

Lecture 9: Stability and frequency properties (mostly recap), error control and software

- Stability (mostly recap), frequency properties (14.6)
- Error control: Automatic adjustment of step-size (14.7)
 - Solver vs Integrator
- Interpolation and events
- Briefly:
 - Multistep methods
 - "Tend to be more efficient than single-step methods for systems with smooth solutions and high accuracy requirements"
 - Often used in advanced modeling software (e.g. Dymola)
 - Differential-algebraic systems
 - Software

Kahoot

- <https://play.kahoot.it/#/k/694ab821-e4e0-421a-a5d6-0d297fd2cf1c>

Implicit Runge-Kutta (IRK) methods

- IVP: $\dot{y} = f(y, t), \quad y(0) = y_0$
- IRK:

$$\begin{aligned}
 k_1 &= f(y_n + h(a_{1,1}k_1 + a_{1,2}k_2 + \dots + a_{1,\sigma}k_\sigma), t_n + c_1h) \\
 k_2 &= f(y_n + h(a_{2,1}k_1 + a_{2,2}k_2 + \dots + a_{2,\sigma}k_\sigma), t_n + c_2h) \\
 k_3 &= f(y_n + h(a_{3,1}k_1 + a_{3,2}k_2 + \dots + a_{3,\sigma}k_\sigma), t_n + c_3h) \\
 &\vdots \\
 k_\sigma &= f(y_n + h(a_{\sigma,1}k_1 + a_{\sigma,2}k_2 + \dots + a_{\sigma,\sigma}k_\sigma), t_n + c_\sigma h) \\
 y_{n+1} &= y_n + h(b_1k_1 + b_2k_2 + \dots + b_\sigma k_\sigma)
 \end{aligned}$$
- Butcher array:

| | | | | | |
|--|------------|----------------|----------------|----------|---------------------|
| $\begin{array}{c c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^\top \end{array}$ | c_1 | a_{11} | a_{12} | \cdots | $a_{1,\sigma}$ |
| | c_2 | a_{21} | a_{22} | \cdots | $a_{2,\sigma}$ |
| | \vdots | \vdots | \vdots | \ddots | |
| | c_σ | $a_{\sigma,1}$ | $a_{\sigma,2}$ | \cdots | $a_{\sigma,\sigma}$ |
| | | b_1 | b_2 | \dots | b_σ |

Recap: Test system, stability function

- One step method (typically: Runge-Kutta):

$$y_{n+1} = y_n + h\phi(y_n, t_n)$$

- Apply it to scalar test system:

$$\dot{y} = \lambda y$$

- We get:

$$y_{n+1} = R(h\lambda)y_n$$

where $R(h\lambda)$ is stability function

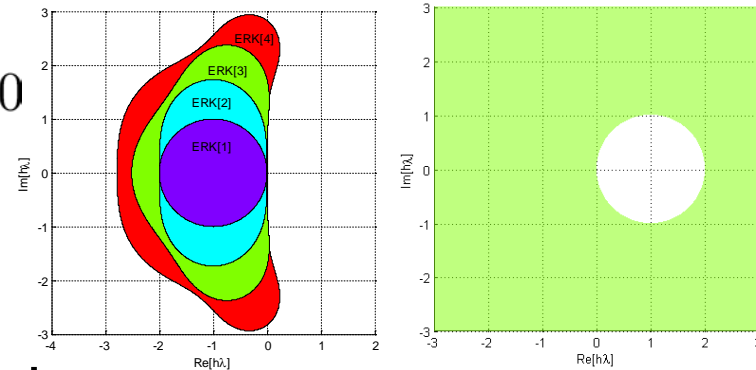
- The method is stable (for test system!) if

$$|R(h\lambda)| \leq 1$$

Linear stability: A- and L-stability

- A-stability: $|R(h\lambda)| \leq 1$ for all $\text{Re } \lambda \leq 0$

- Relevant (mostly) for **stiff** systems
- No explicit methods are A-stable
- Many implicit methods are A-stable

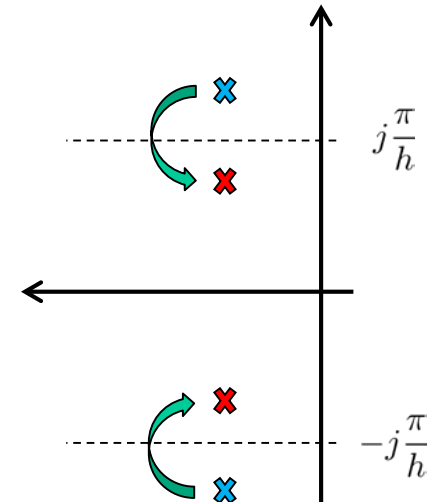


- Aliasing: Frequencies larger than the «Nyquist frequency» $\frac{\pi}{h}$ are mapped to within the Nyquist frequency

- A method is L-stable if it is A-stable, and

$$|R(hj\omega)| \rightarrow 0 \text{ when } \omega \rightarrow \infty$$

- Relevant for (stiff) systems with **oscillatory modes**
- Dampens out fast frequencies
- We often want L-stability in implicit methods, but not always:
 - We typically want to suppress dynamics that are faster than step length («stiff decay»)
 - However, we may want to not dampen oscillatory modes
 - We may want to **not** dissipate energy (numerically) in simulations



Padé approximations

- The local solution to test system:

$$y_L(t_n; t_{n+1}) = e^{h\lambda} y_n$$

- Stability function:

$$y_{n+1} = R(h\lambda) y_n$$

- A method is «good» if

$$R(s) \approx e^s$$

- Explicit Runge-Kutta methods with $\sigma = p$. 4: Taylor expansion!

$$R(s) = 1 + s + \dots + \frac{s^p}{p!}$$

- Implicit Runge-Kutta methods:

$$R(s) = \frac{1 + \beta_1 s + \dots + \beta_k s^k}{1 + \gamma_1 s + \dots + \gamma_m s^m}$$

- Best approximation (for given k and m): Padé-approximation

Padé approximations to e^s

| $\begin{smallmatrix} k \\ m \end{smallmatrix}$ | 0 | 1 | 2 | 3 |
|--|---|--|---|---|
| 0 | $\frac{1}{1}$ | $\frac{1+s}{1}$ | $\frac{1+s+\frac{1}{2}s^2}{1}$ | $\frac{1+s+\frac{1}{2}s^2+\frac{1}{6}s^3}{1}$ |
| 1 | $\frac{1}{1-s}$ | $\frac{1+\frac{1}{2}s}{1-\frac{1}{2}s}$ | $\frac{1+\frac{2}{3}s+\frac{1}{6}s^2}{1-\frac{1}{3}s}$ | $\frac{1+\frac{3}{4}s+\frac{1}{4}s^2+\frac{1}{24}s^3}{1-\frac{1}{4}s}$ |
| 2 | $\frac{1}{1-s+\frac{1}{2}s^2}$ | $\frac{1+\frac{1}{3}s}{1-\frac{2}{3}s+\frac{1}{6}s^2}$ | $\frac{1+\frac{1}{2}s+\frac{1}{12}s^2}{1-\frac{1}{2}s+\frac{1}{12}s^2}$ | $\frac{1+\frac{3}{5}s+\frac{3}{20}s^2+\frac{1}{60}s^3}{1-\frac{2}{5}s+\frac{1}{20}s^2}$ |
| 3 | $\frac{1}{1-s+\frac{1}{2}s^2-\frac{1}{6}s^3}$ | $\frac{1+\frac{1}{4}s}{1-\frac{3}{4}s+\frac{1}{4}s^2-\frac{1}{24}s^3}$ | $\frac{1+\frac{2}{5}s+\frac{1}{20}s^2}{1-\frac{3}{5}s+\frac{3}{20}s^2-\frac{1}{60}s^3}$ | $\frac{1+\frac{1}{2}s+\frac{1}{10}s^2+\frac{1}{120}s^3}{1-\frac{1}{2}s+\frac{1}{10}s^2-\frac{1}{120}s^3}$ |



L-stable



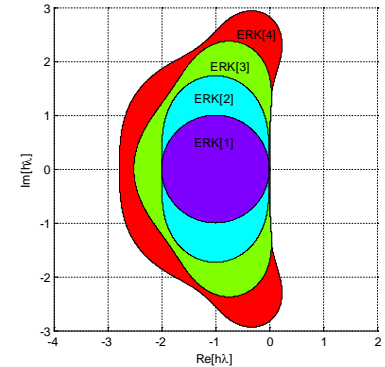
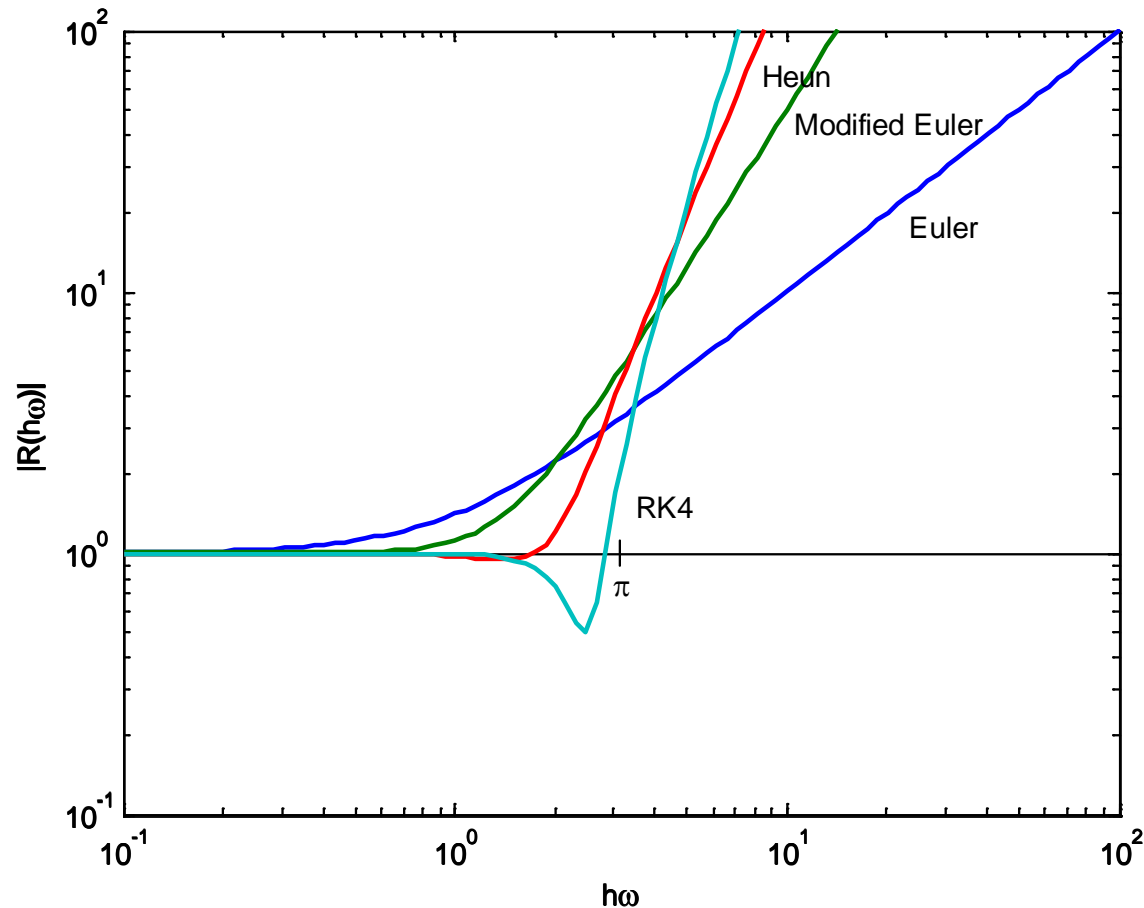
L-stable



A-stable

- $m = 0$: Explicit Runge-Kutta methods with $p = \sigma$
- $m = k$: Gauss, Lobatto IIIA/IIIB (incl. implicit mid-point, trapezoidal)
- $m = k+1$: Radau-methods (incl. implicit Euler)
- $m = k+2$: Lobatto IIIC

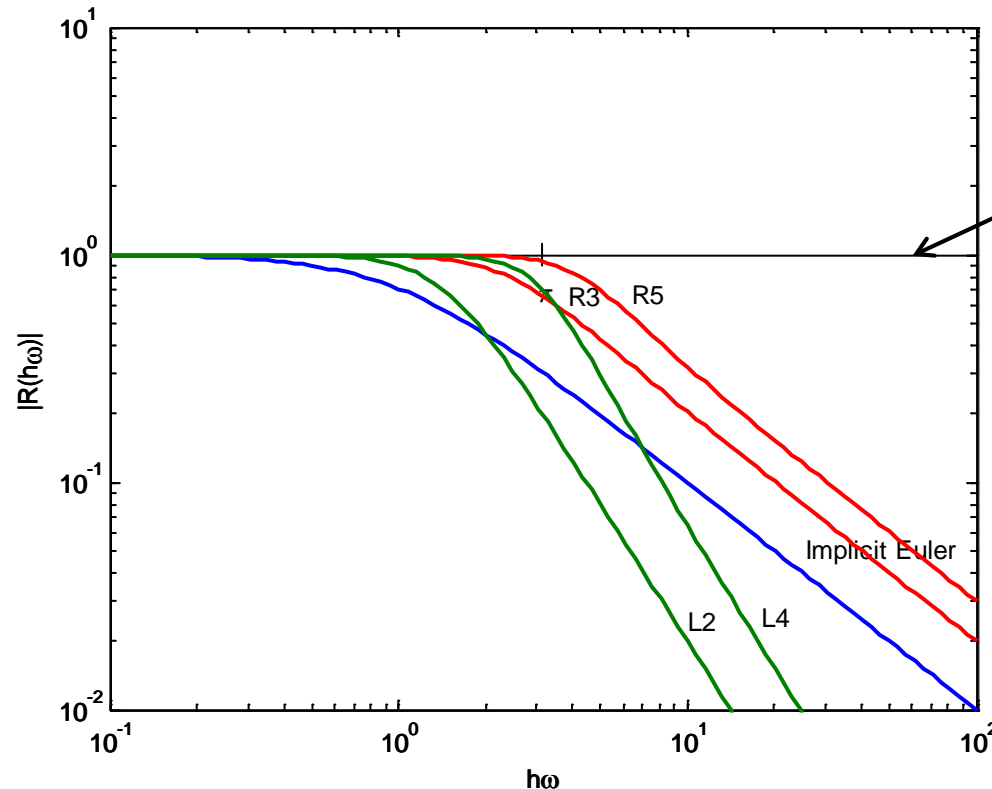
Frequency properties, explicit methods



- Stability functions for explicit Runge-Kutta methods plotted as a function of $s = jh\omega$
- «Nyquist frequency» $h\omega = \pi$ indicated

Frequency properties, implicit methods

| $k \backslash m$ | 0 | 1 | 2 | 3 |
|------------------|---|--|---|---|
| 0 | $\frac{1}{1}$ | $\frac{1+s}{1}$ | $\frac{1+s+\frac{1}{2}s^2}{1}$ | $\frac{1+s+\frac{1}{2}s^2+\frac{1}{6}s^3}{1}$ |
| 1 | $\frac{1}{1-s}$ | $\frac{1+\frac{1}{2}s}{1-\frac{1}{2}s}$ | $\frac{1+\frac{2}{3}s+\frac{1}{6}s^2}{1-\frac{1}{3}s}$ | $\frac{1+\frac{3}{4}s+\frac{1}{4}s^2+\frac{1}{24}s^3}{1-\frac{1}{4}s}$ |
| 2 | $\frac{1}{1-s+\frac{1}{2}s^2}$ | $\frac{1+\frac{1}{3}s}{1-\frac{2}{3}s+\frac{1}{6}s^2}$ | $\frac{1+\frac{1}{2}s+\frac{1}{12}s^2}{1-\frac{1}{2}s+\frac{1}{12}s^2}$ | $\frac{1+\frac{3}{20}s+\frac{3}{20}s^2+\frac{1}{60}s^3}{1-\frac{2}{5}s+\frac{1}{20}s^2}$ |
| 3 | $\frac{1}{1-s+\frac{1}{2}s^2-\frac{1}{6}s^3}$ | $\frac{1+\frac{1}{4}s}{1-\frac{3}{4}s+\frac{1}{4}s^2-\frac{1}{24}s^3}$ | $\frac{1+\frac{2}{5}s+\frac{1}{20}s^2}{1-\frac{3}{5}s+\frac{3}{20}s^2-\frac{1}{60}s^3}$ | $\frac{1+\frac{1}{2}s+\frac{1}{10}s^2+\frac{1}{120}s^3}{1-\frac{1}{2}s+\frac{1}{10}s^2-\frac{1}{120}s^3}$ |



Gauss methods
has $|R(jh\omega)| = 1$,
i.e. no damping

- Stability functions for some implicit Runge-Kutta methods plotted as a function of $s = jh\omega$
- «Nyquist frequency» $h\omega = \pi$ indicated
- Dampens out high frequencies: «Rolloff» of -1 for Radau methods vs -2 for Lobatto IIIC methods

Nonlinear stability

- AN-stability

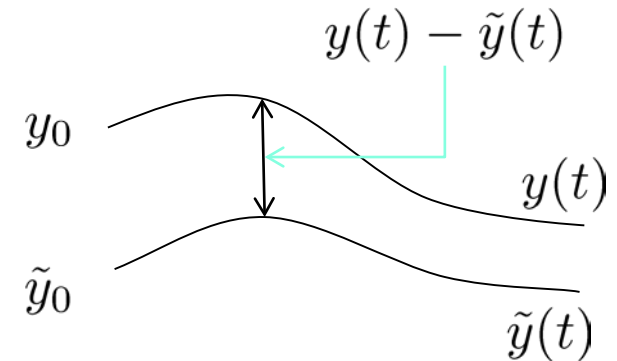
- Stability for time-varying linear system ($\dot{y} = \lambda(t)y$)
- Implies A-stability

- B-stability:

- Given “contracting” system $\dot{y} = f(y, t)$

$$\|y(t) - \tilde{y}(t)\| \rightarrow 0 \text{ exponentially}$$
- A Runge-Kutta method is B-stable if the solutions fulfill

$$\|y_{n+1} - \tilde{y}_{n+1}\| \leq \|y_n - \tilde{y}_n\|$$
for all contracting systems



- B-stability implies AN-stability
- “Difficult” to check (in general), but...

Nonlinear stability, cont'd

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^\top \end{array}$$

Algebraic stability:

- An (implicit) Runge-Kutta method is *algebraically stable* if
 - $b_i \geq 0, \quad i = 1, \dots, \sigma$
 - $\mathbf{M} = \text{diag}(\mathbf{b})\mathbf{A} + \mathbf{A}^\top \text{diag}(\mathbf{b}) - \mathbf{b}\mathbf{b}^\top \geq 0$ (positive semidefinite)
- Easy to check
- The nonlinear stability concepts implies A-stability
 - Algebraic stability implies B-stability
 - B-stability implies AN-stability
 - AN-stability implies A-stability
- Interesting fact:
 - Trapezoid and Implicit Midpoint have same stability function (same linear stability), but only Implicit Midpoint is algebraically stable (and B-stable)

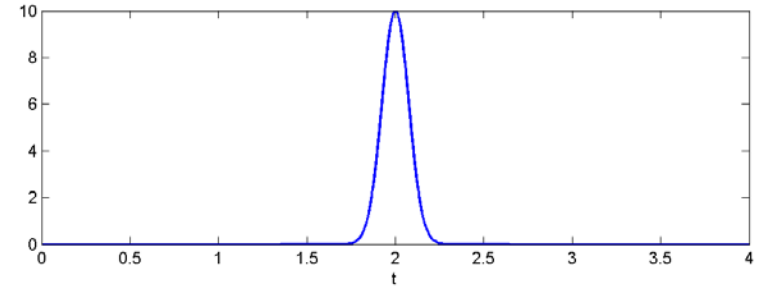
Automatic adjustment of steplength

- We have seen that accuracy depends on step length h
 - (e.g. A-stable methods: Always stable, but not necessarily accurate)
- How to choose step lengths?
 - Systems that are (close to) linear with eigenvalues in limited range:
 - “Easy” to choose h to have stability & desired accuracy everywhere
 - Systems that are (linear or nonlinear) and stiff or highly time-dependent:
 - How to choose h ?
 - h too large: inaccurate (or even unstable) in some periods/regions
 - h too small: inefficient in some periods/regions
- Would it not be nice if we could specify what accuracy we want, and let the ODE solver choose appropriate step-lengths?

Example

- We want to simulate

$$\dot{y} = -0.6y + 10e^{-\frac{(t-2)^2}{2 \cdot 0.075^2}}, \quad y(0) = 0.5$$



- Matlab, using ode23

```
% f(t,y)
f = @(t,y) ( -0.6*y + 10*exp(-(t-2).^2/(2*(0.075^2))) );

% Set desired accuracy
options = odeset('RelTol',10^-3);

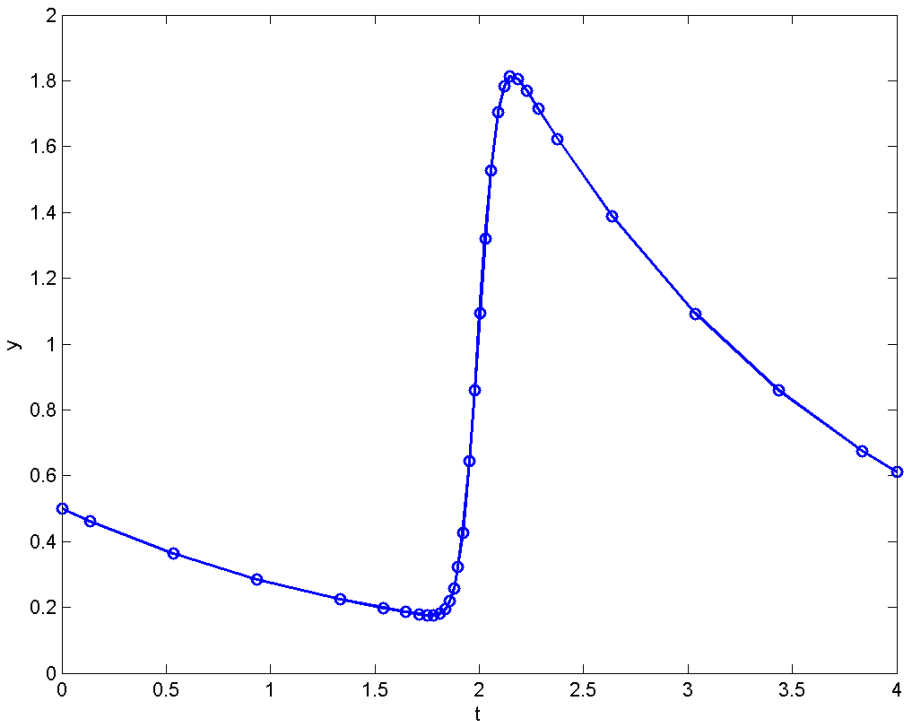
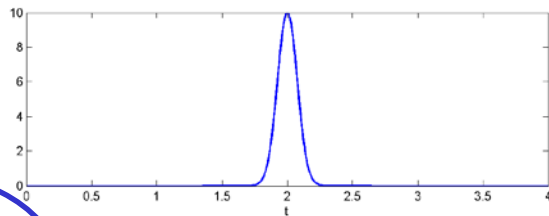
% Simulate
[t,y] = ode23(f, [0 4], 0.5, options);

% Plot solution
plot(t,y,'-o');
```

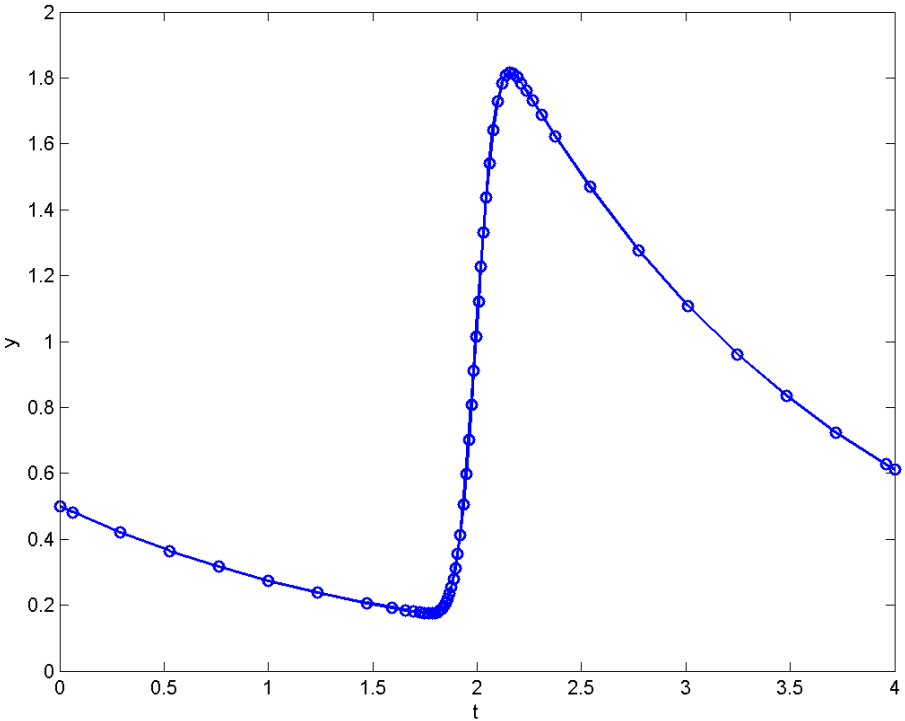
Example, cont'd

RelTol = 10⁻³:

$\dot{y} = -0.6y + 10e^{-\frac{(t-2)^2}{2 \cdot 0.075^2}}, \quad y(0) = 0.5$

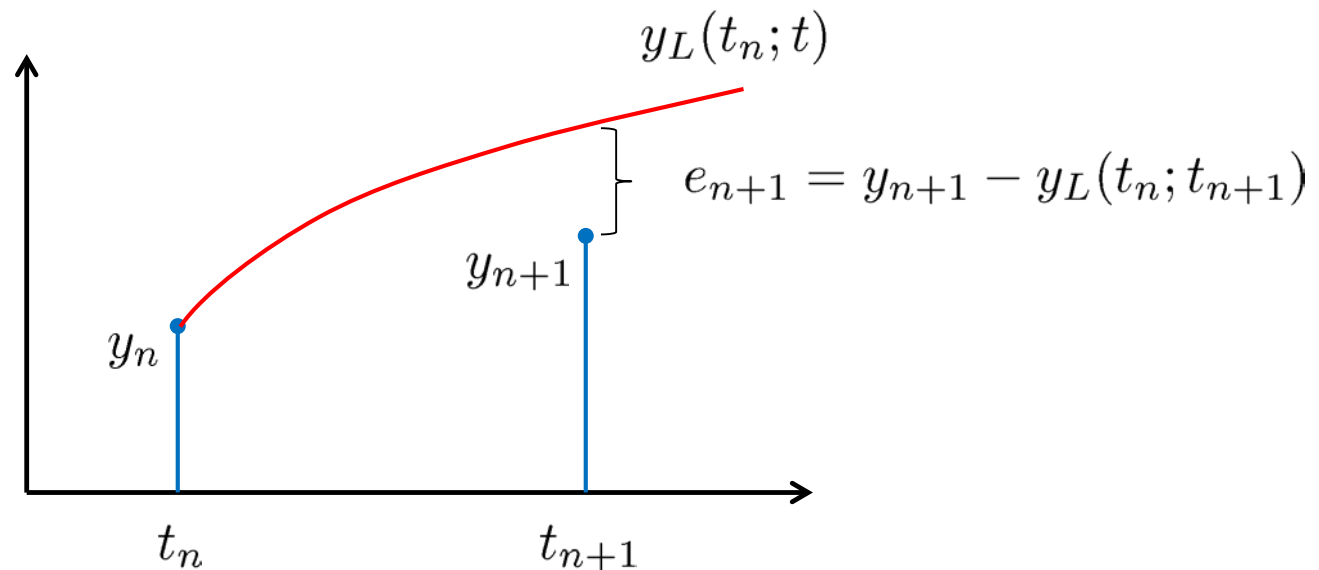


RelTol = 10⁻⁴:



Estimation of local error

- Recall: Local error is error from y_n to y_{n+1}



Estimation of local error

- Starting at y_n ,

Calculate y_{n+1} using (E)RK

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^\top \end{array}$$

with order p

local error $e_{n+1} = O(h^{p+1})$

Calculate \hat{y}_{n+1} using (E)RK

$$\begin{array}{c|c} \hat{\mathbf{c}} & \hat{\mathbf{A}} \\ \hline & \hat{\mathbf{b}}^\top \end{array}$$

with order $p+1$

local error $\hat{e}_{n+1} = O(h^{p+2})$

- Local solution (per def.):

$$y_L(t_n; t_{n+1}) = y_{n+1} + e_{n+1}$$

$$y_L(t_n; t_{n+1}) = \hat{y}_{n+1} + \hat{e}_{n+1}$$

- Combine:

$$y_{n+1} - \hat{y}_{n+1} = e_{n+1} - \hat{e}_{n+1} \approx e_{n+1}$$

- Gives estimate of local error:

$$e_{n+1} \approx \hat{y}_{n+1} - y_{n+1}$$

Efficient calculations

$$e_{n+1} \approx \hat{y}_{n+1} - y_{n+1}$$

$$\begin{array}{c|c} \hat{c} & \hat{A} \\ \hline & \hat{b}^T \end{array} \quad \begin{array}{c|c} c & A \\ \hline & b^T \end{array}$$

Efficient calculations

Choose $C = \hat{C}$ and $A = \hat{A}$

→ same stage calculations

→ "embedded Runge-Kutta"

\hat{y}_{n+1} "embedded solution"

$$\begin{array}{c|c} C & A \\ \hline y & b^T \\ \hat{y} & \hat{b}^T \\ \hline e & E \end{array}$$

$$E = \hat{b} - b$$

RK4(5) Runge-Kutta-Fehlberg (1969)

- $\sigma = 6$, $p = 4$, $\hat{p} = 5$

| | | | | | | |
|-------|-----------|------------|------------|-------------|--------|------|
| 0 | | | | | | |
| 1/4 | 1/4 | | | | | |
| 3/8 | 3/32 | 9/32 | | | | |
| 12/13 | 1932/2197 | -7200/2197 | 7296/2197 | | | |
| 1 | 439/216 | -8 | 3680/513 | -845/4104 | | |
| 1/2 | -8/27 | 2 | -3544/2565 | 1859/4104 | -11/40 | |
| | 25/216 | 0 | 1408/2565 | 2197/4104 | -1/5 | 0 |
| | 16/135 | 0 | 6656/12825 | 28561/56430 | -9/50 | 2/55 |

- Issue: Why use y_{n+1} ($p = 4$) when we have calculated more accurate \hat{y}_{n+1} ($\hat{p} = 5$)?
 - Use \hat{y}_{n+1} instead: “local extrapolation”
 - Some numerical issues/optimizations concerning accuracy comes into play

Methods using local extrapolation

- Dormand-Prince 5(4) – DP5(4) (1980)

| | | | | | | | |
|------|------------|-------------|------------|----------|---------------|----------|------|
| 0 | | | | | | | |
| 1/5 | 1/5 | | | | | | |
| 3/10 | 3/40 | 9/40 | | | | | |
| 4/5 | 44/45 | -56/15 | 32/9 | | | | |
| 8/9 | 19372/6561 | -25360/2187 | 64448/6561 | -212/729 | | | |
| 1 | 9017/3168 | -355/33 | 46732/5247 | 49/176 | -5103/18656 | | |
| 1 | 35/384 | 0 | 500/1113 | 125/192 | -2187/6784 | 11/84 | |
| | 5179/57600 | 0 | 7571/16695 | 393/640 | -92097/339200 | 187/2100 | 1/40 |
| | 35/384 | 0 | 500/1113 | 125/192 | -2187/6784 | 11/84 | 0 |

- Implemented as ode45 in Matlab (and GNU Octave)
 - Freeware Fortran code Dopri5

- Bogacki-Shampine 2(3) – BS2(3) (1989)
 - ode23 in Matlab
 - Faster than ode45 if low accuracy demands

| | | | | |
|-----|------|-----|-----|-----|
| 0 | | | | |
| 1/2 | 1/2 | | | |
| 3/4 | 0 | 3/4 | | |
| 1 | 2/9 | 1/3 | 4/9 | |
| | 2/9 | 1/3 | 4/9 | 0 |
| | 7/24 | 1/4 | 1/3 | 1/8 |

Use local error estimate to adjust step-size I

- Local error estimate $e_{n+1} = (e_{1,n+1}, e_{2,n+1}, \dots, e_{d,n+1})^T$
- Measure of error: $\varepsilon_{n+1} = \|e_{n+1}\|_p$ (for instance $p = \infty$)
- We know that $\varepsilon_{n+1} \approx Ch^{p+1}$
- If $\varepsilon_{n+1} > e_{tol}$:
 - Choose h_{new} such that:

$$e_{new} \approx C h_{new}^{p+1} = e_{tol}$$

$$\rightarrow \frac{e_{tol}}{\varepsilon_{n+1}} \approx \frac{Ch_{new}^{p+1}}{Ch^{p+1}} = \left(\frac{h_{new}}{h}\right)^{p+1}$$

$$\rightarrow h_{new} = h \left(\frac{e_{tol}}{\varepsilon_{n+1}}\right)^{\frac{1}{1+p}}$$

Algorithm:

$\varepsilon_{n+1} \approx e_{tol}$: Do nothing

$\varepsilon_{n+1} > e_{tol}$: Set $h = h_{new}$
(decrease h)

$\varepsilon_{n+1} < e_{tol}$: Set $h = h_{new}$
(increase h)

Use local error estimate to adjust step-size II

We can rewrite:

$$h_{new} = h \left(\frac{e_{tol}}{\varepsilon_{n+1}} \right)^{\frac{1}{1+p}}$$

$$\underbrace{\ln h_{new}}_{u_{n+1}} = \underbrace{\ln h}_{u_n} + \frac{1}{1+p} \left(\underbrace{\ln e_{tol}}_{y_{ref}} - \underbrace{\ln \varepsilon_{n+1}}_{y_n} \right)$$

→ I - controller:

$$u_{n+1} = u_n + \frac{h}{T_i} \underbrace{(y_{ref} - y_n)}_{\tilde{y}}$$

Include also p-part (to improve control)

$$u_{n+1} = u_n + K_p (\tilde{y}_n - \tilde{y}_{n-1}) + K_p \frac{h}{T_i} \tilde{y}_n$$

→ incremental term of PI controller

Example: $K_p = \frac{0.4}{p+1}$

$$T_i = 1.3 h$$

How to choose e_{tol} in practice?

- Say you want to simulate a model of a chemical reaction, using SI units, and have:

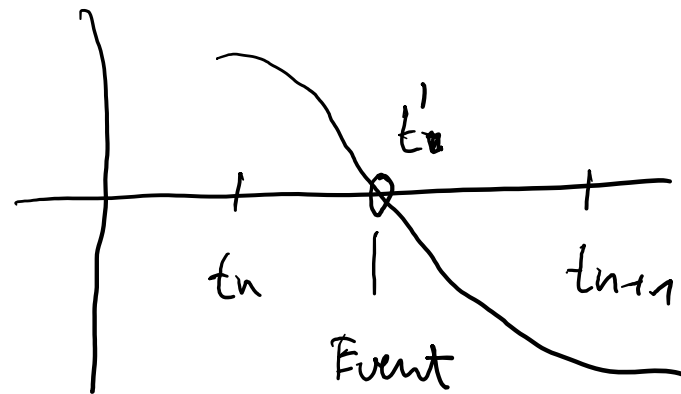
| State | Nominal values | Tolerances |
|---------------|----------------|------------|
| Pressure | 10^5 Pa | 10 Pa |
| Concentration | 0.01 | 10^{-6} |

- To give the solver a single tolerance value, you have to scale your model!
 - (often a good idea also for other reasons)
 - In practice, a perfectly scaled model is difficult to achieve
- Alternatively, use solvers that implement relative tolerance (possibly in addition to absolute tolerance)
 - Matlab: $e_{\text{tol},i} = \max\{r|y_i|, a_i\}$
 - CVode: $e_{\text{tol},i} = r|y_i| + a_i$

r : RelTol (scalar) [10^{-3}]
 a_i : AbsTol (vector) [10^{-6}]

Event-detection and interpolated solutions

"Event" function $g(y, t) = 0$



"dense
output"

If we want to find t'

→ Interpolate $t' = t_n + \alpha h$

$$y_n(1) = y_{n+1}$$

create polynomial: $y(\alpha)$; $y_n(0) = y_n$

→ Find α that solves $g(y_n(\alpha), t + \alpha h) = 0$

Event example: Bouncing ball

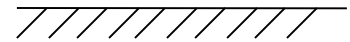
- Newton's law:

$$m\ddot{x} = -mg$$

- State-space:

$$\dot{x} = v$$

$$\dot{v} = -g$$



- Implementation of derivative

```
function dy = f_bb(t,y)
dy = zeros(2,1);      % column vector
g = 9.81;             % gravity
dy(1) = y(2);         % derivative of height
dy(2) = -g;           % derivative of velocity
```

- What if we hit the ground?

```
if (y(1) <= 0),       % Check if we hit the ground
    y(2) = - 0.8*y(2); % 80% elastic
end
```


Bouncing ball: Euler implementation

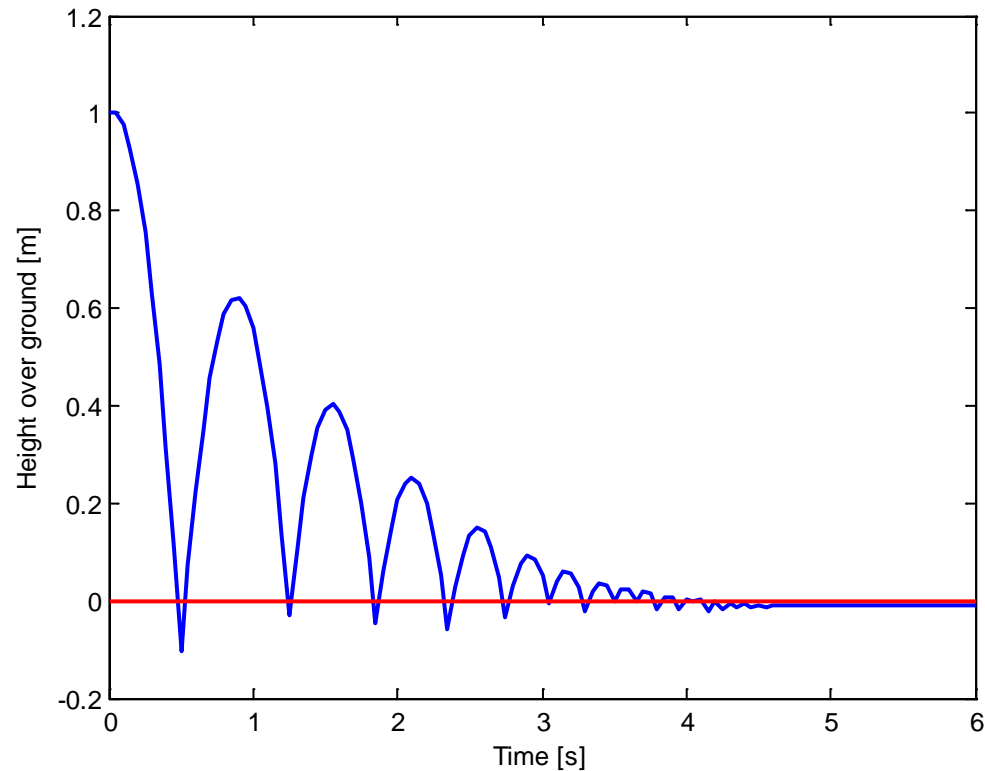
```
t_0 = 0; t_end = 6; h = 0.05; timespan = t_0:h:t_end;
y = zeros(2,length(timespan)+1); % Allocate space

x_0 = 1;           % Initial height above ground [m]
v_0 = 0;           % Initial velocity [m/s]
y(:,1) = [x_0; v_0];

for i = 1:length(timespan),
    t = timespan(i);
    y(:,i+1) = y(:,i) + h*f_bb(t,y(:,i)); % Euler integration

    if (y(1,i+1) <=0), % Check if we hit the ground
        y(2,i+1) = - 0.8*y(2,i); % 80% elastic
    end
end
```

Bouncing ball solved using Euler



ODE-solver with event detection

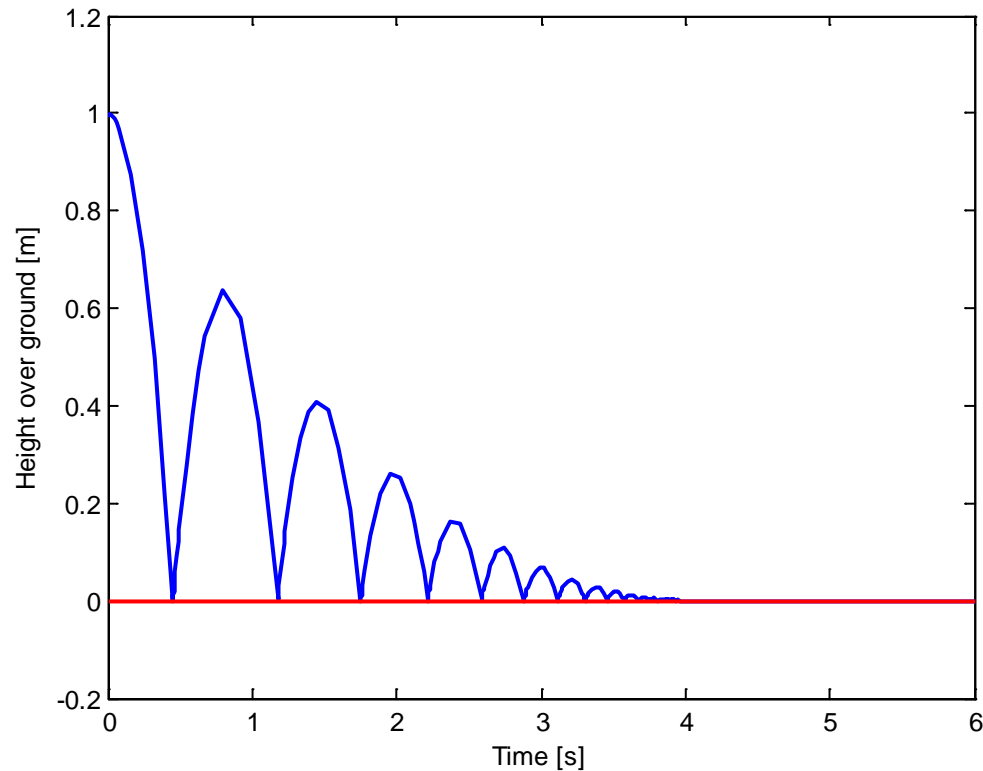
- Event-function

```
function [value,isterminal,direction] = g_bb(t,y)
value = y(1);           % Value of event-function
isterminal = 1;         % Should we stop at event? (1/0)
direction = -1;         % Event at negative to positive (1),
                        % positive to negative (-1), or both (0)
```

- Simulate using ode45 with event detection:

```
options = odeset('events','g_bb');           % Set event-function
t_curr = t_0; x_curr = x_0; v_curr = v_0;
while (t_curr < t_end),
    [ttemp,ytemp] = ode45(@f_bb,[t_curr t_end],[x_curr; v_curr],options);
    t_curr = ttemp(end);
    if t_curr < t_end, % Not simulated to end yet, therefore it is an event
        x_curr = 0;
        v_curr = -.8*ytemp(end,2); % 80% elastic
    end
    t = [t;ttemp]; y = [y;ytemp]; % Add to solution
end
```

Bouncing ball solved using event-function



Multi-step methods

- One-step

$$y_{n+1} = y_n + h\phi(y_n, t_n)$$

- Multi-step

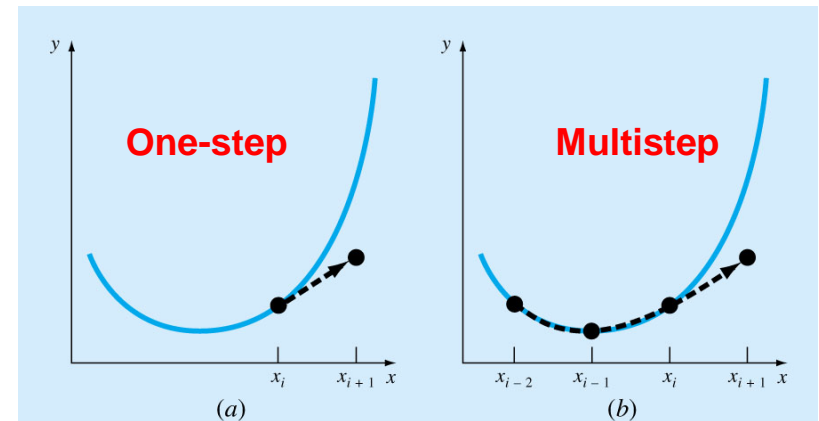
$$y_{n+1} = \alpha_1 y_n + \alpha_2 y_{n-1} + \dots + h(\beta_0 f(y_{n+1}, t_{n+1}) + \beta_1 f(y_n, t_n) + \beta_2 f(y_{n-1}, t_{n-1}) + \dots)$$

- Derived by fitting polynomials to previous steps

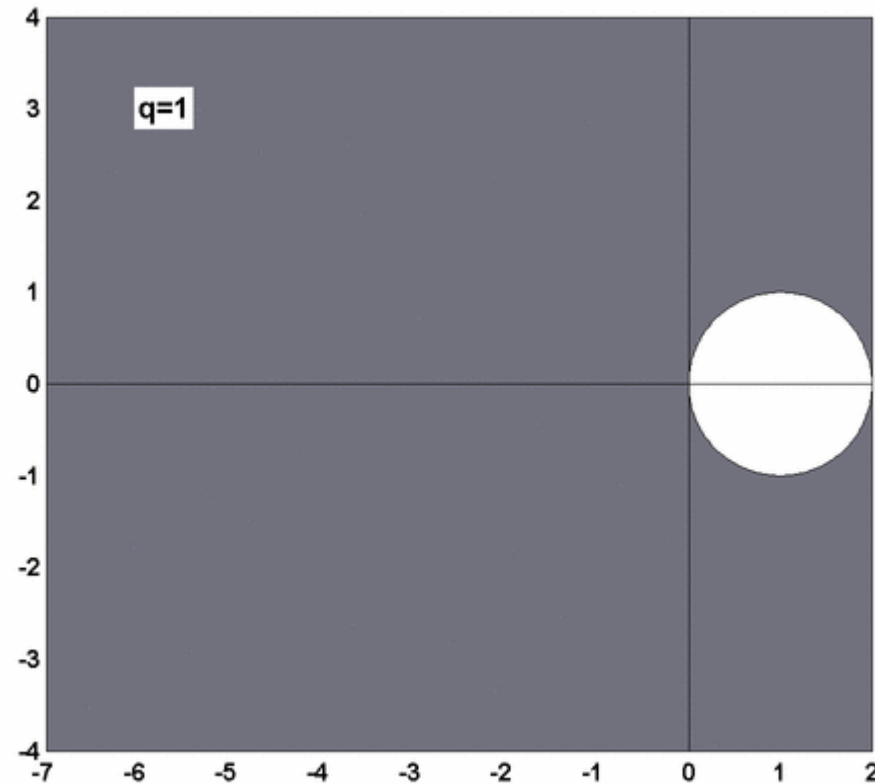
- Multi-step methods:

- Adams-Bashforth (Explicit – $\beta_0 = 0$)
- Adams-Moulton (Implicit)
 - PECE (Adams-Bashforth-Moulton)
- Backward Differentiation Formula (BDF) (Implicit)
 - Numerical Differential Formula (NDF)

- Same stability concepts as for one-step RK methods apply



Stability region for Adams-Moulton



Differential-Algebraic Equations (DAE)

- ODE: $\dot{y} = f(y), \quad y(0) = y_0$
- DAE: $\dot{y} = f(y, z), \quad y(0) = y_0$
 $0 = g(y, z)$
- Advanced simulation tools (like those based on Modelica) in general generate DAEs
- If $\frac{\partial g(y, z)}{\partial z}$ is invertible [**DAE is index 1**] then
 $0 = g(y, z)$ can be solved to $z = z(y)$
either symbolically (by hand/computer) or numerically
- Then the DAE can be written as ODE:
 $\dot{y} = f(y, z(y)) = \tilde{f}(y)$
and ODE solvers (ERK, IRK, BDF, ...) can be used

Differential index DAE

$$\dot{y} = f(y, z, t), \quad y(0) = y_0$$
$$0 = g(y, z, t)$$

The differential index of a DAE system is the minimum number of times $g(y, z, t)$ must be differentiated with respect to time in order to yield a pure ODE system:

$$\dot{y} = f(y, z, t)$$

$$\dot{z} = s(y, z, t)$$

Problems if g is not invertible

- Not all differential variables are state variables of the system
 - The initial values y_0 of the differential variables cannot be chosen freely, but are constrained.
 - The constraints are “not visible” and not explicitly given with the systems equations
 - The initialization of the simulation is not without further ado possible
- Before the simulation an analysis of the properties of the DAE system has to be performed

Differential-Algebraic equations (DAE), II

- Some higher-order index systems ($\frac{\partial g(y,z)}{\partial z}$ not invertible) can be reduced to index 1 [**index reduction**] by using certain tricks
 - And thereby be transformed to ODE system
- Not always possible *nor desirable* to solve DAE as ODE
 - Some numerical solvers can solve (low index) DAE problems directly (especially *implicit* solvers, who must solve nonlinear equations anyway)
 - See book for examples (IRK: 14.12.1, BDF: 14.12.2)
- DAE Software
 - Matlab: ode15s, ode23t ([link](#))
 - DASSL/DASPK
 - Sundials IDA
 - And others...

Example: Index Reduction

- Example: $\dot{x}_2 = x_1 + u_1$

$$0 = x_2 + u_2$$

diff. var. x_2

alg. var. x_1

$$\frac{\partial g}{\partial z} = \frac{\partial (x_2 + u_2)}{\partial x_1} = 0 \rightarrow \text{not index 1}$$

Find:

$$1. \text{Diff: } 0 = \dot{x}_2 + \dot{u}_2$$

$$= x_1 + u_1 + \dot{u}_2$$

$$2. \text{Diff: } 0 = \dot{x}_1 + \dot{u}_1 + \ddot{u}_2$$

$$\rightarrow \dot{x}_1 = -\dot{u}_1 - \ddot{u}_2$$

\Rightarrow Index 2 since 2. diff.

$$x_1 = -\dot{u}_2 - u_1$$

$$x_2 = -u_2$$

Try yourself: Find differential index

- Rewrite to state-space
- Find diff. Index
- Create index 1 system

diff. var : x, \dot{x}, y, \dot{y}
 alg. var : ϕ, F

$$\ddot{x} = -\frac{F}{m} \sin \phi$$

$$\ddot{y} = g - \frac{F}{m} \cos \phi$$

$$0 = x - l \sin \phi$$

$$0 = l^2 - x^2 - y^2$$

Homework

- What is local extrapolation and when it is used?
- What is FSAL? For what is it used?
- What is a dense output, how is the Runge-Kutta method called using it, and when it is useful?
- Implement the *bouncing ball* example using slides 28-32
- Read 14.11
- Find the differential index of the pendulum (slide 40)
- Check out the self-study part (slide 43-49)

Kahoot

- <https://play.kahoot.it/#/k/5199a4d4-e54b-4f4b-81ea-8c8f1c3170e7>

Self-study

Software

ODE solvers:

- Numerical implementations with control over accuracy (variable time-step solvers)

(ODE Integrators: Fixed step solvers)

Matlab ODE-solvers – nonstiff systems

- **ode45** is based on an explicit Runge-Kutta (4,5) formula, the **Dormand-Prince pair**. It is a one-step solver – in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, **ode45** is the best function to apply as a first try for most problems.
- **ode23** is an implementation of an explicit Runge-Kutta (2,3) pair of **Bogacki and Shampine**. It may be more efficient than **ode45** at crude tolerances and in the presence of moderate stiffness. Like **ode45**, **ode23** is a one-step solver.
- **ode113** is a variable order **Adams-Bashforth-Moulton PECE** solver. It may be more efficient than **ode45** at stringent tolerances and when the ODE file function is particularly expensive to evaluate. **ode113** is a multistep solver — it normally needs the solutions at several preceding time points to compute the current solution.

Matlab ODE-solvers – stiff systems

- **ode15s** is a variable order solver based on the **numerical differentiation formulas (NDFs)**. Optionally, it uses the **backward differentiation formulas (BDFs)** that are usually less efficient. Like **ode113**, **ode15s** is a multistep solver. Try **ode15s** when **ode45** fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem.
- **ode23s** is based on a **modified Rosenbrock formula of order 2**. Because it is a one-step solver, it may be more efficient than **ode15s** at crude tolerances. It can solve some kinds of stiff problems for which **ode15s** is not effective.
- **ode23t** is an implementation of **the trapezoidal rule** using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. **ode23t** can solve DAEs.
- **ode23tb** is an implementation of TR-BDF2, **an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two**. By construction, the same iteration matrix is used in evaluating both stages. Like **ode23s**, this solver may be more efficient than **ode15s** at crude tolerances.

When to use?

| 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 | For problems with crude error tolerances or for solving moderately stiff problems. |
| ode113 | Nonstiff | Low to high | For problems with stringent error tolerances or for solving computationally intensive problems. |
| ode15s | Stiff | Low to medium | If ode45 is slow because the problem is stiff. |
| ode23s | Stiff | Low | If using crude error tolerances to solve stiff systems and the mass matrix is constant. |
| ode23t | Moderately Stiff | Low | For moderately stiff problems if you need a solution without numerical damping. |
| ode23tb | Stiff | Low | If using crude error tolerances to solve stiff systems. |

Other ODE packages



- Sundials, <https://computation.llnl.gov/casc/sundials/main.html>
 - SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers) consists of C-code for the following five solvers:
 - **CVODE solves initial value problems for ordinary differential equation (ODE) systems.**
 - CVODES solves ODE systems and includes sensitivity analysis capabilities (forward and adjoint).
 - IDA solves initial value problems for differential-algebraic equation (DAE) systems.
 - IDAS solves DAE systems and includes sensitivity analysis capabilities (forward and adjoint).
 - KINSOL solves nonlinear algebraic systems.
- CVODE
 - Non-stiff systems: Adams-Moulton
 - Stiff systems: BDF
 - With event detection («rootfinding»)

Other solvers: DASSL

- Differential Algebraic System Solver
 - <http://engineering.ucsb.edu/~cse/software.html>
 - Based on BDF
 - Fortran implementation
 - With event detection («rootfinding»)
- Developed in the eighties, widely used
- Default solver in many Modelica packages
 - Dymola, OpenModelica, ...
- Extensions
 - DASPK 2.0: Large-scale systems
 - DASPK 3.1: With sensitivity analysis