Chapter Objectives

- Understanding the meaning of local and global truncation errors and their relationship to step size for one-step methods for solving ODEs.
- Knowing how to implement the following Runge-Kutta (RK) methods for a single ODE:
  - Euler
  - Heun
  - Midpoint
  - Fourth-Order RK
- Knowing how to iterate the corrector of Heun’s method.
- Knowing how to implement the following Runge-Kutta methods for systems of ODEs:
  - Euler
  - Fourth-order RK

Ordinary Differential Equations

- Methods described here are for solving differential equations of the form:

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

- The methods in this chapter are all *one-step* methods and have the general format:

$$y_{i+1} = y_i + \phi h$$

where ϕ is called an *increment function*, and is used to extrapolate from an old value y_i to a new value y_{i+1}.

Euler's Method

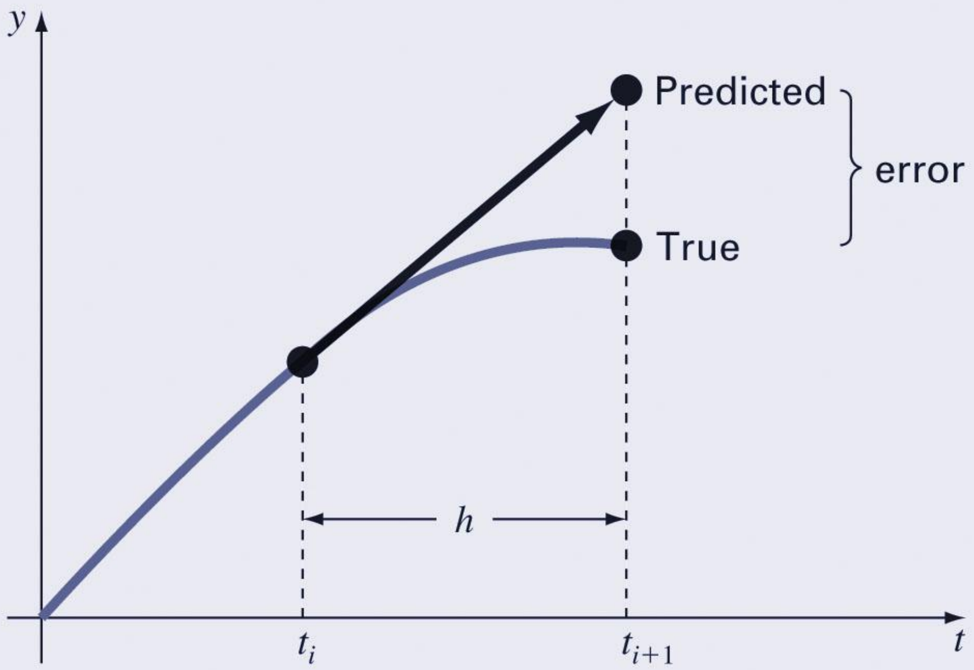
- The first derivative provides a direct estimate of the slope at t_i:

$$\left.\frac{dy}{dt}\right|_{t_i} = f(t_i, y_i)$$

and the Euler method uses that estimate as the increment function:

$$\phi = f(t_i, y_i)$$

$$y_{i+1} = y_i + f(t_i, y_i)h$$



Error Analysis for Euler's Method

- The numerical solution of ODEs involves two types of error:
  - *Truncation errors*, caused by the nature of the techniques employed
  - *Roundoff errors*, caused by the limited numbers of significant digits that can be retained
- The total, or *global* truncation error can be further split into:
  - *local truncation error* that results from an application method in question over a single step, and
  - *propagated truncation error* that results from the approximations produced during previous steps.

Error Analysis for Euler's Method

- The local truncation error for Euler's method is $O(h^2)$ and proportional to the derivative of $f(t,y)$ while the global truncation error is $O(h)$.
- This means:
  - The global error can be reduced by decreasing the step size, and
  - Euler's method will provide error-free predictions if the underlying function is linear.
- Euler's method is *conditionally stable*, depending on the size of h.

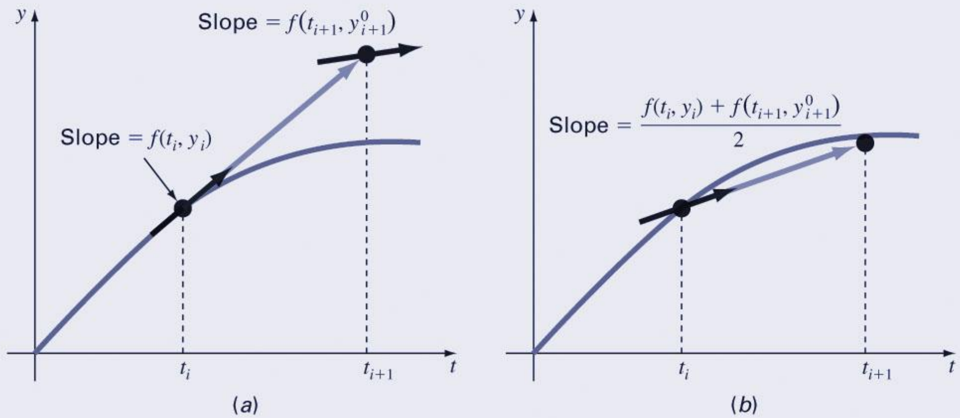
MATLAB Code for Euler’s Method

```
function [t,y] = eulode(dydt,tspan,y0,h,varargin)
% eulode: Euler ODE solver
%   [t,y] = eulode(dydt,tspan,y0,h,p1,p2,...):
%             uses Euler's method to integrate an ODE
% input:
%   dydt = name of the M-file that evaluates the ODE
%   tspan = [ti, tf] where ti and tf = initial and
%             final values of independent variable
%   y0 = initial value of dependent variable
%   h = step size
%   p1,p2,... = additional parameters used by dydt
% output:
%   t = vector of independent variable
%   y = vector of solution for dependent variable

if nargin<4,error('at least 4 input arguments required'),end
ti = tspan(1);tf = tspan(2);
if ~(tf>ti),error('upper limit must be greater than lower'),end
t = (ti:h:tf)'; n = length(t);
% if necessary, add an additional value of t
% so that range goes from t = ti to tf
if t(n)<tf
  t(n+1) = tf;
  n = n+1;
end
y = y0*ones(n,1); %preallocate y to improve efficiency
for i = 1:n-1 %implement Euler's method
  y(i+1) = y(i) + dydt(t(i),y(i),varargin{:})*(t(i+1)-t(i));
end
```

Heun’s Method

- One method to improve Euler’s method is to determine derivatives at the beginning and predicted ending of the interval and average them:

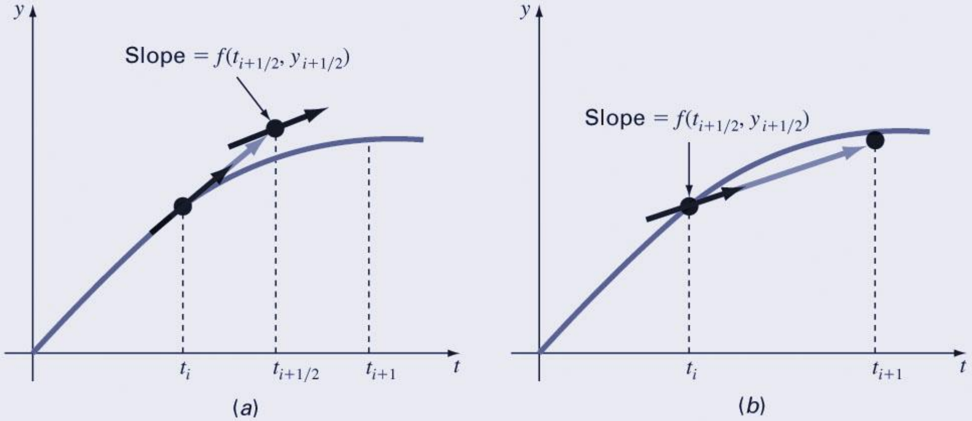


- This process relies on making a prediction of the new value of *y*, then correcting it based on the slope calculated at that new value.
- This predictor-corrector approach can be iterated to convergence:

$$y_{i+1}^{j} \longleftarrow y_i^m + \frac{f(t_i, y_i^m) + f\left(t_{i+1}, y_{i+1}^{j-1}\right)}{2} h$$

Midpoint Method

- Another improvement to Euler's method is similar to Heun's method, but predicts the slope at the midpoint of an interval rather than at the end:



- This method has a local truncation error of $O(h^3)$ and global error of $O(h^2)$

Runge-Kutta Methods

- Runge-Kutta (RK) methods achieve the accuracy of a Taylor series approach without requiring the calculation of higher derivatives.
- For RK methods, the increment function ϕ can be generally written as:

$$\phi = a_1 k_1 + a_2 k_2 + \cdots + a_n k_n$$

where the *a*'s are constants and the *k*'s are

$$\begin{aligned}
k_1 &= f(t_i, y_i) \\
k_2 &= f(t_i + p_1 h, y_i + q_{11} k_1 h) \\
k_3 &= f(t_i + p_2 h, y_i + q_{21} k_1 h + q_{22} k_2 h) \\
&\vdots \\
k_n &= f(t_i + p_{n-1} h, y_i + q_{n-1,1} k_1 h + q_{n-1,2} k_2 h + \cdots + q_{n-1,n-1} k_{n-1} h)
\end{aligned}$$

where the *p*'s and *q*'s are constants.

Classical Fourth-Order Runge-Kutta Method

- The most popular RK methods are fourth-order, and the most commonly used form is:

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)h$$

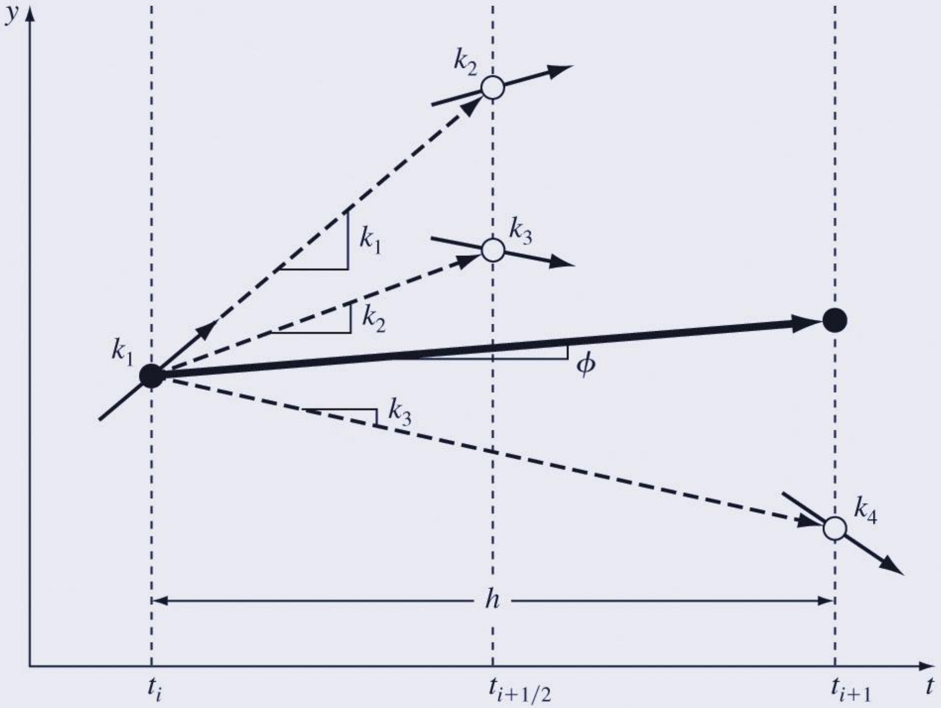
where:

$$k_1 = f\left(t_i, y_i\right)$$

$$k_2 = f\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h\right)$$

$$k_3 = f\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2 h\right)$$

$$k_4 = f\left(t_i + h, y_i + k_3 h\right)$$



Systems of Equations

- Many practical problems require the solution of a *system* of equations:

$$\frac{dy_1}{dt} = f_1(t, y_1, y_2, \cdots, y_n)$$
$$\frac{dy_2}{dt} = f_2(t, y_1, y_2, \cdots, y_n)$$
$$\vdots$$
$$\frac{dy_n}{dt} = f_n(t, y_1, y_2, \cdots, y_n)$$

- The solution of such a system requires that *n* initial conditions be known at the starting value of *t*.

Solution Methods

- Single-equation methods can be used to solve systems of ODE’s as well; for example, Euler’s method can be used on systems of equations - the one-step method is applied for every equation at each step before proceeding to the next step.
- Fourth-order Runge-Kutta methods can also be used, but care must be taken in calculating the *k*’s.

MATLAB RK4 Code

```
function [tp,yp] = rk4sys(dydt,tspan,y0,h,varargin)
% rk4sys: fourth-order Runge-Kutta for a system of ODEs
%   [t,y] = rk4sys(dydt,tspan,y0,h,p1,p2,...): integrates
%           a system of ODEs with fourth-order RK method
% input:
%   dydt = name of the M-file that evaluates the ODEs
%   tspan = [ti, tf]; initial and final times with output
%                     generated at interval of h, or
%   = [t0 t1 ... tf]; specific times where solution output
%   y0 = initial values of dependent variables
%   h = step size
%   p1,p2,... = additional parameters used by dydt
% output:
%   tp = vector of independent variable
%   yp = vector of solution for dependent variables

if nargin<4,error('at least 4 input arguments required'), end
if any(diff(tspan)<=0),error('tspan not ascending order'), end
n = length(tspan);
ti = tspan(1);tf = tspan(n);
if n == 2
  t = (ti:h:tf)'; n = length(t);
  if t(n)<tf
    t(n+1) = tf;
    n = n+1;
  end
else
  t = tspan;
end
tt = ti; y(1,:) = y0;
np = 1; tp(np) = tt; yp(np,:) = y(1,:);
i=1;
while(1)
  tend = t(np+1);
  hh = t(np+1) - t(np);
  if hh>h,hh = h;end
  while(1)
    if tt+hh>tend,hh = tend-tt;end
    k1 = dydt(tt,y(i,:),varargin{:})';
    ymid = y(i,:) + k1.*hh./2;
    k2 = dydt(tt+hh/2,ymid,varargin{:})';
    ymid = y(i,:) + k2*hh/2;
    k3 = dydt(tt+hh/2,ymid,varargin{:})';
    yend = y(i,:) + k3*hh;
    k4 = dydt(tt+hh,yend,varargin{:})';
    phi = (k1+2*(k2+k3)+k4)/6;
    y(i+1,:) = y(i,:) + phi*hh;
    tt = tt+hh;
    i=i+1;
    if tt>=tend,break,end
  end
  np = np+1; tp(np) = tt; yp(np,:) = y(i,:);
  if tt>=tf,break,end
end
```