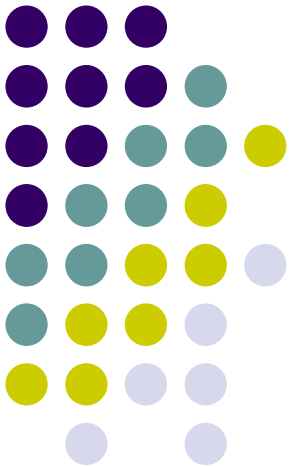
ME751
Advanced Computational Multibody Dynamics

RK-Methods

AB & AM Methods

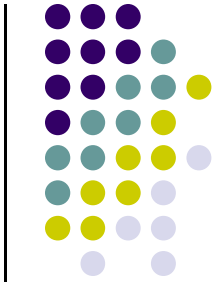
BDF Methods

March 25, 2010



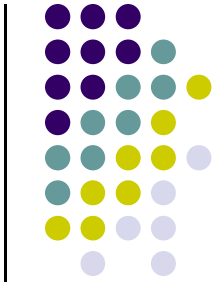
"A Constitution should be short and obscure."
Napoleon Bonaparte

Before we get started…



- Last Time:
  - Numerical solution of IVP
    - Basic Concepts: truncation error, accuracy, zero-stability, convergence, local error, stability
- Today:
  - Finish discussion of Basic Concepts
    - Concentrate on Implicit Integration: why it's hard, and what it buys you
  - Basic Methods
    - Skip altogether Runge-Kutta, Adams-Bashforth, and Adams-Moulton Methods
    - Cover BDF methods
- New HW uploaded on the class website
  - It's ugly
- Trip to John Deere & NADS: need head count by April 8 – email me
  - Transportation and hotel will be covered
  - Leave on May 3 at 6 pm or so, return on May 4 at 10 pm
  - Might also visit software company in Iowa City, they have a simulation tool just like ADAMS

Implicit Methods



- Implicit methods were derived to answer the limitation on the step size noticed for Forward Euler, which is an explicit method

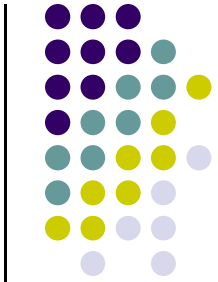
- Simplest implicit method: Backward Euler
  - Given the IVP

$$\begin{cases} \dot{y} &= f(t,y) \\ y(0) &= c \end{cases}$$

  - Backward Euler finds at each time step t_n the solution by solving the following equation for y_n:

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

Explicit vs. Implicit Methods



- A method is called explicit if the approximation of the solution at the next time step is computed straight out of values computed at previous time steps
  - In other words, in the right side of the formula that gives y_n, you only have dependency on y_{n-1}, y_{n-2}, etc. – it's like a recursive formula
  - Example: Forward Euler

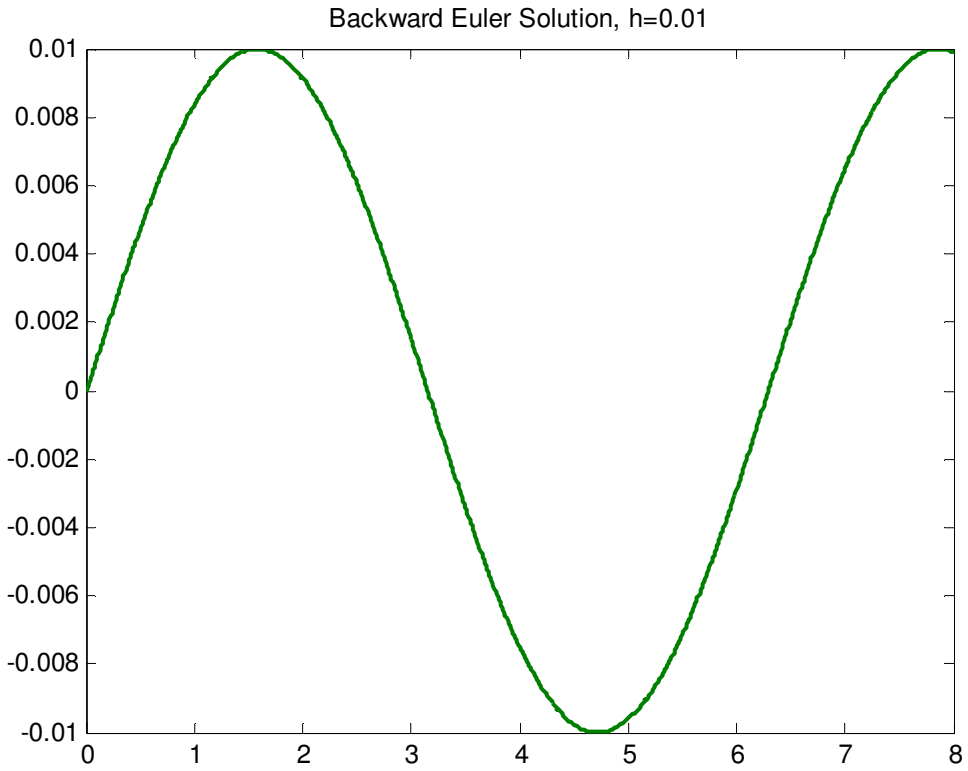
$$y_n = y_{n-1} + hf(t_{n-1}, y_{n-1})$$

- A method is called implicit if the solution at the new time step is found by solving an equation:
  - In other words, in the right side of the formula that gives y_n, you have dependency on y_n, y_{n-1}, y_{n-2}, etc.
  - Example: Backward Euler

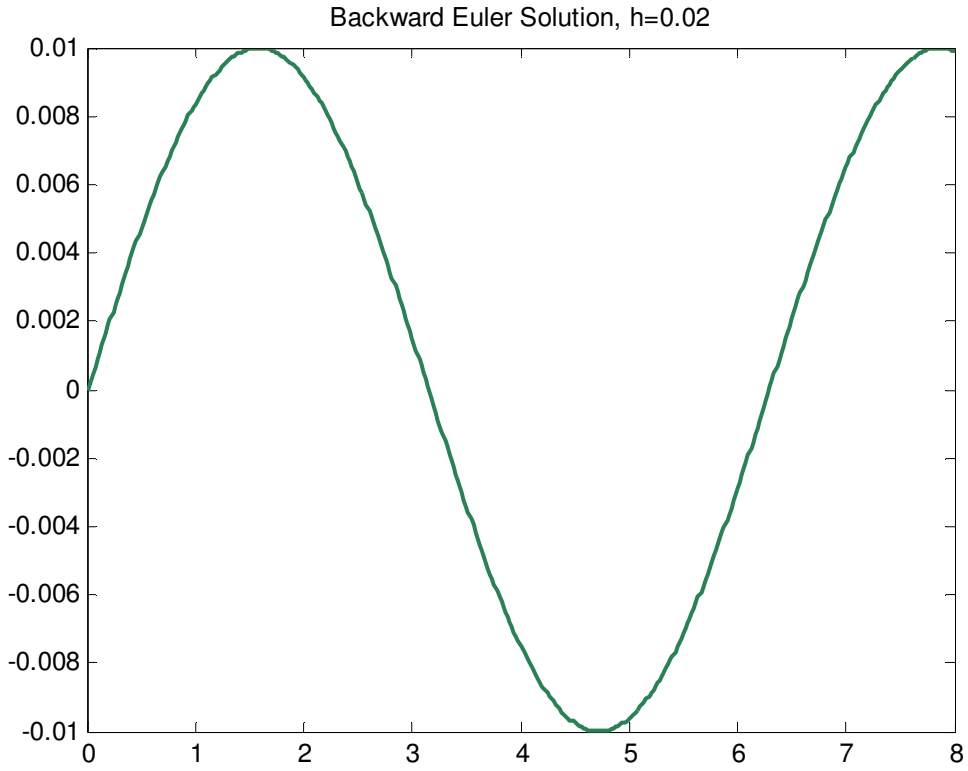
$$y_n = y_{n-1} + hf(t_n, y_n)$$

Example, Approached with Backward Euler: h=0.01

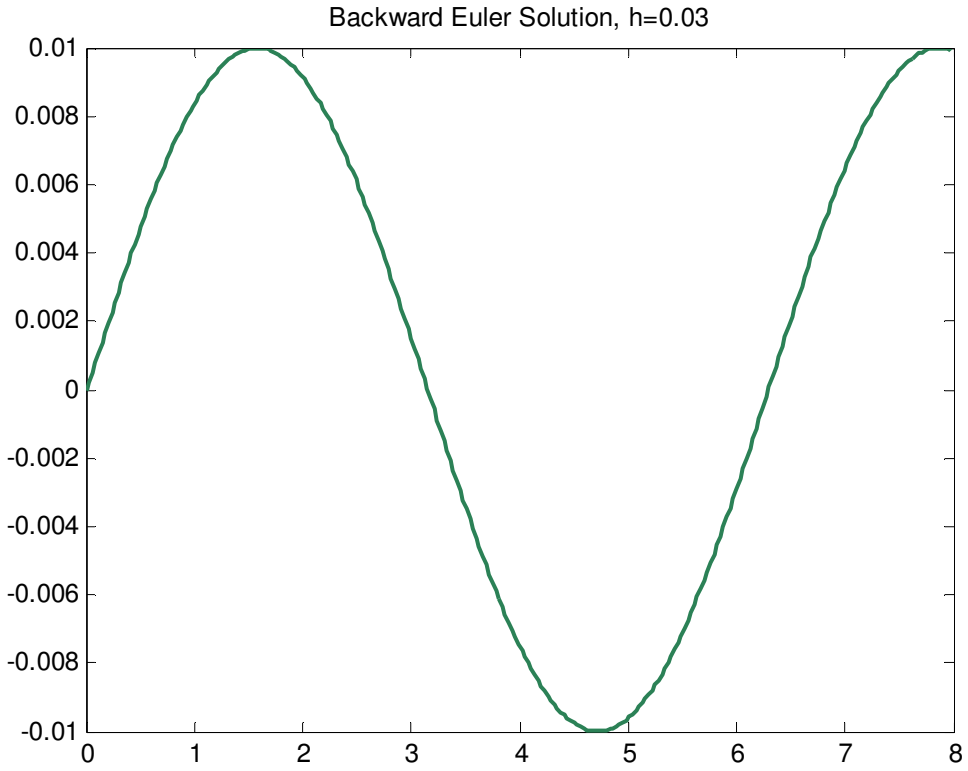
$$\begin{cases} \dot{y} = -100y + \sin(t) \\ y(0) = 0 \end{cases} \qquad t \in [0, 8]$$



Example, Approached with Backward Euler: h=0.02

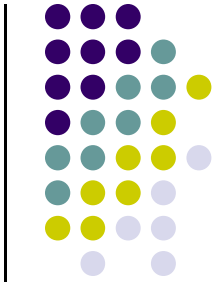


Example, Approached with Backward Euler: h=0.03



- Note that things are good at large values of the integration step size

Exercise, Backward Euler

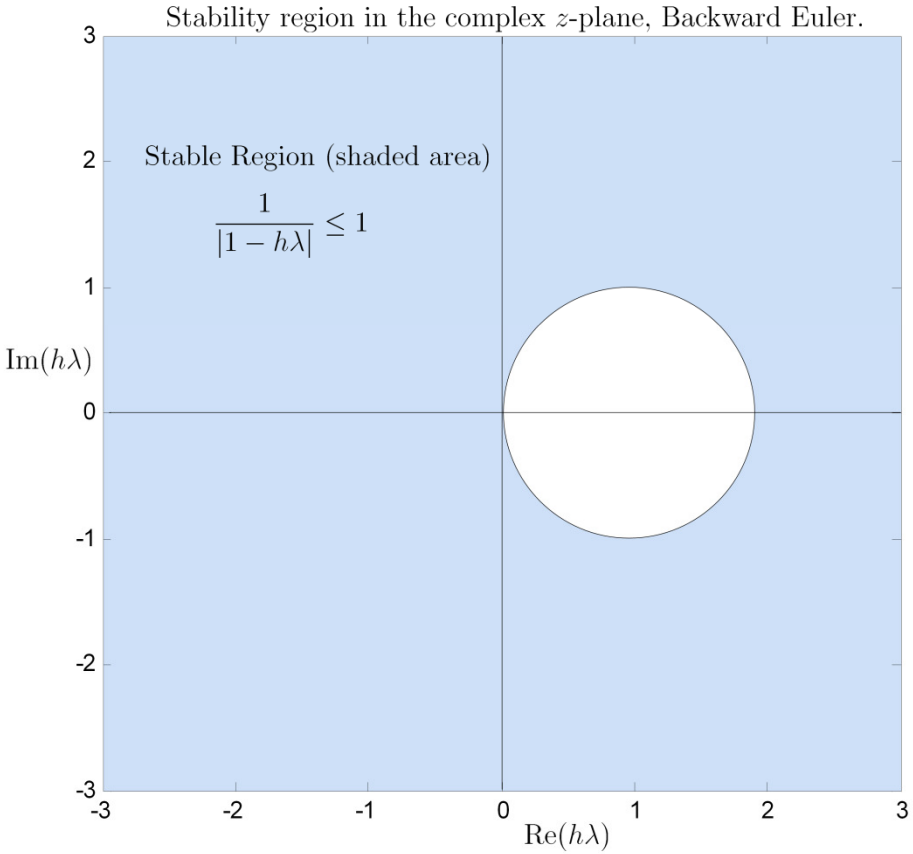


- Prove that
  - Backward Euler is accurate of order 1
  - It satisfies the 0-order stability condition
  - It's convergent with convergence order 1

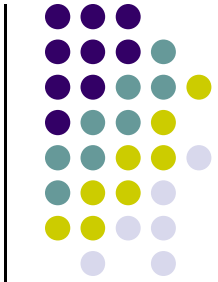
- Generate
  - The stability region of the method and compare to Forward Euler
  - A convergence plot for the IVP

$$\begin{cases} \dot{y} = -100y + \sin(t) \\ y(0) = 0 \end{cases} \qquad t \in [0, 8]$$

Stability Region, Backward Euler



Generating Convergence Plot



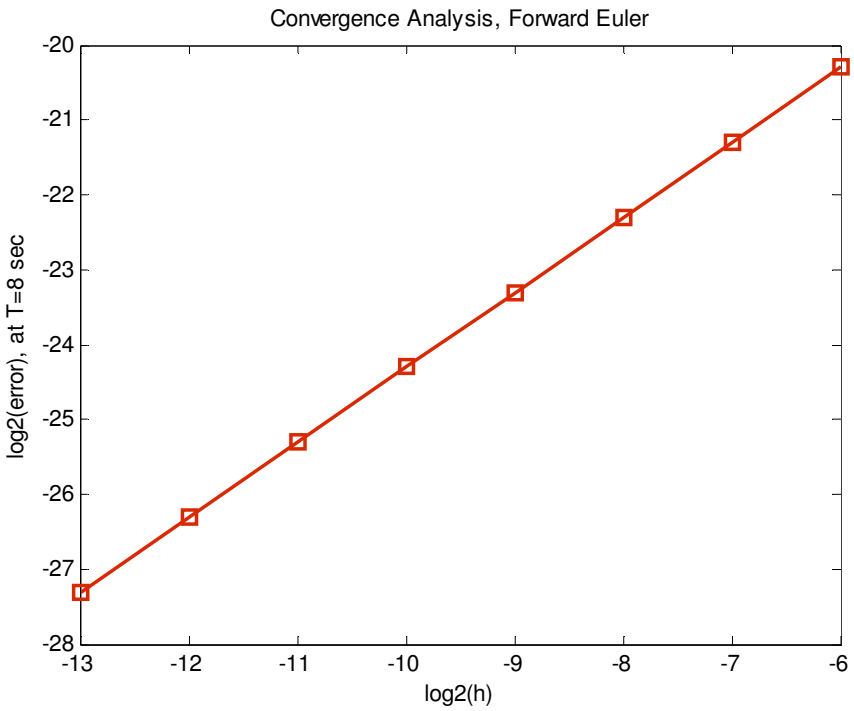
- Procedure to generate Convergence Plot:
  - First, get the exact solution, or some highly accurate numerical solution that can serve as the reference solution
  - Run a sequence of 6 to 8 simulations with decreasing values of step size h
    - Each simulation halves the step-size of the previous simulation
  - For each simulation of the sequence, compare the value of the approximate solution at Tend to the value of the reference solution at Tend

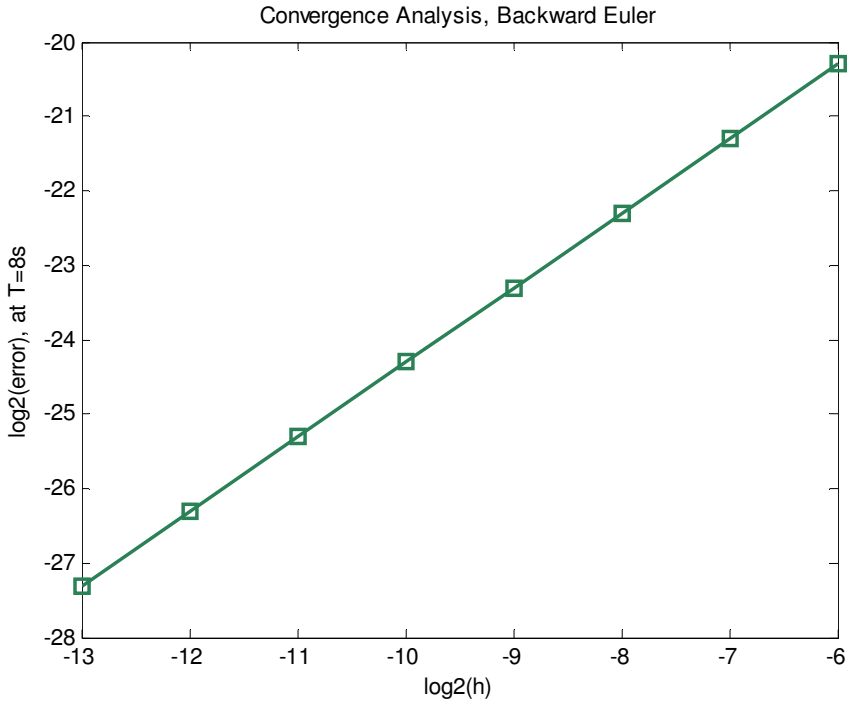
$$error = |y_{end} - y(T_{end})|$$

    - You don't necessarily have to use Tend, some other representative time is ok
    - Generate an array of pairs (h, error), and plot log2(h) vs. log2(error)
      - You should see a line of constant slope. The slope represents the convergence order

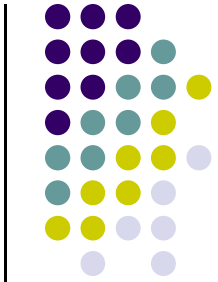
Convergence Plots

$$\begin{cases} \dot{y} = -100y + \sin(t) \\ y(0) = 0 \end{cases} \qquad t \in [0, 8]$$





Code to Generate Convergence Plot



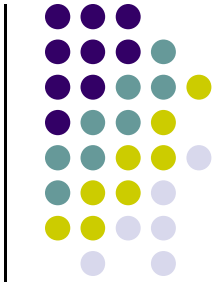
```
nPoints = 8;        % number of points used to generate the convergence plot
hLargest = 2^(-6);  % largest step-size h considered in the convergence analysis
tEnd = 8;           % Tend

hSize = zeros(8,1);
hSize(1) = hLargest;
for i=1:nPoints-1
    hSize(i+1)=hSize(i)/2;
end

% First column of "results" : the step size used for integration
% Second column of "results": the error in the Forward Euler at Tend
% Third column of "results" : the error in the Backward Euler at Tend
results = zeros(nPoints, 3);

% Run a batch of analyses, the step size is gradually smaller
for i=1:nPoints
    yE = zeros(size(0:hSize(i):tEnd))';
    yFE = zeros(size(yE));
    yBE = zeros(size(yE));
    [yE, yFE, yBE] = fEulerVSbEuler(hSize(i), tEnd);
    results(i,1) = hSize(i);
    results(i,2) = abs(yE(end)-yFE(end));
    results(i,3) = abs(yE(end)-yBE(end));
end
```

Implicit Methods, The Ugly Part

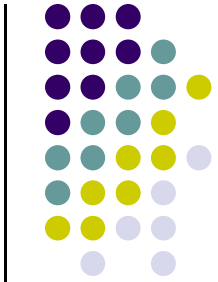


- Why not always use implicit integration methods?
- Implicit method come with some baggage: you need to solve an equation (or system of equations) at *each* integration time step t_n
- Specifically, look at Backward Euler. At each t_n, you need to solve for y_n. This is a nonlinear equation, since f(t,y) in general is a nonlinear function

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

- Solving nonlinear systems is something that I'd avoid if possible…

Implicit Integration, Solving the Nonlinear System

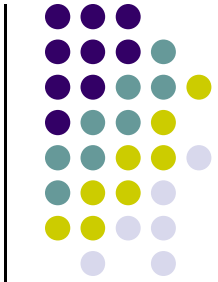


- Note that if you are dealing with a system of ODEs, that is, if y is a vector quantity, you have to solve not a nonlinear equation, but a nonlinear system of equations:

$$\mathbf{g}(\mathbf{y}_n) \equiv \mathbf{y}_n - \mathbf{y}_{n-1} - h\mathbf{f}(t_n, \mathbf{y}_n) = \mathbf{0}$$

- We'll assume in what follows (as almost always the case) that the system above is a nonlinear one
  - Issues that we discuss in this context:
    - The "functional iteration" approach to finding y_n
    - Newton Iteration
    - Approximating the Jacobian associated with the nonlinear system

Nonlinear System Solution: The Functional Iteration



- The basic idea is to solve the system through a functional iteration
  - The superscript (ν+1) indicates the iteration count
  - An initial guess $\mathbf{y}_n^{(0)}$ is needed to "seed" the iterative process

$$\mathbf{y}_n^{(\nu+1)} = \mathbf{y}_{n-1} + h\mathbf{f}(t_n, \mathbf{y}_n^{(\nu)})$$

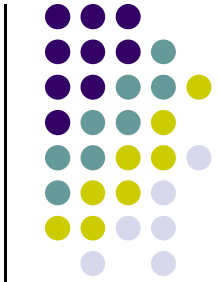
- If this defines a contractive map in a Banach space, the functional iteration leads to a fixed point, which is the solution of interest

- However, for this to be a contractive mapping in some norm, the following needs to hold in a neighborhood of the solution y_n:

$$h \, \| \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \| < 1$$

- For stiff systems, the matrix norm above is very large. This requires small h. And this defeats the purpose of using an implicit formula…

Exercise

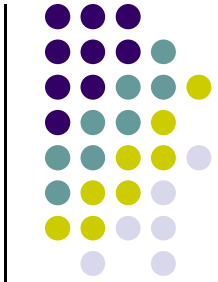


- Analyze the restrictions on the step-size imposed by the requirement that the functional iteration convergence for the following IVP:

$$\begin{cases} \dot{y} = \lambda(ty^2 - 1/t) - 1/t^2 \\ y(1) = 1 \end{cases} \qquad t \in [1, 10]$$

- Here λ<0 is some parameter that determines the stiffness of the IVP
- Note that for λ=-1, the solution is y(t)=1/t

Nonlinear System Solution: The Newton Iteration



- This is simply applying Newton's method to solve the system

$$\mathbf{g}(\mathbf{y}_n) = \mathbf{0}$$

- Boils down to carrying out the iterative process:

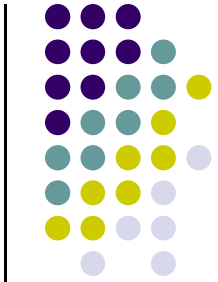
This is where most of the computational effort is spent

$$\begin{cases} \left[\dfrac{\partial \mathbf{g}}{\partial \mathbf{y}}\right] \cdot \Delta \mathbf{y}_n^{(\nu)} = -\mathbf{g}(\mathbf{y}_n^{(\nu)}) \\ \mathbf{y}_n^{(\nu+1)} = \mathbf{y}_n^{(\nu)} + \Delta \mathbf{y}_n^{(\nu)} \end{cases}$$

  - The superscript (ν+1) indicates the iteration count
  - An initial guess $\mathbf{y}_n^{(0)}$ is needed to "seed" the iterative process (take it y_{n-1})
  - Iterative process stopped when correction is smaller than prescribed value
    - NTOL depends on the local error bound that the user aims to achieve
    - Stop when

$$\|\Delta \mathbf{y}_n^{(\nu)}\| \leq \mathrm{NTOL}$$

Nonlinear System Solution: The Newton Iteration

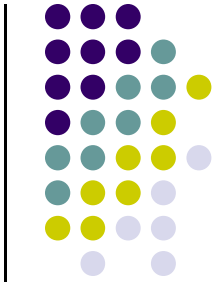


- Iteration matrix:

$$\left[\frac{\partial \mathbf{g}}{\partial \mathbf{y}}\right] = \mathbf{I} - h\,\frac{\partial \mathbf{f}}{\partial \mathbf{y}} \in \mathbb{R}^{m\times m}$$

- Typically, the approach does not place harsh limits on the value of the step size
- Note that the iteration matrix is guaranteed to be nonsingular for small enough values of the step-size h
- The iteration matrix is not updated at each iteration. Updated only when convergence in Newton iteration gets poor
- Note that each update also requires LU factorization of iteration matrix
  - Adding insult to injury…

Nonlinear System Solution: The Newton Iteration



- Iteration matrix, entry (i,j):

$$\left[\frac{\partial \mathbf{g}}{\partial \mathbf{y}}\right]_{ij} = I_{ij} - h\,\frac{\partial f[i]}{\partial y[j]} \in \mathbb{R} \qquad \text{where} \qquad I_{ij} = \begin{cases} 1 & \text{if} \quad i = j \\ 0 & \text{if} \quad i \neq j \end{cases}$$

- The expensive part is computing the partial derivative $\dfrac{\partial f[i]}{\partial y[j]}$
- Ideally, you can compute this exactly
- Otherwise, compute using finite differences:

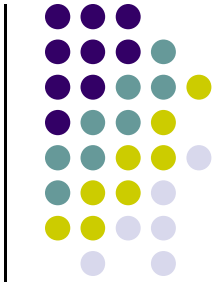
$$\frac{\partial f_i}{\partial y_j} = \lim_{\delta \to 0} \frac{f_i(y_1, \ldots, y_j + \delta, \ldots, y_m) - f_i(y_1, \ldots, y_j, \ldots, y_m)}{\delta} \quad \Rightarrow \quad \frac{\partial f_i}{\partial y_j} \approx \frac{f[i](y_1, \ldots, y[j] + \Delta, \ldots, y_m) - f_i(y_1, \ldots, y_j, \ldots, y_m)}{\Delta}$$

Be aware of notational inconsistency; employed to keep things simple

- Very amenable to parallel computing

Exercise

[AO, Handout]

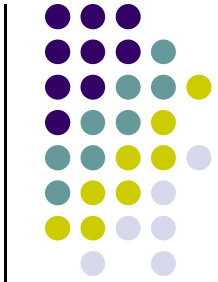


- For IVP below, find iteration matrix when solved with B. Euler
  - Find it analytically
  - Find it using finite differences
  - In both cases use $y[1] = 0 \quad \& \quad y[2] = 2$ for evaluating the matrix

$$\text{IVP:} \quad \begin{cases} \dot{y}[1] = \alpha - y[1] - \frac{4y[1]y[2]}{1+y^2[1]} \\ \dot{y}[2] = \beta y[1]\left(1 - \frac{y[2]}{1+y^2[1]}\right) \qquad t \in [0, 20] \\ y[1](0) = 0 \quad y[2](0) = 2 \end{cases}$$

[Stability, First Flavor]

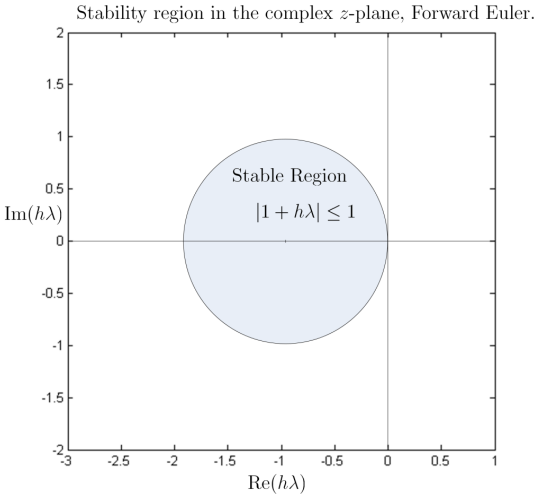
A-Stable Integration Methods

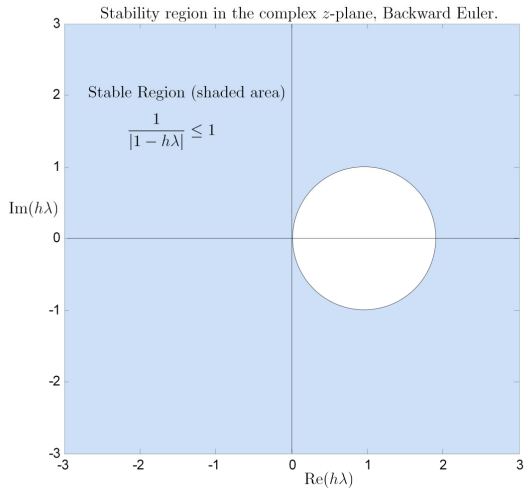


- Definition, A-Stability
  - First, recall the region of absolute stability: defined in conjunction with the test IVP, represents the region where hλ should land so that

$$|y_n| \leq |y_{n-1}|$$

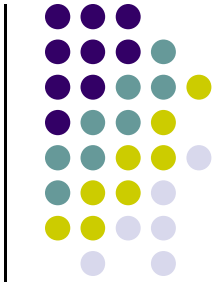
  - By definition, a numerical integration scheme is said to be A-stable if its region of absolute stability covers the entire left half-plane
    - Forward Euler is not A-stable
    - Backward Euler is A-stable





[Stability, Second Flavor]

L-Stable Integration Methods (Methods with Stiff Decay)



- The concept of A-stability is not enough. It only requires that

$$|y_n| \leq |y_{n-1}|$$

- What happens if the problem is super stiff? That is, in the test IVP, λ << 0 (very negative, on the real axis)…

- Consider a new IVP, very similar to the test IVP we worked with:

$$\dot{y} = \lambda(y - g(t))$$

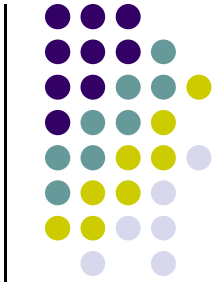
- The assumption is that g(t) is some bounded smooth function

- Note that for the solution we have (after some very short transients)

$$y(t) = g(t)$$

[Stability, Second Flavor]

L-Stable Integration Methods (Methods with Stiff Decay)



- The natural question to ask is this: will my solution y_n get quickly to $g(t_n)$ irrespective of the value of y_{n-1}?

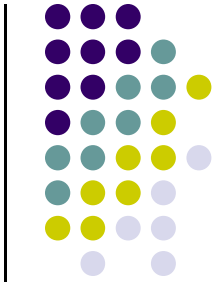
- So I want

$$|y_n - g(t_n)| \to 0 \qquad \text{as} \qquad \lambda \to -\infty$$

- If a numerical integration scheme satisfies this requirement it is said to have “stiff decay”

- What’s the nice thing about methods with stiff decay?
  - They have the ability to skip fine-level (i.e., rapidly varying) solution details and still maintain a decent description of the solution on a coarse level in the very stiff case

Exercise



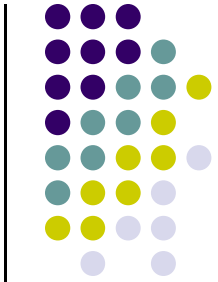
- Prove that Forward Euler doesn't have stiff decay
- Prove that Backward Euler has stiff decay
- Does the trapezoidal formula (provided below) have stiff decay?

$$y_n = y_{n-1} + \frac{h}{2}(f(t_{n-1}, y_{n-1}) + f(t_n, y_n))$$

- Plot the numerical solution of the following IVP, first obtained with Backward Euler and then with the trapezoidal formula. Comment on the relevance of the stiff decay attribute:

$$\begin{cases} \dot{y} = -100(y - \sin(t)) \\ y(0) = 1 \end{cases} \qquad t \in [0, 8]$$

Further Exercises



- Out of Ascher & Petzold book:
  - Problem 3.1
  - Problem 3.2
  - Problem 3.3
  - Problem 3.9

Numerical Integration Methods Taxonomy