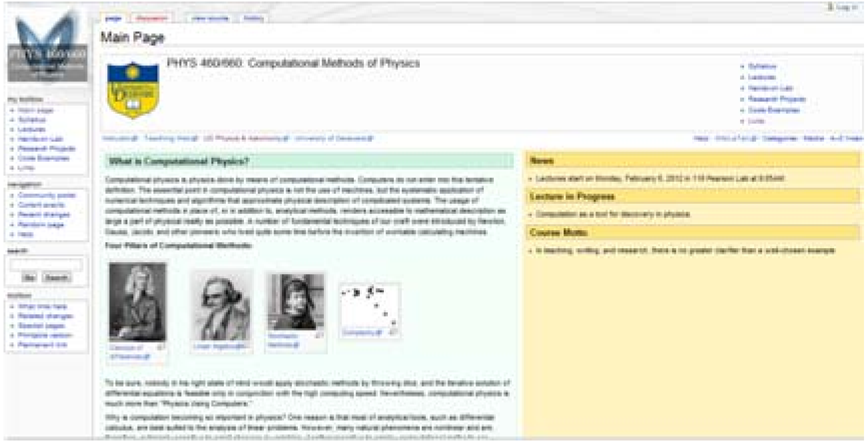
Numerical Methods for Ordinary Differential Equations

Branislav K. Nikolić

Department of Physics & Astronomy, University of Delaware, Newark, DE 19716, U.S.A.

PHYS 460/660: Computational Methods of Physics

http://wiki.physics.udel.edu/phys660



Terminology for ODEs

$$F\left(y, \frac{d}{dt} y(t), \frac{d^2}{dt^2} y(t), \ldots, \frac{d^n}{dt^n} y(t) \right) = 0$$

- **Ordinary:** only one independent variable
- **Differential:** unknown functions enter into the equation through its derivatives
- **Order:** highest derivative in F
- **Degree:** exponent of the highest derivative

$$\text{Example:} \left(\frac{d^2}{dt^2} y(t) \right)^3 - y(t) = 0$$

What Does It Mean to Solve ODE?

$$y = y(t)$$

- A problem involving ODE is not completely specified by its equation
- ODE has to be supplemented with **boundary conditions:**

  - **Initial value problem:** y is given at some starting value t_i, and it is desired to find y at some final points t_f or at some discrete list of points (for example, at tabulated intervals).

  - **Two point bondary value problem:** Boundary conditions are specified at more than one t; typically some of the conditions will be specified at t_i and some at t_f.

What Does it Mean to Numerically Solve ODE with the Initial Value Conditions?

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

- A numerical solution to this problem generates sequence of values for the independent variable

$$t_1, t_2, \ldots, t_n$$

and a corresponding sequence of values of the dependent variable

$$y_1, y_2, \ldots, y_n$$

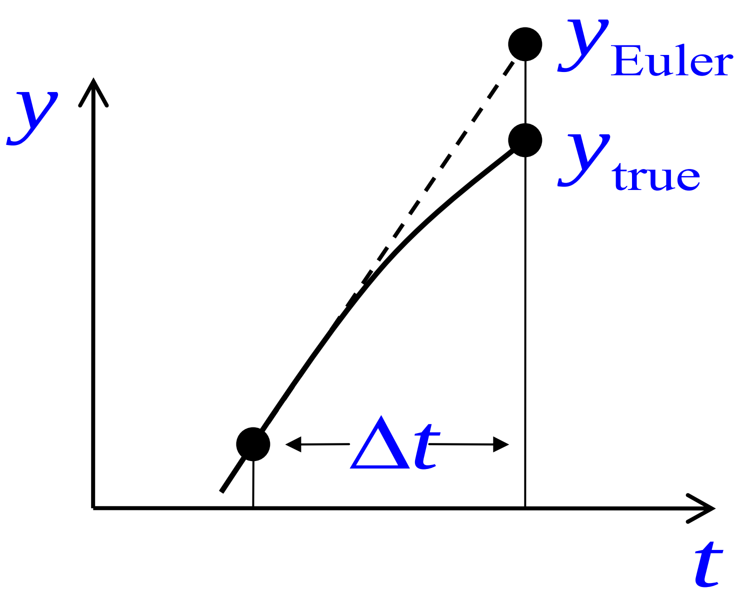
so that each y_n **approximates** solution at t_n:

$$y(t_n) \approx y_n, \quad n = 0, 1, \ldots$$

Euler Method Fundamentals

- All finite difference methods start from the same conceptual idea: Add small increments to your function corresponding to derivatives (right-hand side of the equations) multiplied by the stepsize.

- Euler method is an implementation of this idea in the simplest and most direct form.



DEFICIENCES OF EULER METHOD

- Tiny steps are needed to get even a few digits accuracy.
- The biggest defect of Euler method is actually inability to provide an error estimate.
- Thus, there is no automatic way to determine what step size is needed to achieve a specified accuracy.

Euler Algorithm for First-Order ODE Converted Into MATLAB Code

$$\frac{dy}{dt} = f(t,y) \longrightarrow \Delta y = f(t,y)\Delta t$$

%MATLAB code

$t = t_0;$

$y = y_0;$

while t <= tfinal

$\quad y = y + h * feval(f, t, y)$

$\quad t = t + h$

end

Step Size Effects in Radioactive Decay

$$\text{Analytics:}\ \frac{dN_U}{dt} = -\frac{N_U}{\tau} \Rightarrow N_U = N_U(t=0)e^{-\frac{t}{\tau}}$$

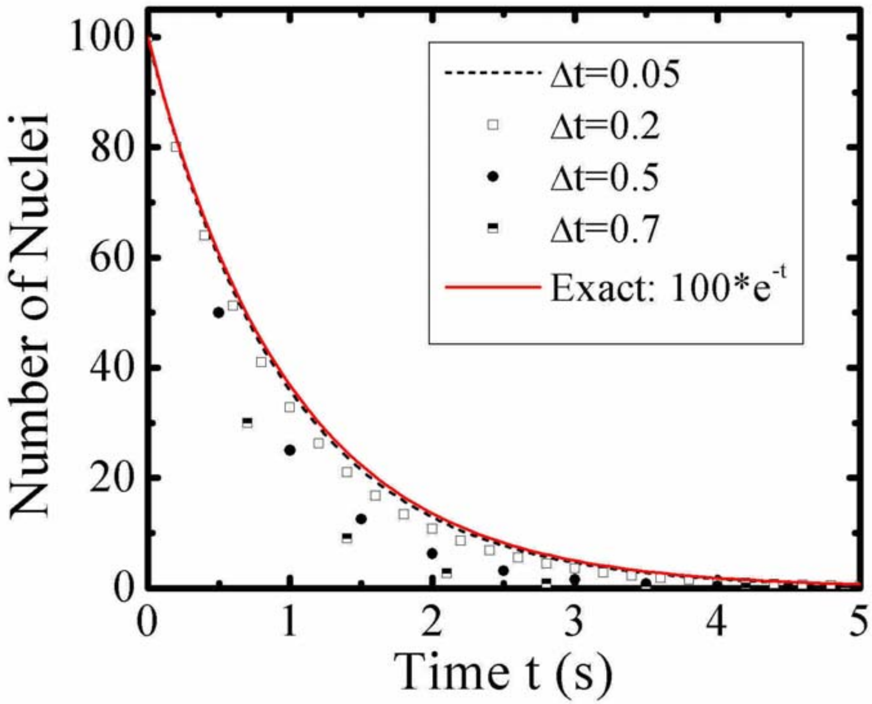
Numerics (Euler):

$$N_U(\Delta t) = N_U(0) + \frac{dN_U}{dt}\Delta t + O\left((\Delta t)^2\right)$$

$$N_{i+1} \approx N_i - \frac{N_i}{\tau}\Delta t$$

Numerical solution will depend on the step size

$$\Delta t$$



Stability of Euler Algorithm

- Step size if often limited by the **stability criterion**:

$$\frac{dy}{dt} = -ay \Rightarrow y(0) = 1,\ y = e^{-at}$$

After n Euler steps of size Δt:

$$y_{n+1} = y_n - ay_n\Delta t \Rightarrow y_n = (1 - a\Delta t)^n$$

Approximate solution will decay monotonically only if Δt is small enough:

$$\Delta t \le \Delta t_{\max} \equiv \frac{1}{a}$$

- For a single decaying exponential-like solution (i.e. if there is only one first order equation) the existence of a stability criterion is not a problem because Δt has to be small for the reasons of **accuracy**.

Accuracy:
Discretization and Roundoff Errors

Integrate over interval: $L = t_f - t_0 \Rightarrow$ Full Error: $Ch^p + \dfrac{L\varepsilon}{h}$

Number of steps for roundoff error to be comparable with the discretization error: $N \approx L\left(\dfrac{C}{L\varepsilon}\right)^{\frac{1}{p+1}}$

❑ <u>Local:</u>

$$\left.\begin{array}{l} \dfrac{du}{dt} = f(u_n, t_n) \\ u_n(t_n) = y_n \end{array}\right\} \Rightarrow LE_n = y_{n+1} - u_{n+1}(t_{n+1})$$

❑ <u>Global:</u>

$$GE_n = y_n - y(t_n)$$

❑ **Method is of order n iff:**

$$\boxed{LE_n = O(h^{n+1}) \Leftrightarrow |LE_n| \le Ch^{n+1}}$$

$$h = t_{n+1} - t_n \equiv \Delta t$$

$$f = f(t) \Rightarrow y(t) = \int_{t_0}^{t_N} f(\tau)d\tau \approx \sum_{n=0}^{N-1} h_n f(t_n)$$

$$LE_n = h_n f(t_n) - \int_{t_n}^{t_{n+1}} f(\tau)d\tau$$

$$GE_n = \sum_{n=0}^{N-1} h_n f(t_n) - \int_{t_0}^{t_N} f(\tau)d\tau$$

$$GE_n = \sum_{n=0}^{N-1} LE_n$$

special case where global error is trivially sum of local errors

Global Discretization Error by Example

- Suppose we want to find the solution over the interval $[0,T]$

→ we first divide the interval into n equal steps $\Delta t = T/n$

$$y(T) = e^{-aT}, \quad y_n = \left(1 - a\frac{T}{n}\right)^n$$

$$y(T) = 1 - aT + \frac{(aT)^2}{2!} - \frac{(aT)^3}{3!} + \ldots$$

$$y_n = 1 - aT + \frac{n(n-1)}{n^2}\frac{(aT)^2}{2!} - \frac{n(n-1)(n-2)}{n^3}\frac{(aT)^3}{3!} + \ldots$$

$$y(T) - y_n = \frac{1}{n}\frac{(aT)^2}{2!} - \frac{3}{n}\frac{(aT)^2}{3!} + \ldots + O\left(\frac{1}{n^2}\right) \sim \frac{a\Delta t}{2} aTe^{-aT}$$

- This is a measure of the global truncation error, i.e., the error over a fixed range in *t*.
- It is proportional to the first power of the step size, and hence the Euler method is a first order method - **do not confuse this with the fact that we are applying it to the case to a first order equation**

Reducing Higher Order ODE to a System of First Order ODE

- Solve higher order ODEs by splitting them into sets of first order equations:

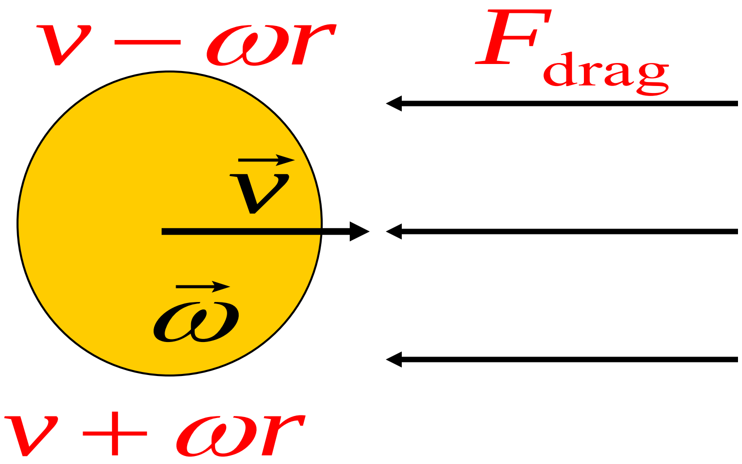
$$\frac{d^2y}{dt^2} + p(t)\frac{dy}{dt} + q(t)y = g(t)$$

$$z = \frac{dy}{dt} \Rightarrow \begin{cases} \dfrac{dz}{dt} = g(t) - p(t)z - q(t)y \\ \dfrac{dy}{dt} = z \end{cases}$$

There is no unique way to do this:

$$z = \frac{dy}{dt} + p(t)y \Rightarrow \begin{cases} \dfrac{dz}{dt} = g(t) + \left(\dfrac{dp(t)}{dt} - q(t)\right)y \\ \dfrac{dy}{dt} = z - p(t)y \end{cases}$$

Example: Realistic Motion of Baseball



$$m\frac{d^2\vec{r}}{dt^2} = m\vec{g} - B_2 v^2 \frac{\vec{v}}{v} + S_0 \vec{v}\times\vec{\omega}$$

initialize $\quad t_1, \vec{y}(t_1)$

do while $i \le n$

$\quad \vec{y}_{i+1} = \vec{y}_i + f(t_i, \vec{y}_i)\Delta t$

$\quad t_{i+1} = t_i + \Delta t$

end do

$\Leftrightarrow$

$$x_{i+1} = x_i + v_i^x \Delta t$$

$$v_{i+1}^x = v_i^x - \frac{B_2}{m} v v_i^x \Delta t$$

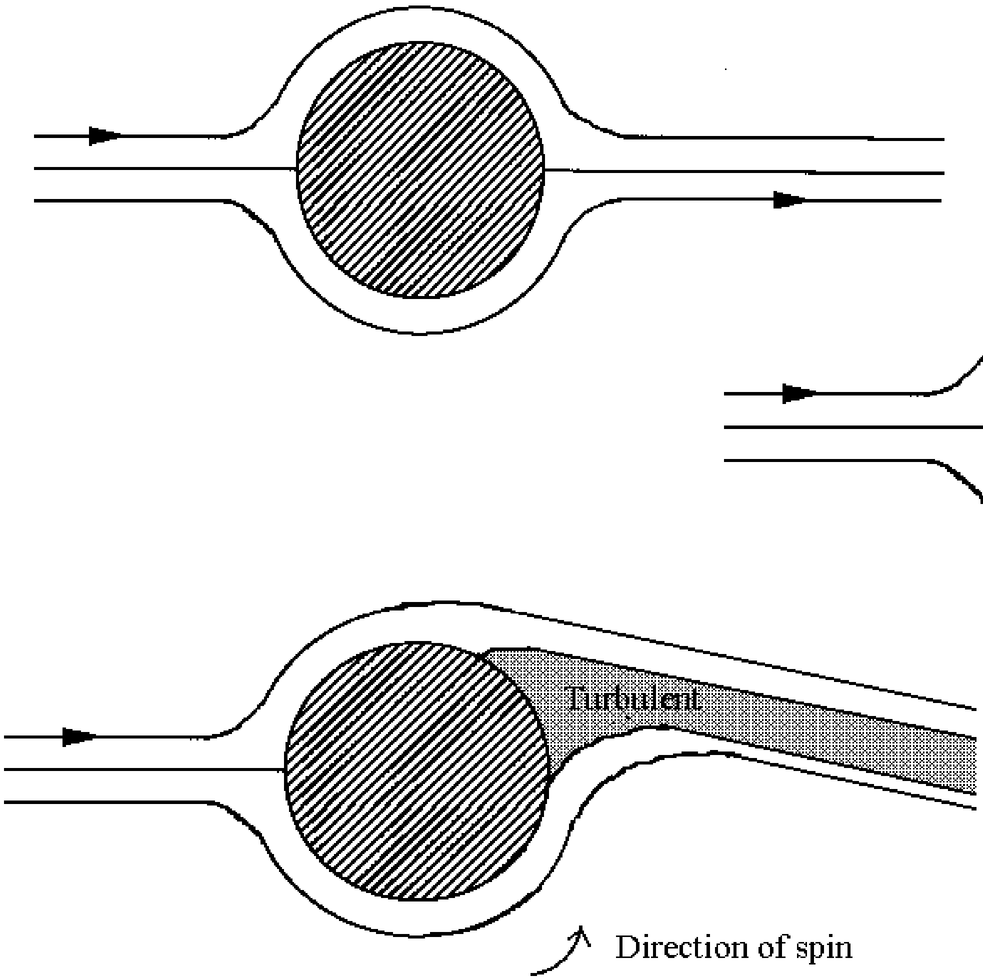
$$y_{i+1} = y_i + v_i^y \Delta t$$

$$v_{i+1}^y = v_i^y - g\Delta t$$

$$z_{i+1} = z_i + v_i^z \Delta t$$

$$v_{i+1}^z = v_i^z - \frac{S_0 v_x \omega}{m}\Delta t$$

More Realistic Modeling Beyond Laminar Air Flow: Turbulence Effects



ODE for Linear Harmonic Oscillator

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin\theta = 0$$

$$\text{for small } \theta \Rightarrow \sin\theta \approx \theta$$

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\theta = 0, \quad \Omega = \sqrt{\frac{g}{l}}$$

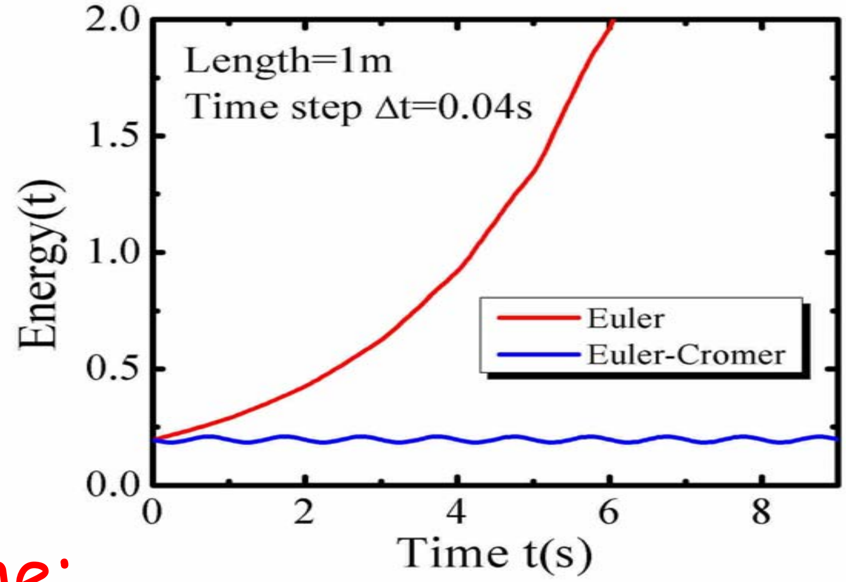
$$E_{total} = \frac{1}{2}ml^2\left(\frac{d\theta}{dt}\right)^2 + \frac{1}{2}mgl\theta^2 \text{ must be conserved!}$$

Euler Method for Linear Harmonic Oscillator

- Switch to **dimensionless** quantities:

$$\frac{d^2\theta}{dt^2}+\theta=0 \Rightarrow \theta=\theta_0\sin(\Omega t+\phi)$$

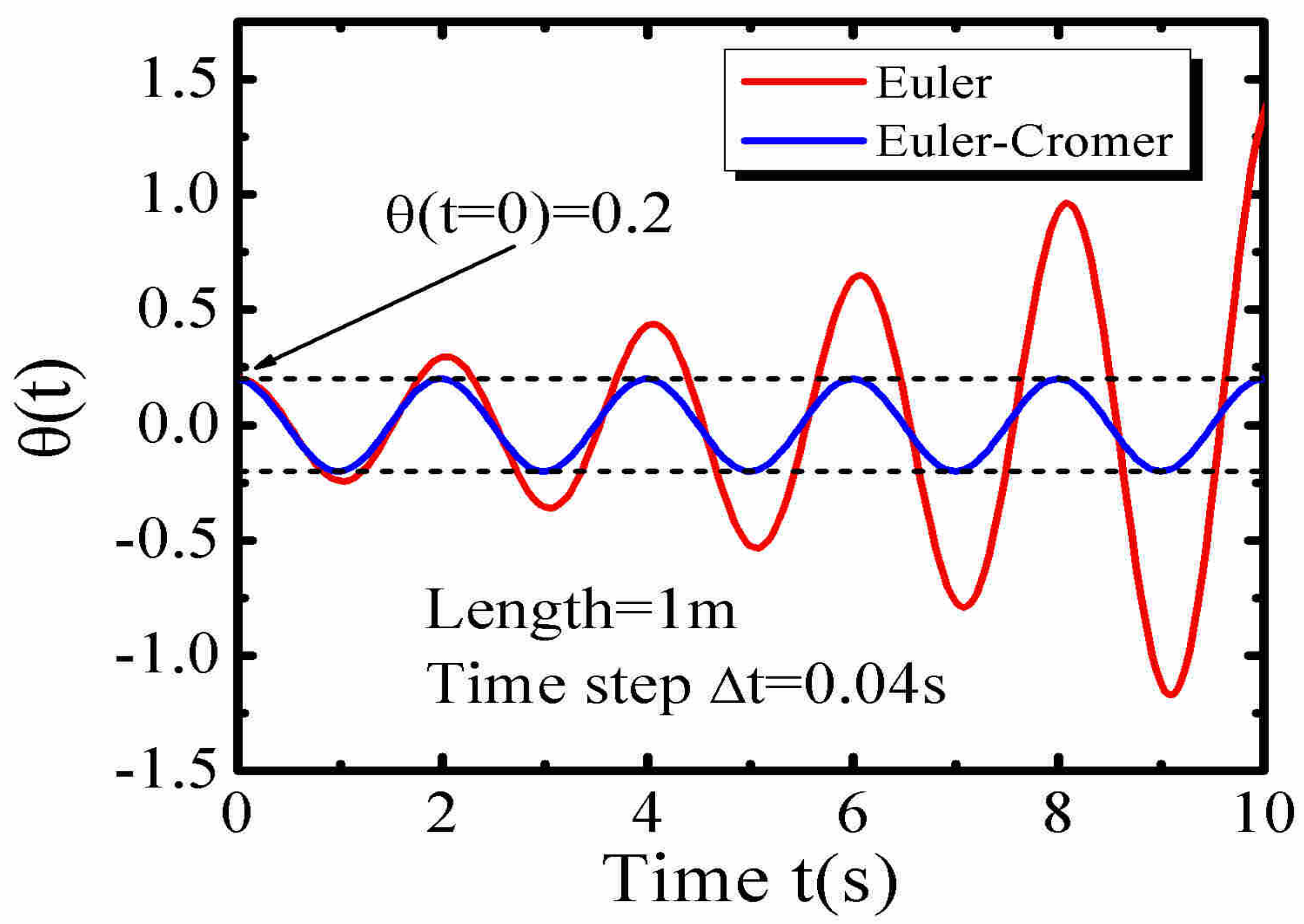
$$E_{total}=\frac{1}{2}\left(\frac{d\theta}{dt}\right)^2+\frac{1}{2}\theta^2$$



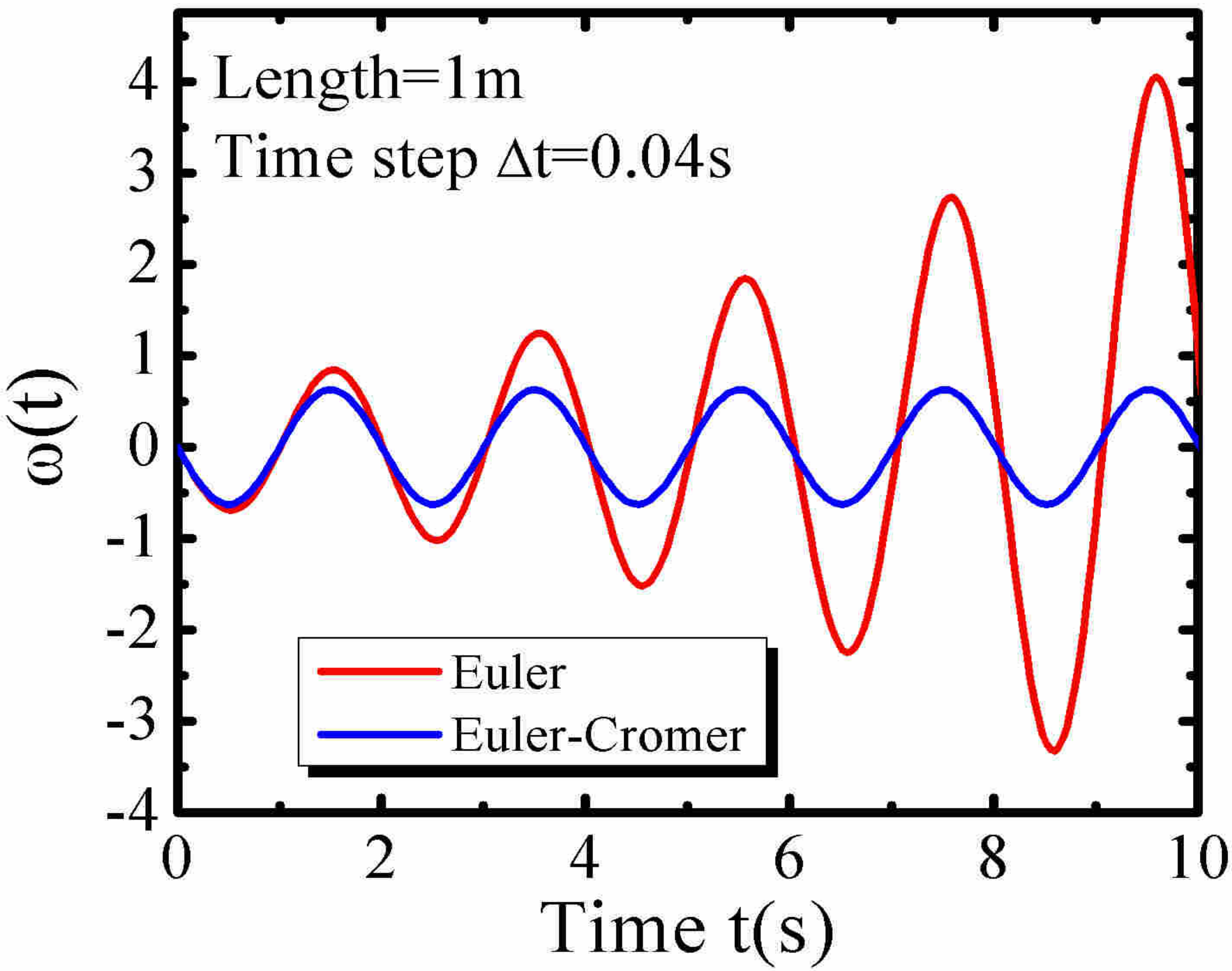
- Euler discretization scheme:

$$\left.\begin{aligned}\omega_{n+1}&=\omega_n-\theta_n\Delta t\\ \theta_{n+1}&=\theta_n+\omega_n\Delta t\\ t_{n+1}&=t_n+\Delta t\end{aligned}\right\}\Rightarrow\left\{\begin{aligned}E_{total}&=\frac{1}{2}\left(\omega^2{}_{n+1}+\theta^2{}_{n+1}\right)\\ E_{total}&=E_n\left(1+\Delta t^2\right)\end{aligned}\right.$$

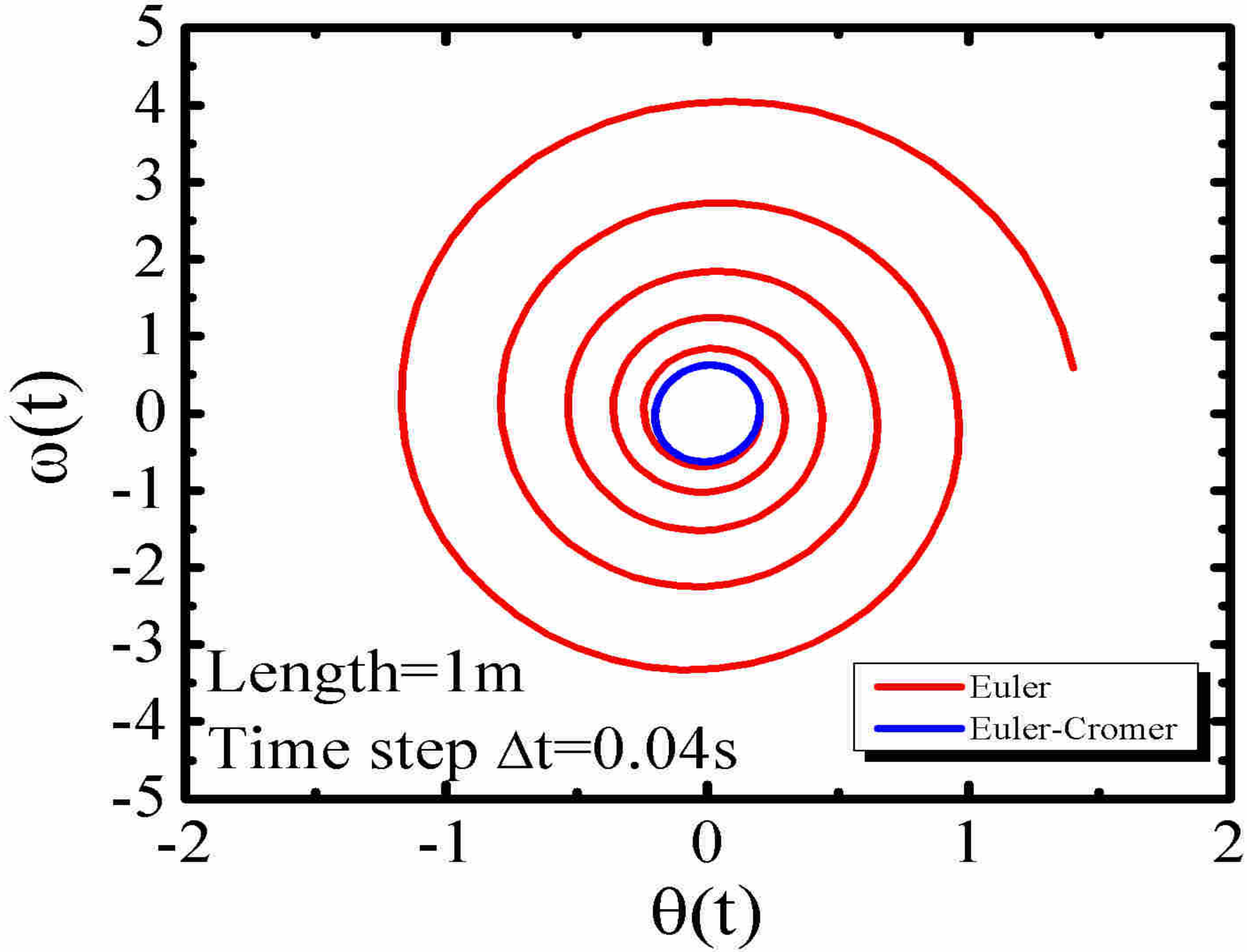
Euler Method Fails for $\theta(t)$



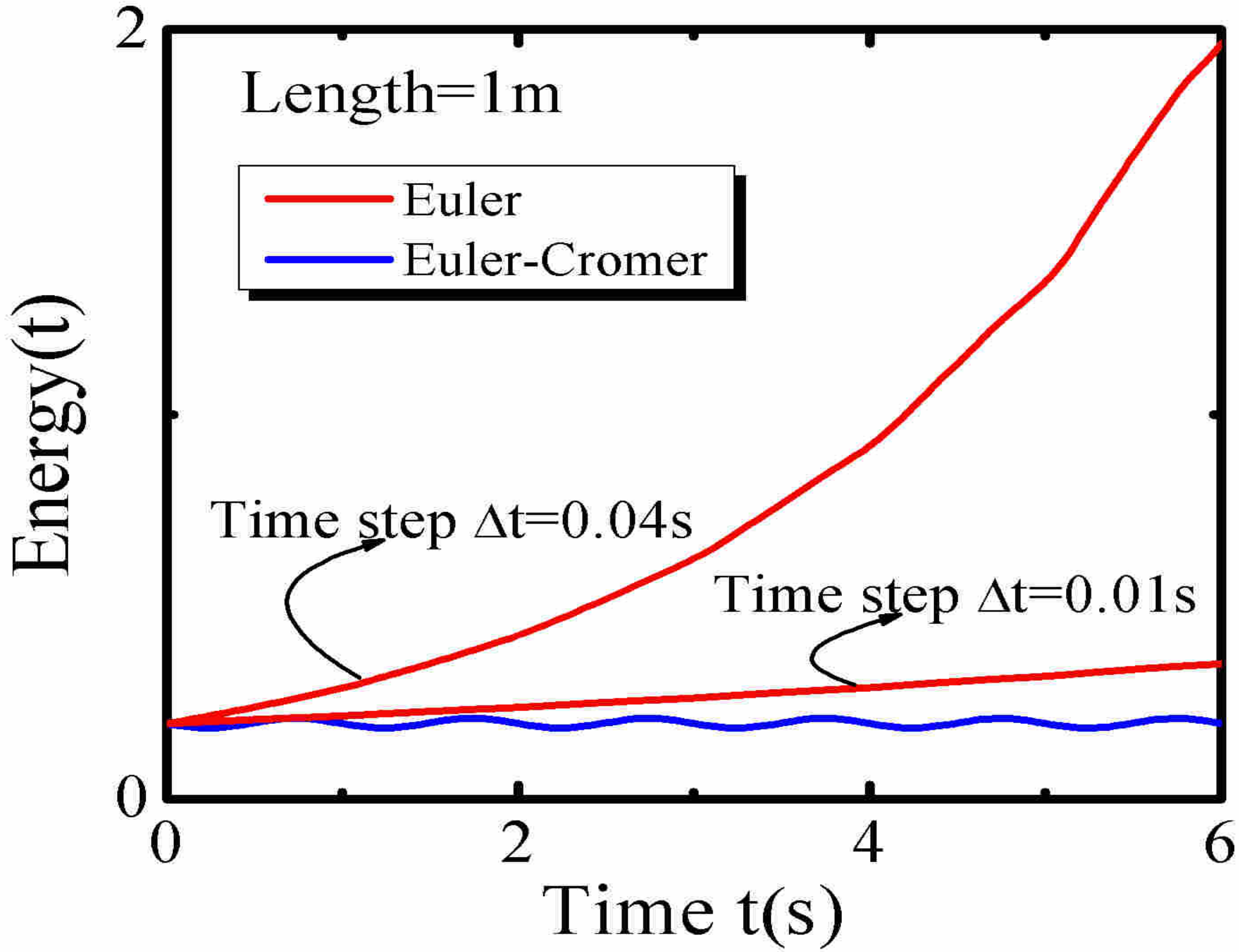
Euler Method Fails for $\omega(t)$



Euler Fails for Phase Space Trajectory



Can We Save Euler Method by Using Smaller Step Size?



Cromer Fix for Euler Method Applied to LHO

$$\omega_n \to \omega_{n+1} \Rightarrow \begin{cases} \omega_{n+1} = \omega_n - \theta_n \Delta t \\ \theta_{n+1} = \theta_n + \omega_{n+1} \Delta t \\ t_{n+1} = t_n + \Delta t \end{cases}$$

- Apparently trivial trick, but:

$$E_{n+1} = E_n + \frac{1}{2}\left(\omega_n{}^2 - \theta_n{}^2\right)\Delta t^2 + O\left(\Delta t^3\right)$$

$$\underbrace{\theta = \theta_0 \sin(t - t_0), \quad \omega = \theta_0 \cos(t - t_0)}_{\left\langle \omega^2 - \theta^2 = \theta_0{}^2 \cos 2(t - t_0) \right\rangle_{over\ a\ period} = 0}$$

From Euler to Higher Order Algorithms

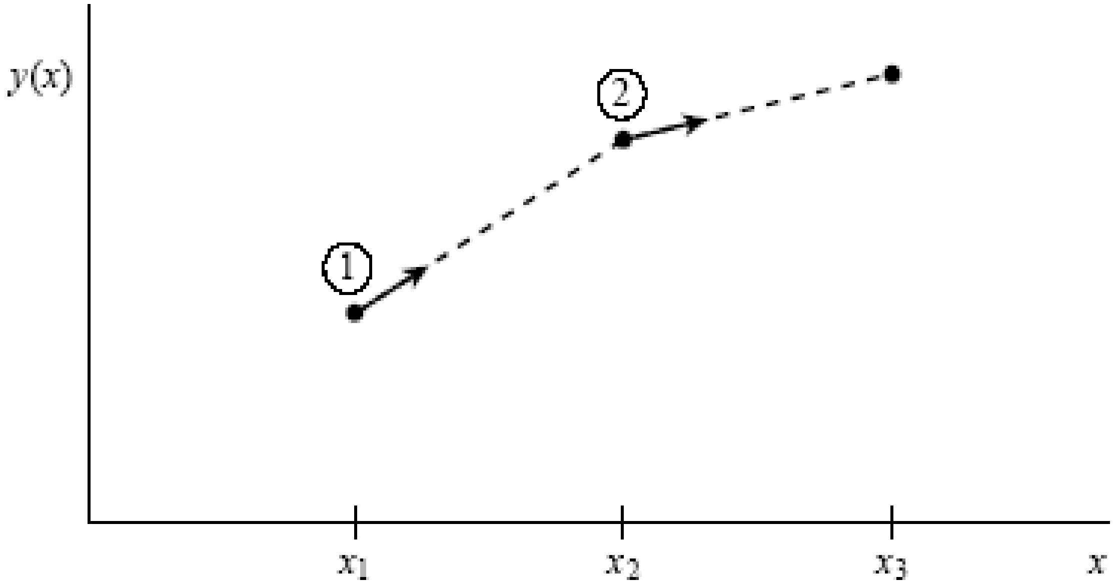


Figure 16.1.1. Euler's method. In this simplest (and least accurate) method for integrating an ODE, the derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.

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

vs.

Mean value theorem

$$y(t+\Delta t) \overset{exact}{=} y(t) + dy/dt\big|_{t_m} \Delta t$$

Midpoint Method: Second Order Runge-Kutta

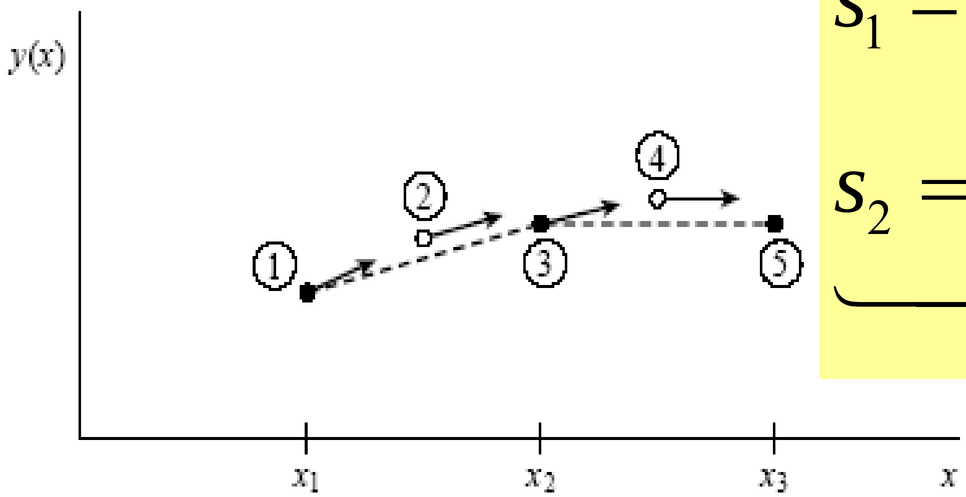


Figure 16.1.2. Midpoint method. Second-order accuracy is obtained by using the initial derivative at each step to find a point halfway across the interval, then using the midpoint derivative across the full width of the interval. In the figure, filled dots represent final function values, while open dots represent function values that are discarded once their derivatives have been calculated and used.

$$s_1 = f(t_n, y_n)$$

$$\underbrace{s_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} s_1\right)}$$

$$y_{n+1} = y_n + h s_2 + O(h^3)$$

$$t_{n+1} = t_n + h$$

Classic Runge-Kutta Method

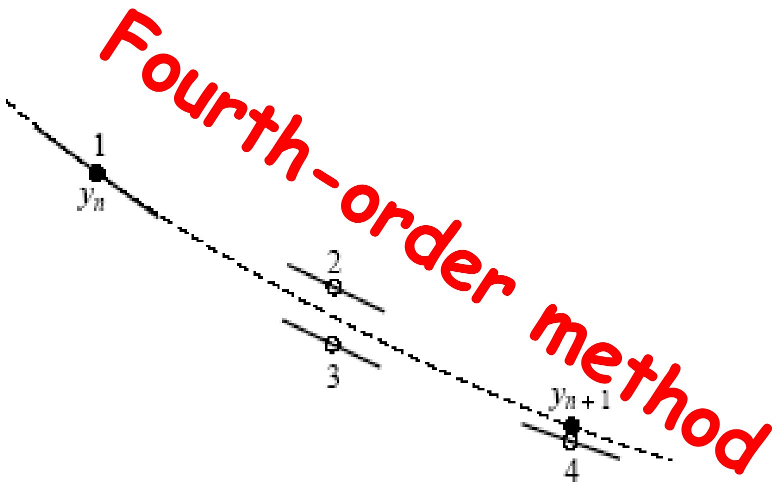


Figure 16.1.3. Fourth-order Runge-Kutta method. In each step the derivative is evaluated four times: once at the initial point, twice at trial midpoints, and once at a trial endpoint. From these derivatives the final function value (shown as a filled dot) is calculated. (See text for details.)

$$\underbrace{\begin{aligned} s_1 &= f(t_n, y_n) \\ s_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} s_1\right) \\ s_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} s_2\right) \\ s_4 &= f(t_n + h, y_n + h s_3) \end{aligned}}$$

$$y_{n+1} = y_n + \frac{h}{6}(s_1 + 2s_2 + 2s_3 + s_4) + O(h^5)$$

$$t_{n+1} = t_n + h$$

Classic Runge-Kutta: Fortran Program vs. Matlab Script

```
SUBROUTINE rk4(y,dydx,n,x,h,yout,derivs)
INTEGER n,NMAX
REAL h,x,dydx(n),y(n),yout(n)
EXTERNAL derivs
PARAMETER (NMAX=50)            Set to the maximum number of functions.
```

Given values for the variables y(1:n) and their derivatives dydx(1:n) known at x, use the fourth-order Runge-Kutta method to advance the solution over an interval h and return the incremented variables as yout(1:n), which need not be a distinct array from y. The user supplies the subroutine derivs(x,y,dydx), which returns derivatives dydx at x.

```
INTEGER i
REAL h6,hh,xh,dym(NMAX),dyt(NMAX),yt(NMAX)
hh=h*0.5
h6=h/6.
xh=x+hh
do 11 i=1,n                    First step.
    yt(i)=y(i)+hh*dydx(i)
enddo 11
call derivs(xh,yt,dyt)         Second step
do 12 i=1,n
    yt(i)=y(i)+hh*dyt(i)
enddo 12
call derivs(xh,yt,dym)         Third step.
do 13 i=1,n
    yt(i)=y(i)+h*dym(i)
    dym(i)=dyt(i)+dym(i)
enddo 13
call derivs(x+h,yt,dyt)        Fourth step.
do 14 i=1,n                    Accumulate increments with proper weights.
    yout(i)=y(i)+h6*(dydx(i)+dyt(i)+2.*dym(i))
enddo 14
return
END
```

MATLAB

```
clc;                                                  % Clears the screen
clear all;

h=1.5;                                                % step size
x = 0:h:3;                                            % Calculates upto y(3)
y = zeros(1,length(x));
y(1) = 5;                                             % initial condition
F_xy = @(t,r) 3.*exp(-t)-0.4*r;                       % change the function as you desire

for i=1:(length(x)-1)                                 % calculation loop
    k_1 = F_xy(x(i),y(i));
    k_2 = F_xy(x(i)+0.5*h,y(i)+0.5*h*k_1);
    k_3 = F_xy((x(i)+0.5*h),(y(i)+0.5*h*k_2));
    k_4 = F_xy((x(i)+h),(y(i)+k_3*h));

    y(i+1) = y(i) + (1/6)*(k_1+2*k_2+2*k_3+k_4)*h;  % main equation
end
```

General Algorithm for Single-Step Methods

- Each of the k stages of the algorithm computes slope s_i by evaluating $f(t,y)$ for a particular value of t and a value of y obtained by taking linear combinations of the previous slopes:

$$s_i = f(t_n + \alpha_i h, y_n + h\sum_{j=1}^{i-1}\beta_{i,j}s_j), i = 1,\ldots,k$$

- The proposed step is also a linear combination of the slopes:

$$y_{n+1} = y_n + h\sum_{i=1}^{k}\gamma_i s_i$$

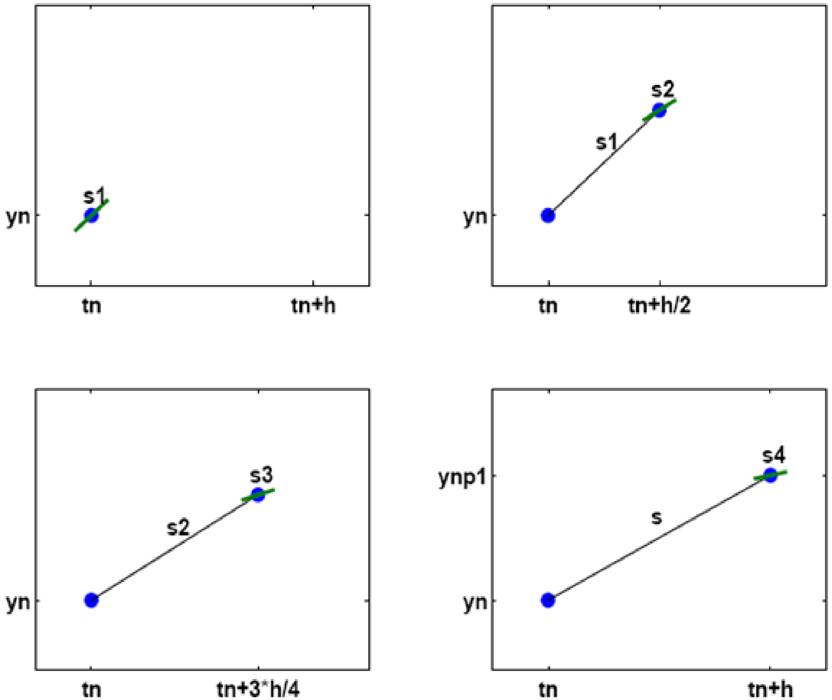
- Error is estimated from yet another linear combination of the slopes:

$$e_{n+1} = h\sum_{i=1}^{k}\delta_i s_i$$

- The parameters are determined by matching terms in the Taylor series expansion of the slopes → **the order of the method is the exponent of the smallest power of h that cannot be matched**

- In MATLAB ODE numerical routines are named as **odennxx**, where **nn** indicates the order and **xx** is some special feature of the method.

Example: MATLAB ode23 Function (Bogacki and Shampine BS23 Algorithm)



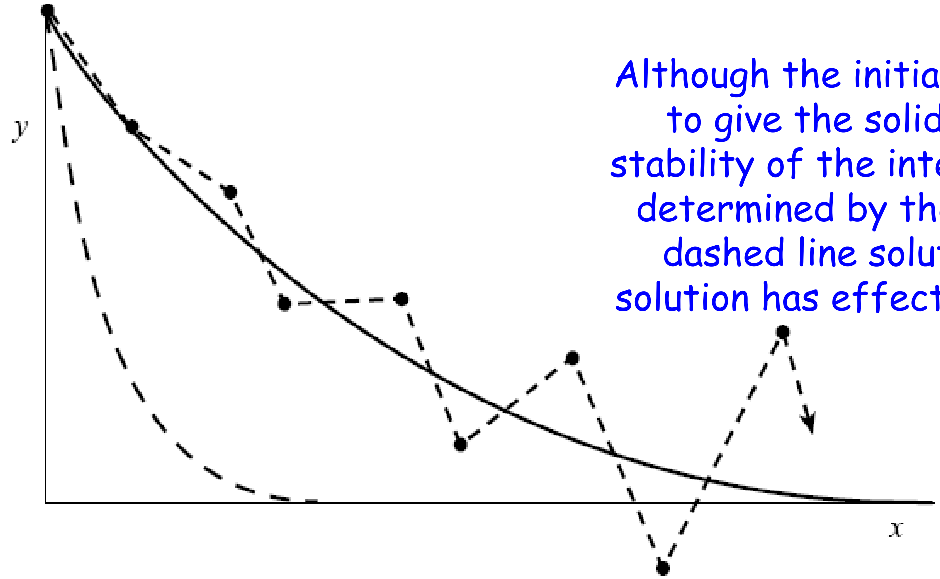
$$\underbrace{\begin{aligned} s_1 &= f(t_n, y_n) \\ s_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} s_1\right) \\ s_3 &= f\left(t_n + \frac{3}{4}h, y_n + \frac{3}{4} h s_2\right) \end{aligned}}$$

$$\left.\begin{aligned} y_{n+1} &= y_n + \frac{h}{9}(2s_1 + 3s_2 + 4s_3) \\ t_{n+1} &= t_n + h;\ s_4 = f(t_{n+1}, y_{n+1}) \end{aligned}\right\}$$

$$e_{n+1} = \frac{h}{72}(-5s_1 + 6s_2 + 8s_3 - 9s_4)$$

Special Numerical Algorithms are Required for the So-Called Stiff ODEs

- ODE is stiff if the solution being sought is varying slowly, but there are nearby solutions that vary rapidly, so the numerical method must take small steps to obtain satisfactory results



Although the initial conditions are such as to give the solid line as solution, the stability of the integration (dotted line) is determined by the more rapidly varying dashed line solution, even after that solution has effectively died away to zero

Implicit methods offer cure for stifness:

$$y' = -cy, c > 0 \overset{\text{explicit}}{\Rightarrow} y_{n+1} = y_n + \Delta t y_n' = (1 - c\Delta t) y_n$$

$$\Delta t > 2/c \Leftrightarrow |y_n| \to \infty \text{ as } n \to \infty$$

$$y' = -cy \overset{\text{implicit}}{\Rightarrow} y_{n+1} = y_n + \Delta t y_{n+1}' \Rightarrow y_{n+1} = \frac{y_n}{1 + c\Delta t}$$