SECTION 6: ORDINARY DIFFERENTIAL EQUATIONS

ESC 440 – Computational Methods for Engineers

Introduction

Ordinary Differential Equations

- Differential equations can be categorized as either ***ordinary*** or ***partial*** differential equations
  - ***Ordinary*** differential equations (ODEs) – functions of a single independent variable
  - ***Partial*** differential equations (PDEs) – functions of two or more independent variables
- We'll focus on ***ordinary differential equations*** only
- Note that we are not making any assumption of linearity here
  - All techniques we'll look at apply equally to ***linear or nonlinear ODEs***

Differential Equation Order

- The ***order*** of a differential equation is the highest derivative it contains
  - First-order ODEs contain only first derivatives
  - Second-order ODEs include second derivatives (possibly first, as well), and so on …
- ***Any $\boldsymbol{n^{th}}$- order ODE can be reduced to a system of $\boldsymbol{n}$ first-order ODEs***
  - Solution requires knowledge of n initial or boundary conditions
- We'll focus on techniques to solve first-order ODEs
  - Can be applied to systems of first-order ODEs representing higher-order ODEs

Initial-Value vs. Boundary-Value Problems

- To solve an n^{th}-order ODE (or a system of n first-order ODEs), n known conditions are required
  - ***Initial-value problems*** – all n conditions are specified at the same value of the independent variable (typically, at $x = 0$ or $t = 0$)
  - ***Boundary-value problems*** – n conditions specified at different values of the independent variable
- In this course, we'll focus exclusively on ***initial-value problems***

Solving ODEs – General Aproach

- Have an ODE that is some function of the independent and dependent variables:

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

- Numerical solutions amounts to approximating $y(t)$
- Starting at some known initial condition, $y(0)$, propagate the solution forward in time:

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

or

$$(next\ y\ value) = (current\ y\ value) + (slope) \times (step\ size)$$

- ϕ is called the ***increment function***
  - Represents a slope, though not necessarily the slope at (t_i, y_i)
- h is the ***time step***: $h = t_{i+1} - t_i$

One-Step vs. Multi-Step Methods

- ***One-step methods***
  - Use only information at ***current value*** of $y(t)$ (i.e. $y(t_i)$, or y_i) to determine the increment function, ϕ, to be used to propagate the solution forward to y_{i+1}
  - Collectively known as ***Runge-Kutta methods***
  - We'll focus on these exclusively in this course
- ***Multi-step methods***
  - Use both ***current and past values*** of $y(t)$ to provide information about the trajectory of $y(t)$
  - Improved accuracy

Euler's Method

We'll first look at three specific Runge-Kutta algorithms, before returning to a development of the Runge-Kutta approach from a more general perspective.

Euler's Method

- Given an ODE of the form

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

approximate the solution, $y(t)$, using the formula

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

where the increment function is the current derivative

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

- That is, assume the slope of $y(t)$ is constant for $t_i \leq t \leq t_{i+1}$
  - Use the slope at (t_i, y_i) to extrapolate to y_{i+1}

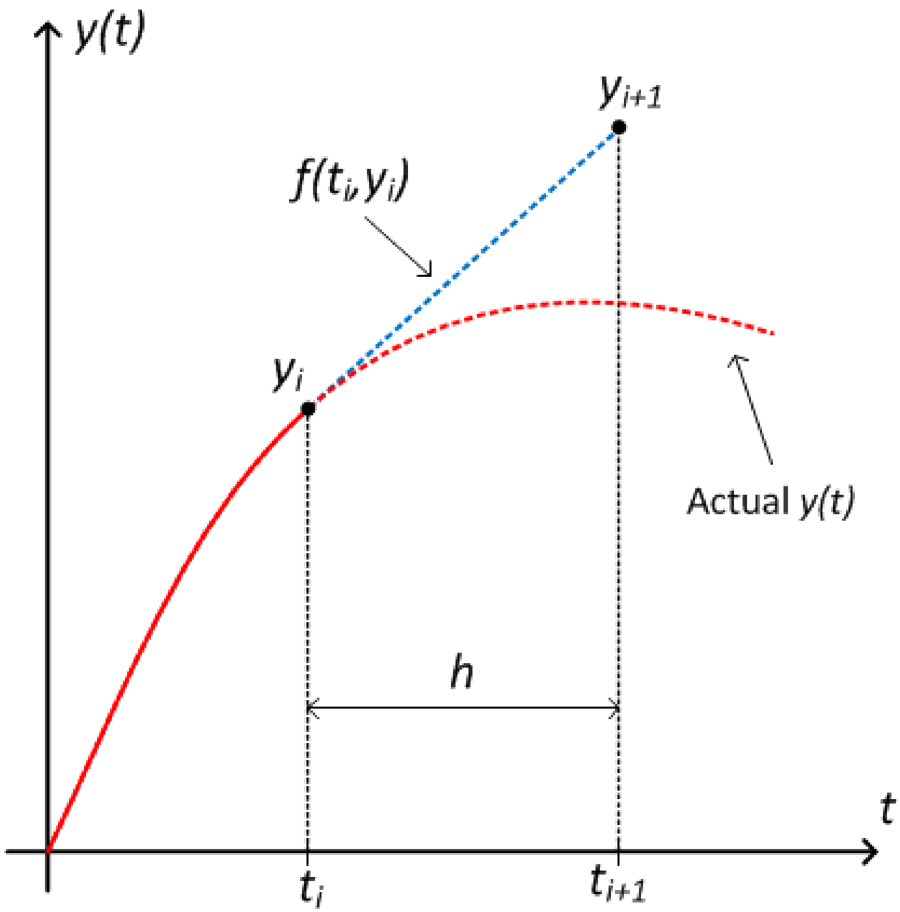
Euler's Method

- Euler's method formula:

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

- Increment function is the current slope:

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



Euler's Method - Example

- Use Euler's method to solve

$$\frac{dy}{dt} = 5e^{-0.5t} - 0.5y$$

given an initial condition of

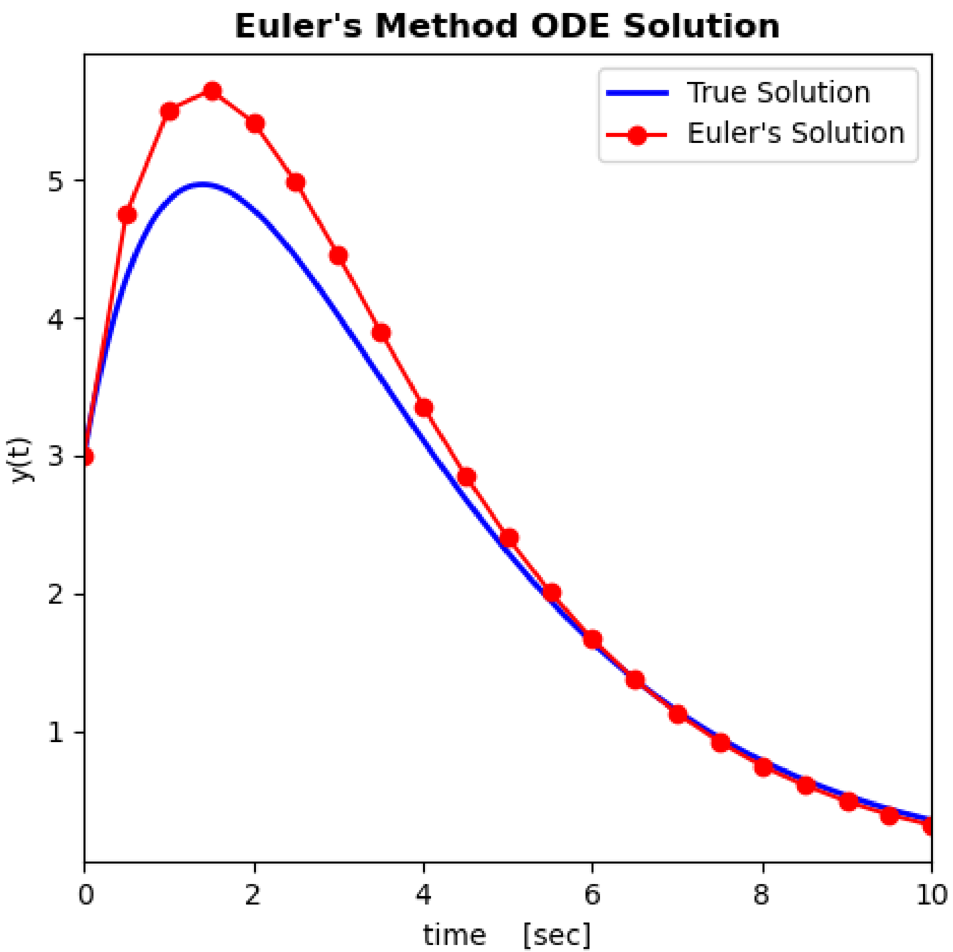
$$y(0) = 3$$

and a step size of

$$h = 0.5\ sec$$

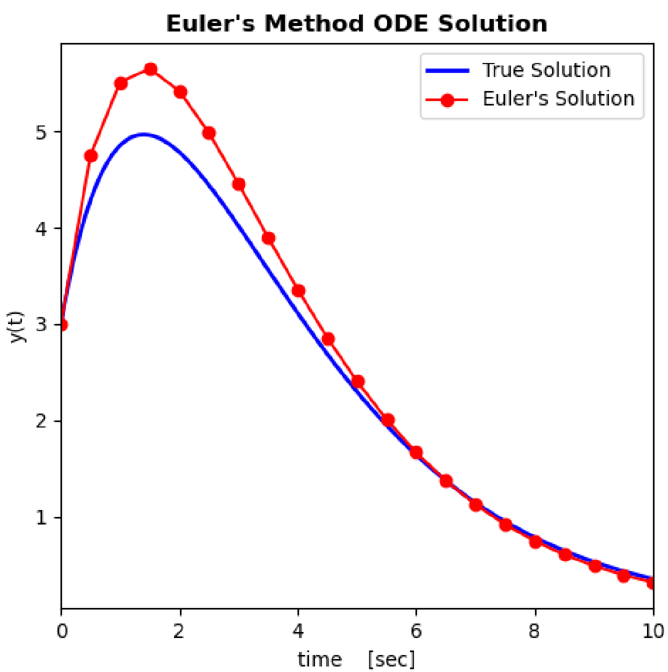
- True solution is:

$$y(t) = e^{-0.5t} + 5t \cdot e^{-0.5t}$$



Euler’s Method - Example


```python
dydt = lambda t, y: 5*np.exp(-0.5*t) - 0.5*y
y0 = 3

t0 = 0
tf = 10
h = 0.5

ttrue = np.linspace(t0,tf,2000)
ytrue = 3*np.exp(-0.5*ttrue) + 5*ttrue*np.exp(-0.5*ttrue)

[t,y] = euler(dydt, [t0,tf], y0, h)
```




```python
# euler.py

import numpy as np
from matplotlib import pyplot as plt

def euler(dydt,tspan,y0,h):
    '''
    Solves an ODE using Euler's method.

    Parameters
    ----------
    dydt : ODE function - function of t and y
    tspan : array of initial and final times: tspan = [t0, tf]
    y0 : initial condition
    h : time step

    Returns
    -------
    t : time vector of solution - will contain tf, so final time
        step may be smaller than h

    '''

    t0 = tspan[0]
    tf = tspan[1]
    t = np.arange(t0, tf+h/2, h)

    # if tspan isn't divisible by h,
    # add tf as final time point
    if t[-1] != tf:
        t = np.append(t, tf)

    n = len(t)

    y = np.zeros(len(t))
    y[0] = y0

    for i in range(n-1):
        y[i+1] = y[i] + dydt(t[i],y[i])*(t[i+1]-t[i])

    return [t, y]
```

Euler's Method - Error

- Two types of truncation error:
  - ***Local*** – error due to the approximation associated with the given method over a single time step
  - ***Global*** – error propagated forward from previous time steps
- Total error is the sum of local and global error
- Representing the solution to the ODE as a Taylor series expansion about (t_i, y_i), the solution at t_{i+1} is:

$$y_{i+1} = y_i + f(t_i, y_i)h + f'(t_i, y_i)\frac{h^2}{2!} + \cdots + f^{(n)}(t_i, y_i)\frac{h^n}{n!} + R_n$$

- Where the remainder term is:

$$R_n = O(h^{n+1})$$

Euler's Method - Error

- Euler's method is the Taylor series, truncated after the first derivative term

$$y_{i+1} = y_i + f(t_i, y_i)h + R_1$$

- For small enough h, the error is dominated by the next term in the series, so

$$E_a = f'(t_i, y_i)\frac{h^2}{2!} \approx R_1 = O(h^2)$$

- ***Local error is proportional to h^2***

- Analysis of the global (i.e. propagated) error is beyond the scope of this course, but the result is that ***global error is proportional to h***

Euler's Method – Stability

- Euler's method will result in error, but worse yet, it may be unstable
  - Unstable if errors grow without bound
- Consider, for example, the following ODE:

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

- The true solution decays exponentially to zero:

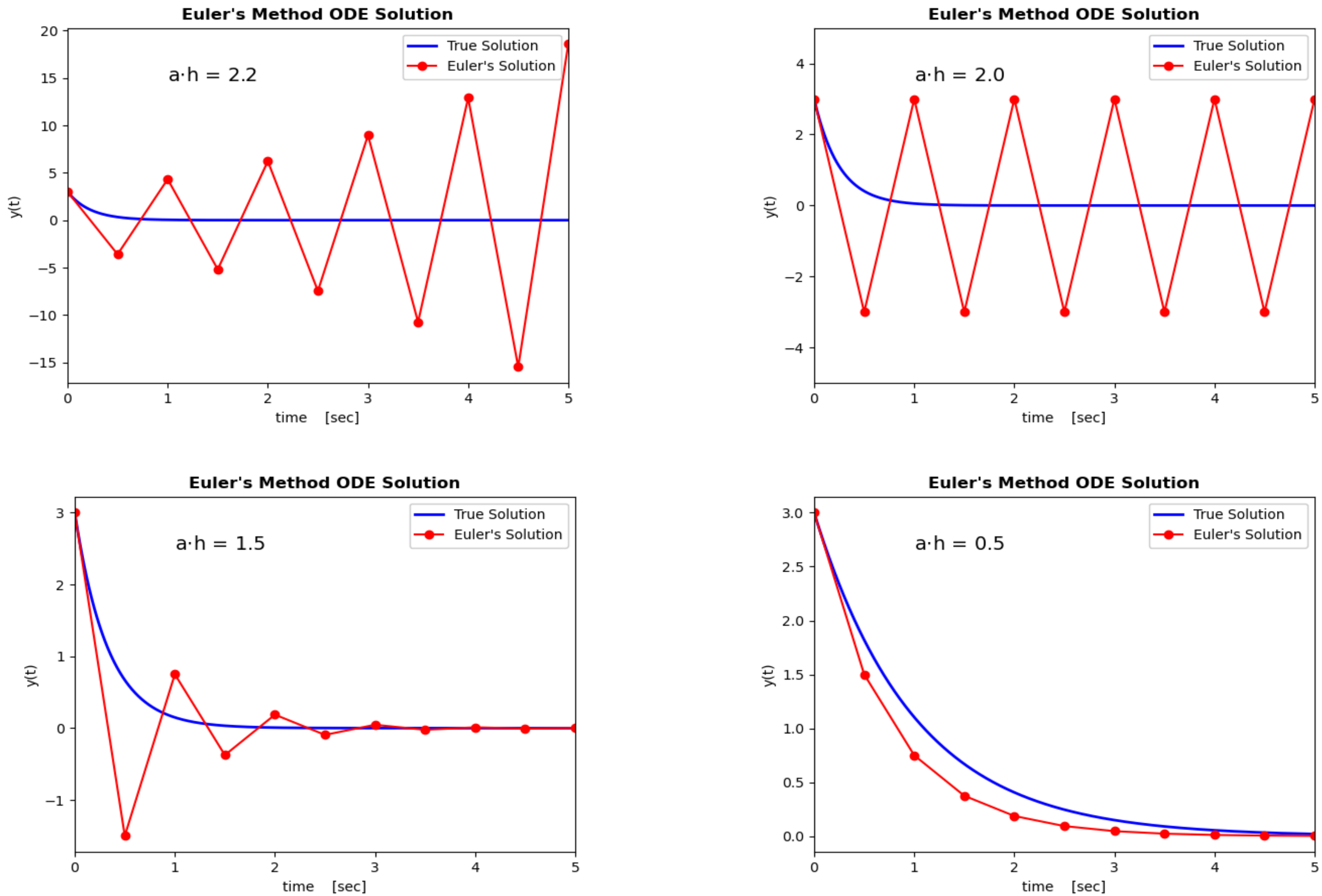
$$y(t) = y_0 e^{-at}$$

- Using Euler's method, the solution is

$$y_{i+1} = y_i - a y_i h = y_i(1 - ah)$$

- This solution will grow without bound if $|1 - ah| > 1$, i.e. if $h > 2/a$
  - If the step size is too large, solution blows up
  - Euler's method is ***conditionally stable***

Stability of Euler's Method – Examples



Heun's Method

Heun's Method

- Euler's assumes a constant slope for the increment function:

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

- Improve accuracy of the solution by using a more accurate slope estimate for $t_i \leq t \leq t_{i+1}$
- Heun's method first applies Euler's method to predict the value of y at t_{i+1} – the ***predictor equation:***

$$y_{i+1}^0 = y_i + f(t_i, y_i)h$$

- This value is then used to predict the slope at t_{i+1}

$$y'_{i+1} = f(t_{i+1}, y_{i+1}^0)$$

Heun's Method

- The increment function is the average of the slope at (t_i, y_i) and the slope at $\left(t_{i+1}, y_{i+1}^0\right)$

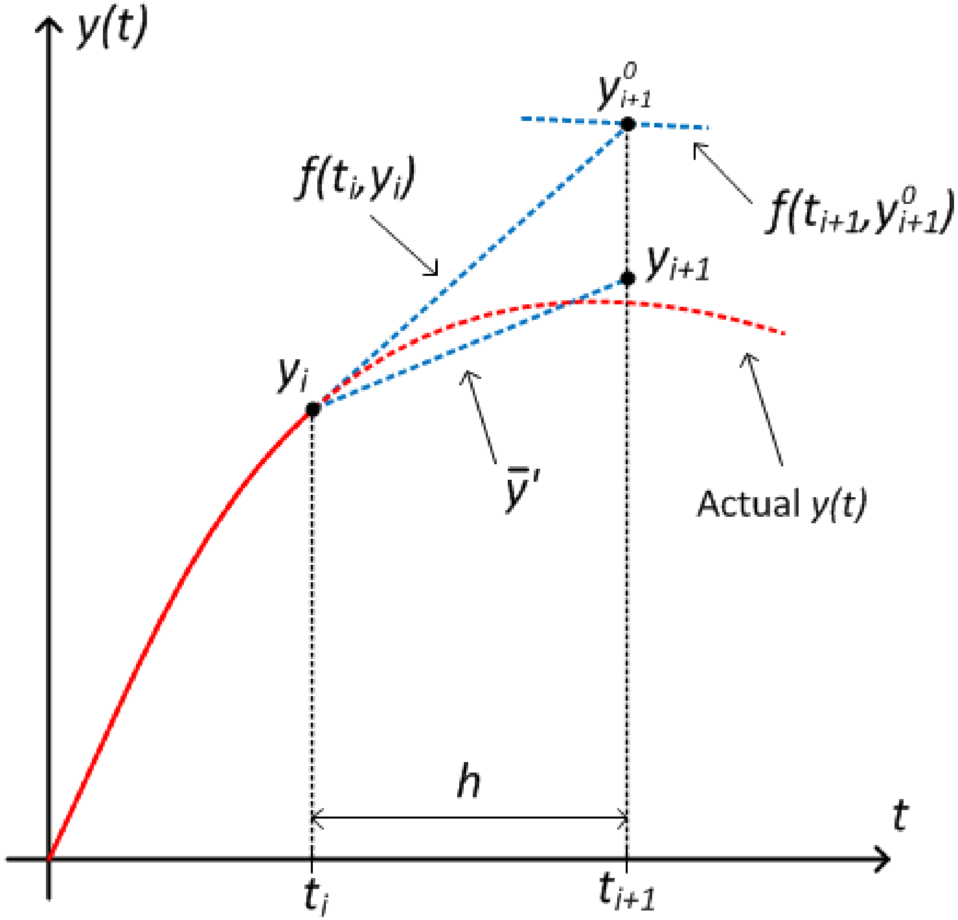
$$\phi = \bar{y}' = \frac{f(t_i, y_i) + f\left(t_{i+1}, y_{i+1}^0\right)}{2}$$

- The next value of $y(t)$ is given by the ***corrector equation:***

$$y_{i+1} = y_i + \frac{f(t_i, y_i) + f\left(t_{i+1}, y_{i+1}^0\right)}{2} h$$

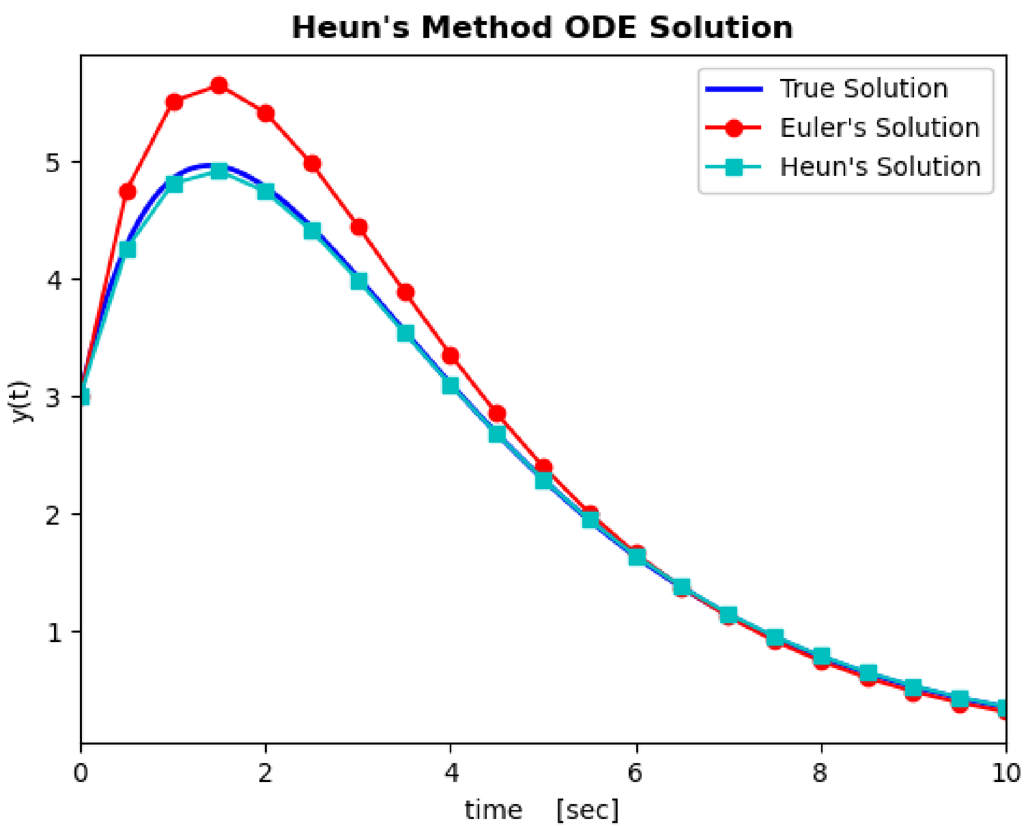
Heun's Method – Summary

- Apply Euler's – the ***predictor equation*** – to predict y_{i+1}^0
- Calculate slope at (t_{i+1}, y_{i+1}^0)
- Compute average of the two slopes
- Use slope average to propagate the solution forward to y_{i+1} – the ***corrector equation***



Heun's Method – Example


```python
def heun(dydt,tspan,y0,h):
    '''
    Solves an ODE using Heun's method

    Parameters
    ----------
    dydt : ODE function - function of t and y
    tspan : array of initial and final times: tspan = [t0, tf]
    y0 : initial condition
    h : time step

    Returns
    -------
    t : time vector of solution - will contain tf, so final time
        step may be smaller than h
    y : solution vector

    '''

    t0 = tspan[0]
    tf = tspan[1]
    t = np.arange(t0, tf+h/2, h)

    # make sure last time point is tf
    if t[-1] != tf:
        t = np.append(t, tf)

    n = len(t)

    y = np.zeros(len(t))
    y[0] = y0

    for i in range(n-1):
        # predictor equation
        yp = y[i] + dydt(t[i],y[i])*(t[i+1]-t[i])
        # predicted slope at t(i+1)
        dydtp = dydt(t[i+1],yp)
        # increment function - avg. slope
        phi = (dydt(t[i],y[i]) + dydtp)/2
        # corrector equation
        y[i+1] = y[i] + phi*(t[i+1]-t[i])

    return [t, y]
```



Heun's Method with Iteration

- ***Predictor equation***:

$$y_{i+1}^{0} = y_i + f(t_i, y_i)h$$

- ***Corrector equation***:

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

- ***The corrector equation can be applied iteratively***, providing a refined estimate of y_{i+1}
- Iterate until approximate error falls below some stopping criterion

$$|\varepsilon_a| = \left| \frac{y_{i+1}^{j} - y_{i+1}^{j-1}}{y_{i+1}^{j}} \right| \cdot 100\% \leq \varepsilon_s$$

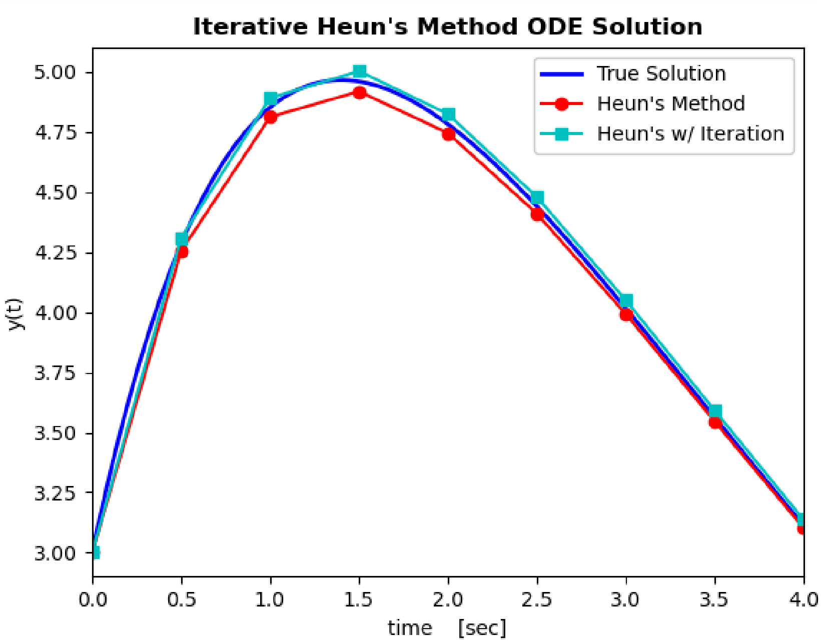
Iterative Heun's Method – Algorithm

- $y_{i+1}^{0} = y_i + f(t_i, y_i)h$
- $j = 1$
- While $|\varepsilon_a| > \varepsilon_s$
  - $y_{i+1}^{j} = y_i + \frac{f(t_i, y_i) + f\left(t_{i+1}, y_{i+1}^{j-1}\right)}{2} h$
  - $|\varepsilon_a| = \left|\frac{y_{i+1}^{j} - y_{i+1}^{j-1}}{y_{i+1}^{j}}\right| \cdot 100\%$
  - $j = j + 1$

- Does not necessarily converge to the correct solution, though ε_a will converge to a finite value

Iterative Heun’s Method – Example


```python
def heuniter(dydt,tspan,y0,h,reltol):
    '''
    Solves an ODE using the iterative Heun's method

    Parameters
    ----------
    dydt : ODE function - function of t and y
    tspan : array of initial and final times: tspan = [t0, tf]
    y0 : initial condition
    h : time step

    Returns
    -------
    t : time vector of solution - will contain tf, so final time
        step may be smaller than h
    y : solution vector

```

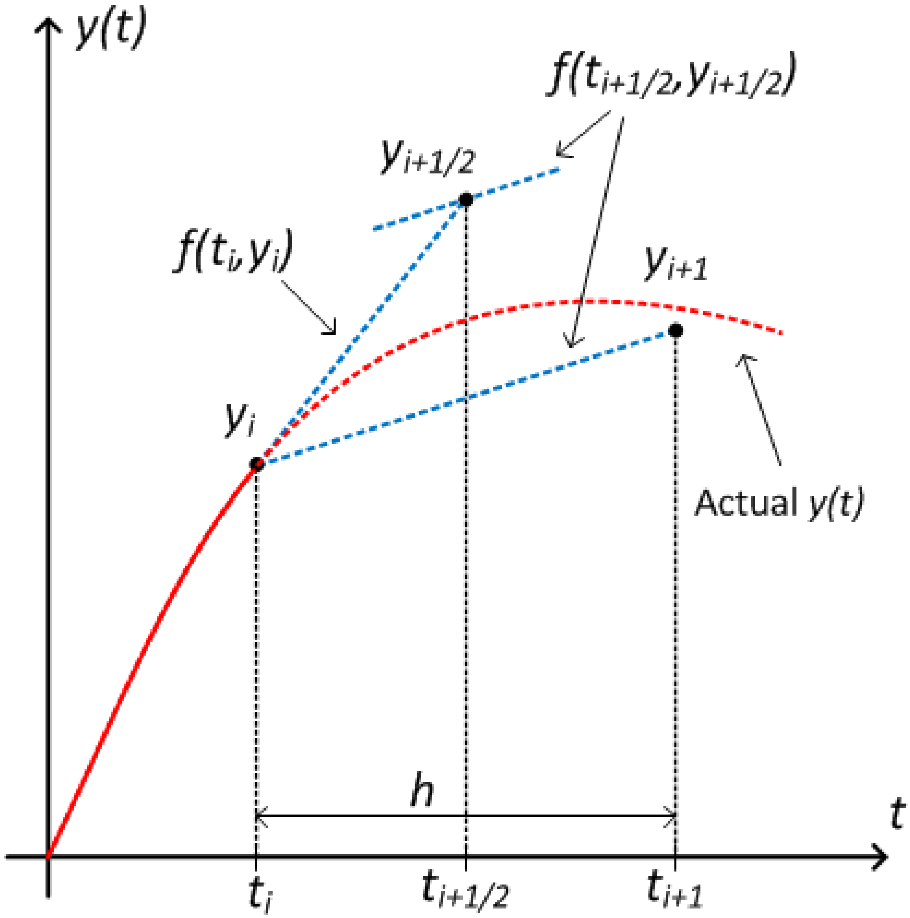



```python
    t0 = tspan[0]
    tf = tspan[1]
    t = np.arange(t0, tf+h/2, h)

    # make sure last time point is tf
    if t[-1] != tf:
        t = np.append(t, tf)

    n = len(t)

    y = np.zeros(len(t))
    y[0] = y0
    ea = 100

    for i in range(n-1):
        # predictor equation
        yp_old = y[i] + dydt(t[i],y[i])*(t[i+1]-t[i])
        while ea >= reltol:
            # predicted slope at (t(i+1),yp_old)
            dydtp = dydt(t[i+1],yp_old)
            # increment function
            phi = (dydt(t[i],y[i]) + dydtp)/2
            # next estimate
            yp = y[i] + phi*(t[i+1]-t[i])
            # estimate the error
            ea = np.abs((yp-yp_old)/yp)*100
            yp_old = yp

        # result of iteration is next y value
        y[i+1] = yp
        ea = 100    # reset ea for next time step

    return [t, y]
```

Midpoint Method

Midpoint Method

- The ***slope at the midpoint of a time interval*** used as the increment function
- Provides a more accurate estimate of the slope across the entire time interval



Midpoint Method

- Apply Euler's method to approximate y at midpoint

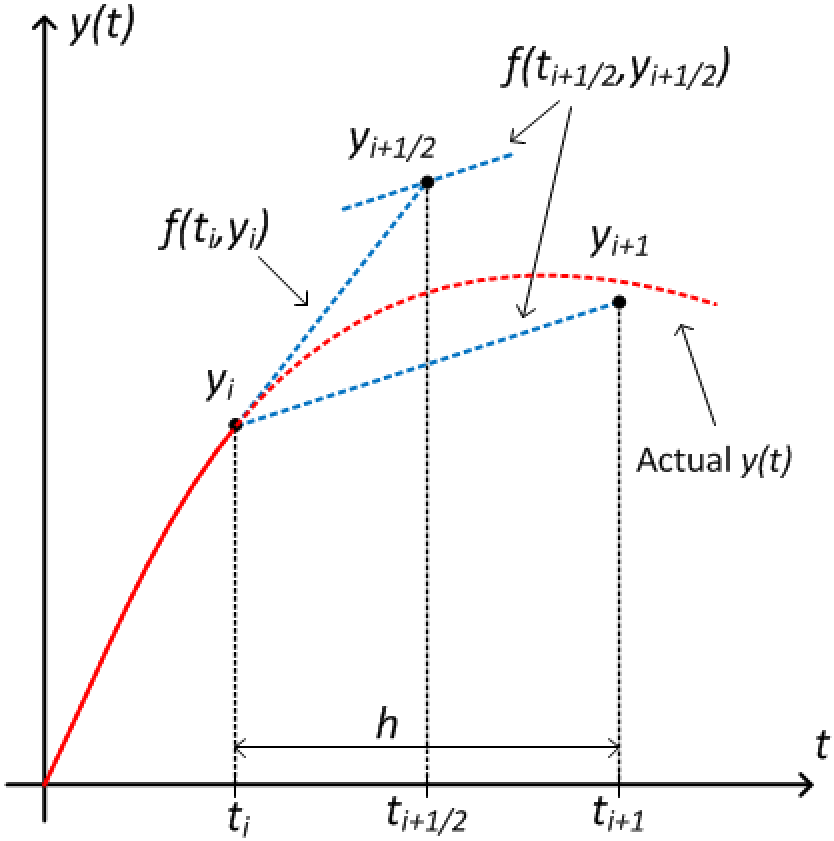
$$y_{i+\frac{1}{2}} = y_i + f(t_i, y_i)\frac{h}{2}$$

- Slope estimate at midpoint:

$$y'_{i+\frac{1}{2}} = f\left(t_{i+\frac{1}{2}}, y_{i+\frac{1}{2}}\right)$$

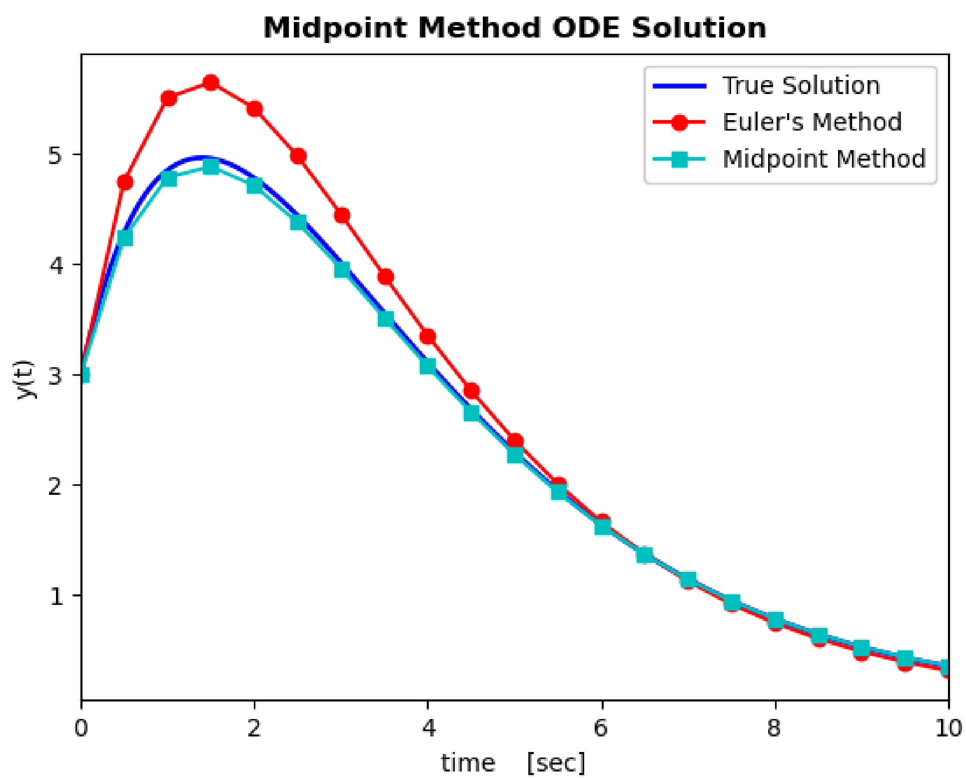
- Midpoint slope estimate is increment function

$$y_{i+1} = y_i + f\left(t_{i+\frac{1}{2}}, y_{i+\frac{1}{2}}\right)h$$



Midpoint Method – Example


```python
def midpt(dydt,tspan,y0,h):
    '''
    Solves an ODE using the midpoint method

    Parameters
    ----------
    dydt : ODE function - function of t and y
    tspan : array of initial and final times: tspan = [t0, tf]
    y0 : initial condition
    h : time step

    Returns
    -------
    t : time vector of solution - will contain tf, so final time
        step may be smaller than h
    y : solution vector

    '''

    t0 = tspan[0]
    tf = tspan[1]
    t = np.arange(t0, tf+h/2, h)

    # make sure last time point is tf
    if t[-1] != tf:
        t = np.append(t, tf)

    n = len(t)

    y = np.zeros(len(t))
    y[0] = y0

    for i in range(n-1):
        # apply Euler's to get y(i+1/2)
        h = t[i+1] - t[i]
        ymp = y[i] + dydt(t[i],y[i])*h/2
        # increment function - midpoint slope
        phi = dydt(t[i]+h/2, ymp)
        # propagate y forward one time step
        y[i+1] = y[i] + phi*h

    return [t, y]

```



One-Step Methods – Error

Method	Local Error	Global Error
Euler's	$O(h^2)$	$O(h)$
Heun's (w/o iter.)	$O(h^3)$	$O(h^2)$
Midpoint	$O(h^3)$	$O(h^2)$

Runge-Kutta Methods

Runge-Kutta Methods

- Euler's, Heun's, and midpoint methods are specific cases of the broader category of one-step methods known as ***Runge-Kutta methods***
- Runge-Kutta methods all have the same general form

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

- The increment function has the following form

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

- n is the order of the Runge-Kutta method
  - We'll see that Euler's is a first-order method, while Heun's and midpoint are both second-order

Runge-Kutta Methods

- The increment function is

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

where

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

- The a's, p's, and q's are constants
- Can see that Euler's method is first-order with $a_1 = 1$

Runge-Kutta Methods

- To determine values of a's, p's, and q's:
  - Set the Runge-Kutta formula equal to a Taylor series of the same order
  - Equate coefficients
  - An under-determined system results
  - Arbitrarily set one constant and solve for others
- Procedure is the same for all orders
  - We'll step through the derivation of the second-order Runge-Kutta formulas

Second-Order Runge-Kutta Methods

- Second-order Runge-Kutta:

$$y_{i+1} = (a_1 k_1 + a_2 k_2)h \qquad (1)$$

where

$$k_1 = f(t_i, y_i) \qquad (2)$$

$$k_2 = f(t_i + p_1 h, y_i + q_{11} k_1 h) \qquad (3)$$

- Second-order Taylor series:

$$y_{i+1} = y_i + f(t_i, y_i)h + \frac{f'(t_i, y_i)}{2!} h^2 \qquad (4)$$

where

$$f'(t_i, y_i) = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} \frac{dy}{dt} \qquad (5)$$

Second-Order Runge-Kutta Methods

- Substituting (5) into (4), and recognizing that $\frac{dy}{dt} = f(t_i, y_i)$, the Taylor series becomes

$$y_{i+1} = y_i + f(t_i, y_i)h + \left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f(t_i, y_i)\right)\frac{h^2}{2!} \tag{6}$$

- Next, represent (3) as a first-order Taylor series
  - It's a function of two variables, for which the first-order Taylor series has the following form

$$g(x + \Delta x, y + \Delta y) = g(x, y) + \Delta x \frac{\partial g}{\partial x} + \Delta y \frac{\partial g}{\partial y} + O(h^2) \tag{7}$$

- Using (7), (3) becomes

$$k_2 = f(t_i, y_i) + p_1 h \frac{\partial f}{\partial t} + q_{11} k_1 h \frac{\partial f}{\partial y} + O(h^2) \tag{8}$$

Second-Order Runge-Kutta Methods

- Substituting (2) and (8) into (1)

$$y_{i+1} = y_i + a_1 h f(t_i, y_i) + a_2 h f(t_i, y_i)$$
$$+ a_2 p_1 h^2 \frac{\partial f}{\partial t} + a_2 q_{11} h^2 \frac{\partial f}{\partial y} f(t_i, y_i) \qquad (9)$$

- Now, set (9) equal to (6), the Taylor series

$$y_i + a_1 h f(t_i, y_i) + a_2 h f(t_i, y_i) + a_2 p_1 h^2 \frac{\partial f}{\partial t} + a_2 q_{11} h^2 \frac{\partial f}{\partial y} f(t_i, y_i)$$
$$= y_i + f(t_i, y_i) h + \frac{\partial f}{\partial t} \frac{h^2}{2} + \frac{\partial f}{\partial y} \frac{h^2}{2} f(t_i, y_i) \qquad (10)$$

- Equating the coefficients in (10) gives three equations with four unknowns:

$$a_1 + a_2 = 1 \qquad (11)$$
$$a_2 p_1 = \frac{1}{2} \qquad (12)$$
$$a_2 q_{11} = \frac{1}{2} \qquad (13)$$

Second-Order Runge-Kutta Methods

- We have three equations in four unknowns

$$a_1 + a_2 = 1 \qquad (11)$$

$$a_2 p_1 = \frac{1}{2} \qquad (12)$$

$$a_2 q_{11} = \frac{1}{2} \qquad (13)$$

- An under-determined system
  - An infinite number of solutions
  - Arbitrarily set one constant – a_2 – to a certain value and solve for the other three constants
  - Different solution for each value of a_2 – a ***family*** of solutions

$a_2 = 1/2$ – Heun's Method

- Arbitrarily set a_2 and solve for the other constants

$$a_1 = \frac{1}{2}, \quad a_2 = \frac{1}{2}, \quad p_1 = 1, \quad q_{11} = 1$$

- The second-order Runge-Kutta formula becomes

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

where

$$k_1 = f(t_i, y_i)$$

$$k_2 = f(t_i + p_1 h, y_i + q_{11} k_1 h) = f(t_i + h, y_i + k_1 h)$$

- This is ***Heun's method***

$$y_{i+1} = y_i + \frac{f(t_i, y_i) + f\left(t_{i+1}, y_{i+1}^0\right)}{2} h$$

$a_2 = 1$ – Midpoint Method

- Arbitrarily set a_2 and solve for the other constants

$$a_1 = 0, \quad a_2 = 1, \quad p_1 = \tfrac{1}{2}, \quad q_{11} = \tfrac{1}{2}$$

- The second-order Runge-Kutta formula becomes

$$y_{i+1} = y_i + k_2 h$$

where

$$k_1 = f(t_i, y_i)$$

$$k_2 = f(t_i + p_1 h, y_i + q_{11} k_1 h) = f(t_i + \frac{h}{2}, y_i + k_1 \frac{h}{2})$$

- This is the ***midpoint method***

$$y_{i+1} = y_i + f\left(t_{i+\frac{1}{2}}, y_{i+\frac{1}{2}}\right) h$$

Fourth-Order Runge-Kutta

- The most commonly used Runge-Kutta method is the ***fourth-order*** method
- Derivation proceeds similar to that of the second-order method
  - Under-determined system – ***family of solutions***
- Most common ***fourth-order Runge-Kutta method***:

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

where

$$k_1 = f(t_i, y_i)$$

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

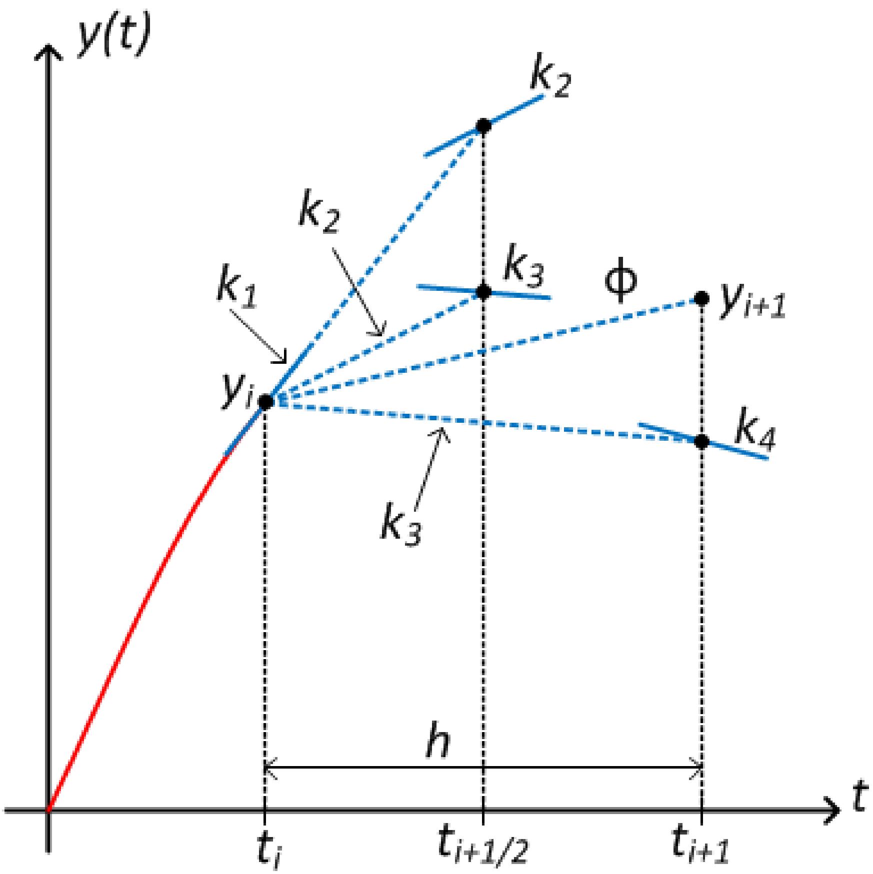
$$k_3 = f\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2 h\right)$$

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

- ***The increment function is a weighted average of four different slopes***

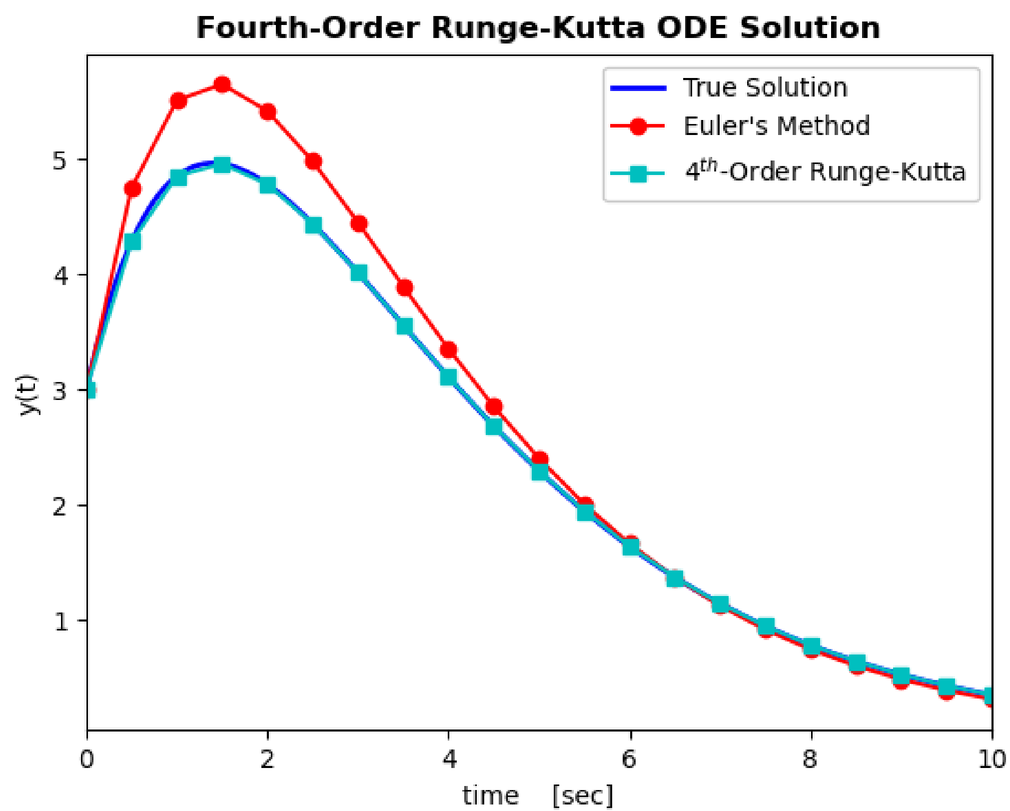
4^{th}-Order Runge-Kutta – Algorithm

1. Calculate the slope at (t_i, y_i) $\rightarrow$ this is k_1
2. Use k_1 to approximate $y_{i+1/2}$ from y_i. Calculate the slope here $\rightarrow$ this is k_2
3. Use k_2 to re-approx. $y_{i+1/2}$ from y_i. Calculate the slope here $\rightarrow$ this is k_3
4. Use k_3 to approx. y_{i+1} from y_i. Calculate the slope here $\rightarrow$ this is k_4
5. Calculate ϕ as a weighted average of the four slopes



Fourth-Order Runge-Kutta – Example


```python
def rk4ode(dydt,tspan,y0,h):
    '''
    Solves an ODE using the 4th-order Runge-Kutta method

    Parameters
    ----------
    dydt : ODE function - function of t and y
    tspan : array of initial and final times: tspan = [t0, tf]
    y0 : initial condition
    h : time step

    Returns
    -------
    t : time vector of solution - will contain tf, so final time
        step may be smaller than h
    y : solution vector

    '''

    t0 = tspan[0]
    tf = tspan[1]
    t = np.arange(t0, tf+h/2, h)

    # make sure last time point is tf
    if t[-1] != tf:
        t = np.append(t, tf)

    n = len(t)

    y = np.zeros(len(t))
    y[0] = y0

    for i in range(n-1):
        # calculate slopes
        k1 = dydt(t[i],y[i])
        k2 = dydt(t[i]+h/2,y[i]+k1*h/2)
        k3 = dydt(t[i]+h/2,y[i]+k2*h/2)
        k4 = dydt(t[i]+h,y[i]+k3*h)
        # increment function
        phi = 1/6*(k1 + 2*k2 + 2*k3 + k4)
        # propagate y forward one time step
        y[i+1] = y[i] + phi*h

    return [t, y]
```



Systems of Equations

Higher-Order Differential Equations

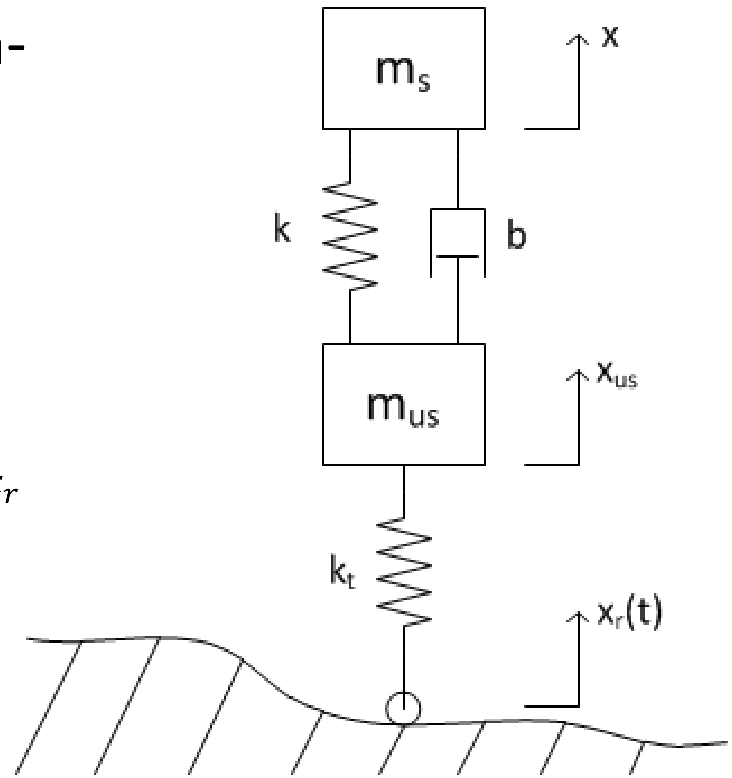
- The ODE solution techniques we've looked at so far pertain to first-order ODEs
- Can be extended to higher-order ODEs by reducing to systems of first-order equations
  - ***An n^{th}-order ODE can be represented as a system of n first-order ODEs***
- Solution method is applied to each equation at each time step before advancing to the next time step
- We'll now illustrate the process with a fourth-order quarter-car model example

Fourth-Order ODE – Example

- Consider a quarter-car model of a vehicle's suspension system
- Apply Newton's second law to each mass to derive the governing fourth-order ODE
  - Single 4th-order equation, or
  - Two 2nd-order equations

$$\ddot{x} + \frac{k}{m_s}(x - x_{us}) + \frac{b}{m_s}(\dot{x} - \dot{x}_{us}) = 0$$

$$\ddot{x}_{us} + \frac{b}{m_{us}}(\dot{x}_{us} - \dot{x}) + \frac{k}{m_{us}}(x_{us} - x) + \frac{k_t}{m_{us}}x_{us} = \frac{k_t}{m_{us}}x_r$$



- Want to reduce to a system of four first-order ODEs
  - Put into state-space form

Fourth-Order ODE – Example

$$\ddot{x} + \frac{k}{m_s}(x - x_{us}) + \frac{b}{m_s}(\dot{x} - \dot{x}_{us}) = 0 \qquad (1)$$

$$\ddot{x}_{us} + \frac{b}{m_{us}}(\dot{x}_{us} - \dot{x}) + \frac{k}{m_{us}}(x_{us} - x) + \frac{k_t}{m_{us}} x_{us} = \frac{k_t}{m_{us}} x_r(t) \qquad (2)$$

- Reducing the ODE to a system of first-order ODEs amounts to representing our system in state-space form:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{b}u$$

- Define a ***state vector*** of displacements and velocities:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} x \\ x_{us} \\ v \\ v_{us} \end{bmatrix} \qquad (3)$$

Fourth-Order ODE – Example

- Rewrite (1) and (2) using the ***state variables*** defined in (3)

$$\dot{v} = \dot{x}_3 = -\frac{k}{m_s}x_1 + \frac{k}{m_s}x_2 - \frac{b}{m_s}x_3 + \frac{b}{m_s}x_4 = 0 \qquad (4)$$

$$\dot{v}_{us} = \dot{x}_4 = -\frac{b}{m_{us}}x_4 + \frac{b}{m_{us}}x_3 - \frac{k}{m_{us}}x_2 + \frac{k}{m_{us}}x_1 - \frac{k_t}{m_{us}}x_2 + \frac{k_t}{m_{us}}x_r(t) \qquad (5)$$

- The ***state variable representation*** of the system is

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{x}_{us} \\ \dot{v} \\ \dot{v}_{us} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{k}{m_s} & \frac{k}{m_s} & -\frac{b}{m_s} & \frac{b}{m_s} \\ \frac{k}{m_{us}} & -\frac{k+k_t}{m_{us}} & \frac{b}{m_{us}} & -\frac{b}{m_{us}} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{k_t}{m_{us}} \end{bmatrix} \cdot x_r(t) \qquad (6)$$

Fourth-Order ODE – Example

- Equation (6) clearly shows our system of four first-order ODEs
  - Alternatively, could have derived the state-space equations directly (e.g. using a ***bond graph*** approach)
- In Python, we'll represent our system as an ***n-dimensional function***
  - A vector of n functions:

$$\dot{x}_1 = x_3 \tag{7}$$

$$\dot{x}_2 = x_4 \tag{8}$$

$$\dot{x}_3 = -\frac{k}{m_s}x_1 + \frac{k}{m_s}x_2 - \frac{b}{m_s}x_3 + \frac{b}{m_s}x_4 \tag{9}$$

$$\dot{x}_4 = \frac{k}{m_{us}}x_1 - \frac{k+k_t}{m_{us}}x_2 + \frac{b}{m_{us}}x_3 - \frac{b}{m_{us}}x_4 + \frac{k_t}{m_{us}}x_r(t) \tag{10}$$

Fourth-Order ODE – Example

- In Python, define the n^{th}-order system of ODEs as shown below
  - An n-dimensional function

```
def qcarode(t,y,ms,mus,k,kt,b,xr):
    # system of first-order ODEs
    dy = np.zeros(4)
    dy[0] = y[2]
    dy[1] = y[3]
    dy[2] = -k/ms*y[0] + k/ms*y[1] - b/ms*y[2] +b/ms*y[3]
    dy[3] = k/mus*y[0] - (k+kt)/mus*y[1] + b/mus*y[2] - b/mus*y[3] + kt/mus*xr

    return dy
```

- Here, the ODE function includes parameters (m_s, k, etc.) in addition to variables t and y
  - Can create a lambda function wrapper to simplify the passing of parameters

Fourth-Order ODE – Example

- Basic formula remains the same
  - Advance the solution to the next time step using the increment function

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

- Now, the *output* is the vector of states, and the increment function is an n-dimensional vector

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \boldsymbol{\phi} h$$

or

$$\left[x_{1,i+1}, x_{2,i+1}, \dots, x_{n,i+1}\right] = \left[x_{1,i}, x_{2,i}, \dots, x_{n,i}\right] + [\phi_1, \phi_2, \dots, \phi_n]h$$

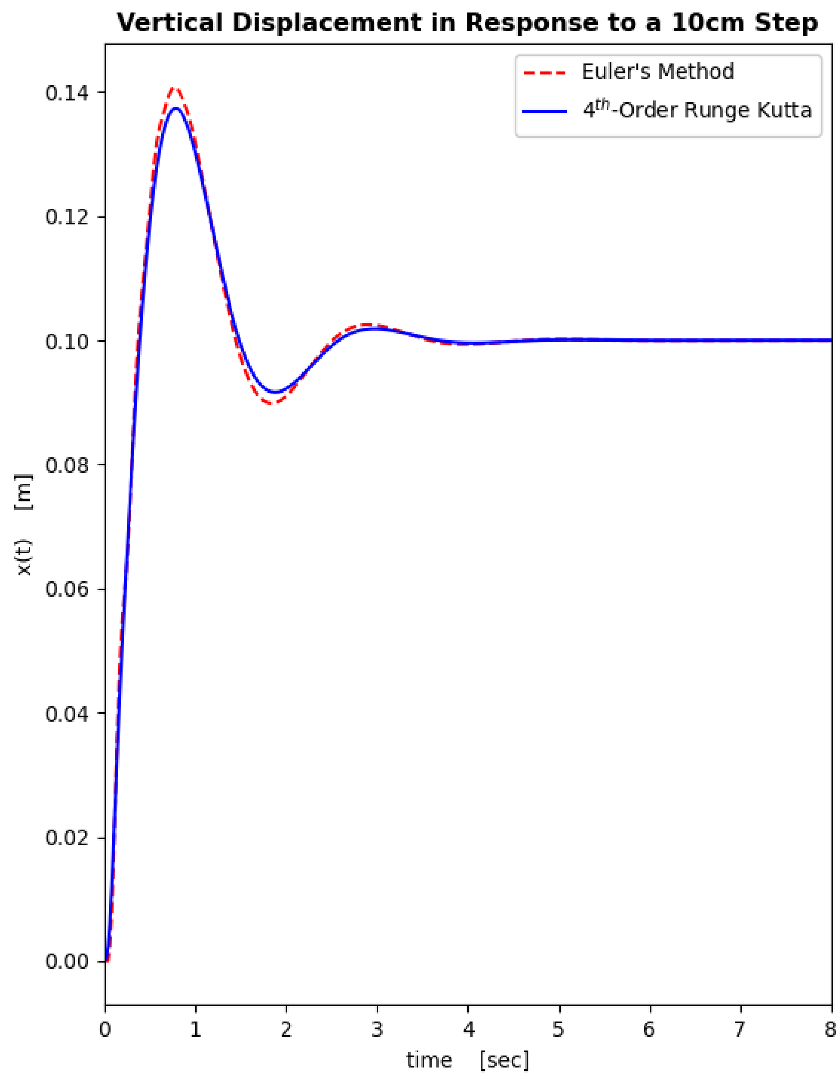
- Requires only a minor modification of the code written for first-order ODEs to accommodate n-dimensional functions

Fourth-Order ODE – Example

- Often want to pass ***parameters*** (i.e., Input arguments in addition to t and y) to the ODE function
- Create a lambda function wrapper for the ODE function, e.g.:

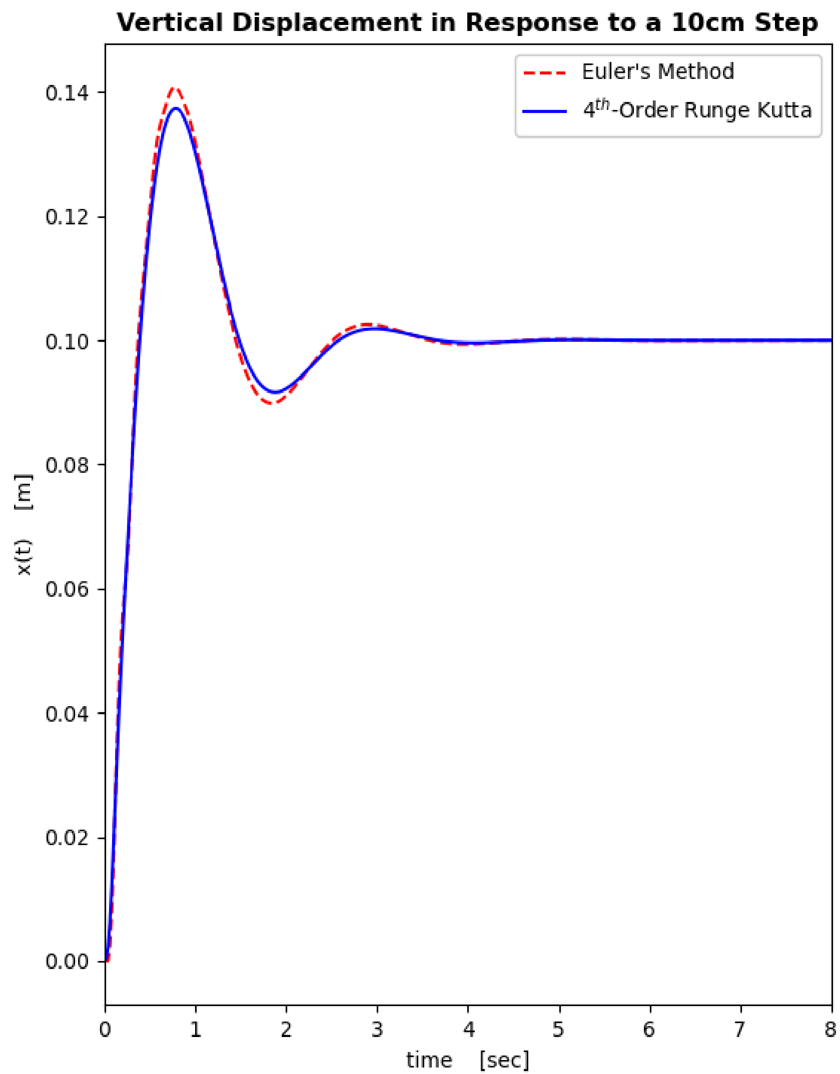
```
# physical system parameters
ms = 973            # sprung mass
k = 10e3            # shock absorber spring constant
b = 3000            # shock absorber damping
kt = 101115         # tire spring constant
mus = 114           # unsprung mass

# input displacement step
xr = 0.1            # 10 cm

# lambda function wrapper to allow
# for passing parameters
xdot = lambda t, y: qcarode(t,y,ms,mus,k,kt,b,xr)
```

Fourth-Order ODE – Example



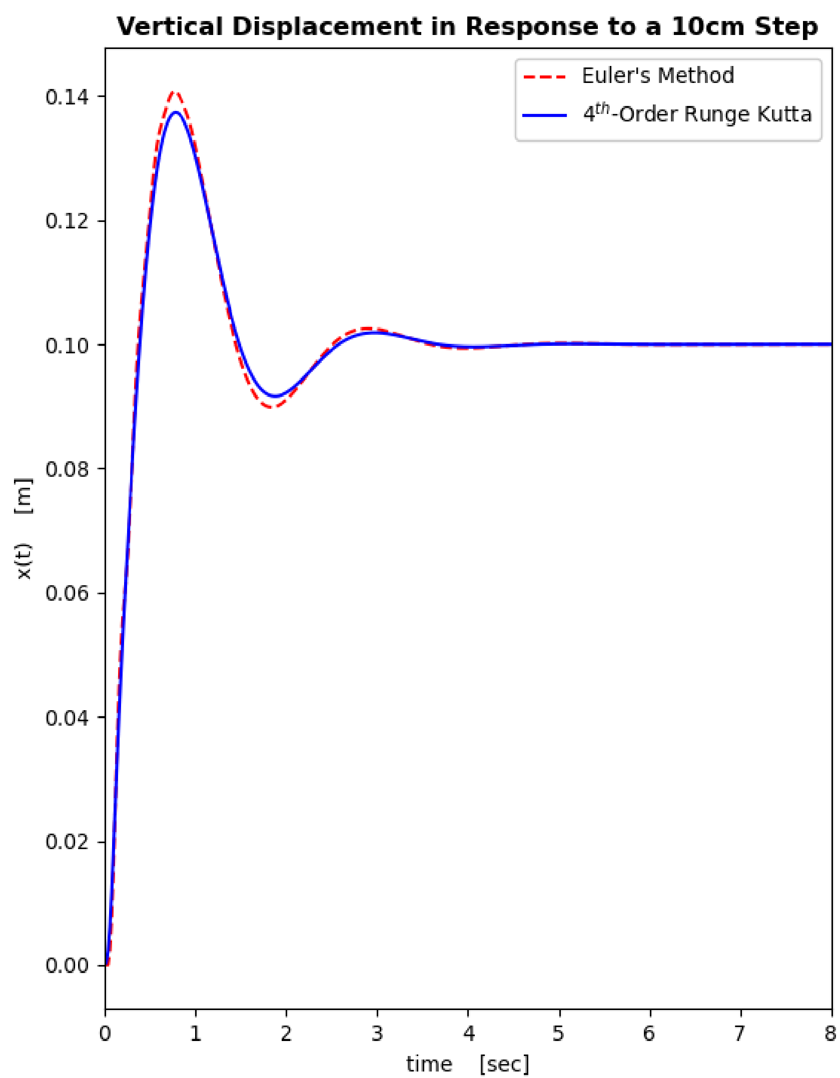

```python
t0 = 0
tf = 8
h = 2e-2

# physical system parameters
ms = 973          # sprung mass
k = 10e3          # shock absorber spring constant
b = 3000          # shock absorber damping
kt = 101115       # tire spring constant
mus = 114         # unsprung mass

# input displacement step
xr = 0.1          # 10 cm

# lambda function wrapper to allow
# for passing parameters
xdot = lambda t, y: qcarode(t,y,ms,mus,k,kt,b,xr)

x0 = [0,0,0,0]

[te, xe] = eulern(xdot, [t0, tf], x0, h)
[trk4, xrk4] = rk4oden(xdot, [t0, tf], x0, h)
```

Fourth-Order ODE – Example




```python
def eulern(dydt,tspan,y0,h):
    '''
    Solves an Nth-order ODE using Euler's method.

    Parameters
    ----------
    dydt : ODE function - function of t and y
    tspan : array of initial and final times: tspan = [t0, tf]
    y0 : initial condition
    h : time step

    Returns
    -------
    t : time vector of solution - will contain tf, so final time
        step may be smaller than h
    y : solution vector

    '''

    t0 = tspan[0]
    tf = tspan[1]
    t = np.arange(t0, tf+h/2, h)

    # if tspan isn't divisible by h,
    # add tf as final time point
    if t[-1] != tf:
        t = np.append(t, tf)

    n = len(t)
    N = len(y0)

    y = np.zeros((n, N))
    y[0,:] = y0

    for i in range(n-1):
        y[i+1,:] = y[i,:] + dydt(t[i],y[i,:])*(t[i+1]-t[i])

    return [t, y]
```

Fourth-Order ODE – Example




```python
def rk4oden(dydt,tspan,y0,h):
    '''
    Solves an Nth-order ODE using the 4th-order Runge-Kutta method

    Parameters
    ----------
    dydt : ODE function - function of t and y
    tspan : array of initial and final times: tspan = [t0, tf]
    y0 : initial condition
    h : time step

    Returns
    -------
    t : time vector of solution - will contain tf, so final time
        step may be smaller than h
    y : solution vector

    '''

    t0 = tspan[0]
    tf = tspan[1]
    t = np.arange(t0, tf+h/2, h)

    # make sure last time point is tf
    if t[-1] != tf:
        t = np.append(t, tf)

    n = len(t)
    N = len(y0)

    y = np.zeros((n, N))
    y[0,:] = y0

    for i in range(n-1):
        # calculate slopes
        k1 = dydt(t[i],y[i,:])
        k2 = dydt(t[i]+h/2,y[i,:]+k1*h/2)
        k3 = dydt(t[i]+h/2,y[i,:]+k2*h/2)
        k4 = dydt(t[i]+h,y[i,:]+k3*h)
        # increment function
        phi = 1/6*(k1 + 2*k2 + 2*k3 + k4)
        # propagate y forward one time step
        y[i+1,:] = y[i,:] + phi*h

    return [t, y]
```

Solving ODEs in Python

SciPy's ODE Solvers

- SciPy's `solve_ivp()` has several ODE solvers
  - RK45 is the default and should usually be first choice for ***non-stiff*** problems
- ***Stiff*** ODEs are those with a large range of eigenvalues – i.e., both very fast and very slow system poles
  - Numerical solution is difficult
- From the SciPy documentation:

Solver	Stiffness	Accuracy	When to use
RK45	Non-stiff	Medium	Most of the time. First choice.
RK23		Low	For problems with crude error tolerances or for solving moderately stiff problems.
DOP853		High	For problems requiring high precision (low values of rtol and atol).
Radau	Stiff	Low to medium	If ode45 is slow or non-convergent because the problem is stiff.
BDF			

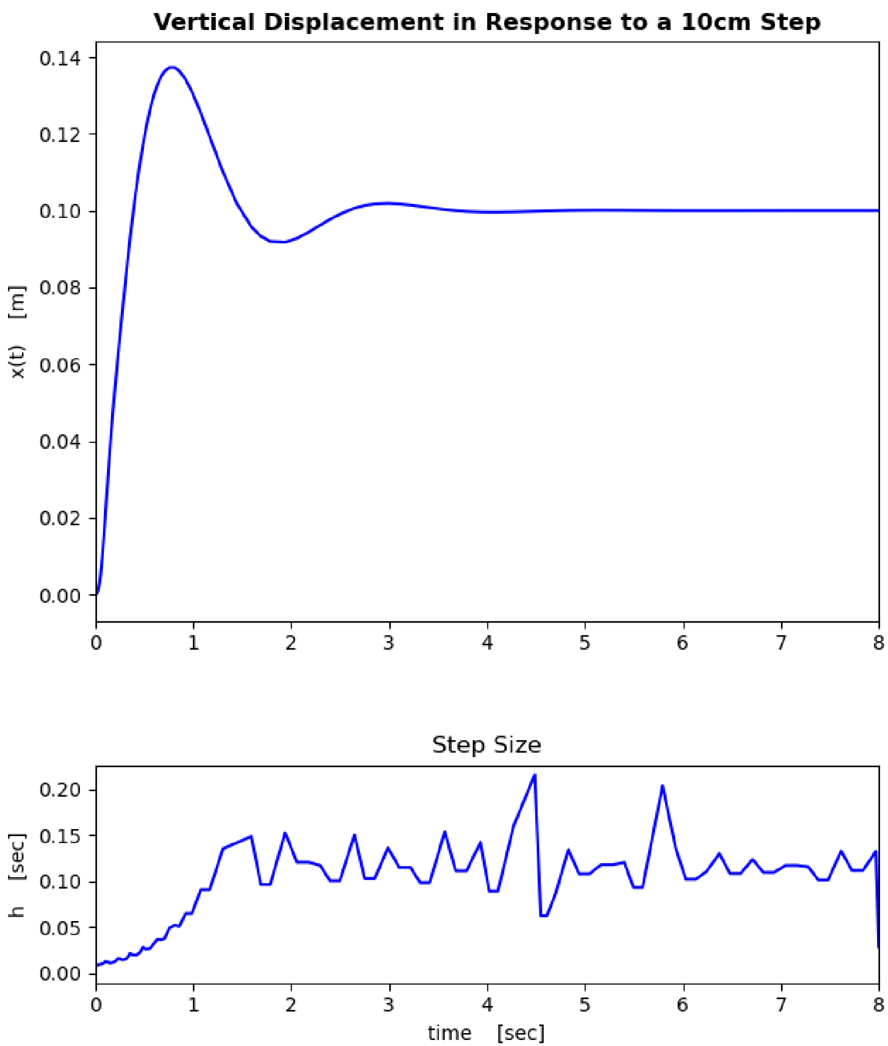
Solving ODEs with SciPy – `solve_ivp()`


```
sol = solve_ivp(dydt, tspan, y0, method='RK45')
```

- `dydt`: ODE function object – n-dimensional
- `tspan`: array of initial and final times – `[ti,tf]`
- `y0`: initial conditions – an n-vector
- `method`: solver to use – optional – default: 'RK45'
- `sol`: an `OdeResult` object with several fields, including:
  - `sol.y`: solution vector
  - `sol.t`: time vector for the solution

- Default method, RK45, is an adaptive algorithm that uses fourth- and fifth-order Runge-Kutta formulas
  - Variable step size

Fourth-Order ODE – Example




```python
def qcarode(t,y,ms,mus,k,kt,b,xr):
    # system of first-order ODEs
    dy = np.zeros(4)
    dy[0] = y[2]
    dy[1] = y[3]
    dy[2] = -k/ms*y[0] + k/ms*y[1] - b/ms*y[2] +b/ms*y[3]
    dy[3] = k/mus*y[0] - (k+kt)/mus*y[1] + b/mus*y[2] \
        - b/mus*y[3] + kt/mus*xr

    return dy

t0 = 0
tf = 8

# physical system parameters
ms = 973        # sprung mass
k = 10e3        # shock absorber spring constant
b = 3000        # shock absorber damping
kt = 101115     # tire spring constant
mus = 114       # unsprung mass

# input displacement step
xr = 0.1        # 10 cm

x0 = [0,0,0,0]

reltol = 1e-6
# use args to pass parameters:
sol = solve_ivp(qcarode, [t0, tf], x0,
                args=(ms,mus,k,kt,b,xr), rtol=reltol)
x = sol.y.transpose()
t = sol.t

h_ivp = np.diff(t)
th = t[1:]
```

Tolerance set with rtol

Pass parameters with args