
- 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 \( y_m \) values to be updated
RK4 Code – Multiple ODEs

\[ h = \frac{(x_{\text{End}} - x_{\text{Start}})}{\text{nSteps}} \]

For step = 1 To nSteps

\[ x = x_{\text{Start}} + h \times (\text{step} - 1) \]

\[ f = fFct(x, y) \quad \text{‘initial y values} \]

For m = 1 To N

\[ k1(m) = h \times f(m) \]

\[ y_{\text{Temp}}(m) = y(m) + 0.5 \times k1(m) \]

Next m

\[ f = fFct(x + h / 2, y_{\text{Temp}}) \]

For m = 1 To N

\[ k2(m) = h \times f(m) \]

\[ y_{\text{Temp}}(m) = y(m) + 0.5 \times k2(m) \]

Next m

\[ f = fFct(x + h / 2, y_{\text{Temp}}) \]

For m = 1 To N

\[ k3(m) = h \times f(m) \]

\[ y_{\text{Temp}}(m) = y(m) + k3(m) \]

Next m

\[ f = fFct(x + h, y_{\text{Temp}}) \]

For m = 1 To N

\[ k4(m) = h \times f(m) \]

\[ y(m) = y(m) + \frac{(k1(m) + 2 \times k2(m) + 2 \times k3(m) + k4(m))}{6} \]

Next m

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

Example: 4th-order Runge-Kutta

\[ \text{Call } f\text{Fct}(x + h / 2, y_{\text{Temp}}, f) \]

For m = 1 To N

\[ k2(m) = h \times f(m) \]

\[ y_{\text{Temp}}(m) = y(m) + 0.5 \times k2(m) \]

Next m

\[ f = f\text{Fct}(x + h / 2, y_{\text{Temp}}) \]

For m = 1 To N

\[ k3(m) = h \times f(m) \]

\[ y_{\text{Temp}}(m) = y(m) + k3(m) \]

Next m

\[ f = f\text{Fct}(x + h, y_{\text{Temp}}) \]

For m = 1 To N

\[ k4(m) = h \times f(m) \]

\[ y(m) = y(m) + \frac{(k1(m) + 2 \times k2(m) + 2 \times k3(m) + k4(m))}{6} \]

Next 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 \([t, y] = \text{solverName(derivativeF, tSpan, y0, options)}, where\)
  – 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 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
Other Numerical ODE Algorithms

ME 309 – Numerical Analysis of Engineering Systems

April 28, 2014

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-Bashforth-Moulton approach
  – Application information for solvers from MATLAB help on next slide

MATLAB Solver Help

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

MATLAB ode45 Example

>> type odeF.m
function f = odeF(t, y)
    f = zeros(3,1);
    f(1) = -y(2)*y(2)/y(3);
    f(2) = -2*y(2)*y(3)/y(1)^3;
    f(3) = -3*y(1)*y(2);
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  0.7408  0.6703
  0.4000  0.6703  0.5488  0.4493
  0.6000  0.5488  0.4493  0.3012
  0.8000  0.4493  0.3012  0.1653
  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
    \[ y_{n+1} - y_n = \frac{h}{2} \left[ f_n + f_\prime \left( y_n \right) + \frac{h}{2} \frac{d^2 f}{dy^2} \left( y_n \right) \right] + O(h^3) \]
- Have to compute (partial derivatives)

\[ \frac{h f_n + \frac{1}{2} \frac{d^2 f}{dy^2} \left( y_n \right) h^2}{1 - \frac{h}{2} \frac{d^2 f}{dy^2} h} \]

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 \( y_{n+1} \)
      \[ y_{n+1}^{(0)} = y_n + hf_n \]
      \[ y_{n+1}^{(m+1)} = y_n + \frac{h}{2} \left[ f_n + f \left( x_{n+1}, y_{n+1}^{(m)} \right) \right] \]

Trapezoid Method III

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

\[ y_{n+1}^{(m+1)} = y_{n+1}^{(m)} - \frac{hf_n}{2} - \frac{hf(x_{n+1}, y_{n+1})}{2} \]

Trapezoid Method Derivation

- Subtract series expansion for \( y_n \) from series for \( y_{n+1} \) about \( y_n \)
  \[ y_{n+1} = y_n + f_n h + \frac{h^2 y_n''}{2} + O(h^3) \]
  \[ y_n = y_{n+1} - f_{n+1} h + \frac{h^2 y_{n+1}''}{2} + O(h^3) \]

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

Trapezoid Method Derivation II

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

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

Trapezoid Method Example

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

\[ y_{n+1} = y_n + \frac{2hf_n + \frac{1}{2} \frac{d^2 f}{dy^2} h^2}{2 - \frac{h}{2} \frac{d^2 f}{dy^2} h} \]

\[ y_{n+1} = y_n + \frac{2h(2 + ha) - 2hay_n}{2 + ha} \]

So \( y_{n+1} = G y_n \) with \( G = (2 - ha)/(2 + ha) \)

- Will use this later in stability discussion
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

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

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 \( y^p \)
  \[ y_{n+1}^C = y_n + \frac{h}{24} \left[ 5 f_n f_{n+1} + 19 f_n - 5 f_{n+1} + f_{n+2} \right] \]

Adams-Bashforth-Moulton II

- Use difference between predictor and corrector results to get error estimate
  \[ y_{n+1}^p = y_n + \frac{h}{24} \left( 55 f_n - 59 f_{n+1} + 37 f_{n+2} - 9 f_{n+3} \right) \]
  \[ y_{n+1}^C = y_n + \frac{h}{24} \left[ 5 f_n f_{n+1} + 19 f_n - 5 f_{n+1} + f_{n+2} \right] \]
- Derivation result (next two slides) gives error estimate in terms of \( (y^p - y^C)_{n+1} \)
  \[ E_c = \frac{-19}{720} h^4 y^{(4)}(\xi_c) = \frac{-19}{270} \left( y_{n+1}^c - y_{n+1}^p \right) \]

Derive Error Equation

- From an error analysis of the integrated interpolation polynomials we can find
  \[ y(x_{n+1}) = y_{n+1}^p + \frac{251}{720} h^3 y^{(3)}(\xi) \]
  1. Subtract equations
  2. Subtract and add \( y^{(1)}(\xi_c) \) term
  \[ y(x_{n+1}) = y_{n+1}^C + \frac{19}{720} h^3 y^{(3)}(\xi) \]
  \[ 0 = y_{n+1}^C - y_{n+1}^p + \frac{251}{720} \frac{19}{720} h^3 y^{(3)}(\xi_c) + \frac{251}{720} \frac{19}{720} h^3 \left[ y^{(3)}(\xi_c) - y^{(1)}(\xi_c) \right] \]
- Neglect \( y^{(1)}(\xi_c) \cdot y^{(1)}(\xi_c) \)
  \[ y_{n+1}^C - y_{n+1}^p = \frac{251}{720} \frac{19}{720} h^3 y^{(3)}(\xi_c) \]

Derive Error Equation

- Solve for \( E_c \), the corrector error
  \[ E_c = y(x_{n+1}) - y_{n+1}^C = -\frac{19}{720} h^4 y^{(4)}(\xi_c) \]
  \[ y_{n+1}^C - y_{n+1}^p = \left( \frac{251}{720} + \frac{19}{720} \right) h^3 y^{(3)}(\xi) = \frac{270}{720} h^3 y^{(3)}(\xi) \]
  \[ E_c = -\frac{19}{720} h^4 y^{(4)}(\xi_c) = \frac{-19}{270} \frac{270}{720} \left( y_{n+1}^C - y_{n+1}^p \right) = \frac{19}{270} \left( y_{n+1}^p - y_{n+1}^{C*} \right) \]
- Error estimate gives step size control
- How to change \( h \) in multistep method?
Other Numerical ODE Algorithms

Step Size Control

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

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

- Interpolated points

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

- Error very small: Double grid size
  
  $i-5$  $i-4$  $i-3$  $i-2$  $i-1$  $i$  $i+1$ (old step)

- Retained to use for doubling

Grid Halving and Doubling

- Keep extra values $f_{i-4}$ and $f_{i-5}$ in memory to be ready for grid doubling
  
  $f_{i-3,\text{new}} = f_{i-6}$; $f_{i-2,\text{new}} = f_{i-3}$; $f_{i-1,\text{new}} = f_{i-2}$; $f_{i,\text{new}} = f_{i+1}$

- Grid halving requires interpolation for missing values in old grid
  
  $f_{i-2,\text{new}} = f_{i-1}$; $f_{i,\text{new}} = f_i$

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

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

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

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