

# Numerical Integration of Ordinary Differential Equations for Initial Value Problems

Gerald Recktenwald  
Portland State University  
Department of Mechanical Engineering  
[gerry@me.pdx.edu](mailto:gerry@me.pdx.edu)

These slides are a supplement to the book *Numerical Methods with MATLAB: Implementations and Applications*, by Gerald W. Recktenwald, © 2001, Prentice-Hall, Upper Saddle River, NJ. These slides are © 2001 Gerald W. Recktenwald. The PDF version of these slides may be downloaded or stored or printed only for noncommercial, educational use. The repackaging or sale of these slides in any form, without written consent of the author, is prohibited.

The latest version of this PDF file, along with other supplemental material for the book, can be found at [www.prenhall.com/recktenwald](http://www.prenhall.com/recktenwald).

Version 0.91      November 27, 2001

# Overview

- Motivation: ODE's arise as models of many applications
- Euler's method
  - ▷ A low accuracy prototype for other methods
  - ▷ Development
  - ▷ Implementation
  - ▷ Analysis
- Midpoint method
- Heun's method
- Runge-Kutta method of order 4
- MATLAB's adaptive stepsize routines
- Systems of equations
- Higher order ODEs

# Newton's Law of Motion

$$F = ma$$

Acceleration is the time derivative of velocity, so

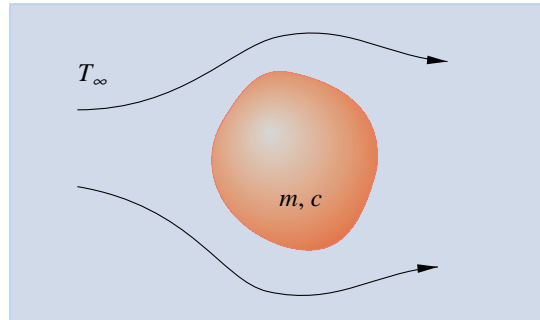
$$\frac{dv}{dt} = a$$

and

$$\frac{dv}{dt} = \frac{F}{m}$$

If  $F(t)$  and  $v(0)$  are known, we can (at least in principle) integrate the preceding equation to find  $v(t)$

# Newton's Law of Cooling



$$Q = hA(T_s - T_\infty)$$

The variation of temperature with time for a submerged object is described by an ODE.

## Newton's Law of Cooling

Apply an energy balance

$$mc \frac{dT}{dt} = -Q = -hA(T_s - T_\infty)$$

Assume material is highly conductive  $\Rightarrow T_s = T$

$$mc \frac{dT}{dt} = -hA(T - T_\infty)$$

or

$$\frac{dT}{dt} = -\frac{hA}{mc}(T - T_\infty)$$

## Analytical Solution

**Example:**

$$\frac{dy}{dt} = -y \quad y(0) = y_0$$

This equation can be integrated directly

$$\frac{dy}{y} = -dt$$

$$\ln y = -t + C$$

$$\ln y - \ln C_2 = -t$$

$$\ln \frac{y}{C_2} = -t$$

$$y = C_2 e^{-t}$$

$$y = y_0 e^{-t}$$

# Numerical Integration of ODEs

Given

$$\frac{dy}{dt} = f(t, y); \quad y(0) = y_0$$

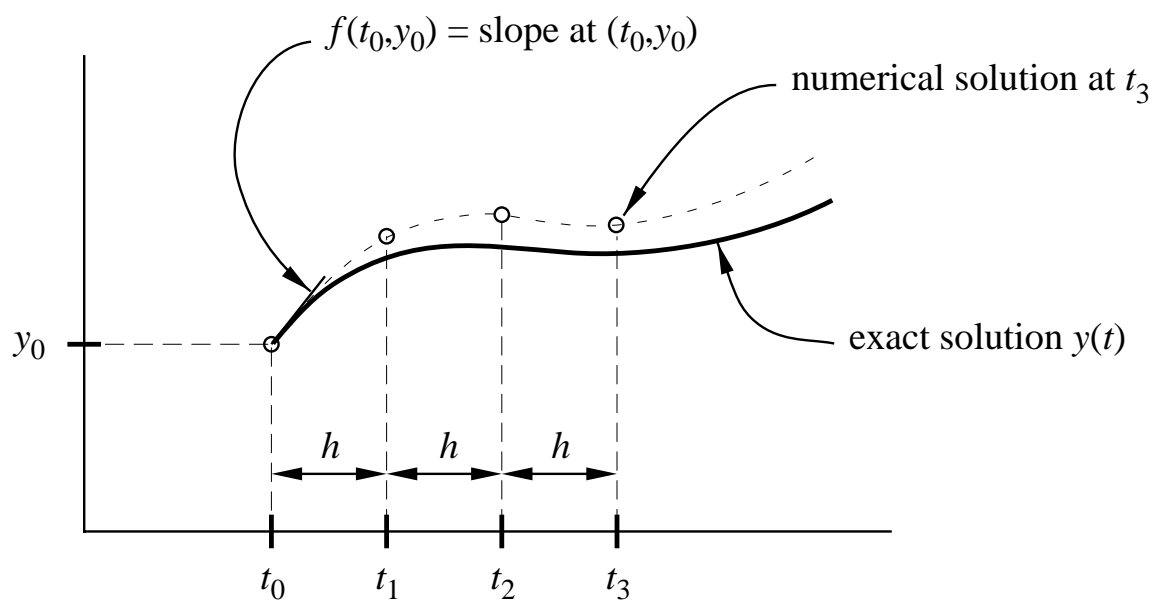
Find approximate solution at discrete values of  $t$

$$t_j = t_0 + jh$$

where  $h$  is the “stepsize”

# Numerical Integration of ODEs

## Graphical Interpretation





# Nomenclature

$y(t)$  = exact solution

$y(t_j)$  = exact solution evaluated at  $t_j$

$y_j$  = approximate solution at  $t_j$

$f(t_j, y_j)$  = approximate r.h.s. at  $t_j$

## Euler's Method (1)

Consider a Taylor series expansion in the neighborhood of  $t_0$

$$y(t) = y(t_0) + (t - t_0) \left. \frac{dy}{dt} \right|_{t_0} + \frac{(t - t_0)^2}{2} \left. \frac{d^2y}{dt^2} \right|_{t_0} + \dots$$

Retain only first derivative term and define

$$f(t_0, y_0) \equiv \left. \frac{dy}{dt} \right|_{t_0}$$

to get

$$y(t) \approx y(t_0) + (t - t_0) f(t_0, y_0)$$

## Euler's Method (2)

Given  $h = t_1 - t_0$  and initial condition,  $y = y(t_0)$ , compute

$$y_1 = y_0 + h f(t_0, y_0)$$

$$y_2 = y_1 + h f(t_1, y_1)$$

$$\vdots \quad \quad \quad \vdots$$

$$y_{j+1} = y_j + h f(t_j, y_j)$$

or

$$y_j = y_{j-1} + h f(t_{j-1}, y_{j-1})$$

## Example: Euler's Method (1)

Use Euler's method to integrate

$$\frac{dy}{dt} = t - 2y \quad y(0) = 1$$

The exact solution is

$$y = \frac{1}{4} \left[ 2t - 1 + 5e^{-2t} \right]$$

## Example: Euler's Method (2)

$j$	$t_j$	$f(t_{j-1}, y_{j-1})$	Euler $y_j = y_{j-1} + h f(t_{j-1}, y_{j-1})$	Exact $y(t_j)$	Error $y_j - y(t_j)$
0	0.0	NA	(initial condition) 1.0000	1.0000	0
1	0.2	$0 - (2)(1) = -2.000$	$1.0 + (0.2)(-2.0) = 0.6000$	0.6879	-0.0879
2	0.4	$0.2 - (2)(0.6) = -1.000$	$0.6 + (0.2)(-1.0) = 0.4000$	0.5117	-0.1117
3	0.6	$0.4 - (2)(0.4) = -0.400$	$0.4 + (0.2)(-0.4) = 0.3200$	0.4265	-0.1065

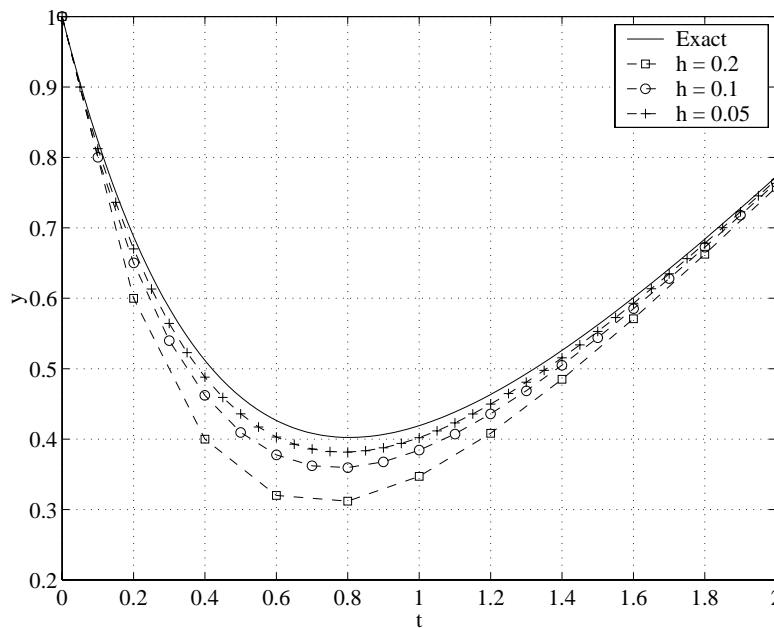
## Demonstration of GDE (1)

Reducing stepsize improves accuracy.

Use Euler's method to integrate

$$\frac{dy}{dt} = t - 2y \quad y(0) = 1$$

for a sequence of smaller  $h$  (see demoEuler)



For a given  $h$ , the largest error in the numerical solution is the *Global Discretization Error* or *GDE*.

## Demonstration of GDE (2)

Local error at any time step is

$$e_j = y_j - y(t_j)$$

where  $y(t_j)$  is the exact solution evaluated at  $t_j$ .

$$\text{GDE} = \max(e_j), \quad j = 1, \dots$$

For Euler's method, GDE decreases linearly with  $h$ .

Here are results for the sample problem plotted on previous slide:  
 $dy/dt = t - 2y$ ,  $y(0) = 1$

$h$	$\max(e_j)$
0.200	0.1117
0.100	0.0502
0.050	0.0240
0.025	0.0117

# Implementation of Euler's Method

A general implementation of Euler's method separates the evaluation of  $f$  (the right hand side function) from the basic algorithm of advance the ODE.

```
function [t,y] = odeEuler(diffeq,tn,h,y0)
% odeEuler Euler's method for integration of a single, first order ODE
%
% Synopsis:    [t,y] = odeEuler(diffeq,tn,h,y0)
%
% Input:       diffeq = (string) name of the m-file that evaluates the right
%               hand side of the ODE written in standard form
%               tn    = stopping value of the independent variable
%               h     = stepsize for advancing the independent variable
%               y0    = initial condition for the dependent variable
%
% Output:      t = vector of independent variable values:  t(j) = (j-1)*h
%               y = vector of numerical solution values at the t(j)

t = (0:h:tn)';           % Column vector of elements with spacing h
n = length(t);           % Number of elements in the t vector
y = y0*ones(n,1);        % Preallocate y for speed

% Begin Euler scheme; j=1 for initial condition
for j=2:n
    y(j) = y(j-1) + h*feval(diffeq,t(j-1),y(j-1));
end
```



## Analysis of Euler's Method (1)

Rewrite the discrete form of Euler's method as

$$\frac{y_j - y_{j-1}}{h} = f(t_{j-1}, y_{j-1}) \quad \text{discrete}$$

Compare with original ODE

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

If the exact solution is substituted back into the discrete approximation to the ODE we get

$$\frac{y(t_j) - y(t_{j-1})}{h} - f(t_{j-1}, y(t_{j-1})) \neq 0$$

## Analysis of Euler's Method (2)

Introduce a family of functions  $z_j(t)$ , which are the exact solutions to the ODE given the approximate solution produced by Euler's method at step  $j$ .

$$\frac{dz_j}{dt} = f(t, y) \quad z_j(t_{j-1}) = y_{j-1}$$

Due to truncation error, Euler's method produces a value of  $y_{j+1}$  that is different from  $z_j(t_{j+1})$  even though by design,  $y_j = z(t_j)$ . In other words

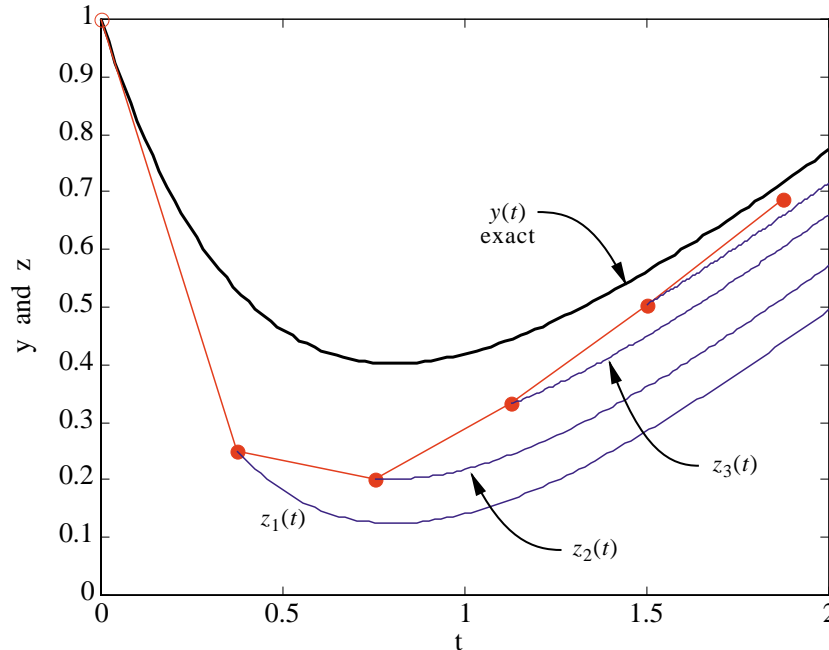
$$y_{j+1} - z(t_{j+1}) \neq 0 \quad \text{due to truncation error}$$

## Analysis of Euler's Method (3)

### Example Solve

$$\frac{dy}{dt} = t - 2y \quad y(0) = 1$$

The plot shows the numerical solution (●) obtained with  $h = 0.5$ . The  $z(t)$  curve starting at each of the numerical solution points is shown as a solid line.



## Analysis of Euler's Method (4)

The **local discretization error** (LDE) is the residual obtained when the exact  $z_j(t)$  is substituted into the discrete approximation to the ODE

$$\tau(t, h) = \frac{z(t_j) - z(t_{j-1})}{h} - f(t_{j-1}, z_j(t_{j-1}))$$

- Note:**
- User chooses  $h$ , and this affects LDE
  - LDE also depends on  $t$ , the position in the interval

## Analysis of Euler's Method (5)

Using a Taylor series expansion for  $z_j(x)$  we find that

$$\frac{z(t_j) - z(t_{j-1})}{h} - f(t_{j-1}, z_j(t_{j-1})) = \frac{h}{2} z_j''(\xi)$$

where  $t_{j-1} \leq \xi \leq t_j$  and  $z_j'' \equiv d^2 z_j / dt^2$ .

Thus, for Euler's method the local discretization error is

$$\tau(t, h) = \frac{h}{2} z_j''(\xi)$$

Since  $\xi$  is not known, the value of  $z_j''(\xi)$  cannot be computed.

## Analysis of Euler's Method (6)

Assume that  $z_j''(\xi)$  is bounded by  $M$  in the interval  $t_0 \leq t \leq t_N$ . Then

$$\tau(t, h) \leq \frac{hM}{2} \quad \text{LDE for Euler's method}$$

Although  $M$  is unknown we can still compute the effect of reducing the stepsize by taking the ratio of  $\tau(t, h)$  for two different choices of  $h$

$$\boxed{\frac{\tau(t, h_2)}{\tau(t, h_1)} = \frac{h_2}{h_1}}$$

## Global Discretization Error for Euler's Method

General application of Euler's method requires several steps to compute the solution to the ODE in an interval,  $t_0 \leq t \leq t_N$ . The local truncation error at each step accumulates. The result is a **global discretization error** (GDE)

The global discretization error for Euler's method is

$$\text{GDE} = \mathcal{O}(h)$$

Thus

$$\boxed{\frac{\text{GDE}(h_1)}{\text{GDE}(h_2)} = \frac{h_1}{h_2}}$$

## Summary of Euler's Method

Development of Euler's method has demonstrated the following *general* ideas

- The numerical integration scheme is derived from a truncated Taylor series approximation of the ODE.
- The local discretization error (LDE) accounts for the error at each time step.

$$LDE = \mathcal{O}(h^p)$$

where  $h$  is the stepsize and  $p$  is an integer  $p \geq 1$ .

- The global discretization error (GDE) includes the accumulated effect of the LDE when the ODE integration scheme is applied to an interval using several steps of size  $h$ .

$$GDE = \mathcal{O}(h^p)$$

- The implementation separates the logic of the ODE integration scheme from the evaluation of the right hand side,  $f(t, y)$ . A general purpose ODE solver requires the user to supply a small m-file for evaluating  $f(t, y)$ .



## Higher Order Methods

We now commence a survey of one-step methods that are more accurate than Euler's method.

- Not all methods are represented here
- Objective is a logical progression leading to RK-4
- Sequence is in order of increasing accuracy *and* increasing computational efficiency

Methods with increasing accuracy, lower GDE

Method	GDE
Euler	$\mathcal{O}(h)$
Midpoint	$\mathcal{O}(h^2)$
Heun's	$\mathcal{O}(h^2)$
RK-4	$\mathcal{O}(h^4)$

Note that since  $h < 1$ , a GDE of  $\mathcal{O}(h^4)$  is much smaller than a GDE of  $\mathcal{O}(h)$ .

## Midpoint Method (1)

Increase accuracy by evaluating slope twice in each step of size  $h$

$$k_1 = f(t_j, y_j)$$

Compute a tentative value of  $y$  at the midpoint

$$y_{j+1/2} = y_j + \frac{h}{2}f(t_j, y_j)$$

re-evaluate the slope

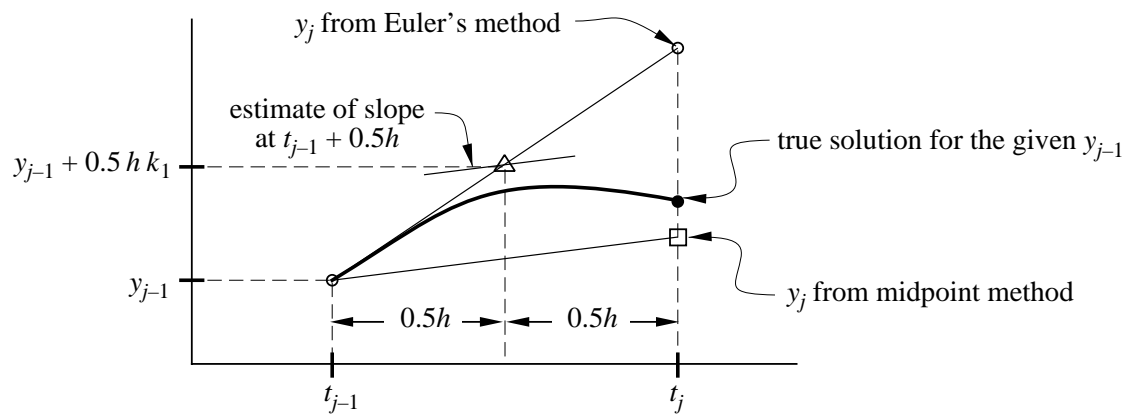
$$k_2 = f(t_j + \frac{h}{2}, y_j + \frac{h}{2}k_1)$$

Compute final value of  $y$  at the end of the full interval

$$y_{j+1} = y_j + hk_2$$

$$\text{LDE} = \text{GDE} = \mathcal{O}(h^2)$$

## Midpoint Method (2)



## Midpoint Method (3)

### Implementation in odeMidpt.m

```
function [t,y] = odeMidpt(diffeq,tn,h,y0)
% odeMidpt   Midpoint method for integration of a single, first order ODE
%
% Synopsis:   [t,y] = odeMidpt(diffeq,tn,h,y0)
%
% Input:      diffeq = (string) name of the m-file that evaluates the right
%              hand side of the ODE written in standard form
%              tn    = stopping value of the independent variable
%              h     = stepsize for advancing the independent variable
%              y0    = initial condition for the dependent variable
%
% Output:     t = vector of independent variable values:  t(j) = (j-1)*h
%              y = vector of numerical solution values at the t(j)

t = (0:h:tn)';           % Column vector of elements with spacing h
n = length(t);           % Number of elements in the t vector
y = y0*ones(n,1);        % Preallocate y for speed
h2 = h/2;                % Avoid repeated evaluation of this constant

% Begin Midpoint scheme; j=1 for initial condition
for j=2:n
    k1 = feval(diffeq,t(j-1),y(j-1));
    k2 = feval(diffeq,t(j-1)+h2,y(j-1)+h2*k1);
    y(j) = y(j-1) + h*k2;
end
```

## Midpoint Method (4)

Midpoint method requires twice as much work per time step.  
Does the extra effort pay off?

Consider integration with Euler's method and  $h = 0.1$ . Formal accuracy is  $\mathcal{O}(0.1)$ .

Repeat calculations with the midpoint method and  $h = 0.1$ . Formal accuracy is  $\mathcal{O}(0.01)$ .

For Euler's method to obtain the same accuracy, the stepsize would have to be reduced by a factor of 10. The midpoint method, therefore, achieves the same (formal) accuracy with one fifth the work!

## Comparison of Midpoint Method with Euler's Method

Solve

$$\frac{dy}{dt} = -y \quad y(0) = 1 \quad 0 \leq t \leq 1$$

The exact solution is  $y = e^{-t}$ .

```
>> compEM
```

h	flopsE	errE	flopsM	errM
0.20000	31	4.02e-02	57	2.86e-03
0.10000	61	1.92e-02	112	6.62e-04
0.05000	121	9.39e-03	222	1.59e-04
0.02500	241	4.65e-03	442	3.90e-05
0.01250	481	2.31e-03	882	9.67e-06
0.00625	961	1.15e-03	1762	2.41e-06

For comparable accuracy:

- Midpoint method with  $h = 0.2$  needs 57 flops, and gives max error of  $2.86 \times 10^{-3}$
- Euler's method with  $h = 0.0125$  needs 481 flops, and gives max error of  $2.31 \times 10^{-3}$

## Heun's Method (1)

Compute the slope at the starting point

$$k_1 = f(t_j, y_j)$$

Compute a tentative value of  $y$  at the endpoint

$$y_j^* = y_j + hf(t_j, y_j)$$

re-evaluate the slope

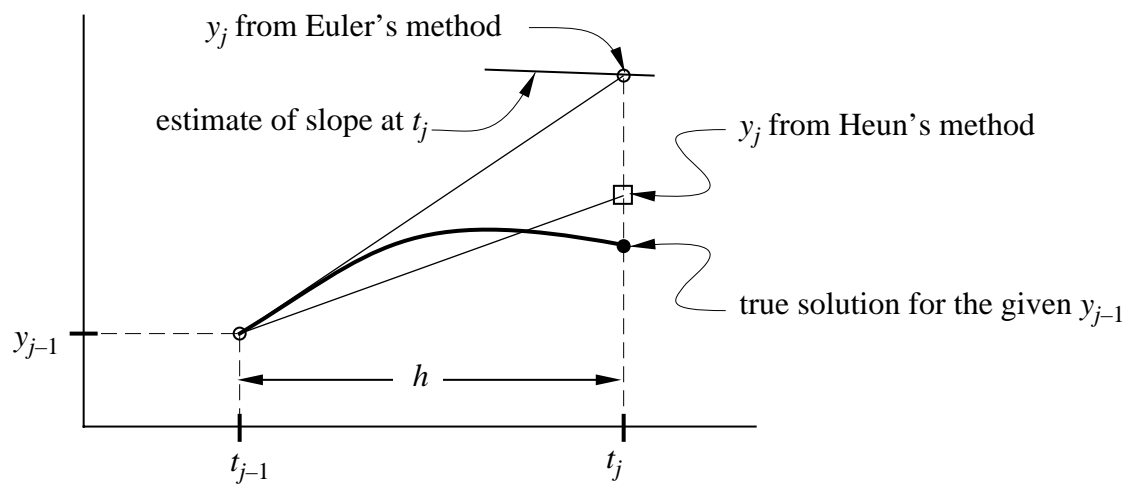
$$k_2 = f(t_j + h, y_j^*) = f(t_j + h, y_j + hk_1)$$

Compute final value of  $y$  with an average of the two slopes

$$y_{j+1} = y_j + h \frac{k_1 + k_2}{2}$$

$$\text{LDE} = \text{GDE} = \mathcal{O}(h^2)$$

## Heun's Method (2)





## Summary So Far

- Euler's method evaluates slope at beginning of the step
- Midpoint method evaluates slope at beginning and at midpoint of the step
- Heun's method evaluates slope at beginning and at end of step

Can we continue to get more accurate schemes by evaluating the slope at more points in the interval? Yes, but there is a limit beyond which additional evaluations of the slope increase in cost (increased flops) faster than they improve the accuracy.

## Runge-Kutta Methods

Generalize the idea embodied in Heun's method. Use a *weighted average of the slope* evaluated at multiple in the step

$$y_{j+1} = y_j + h \sum \gamma_m k_m$$

where  $\gamma_m$  are weighting coefficients and  $k_m$  are slopes evaluated at points in the interval  $t_j \leq t \leq t_{j+1}$

In general,

$$\sum \gamma_m = 1$$

## Fourth Order Runge-Kutta

Compute slope at four places within each step

$$k_1 = f(t_j, y_j)$$

$$k_2 = f\left(t_j + \frac{h}{2}, y_j + \frac{h}{2}k_1\right)$$

$$k_3 = f\left(t_j + \frac{h}{2}, y_j + \frac{h}{2}k_2\right)$$

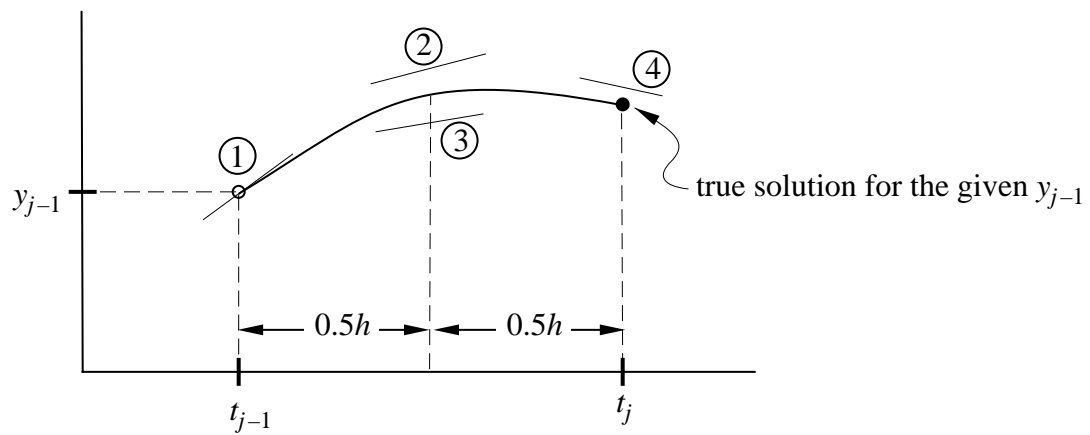
$$k_4 = f(t_j + h, y_j + hk_3)$$

Use weighted average of slopes to obtain  $y_{j+1}$

$$y_{j+1} = y_j + h \left( \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \right)$$

$$\text{LDE} = \text{GDE} = \mathcal{O}(h^4)$$

## Fourth Order Runge-Kutta



# Implementation of RK4

```
function [t,y] = odeRK4(diffeq,tn,h,y0)
% odeRK4 Fourth order Runge-Kutta method for a single, first order ODE
%
% Synopsis: [t,y] = odeRK4(fun,tn,h,y0)
%
% Input:      diffeq = (string) name of the m-file that evaluates the right
%              hand side of the ODE written in standard form
%              tn   = stopping value of the independent variable
%              h    = stepsize for advancing the independent variable
%              y0   = initial condition for the dependent variable
%
% Output:     t = vector of independent variable values: t(j) = (j-1)*h
%              y = vector of numerical solution values at the t(j)

t = (0:h:tn)'; % Column vector of elements with spacing h
n = length(t); % Number of elements in the t vector
y = y0*ones(n,1); % Preallocate y for speed
h2 = h/2; h3 = h/3; h6 = h/6; % Avoid repeated evaluation of constants

% Begin RK4 integration; j=1 for initial condition
for j=2:n
    k1 = feval(diffeq, t(j-1), y(j-1) );
    k2 = feval(diffeq, t(j-1)+h2, y(j-1)+h2*k1 );
    k3 = feval(diffeq, t(j-1)+h2, y(j-1)+h2*k2 );
    k4 = feval(diffeq, t(j-1)+h, y(j-1)+h*k3 );
    y(j) = y(j-1) + h6*(k1+k4) + h3*(k2+k3);
end
```

## Comparison of Euler, Midpoint and RK4

Solve

$$\frac{dy}{dt} = -y \quad y(0) = 1 \quad 0 \leq t \leq 1$$

```
>> compEMRK4
```

h	flopsE	errE	flopsM	errM	flops4	err4
0.20000	31	4.02e-02	57	2.86e-03	129	5.80e-06
0.10000	61	1.92e-02	112	6.62e-04	254	3.33e-07
0.05000	121	9.39e-03	222	1.59e-04	504	2.00e-08
0.02500	241	4.65e-03	442	3.90e-05	1004	1.22e-09
0.01250	481	2.31e-03	882	9.67e-06	2004	7.56e-11
0.00625	961	1.15e-03	1762	2.41e-06	4004	4.70e-12

## Summary Comparison

	Error	step size	flops
Euler	$1.2 \times 10^{-3}$	0.00625	961
Midpoint	$6.6 \times 10^{-4}$	0.1	112
Midpoint	$2.4 \times 10^{-6}$	0.00625	1762
RK-4	$3.3 \times 10^{-7}$	0.1	254

### Conclusion:

- RK-4 is much more accurate (smaller GDE) than Midpoint or Euler
- Although RK-4 takes more flops per step, it can achieve comparable accuracy with much larger time steps. The net effect is that **RK-4 is more accurate and more efficient**

## Summary: Accuracy of ODE Integration Schemes

- GDE decreases as  $h$  decreases
- Need an upper limit on  $h$  to achieve a desired accuracy
- Example: Euler's Method

$$y_j = y_{j-1} + h f(t_{j-1}, y_{j-1})$$

when  $h$  and  $|f(t_j, y_j)|$  are large, the change in  $y$  is large

- The product,  $h f(t_j, y_j)$ , determines accuracy



## Procedure for Using Fixed Stepsize Algorithms

- Develop an m-file to evaluate the right hand side
- Use a high order method, e.g. RK-4
- Compare solutions for a sequence of smaller  $h$
- When the change in the solution between successively smaller  $h$  is “small enough”, accept that as the  *$h$ -independent* solution.

The goal is to obtain a solution that does not depend (in a significant way) on  $h$ .

## Adaptive Stepsize Algorithms

Let the solution algorithm determine  $h$  *at each time step*

- Set a tolerance on the error
- When  $|f(t_{j-1}, y_{j-1})|$  is decreases, increase  $h$  to increase efficiency and decrease round-off
- When  $|f(t_{j-1}, y_{j-1})|$  is increases, decrease  $h$  to maintain accuracy

## Adaptive Stepsize Algorithms (2)

How do we find the “error” at each step in order to judge whether the stepsize needs to be reduced or increased?

Two related strategies:

- Use two  $h$  values at each step:
  1. Advance the solution with  $h = h_1$
  2. Advance the solution with two steps of size  $h_2 = h/2$
  3. If solutions are close enough, accept the  $h_1$  solution, stop
  4. Otherwise, replace  $h_1 = h_2$ , go back to step 2
- Use *embedded* Runge-Kutta methods

## Embedded Runge-Kutta Methods

There is a **pair of RK methods** that use the **same six  $k$  values**

**Fourth Order RK:**

$$y_{j+1} = y_j + c_1 k_1 + c_2 k_2 + c_3 k_3 \\ + c_4 k_4 + c_5 k_5 + c_6 k_6 + \mathcal{O}(h^4)$$

**Fifth Order RK:**

$$y_{j+1}^* = y_j + c_1^* k_1 + c_2^* k_2 + c_3^* k_3 \\ + c_4^* k_4 + c_5^* k_5 + c_6^* k_6 + \mathcal{O}(h^5)$$

Therefore, at each step an estimate of the truncation error is

$$\Delta = y_{j+1} - y_{j+1}^*$$

Possible outcomes

- If  $\Delta$  is smaller than tolerance, accept the  $y_{j+1}$  solution.
- If  $\Delta$  is *much* smaller than tolerance, accept the  $y_{j+1}$  solution, *and* try increasing the stepsize.
- If  $\Delta$  is larger than tolerance, reduce  $h$  and try again.

## MATLAB ode45 **Function**

- User supplies error tolerance, *not* stepsize
- Simultaneously compute  $4^{th}$  and  $5^{th}$  order Runge-Kutta solution
- Compare two solutions to determine accuracy
- Adjust step-size so that error tolerance is maintained

## MATLAB ode45 **Function (2)**

Error tolerances are

$$\tilde{\tau} < \max(\text{RelTol} \times |y_j|, \text{AbsTol})$$

where  $\tilde{\tau}$  is an estimate of the local truncation error, and RelTol and AbsTol are the error tolerances, which have the default values of

$$\text{RelTol} = 1 \times 10^{-3} \qquad \text{AbsTol} = 1 \times 10^{-6}$$

## Using ode45

### Syntax:

```
[t,Y] = ode45(diffeq,tn,y0)  
[t,Y] = ode45(diffeq,[t0 tn],y0)  
[t,Y] = ode45(diffeq,[t0 tn],y0,options)  
[t,Y] = ode45(diffeq,[t0 tn],y0,options,arg1,arg2,...)
```

User writes the *diffeq* m-file to evaluate the right hand side of the ODE.

Solution is controlled with the *odeset* function

```
options = odeset('parameterName',value,...)  
[y,t] = ode45('rhsFun',[t0 tN],y0,options,...)
```

Syntax for *ode23* and other solvers is the same.

## MATLAB's Built-in ODE Routines

Function	Description
ode113	Variable order solution to nonstiff systems of ODEs. ode113 uses an explicit predictor-corrector method with variable order from 1 to 13.
ode15s	Variable order, multistep method for solution to stiff systems of ODEs. ode15s uses an implicit multistep method with variable order from 1 to 5.
ode23	Lower order adaptive stepsize routine for non-stiff systems of ODEs. ode23 uses Runge-Kutta schemes of order 2 and 3.
ode23s	Lower order adaptive stepsize routine for moderately stiff systems of ODEs. ode23s uses Runge-Kutta schemes of order 2 and 3.
ode45	Higher order adaptive stepsize routine for non-stiff systems of ODEs. ode45 uses Runge-Kutta schemes of order 4 and 5.



## Interpolation Refinement ode45 (1)

The ode45 function attempts to obtain the solution to within the user-specified error tolerances. In some situations the solution can be obtained within the tolerance by taking so few time steps that the solution appears to be unsmooth. To compensate for this, ode45 automatically interpolates the solution between points that are obtained from the solver.

Consider

$$\frac{dy}{dt} = \cos(t), \quad y(0) = 0$$

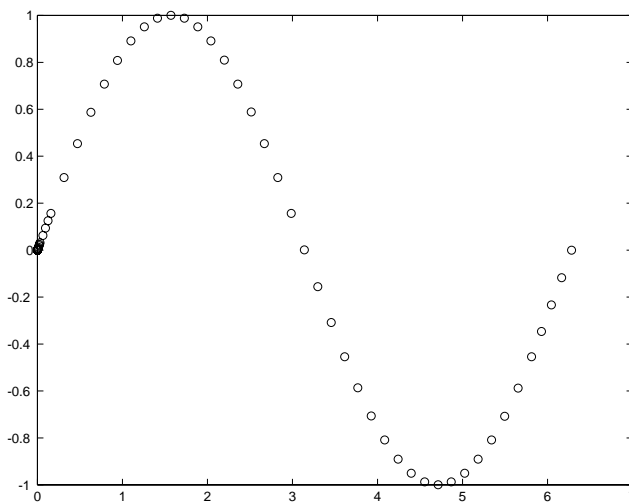
The following statements obtain the solution with the default parameters.

```
>> rhs = inline('cos(t)','t','y');  
>> [t,Y] = ode45(rhs,[0 2*pi],0);  
>> plot(t,Y,'o')
```

(See plot on next slide)

## Interpolation Refinement ode45 (2)

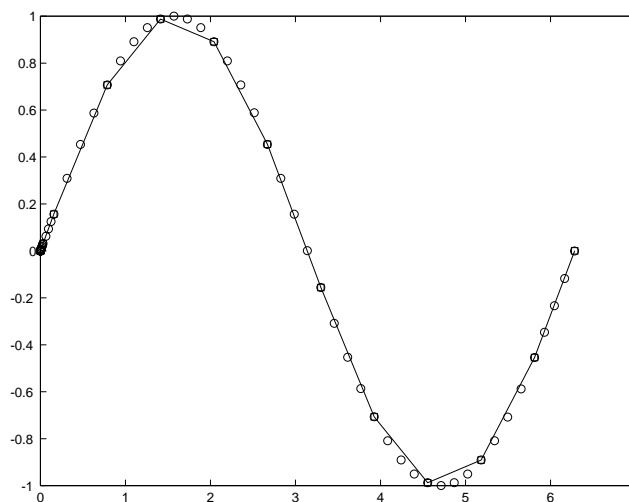
```
>> rhs = inline('cos(t)','t','y');  
>> [t,Y] = ode45(rhs,[0 2*pi],0);  
>> plot(t,Y,'o')
```



## Interpolation Refinement ode45 (3)

Repeat solution without interpolation:

```
>> options = odeset('Refine',1)
>> [t2,Y2] = ode45(rhs,[0 2*pi],0,options);
>> hold on
>> plot(t2,Y2,'rs-')
```



The two solutions are identical at the points obtained from the Runge-Kutta algorithm

```
>> max(Y2 - Y(1:4:end))
ans =
    0
```

## Coupled ODEs

Consider

$$\frac{dy_1}{dt} = f_1(t, y_1, y_2)$$

$$\frac{dy_2}{dt} = f_2(t, y_1, y_2)$$

These equations must be advanced simultaneously.

## Coupled ODEs (2)

Apply the 4<sup>th</sup> Order Runge-Kutta Scheme:

$$k_{1,1} = f_1(t_j, y_{j,1}, y_{j,2})$$

$$k_{1,2} = f_2(t_j, y_{j,1}, y_{j,2})$$

$$k_{2,1} = f_1\left(t_j + \frac{h}{2}, y_{j,1} + \frac{h}{2}k_{1,1}, y_{j,2} + \frac{h}{2}k_{1,2}\right)$$

$$k_{2,2} = f_2\left(t_j + \frac{h}{2}, y_{j,1} + \frac{h}{2}k_{1,1}, y_{j,2} + \frac{h}{2}k_{1,2}\right)$$

$$k_{3,1} = f_1\left(t_j + \frac{h}{2}, y_{j,1} + \frac{h}{2}k_{2,1}, y_{j,2} + \frac{h}{2}k_{2,2}\right)$$

$$k_{3,2} = f_2\left(t_j + \frac{h}{2}, y_{j,1} + \frac{h}{2}k_{2,1}, y_{j,2} + \frac{h}{2}k_{2,2}\right)$$

$$k_{4,1} = f_1(t_j + h, y_{j,1} + hk_{3,1}, y_{j,2} + hk_{3,2})$$

$$k_{4,2} = f_2(t_j + h, y_{j,1} + hk_{3,1}, y_{j,2} + hk_{3,2})$$

## Coupled ODEs (3)

Update  $y_1$  and  $y_2$  only after all slopes are computed

$$y_{j+1,1} = y_{j,1} + h \left( \frac{k_{1,1}}{6} + \frac{k_{2,1}}{3} + \frac{k_{3,1}}{3} + \frac{k_{4,1}}{6} \right)$$

$$y_{j+1,2} = y_{j,2} + h \left( \frac{k_{1,2}}{6} + \frac{k_{2,2}}{3} + \frac{k_{3,2}}{3} + \frac{k_{4,2}}{6} \right)$$

## Example: Predator-Prey Equations

$$\frac{dp_1}{dt} = \alpha_1 p_1 - \delta_1 p_1 p_2 \quad \text{prey}$$

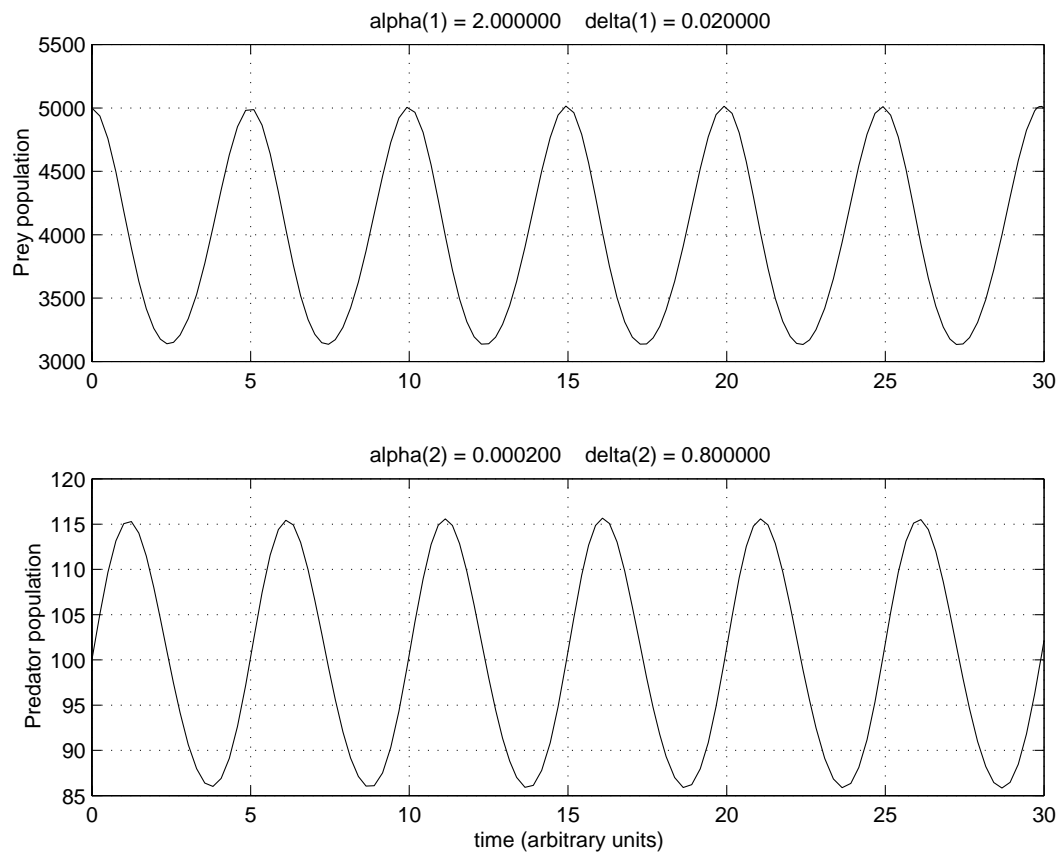
$$\frac{dp_2}{dt} = \alpha_2 p_1 p_2 - \delta_2 p_2 \quad \text{preditor}$$

## Evaluate RHS of Predator-Prey Model

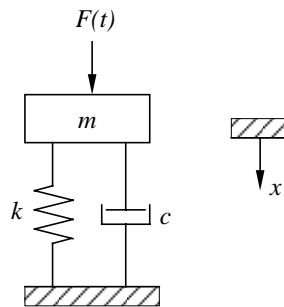
```
function dpdt = rhsPop2(t,p,flag,alpha,delta)
% rhsPop2 Right hand sides of coupled ODEs for 2 species predator-prey system
%
% Synopsis: dpdt = rhsPop2(t,p,flag,alpha,delta)
%
% Input:    t = time. Not used in this m-file, but needed by ode45
%           p = vector (length 2) of populations of species 1 and 2
%           flag = (not used) placeholder for compatibility with ode45
%           alpha = vector (length 2) of growth coefficients
%           delta = vector (length 2) of mortality coefficients
%
% Output:   dpdt = vector of dp/dt values
dpdt = [ alpha(1)*p(1)      - delta(1)*p(1)*p(2);
        alpha(2)*p(1)*p(2) - delta(2)*p(2);      ];
```



# Predator-Prey Results



## Example: Second Order Mechanical System



$$\sum F = ma$$

Forces acting on the mass are

$$F_{spring} = -kx$$

$$F_{damper} = -c\dot{x}$$

$$F(t) - kx - c\dot{x} = m\ddot{x}$$

## Second Order Mechanical System

Governing equation is a second order ODE

$$\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2x = \frac{F}{m}$$

$$\zeta \equiv \frac{c}{2\sqrt{km}}$$

$$\omega_n \equiv \sqrt{k/m}$$

$\zeta$  and  $\omega_n$  are the only (dimensionless) parameters

## A Second Order System is Equivalent to Two, Coupled, First Order Systems

Define

$$y_1 \equiv x \quad y_2 \equiv \dot{x}$$

then

$$\frac{dy_1}{dt} = \dot{x} = y_2$$

$$\frac{dy_2}{dt} = \ddot{x}$$

$$= \frac{F}{m} - 2\zeta\omega_n\dot{x} - \omega_n^2x$$

$$= \frac{F}{m} - 2\zeta\omega_n y_2 - \omega_n^2 y_1$$

## Solve Second Order System with ODE45

```
function demoSmd(zeta,omegan,tstop)
% demoSmd    Second order system of ODEs for a spring-mass-damper system
%
% Synopsis:   smdsys(zeta,omegan,tstop)
%
% Input:      zeta    = (optional) damping ratio; Default: zeta = 0.1
%             omegan  = (optional) natural frequency; Default: omegan = 35
%             tstop   = (optional) stopping time; Default: tstop = 1.5
%
% Output:     plot of displacement and velocity versus time

if nargin<1, zeta = 0.1; end
if nargin<2, omegan = 35; end
if nargin<3, tstop = 1.5; end

y0 = [0; 0]; a0 = 9.8; % Initial conditions and one g force/mass
[t,y] = ode45('rhssmd',tstop,y0,[],zeta,omegan,a0);

subplot(2,1,1);
plot(t,y(:,1)); ylabel('Displacement'); grid;
title(sprintf('zeta = %5.3f    omegan = %5.1f',zeta,omegan));
subplot(2,1,2);
plot(t,y(:,2)); xlabel('Time (s)'); ylabel('Velocity'); grid;
```

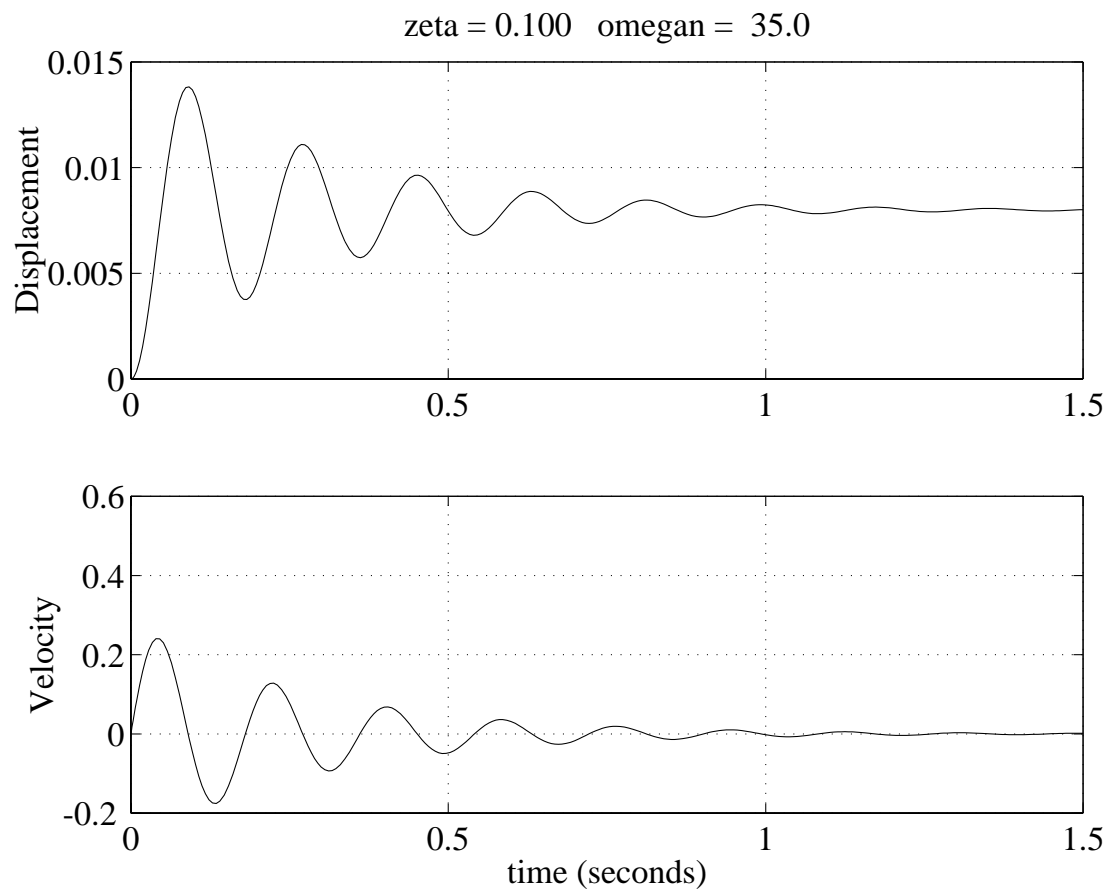
## Solve Second Order System with ODE45

```
function dydt = rhsSmd(t,y,flag,zeta,omegan,a0)
% rhsSmd Right-hand sides of coupled ODEs for a spring-mass-damper system
%
% Synopsis: dydt = rhsSmd(t,y,flag,zeta,omegan,a0)
%
% Input:    t      = time, the independent variable
%           y      = vector (length 2) of dependent variables
%                y(1) = displacement and y(2) = velocity
%           flag   = dummy argument for compatibility with ode45
%           zeta   = damping ratio (dimensionless)
%           omegan = natural frequency (rad/s)
%           a0     = input force per unit mass
%
% Output:    dydt = column vector of dy(i)/dt values

if t<=0, fonm = 0.0;
else,    fonm = a0;    % Force/mass (acceleration)
end

dydt = [ y(2);  fonm - 2*zeta*omegan*y(2) - omegan*omegan*y(1)];
```

## Response of Second Order System to a Step Input



## General Procedure for Higher Order ODEs

Given

$$\frac{d^n u}{dt^n} = f(t, u)$$

the transformation is

Define $y_i$	ODE for $y_i$
$y_1 = u$	$\frac{dy_1}{dt} = y_2$
$y_2 = \frac{du}{dt}$	$\frac{dy_2}{dt} = y_3$
$y_3 = \frac{d^2 u}{dt^2}$	$\frac{dy_3}{dt} = y_4$
$\vdots$	$\vdots$
$y_n = \frac{d^{n-1} u}{dt^{n-1}}$	$\frac{dy_n}{dt} = f(t, u)$