Chapter 22

Initial-Value Problems

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** and a **system** of ODE:

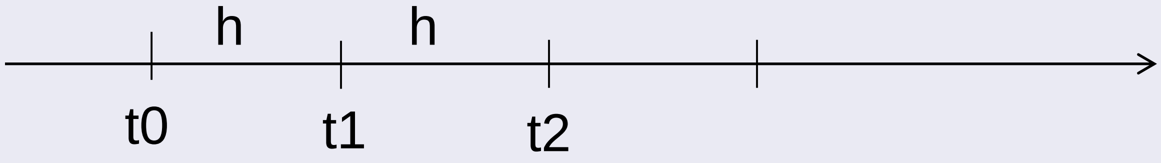
  - Euler (RK1)
  - Heun (RK2)
  - Fourth-Order RK (RK4)

Ordinary Differential Equations (ODEs)

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

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

- With initial conditions $y(t_0) = y_0$
- Divide t-space into segments spaced by h intervals apart.



Numerical integration of an ODE means to get an array of dependent variable versus independent variable

t	y
t_0	y_0
t_1	y_1
..	..
t_n	y_n
..	..
..	..
t_N	y_N

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

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

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

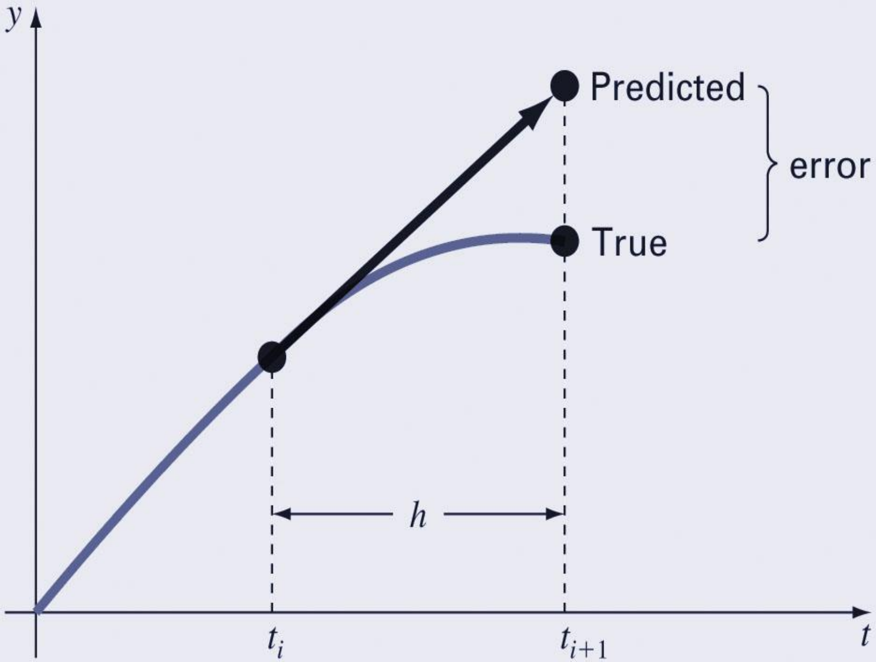
1. Euler's Method

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:

$$\phi = f(t_i, y_i) = k1$$
$$y_{i+1} = y_i + f(t_i, y_i)h$$



Recall Taylor power series (written in t)

$$F(t+h) = F(t) + F'(t)h + \frac{1}{2!}F''(t)h^2 + \cdots$$

At time $t = t_i$ with $y = F$

$$F(t) = F(t_i) = y_i$$

$$F(t+h) = F(t_i + h) = y_{i+1}$$

$$F'(t) = F'(t_i) = y_i' = f(t_i, y_i)$$

$$F''(t) = F''(t_i) = \frac{dF'(t_i)}{dt_i} = \frac{df(t_i, y_i)}{dt_i}$$

$$= \left[\frac{\partial f}{\partial t_i} + \frac{\partial f}{\partial y_i}\frac{dy_i}{dt_i}\right] = \left[\frac{\partial f}{\partial t_i} + \frac{\partial f}{\partial y_i}f\right]$$

Hence to 1st order in h

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

This is Euler method with incremental function

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

The error in this case is

$$R = \frac{1}{2}\left[\frac{\partial f}{\partial t_i} + \frac{\partial f}{\partial y_i} f\right] h^2$$

Error Analysis for Euler's Method

- Recall computation has two types of errors:

  Truncation and Roundoff

- The total, or *global* truncation error can be further split into:

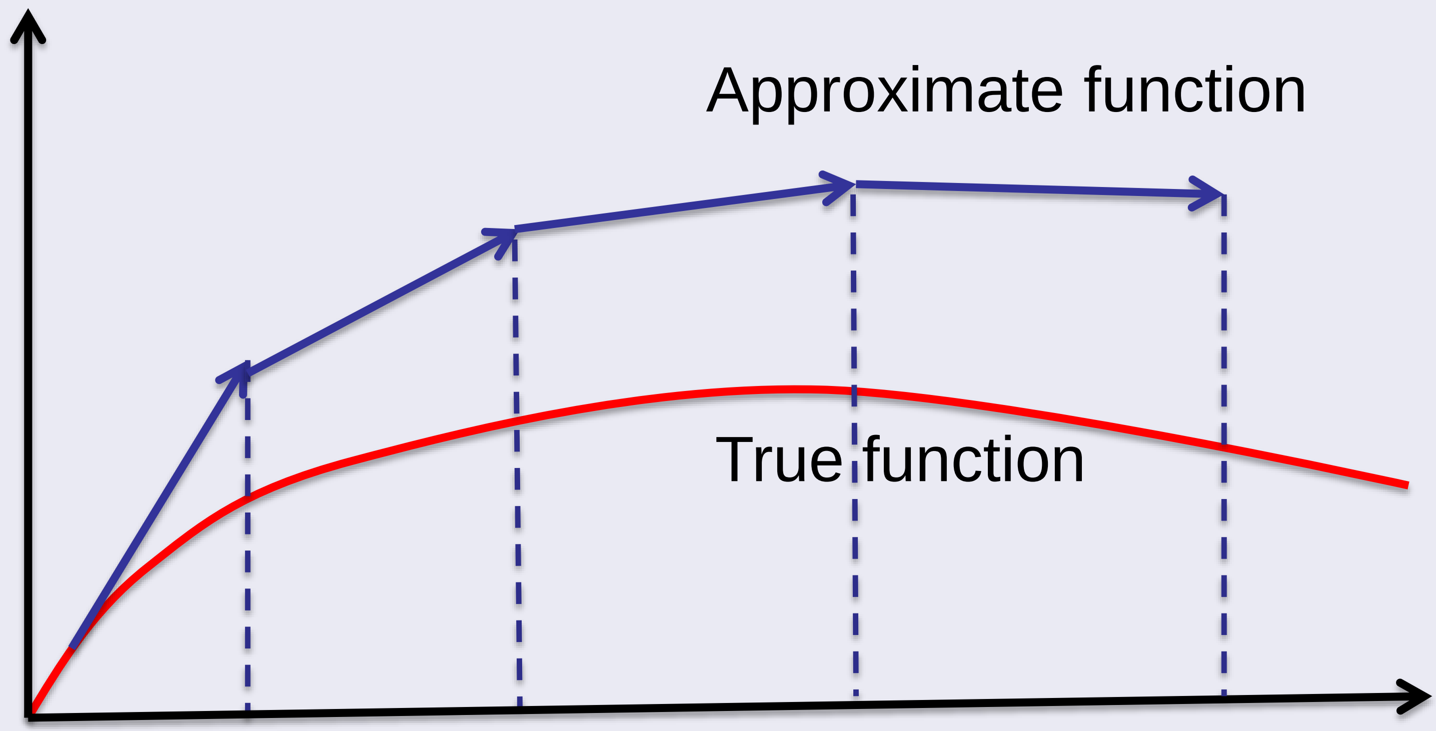
  - *local truncation error* that results from an application method in question over a single step

  - *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 derivatives 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.
  - Euler's method will provide error-free predictions if the underlying function is linear.

Local vs. Global Error



Stability of Euler Method

- Euler's method is *conditionally stable*, depending on the size of h.
- The solution may oscillate or diverge for values of h above a critical value

Example

Consider $y' = -y$ with IV $y(0) = y_0 = 1$

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

$$y_1 = y_0 - y_0 h = (1 - h)$$
$$y_2 = (1 - h)y_1 = (1 - h)^2$$
$$y_n = (1 - h)^n$$

- If $|1 - h| < 1$ the solution converges (stable)
- If $|1 - h| > 1$ the solution diverges (unstable)

True solution $y = e^{-t}$ convergent

Coding of Euler Method

- Solve the equation $y' = -y,\ y(0) = 1$

- Define a function: name it **myeuler**

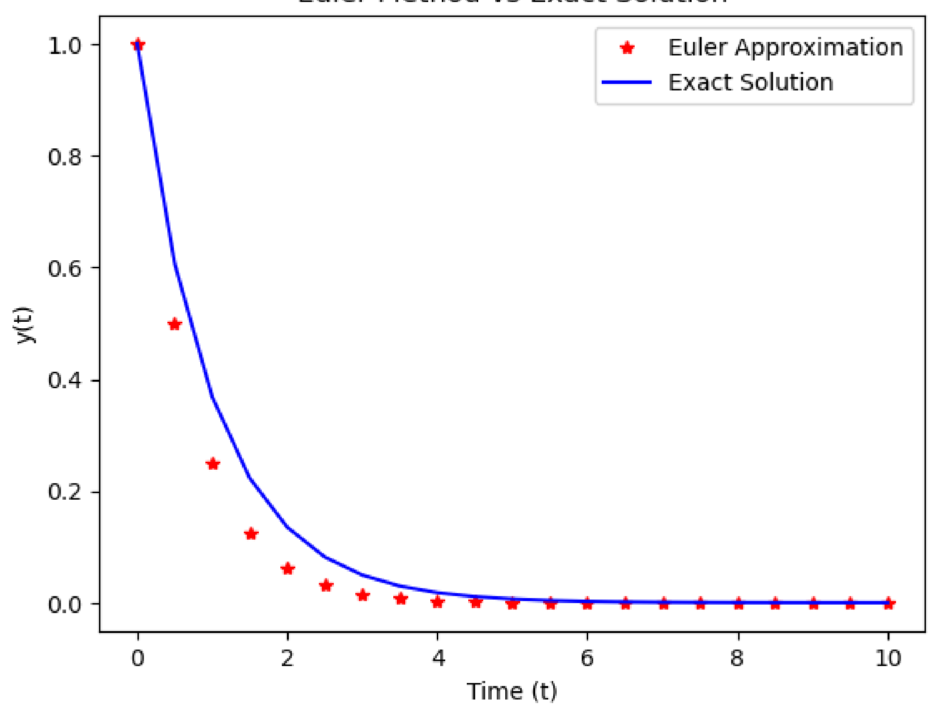
- Enter initial conditions, h step and initial and final times, say 0 to 10.

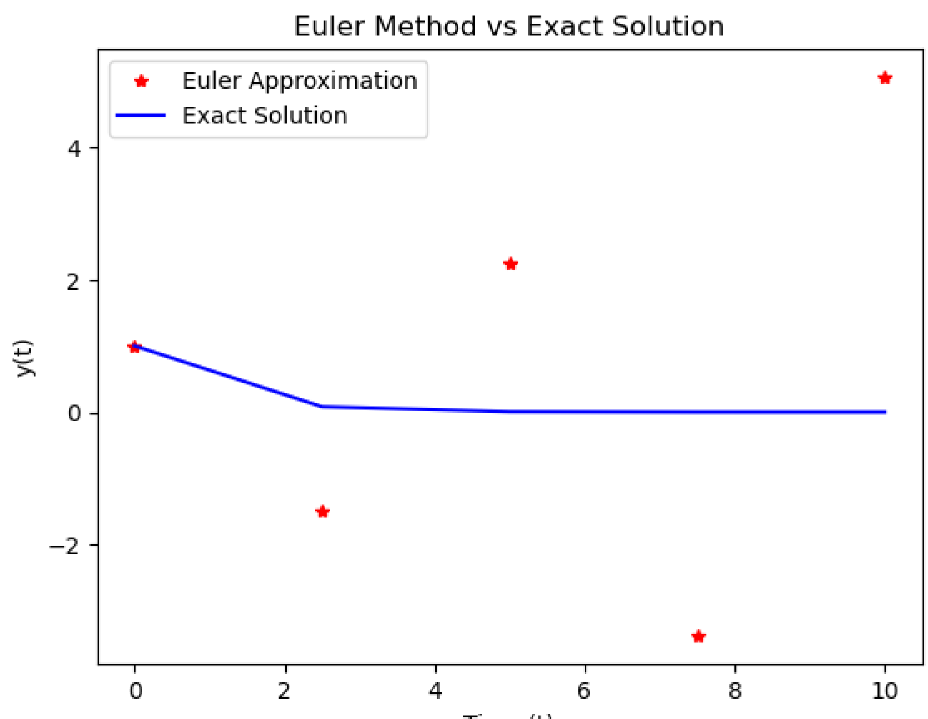
Application of Euler Code

Write a code that calls the euler function to solve $y' = -y,\ y(0) = 1$ and plot the output together with the exact solution.

```python
import numpy as np
def eulode(dydt,tspan,y0,h,*args):
    """
    solve initial-value single ODEs with the Euler method
    input:
    dydt = function name that evaluates the derivative
    tspan = array of [ti,tf] where
    ti and tf are the initial and final values
    of the independent variable
    y0 = initial value of the dependent variable
    h = step size
    *args = additional argument to be passed to dydt
    output:
    t = an array of independent variable values
    y - an array of dependent variable values
    """
    ti = tspan[0] ; tf = tspan[1]
    if not(tf>ti+h):
        return 'upper limit must be greater than lower limit'
    t = []
    t.append(ti) # start the t array with ti
    nsteps = int((tf-ti)/h)
    for i in range(nsteps): # add the rest of the t values
        t.append(ti+(i+1)*h)
    n = len(t)
    if t[n-1] < tf: # check if t array is short of tf
        t.append(tf)
        n = n+1
    y = np.zeros((n)) ; y[0] = y0 # initialize y array
    for i in range(n-1):
        y[i+1] = y[i] + dydt(t[i],y[i],*args)*(t[i+1]-t[i]) # Euler step
    return t,y
```

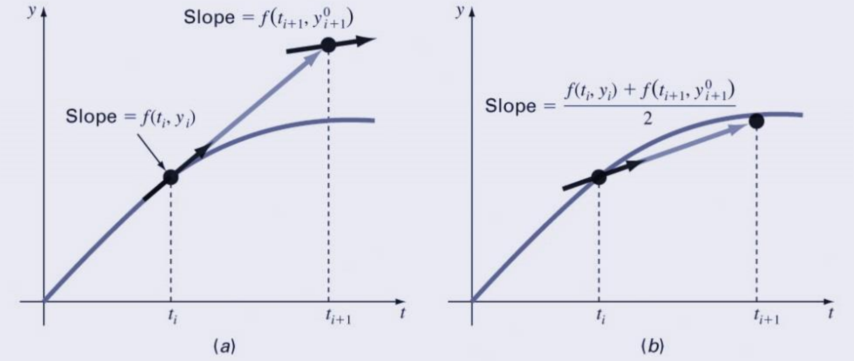
```python
import numpy as np
import matplotlib.pyplot as plt

def dydt(x, y):
    """Subfunction representing the differential equation."""
    return -y

def myeuler():
    """Implementation of the Euler method."""
    t0 = 0
    y0 = 1
    h = 0.5
    tf = 10.0
    t = t0
    y = y0
    tspan = [t0, tf]
    t,y = eulode(dydt,tspan,y0,h)
    #print (t, y)
    t= np.array(t)
    y= np.array(y)
    Ye = np.exp(-t)  # Analytical solution

    plt.plot(t, y, '*', label='Euler Approximation')
    plt.plot(t, Ye, '-b', label='Exact Solution')
    plt.xlabel('Time (t)')
    plt.ylabel('y(t)')
    plt.legend()
    plt.title('Euler Method vs Exact Solution')
    plt.show()
# Run the Euler method
myeuler()
```


Euler Approximation
Exact Solution
1.0
0.8
0.6
0.4
0.2
0.0
y(t)
0
2
4
6
8
10
Time (t)


Euler Method vs Exact Solution
Euler Approximation
Exact Solution
4
2
0
−2
y(t)
0
2
4
6
8
10

2. Heun's (Improved Euler, rk2) Method

- One way to improve Euler's method is to determine derivatives at the beginning and end of the interval and average them:



$$y^0_{i+1} = y_i + f(t_i, y_i)h \qquad \text{(Euler Step)}$$

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

- Implemented as:

$$\mathbf{k1 = f(t, y);\ k2 = f(t + h, y + k1 * h);}$$

$$\mathbf{y = y + \frac{1}{2}(k1 + k2) * h}$$

This is equivalent to taking the average slope at (t_n, y_n) and (t_{n+1}, y_{n+1}) and set the average equal to $\frac{y_{n+1}-y_n}{h}$. That is

$$\frac{y_{n+1} - y_n}{h} = \frac{f(t_n, y_n) + f(t_{n+1}, y_{n+1})}{2}$$

Hence

$$y_{n+1} = y_n + \frac{f(t_n, y_n) + f(t_{n+1}, y_{n+1})}{2} h$$

Since y_{n+1} arises on the RHS we have to find a way to evaluate it.

This is done using Euler step on the RHS.

$$\boldsymbol{k1 = f(t_n, y_n)}$$
$$\boldsymbol{k2 = f(t_n + h, y_n + k1 * h)}$$
$$\boldsymbol{y_{n+1} = y_n + 0.5(k1 + k2)h}$$

Note ϕ in this case is

$$\boldsymbol{\phi = \frac{1}{2}(k1 + k2)}$$

Coding of Heuen Method

- Solve the equation $y' = -y,\ y(0) = 1$

- Define a function : name myheun

- Enter initial conditions, h step and initial and final times

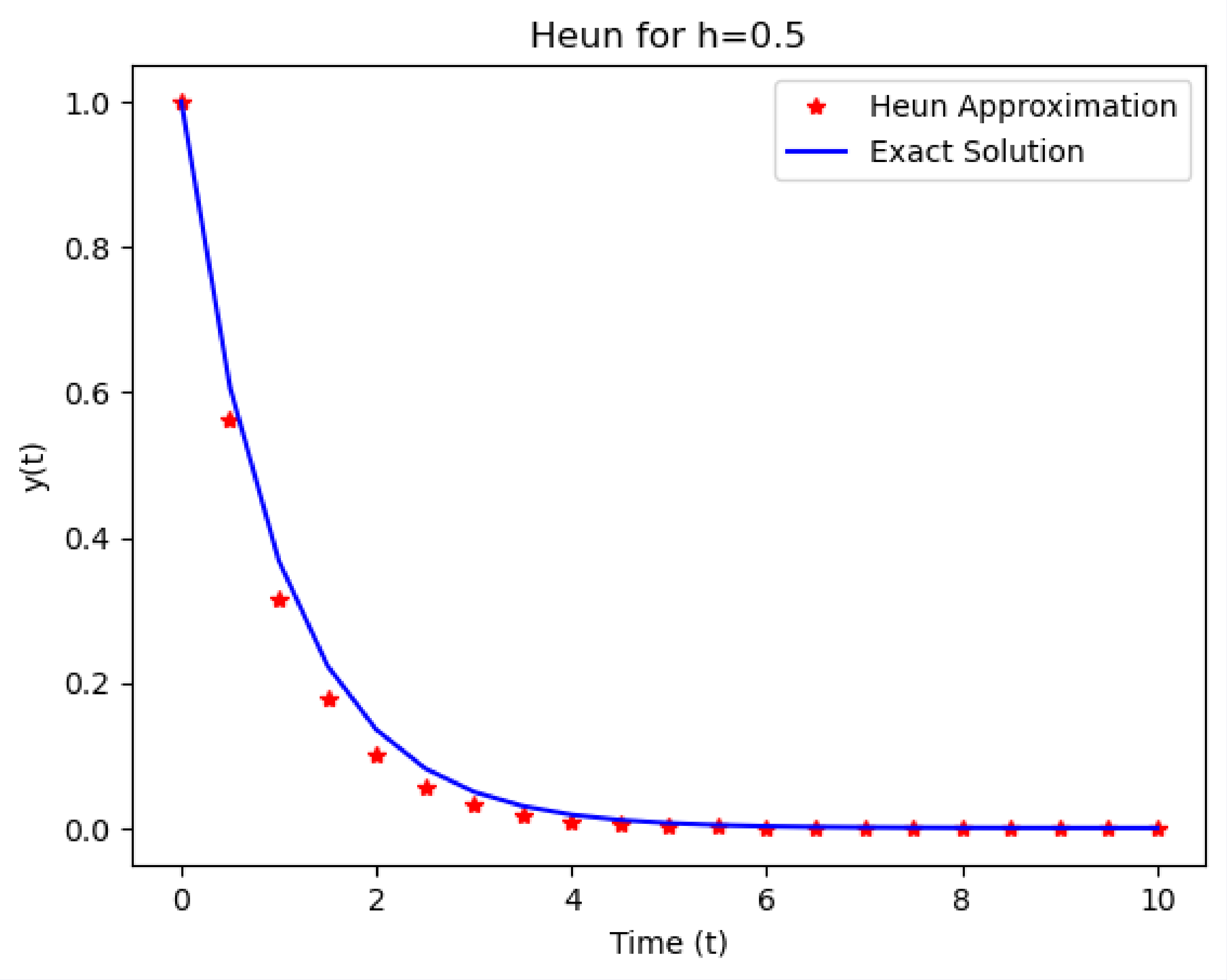
- Plot y vs. t

Application of Heun

Write a code that calls the heun function to solve $y' = -y, \;\; y(0) = 1$ and plot the output together with the exact solution.

```python
import numpy as np
def Heun(dydt,tspan,y0,h,*args):
    """
    solve initial-value single ODEs with the Heun method

    output:
    t = an array of independent variable values
    y - an array of dependent variable values
    """
    ti = tspan[0] ; tf = tspan[1]
    if not(tf>ti+h):
        return 'upper limit must be greater than lower limit'
    t = []
    t.append(ti) # start the t array with ti
    nsteps = int((tf-ti)/h)
    for i in range(nsteps): # add the rest of the t values
        t.append(ti+(i+1)*h)
    n = len(t)
    if t[n-1] < tf: # check if t array is short of tf
        t.append(tf)
        n = n+1
    y = np.zeros((n)) ; y[0] = y0 # initialize y array
    for i in range(n-1):
        hh = t[i+1] - t[i]
        k1 = dydt(t[i],y[i],*args)
        ymid = y[i] + k1*hh/2.
        k2 = dydt(t[i]+hh/2.,ymid,*args)
        phi = (k1 + k2 )/2.
        y[i+1] = y[i] + phi*hh
    return t,y
```

```python
import numpy as np
import matplotlib.pyplot as plt

def dydt(x, y):
    """Subfunction representing the differential equation."""
    return -y

def myheun():
    """Implementation of the Euler method."""
    t0 = 0
    y0 = 1
    h = .5
    tf = 10.0
    t = t0
    y = y0
    tspan = [t0, tf]
    t,y = Heun(dydt,tspan,y0,h)
    #print (t, y)
    t= np.array(t)
    y= np.array(y)
    Ye = np.exp(-t)  # Analytical solution

    plt.plot(t, y, '*r', label='Heun Approximation')
    plt.plot(t, Ye, '-b', label='Exact Solution')
    plt.xlabel('Time (t)')
    plt.ylabel('y(t)')
    plt.legend()
    plt.title('Heun for h=0.5')
    plt.show()
# Run the Euler method
myheun()
```



Heun for h=0.5
Heun Approximation
Exact Solution
y(t)
Time (t)
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10

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

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

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

Example

For the equation $y' = \cos(t + y)$ write RK2 step in full.

$$K1 = \cos(t_n + y_n)$$
$$K2 = \cos(t_n + h + y_n + \cos(t_n + y_n)\, h)$$

$$y_{n+1} = y_n + \frac{h}{2}[\cos(t_n + y_n) + \cos(t_n + h + y_n + \cos(t_n + y_n)\, h)]$$

If $y(0) = 0$, analytic solution is
$$y = 2\tan^{-1} t - t$$

3. Classical Fourth-Order Runge-Kutta Method

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

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

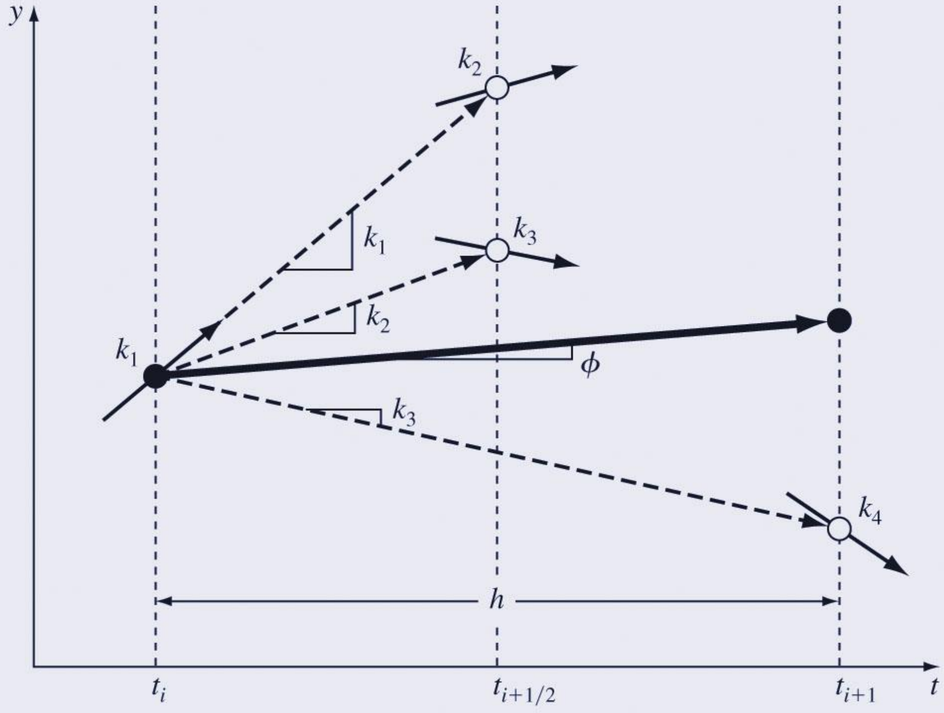
where:

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

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

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

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



The slope K1 at t is used to predict y at mid-point t+h/2 and the slope there K2.

K2 is used at t to predict y at mid-point t+h/2 and slope there K3.

K3 is used at t to predict y at t+h and the slope there K4.

The slopes K1 and K4 used once and K2 and K3 used twice, hence their waits are 1 and 2 respectively.

Coding of rk4 Method

- Solve the equation $y' = -y,\ \ y(0) = 1$

- Define a function: name myrk4

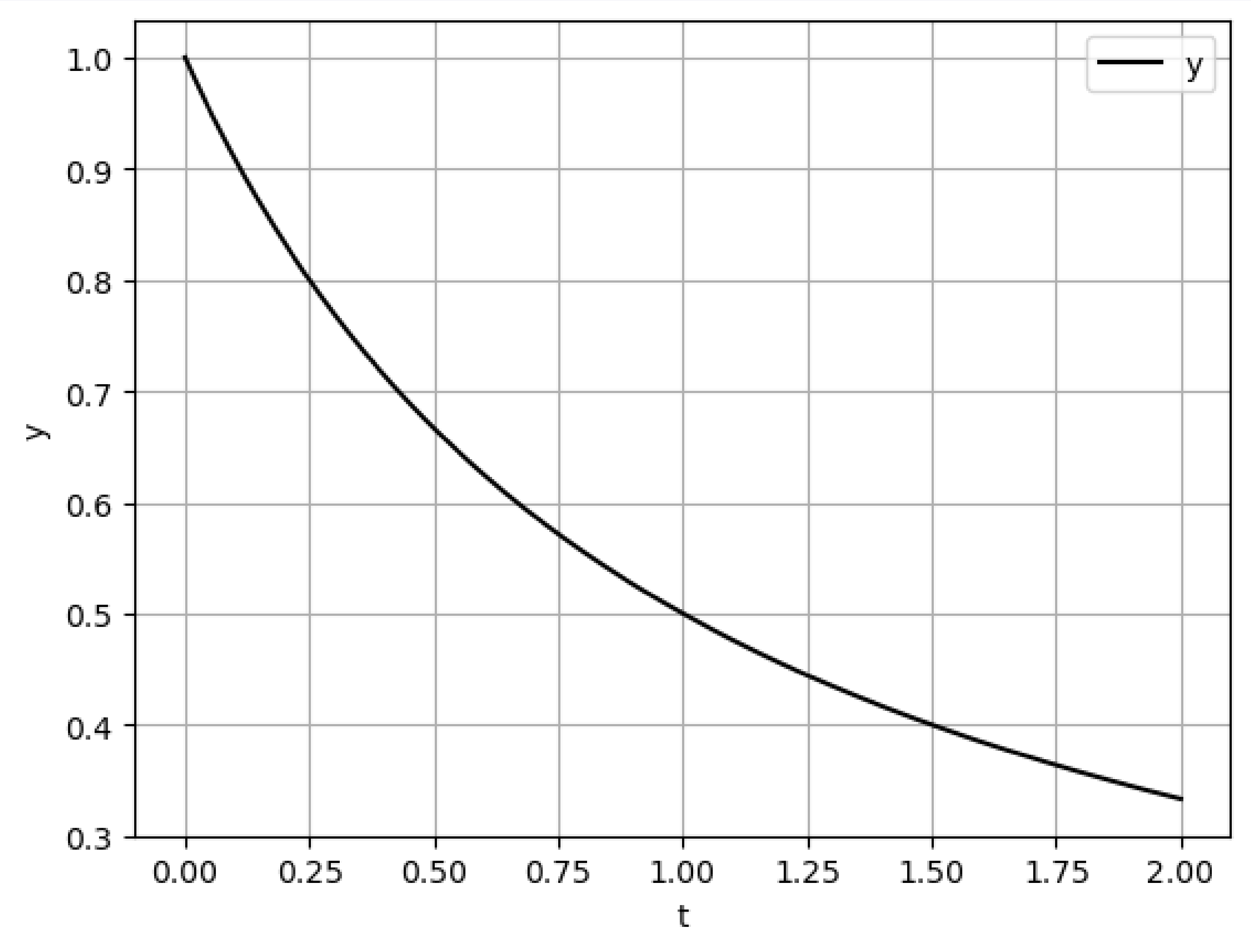
- Enter initial conditions, h step and initial and final times

```python
import numpy as np
def rk4sys(dydt,tspan,y0,h= -1.,*args):

    if np.any(np.diff(tspan) < 0):
        return 'tspan times must be ascending'
    # check if only ti and tf spec'd and no value for h
    if len(tspan) == 2 and h != -1.:
        ti = tspan[0] ; tf = tspan[1]
        nsteps = int((tf-ti)/h)
        t = []
        t.append(ti)
        for i in range(nsteps): # add the rest of the t values
            t.append(ti+(i+1)*h)
        n = len(t)
        if t[n-1] < tf: # check if t array is short of tf
            t.append(tf)
            n = n+1
    else:
        n = len(tspan) # here if tspan contains step times
        t = tspan
    neq = len(y0)
    y = np.zeros((n,neq)) # set up 2-D array for dependent variables
    for j in range(neq):
        y[0,j] = y0[j] # set first elements to initial conditions
    for i in range(n-1): # 4th order RK
        hh = t[i+1] - t[i]
        k1 = dydt(t[i],y[i,:],*args)
        ymid = y[i,:] + k1*hh/2.
        k2 = dydt(t[i]+hh/2.,ymid,*args)
        ymid = y[i,:] + k2*hh/2.
        k3 = dydt(t[i]+hh/2.,ymid,*args)
        yend = y[i,:] + k3*hh
        k4 = dydt(t[i]+hh,yend,*args)
        phi = (k1 + 2.*(k2+k3) + k4)/6.
        y[i+1,:] = y[i,:] + phi*hh
    return t,y
```

Application of rk4 General Code

- Write a code that calls the rk4 function to solve $y' = -y^2,\ y(0) = 1$ and plot the output together with the exact solution.

```python
def dydtsys(t,y):
    dy = -y**2
    return dy
h = 0.01
ti = 0. ; tf = 2.
tspan = [0.,2.]
y0 = np.array([1])
t,y = rk4sys(dydtsys,tspan,y0,h)

import pylab
pylab.plot(t,y,c='k',label='y')
pylab.grid()
pylab.xlabel('t')
pylab.ylabel('y')
pylab.legend()
```


1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
y
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
t
y

4. 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*.

Array Format

Write

$$\mathrm{y} = [y_1; y_2; \ldots \ldots; y_n]$$
$$\mathrm{F} = [f_1; f_2; \ldots \ldots; f_3]$$

Then the system of equations become

$$\frac{d\mathrm{y}}{dt} = \mathrm{F}$$

This is a first order vector equation

Example

A system of equations is given by

$$\frac{dx}{dt} = x - y + tz,$$
$$\frac{dy}{dt} = -yxz,$$
$$\frac{dz}{dt} = e^t \sin(xy)$$

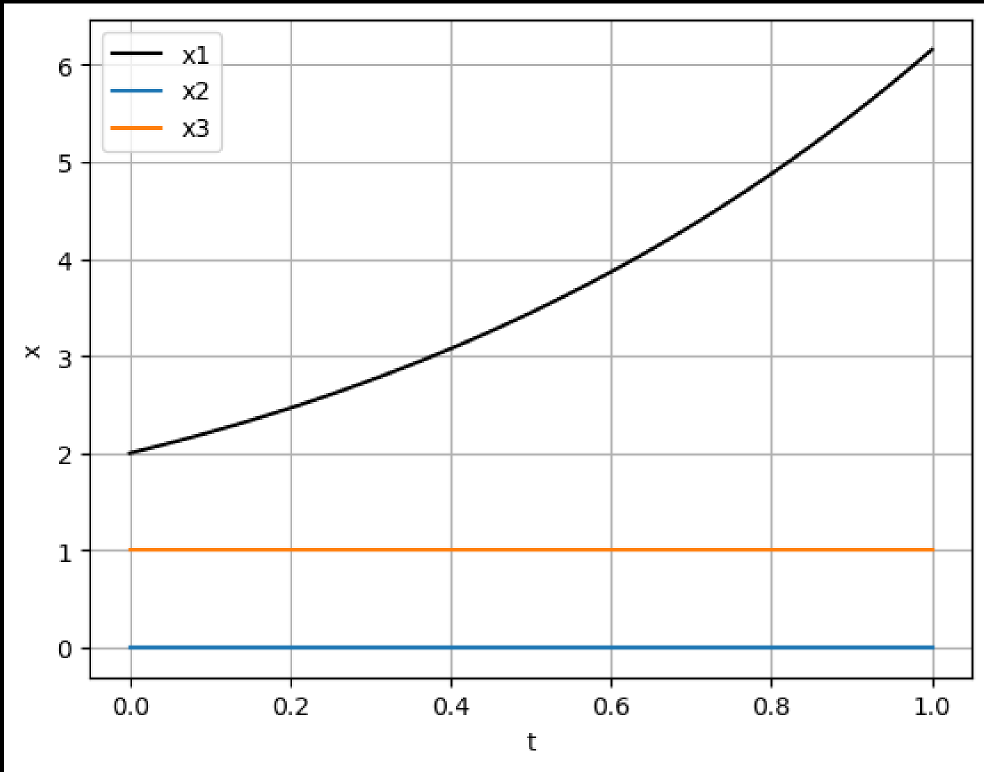
Write $x = x_1, \quad y = x_2, \;\; z = x_3$

The system becomes

$$\frac{dx_1}{dt} = x_1 - x_2 + tx_3$$
$$\frac{dx_2}{dt} = -x_1x_2x_3$$
$$\frac{dx_3}{dt} = e^t \sin(x_1x_2)$$
$$\mathbf{x} = [x_1; x_2; x_3]$$

$$\boldsymbol{f}(t, \mathbf{x}) = [x_1 - x_2 + tx_3; -x_1x_2x_3; e^t \sin(x_1x_2)]$$
$$\frac{d\mathbf{x}}{dt} = \boldsymbol{f}(t, \mathbf{x})$$

```python
def dxdtsys(t,x):
    n = len(x)
    dx = np.zeros((n))
    dx[0]=x[0]-x[1]+t*x[2]
    dx[1]=-x[0]*x[1]*x[2]
    dx[2]=np.exp(t)*np.sin(x[0]*x[1])
    return dx
h = 0.01
ti = 0. ; tf = 3.
tspan = [0.,1.]
x0 = np.array([2.,0.,1])
t,x = rk4sys(dxdtsys,tspan,x0,h)

import pylab
pylab.plot(t,x[:,0],c='k',label='x1')
pylab.plot(t,x[:,1],label='x2')
pylab.plot(t,x[:,2],label='x3')

pylab.grid()
pylab.xlabel('t')
pylab.ylabel('y')
pylab.legend()
```

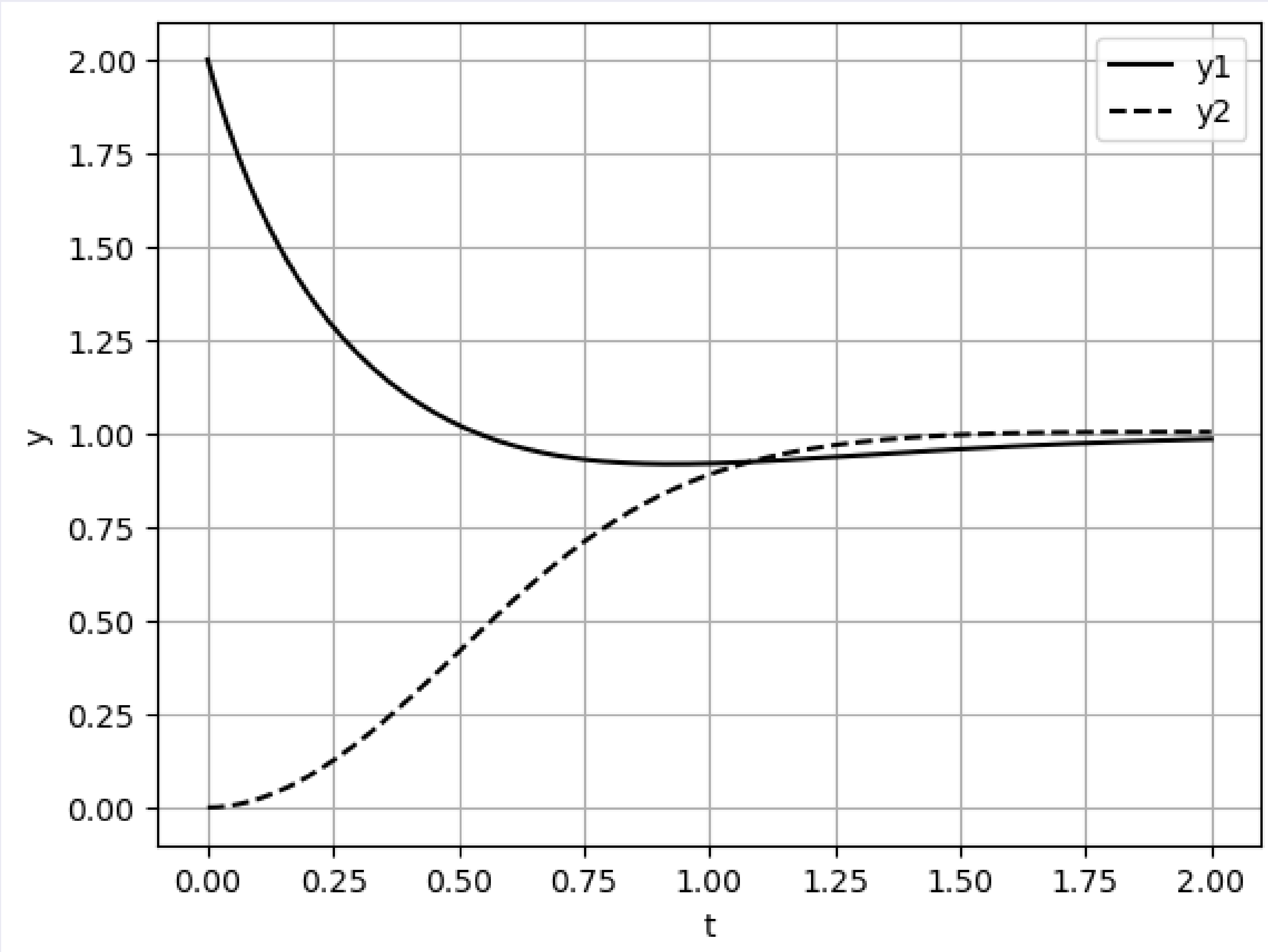


Employ the `rk4sys` function to solve the following coupled differential equations:

$$\frac{dy_1}{dt} = -2y_1^2 + 2y_1 + y_2 - 1 \qquad y_1(0) = 2$$
$$\frac{dy_2}{dt} = -y_1 - 3y_2^2 + 2y_2 + 2 \qquad y_2(0) = 0$$

over the domain $0 \leq t \leq 2$ with a step size of $h = 0.01$.

```python
def dydtsys(t,y):
    n = len(y)
    dy = np.zeros((n))
    dy[0] = -2.*y[0]**2 +2.*y[0] + y[1] - 1.
    dy[1] = -y[0] -3*y[1]**2 +2.*y[1] + 2.
    return dy
h = 0.01
ti = 0. ; tf = 2.
tspan = [0.,2.]
y0 = np.array([2.,0.])
t,y = rk4sys(dydtsys,tspan,y0,h)

import pylab
pylab.plot(t,y[:,0],c='k',label='y1')
pylab.plot(t,y[:,1],c='k',ls='--',label='y2')
pylab.grid()
pylab.xlabel('t')
pylab.ylabel('y')
pylab.legend()
```


2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
y
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
t
y1
y2

Higher Order to First Order

- Consider the higher order equation

$$\frac{d^3x}{dt^3} = f(t, x, \frac{dx}{dt}, \frac{d^2x}{dt^2})$$

- Let $x = x_1$, $\frac{dx}{dt} = x_2$, $\frac{d^2x}{dt^2} = x_3$, then eqn. is

$$\frac{dx_1}{dt} = x_2, \quad \frac{dx_2}{dt} = x_3, \quad \frac{dx_3}{dt} = f(t, x_1, x_2, x_3)$$

- Using vector $\mathbf{x} = [x_1; x_2; x_3]$ we have

$$\frac{d\mathbf{x}}{dt} = \mathbf{F}(t, \mathbf{x}) \ , \quad \mathbf{F} = [x_2, ; x_3, ; f(t, x_1, x_2, x_3)]$$

Example: Projectile

Consider a projectile moving under linear air resistance

The equations of motion are

$$\ddot{x} = -k\dot{x}$$
$$\ddot{y} = -g - k\dot{y}$$

The initial conditions are

$$t = 0, (x, y) = (0,0),$$
$$(\dot{x}, \dot{y}) = v_0(\cos(\theta), \sin(\theta))$$

This is analytically soluble with

$$x = \frac{v_0 \cos(\theta)}{k} \left(1 - e^{-kt}\right)$$

$$y = -\frac{g}{k} t + \frac{1}{k}\left(v_0 \sin(\theta) + \frac{g}{k}\right)\left(1 - e^{-kt}\right)$$

And

$$\dot{x} = v_0 \cos(\theta)\, e^{-kt}$$

$$\dot{y} = -\frac{g}{k} + \left(v_0 \sin(\theta) + \frac{g}{k}\right) e^{-kt}$$

Max height is obtained by setting $\dot{y} = 0$

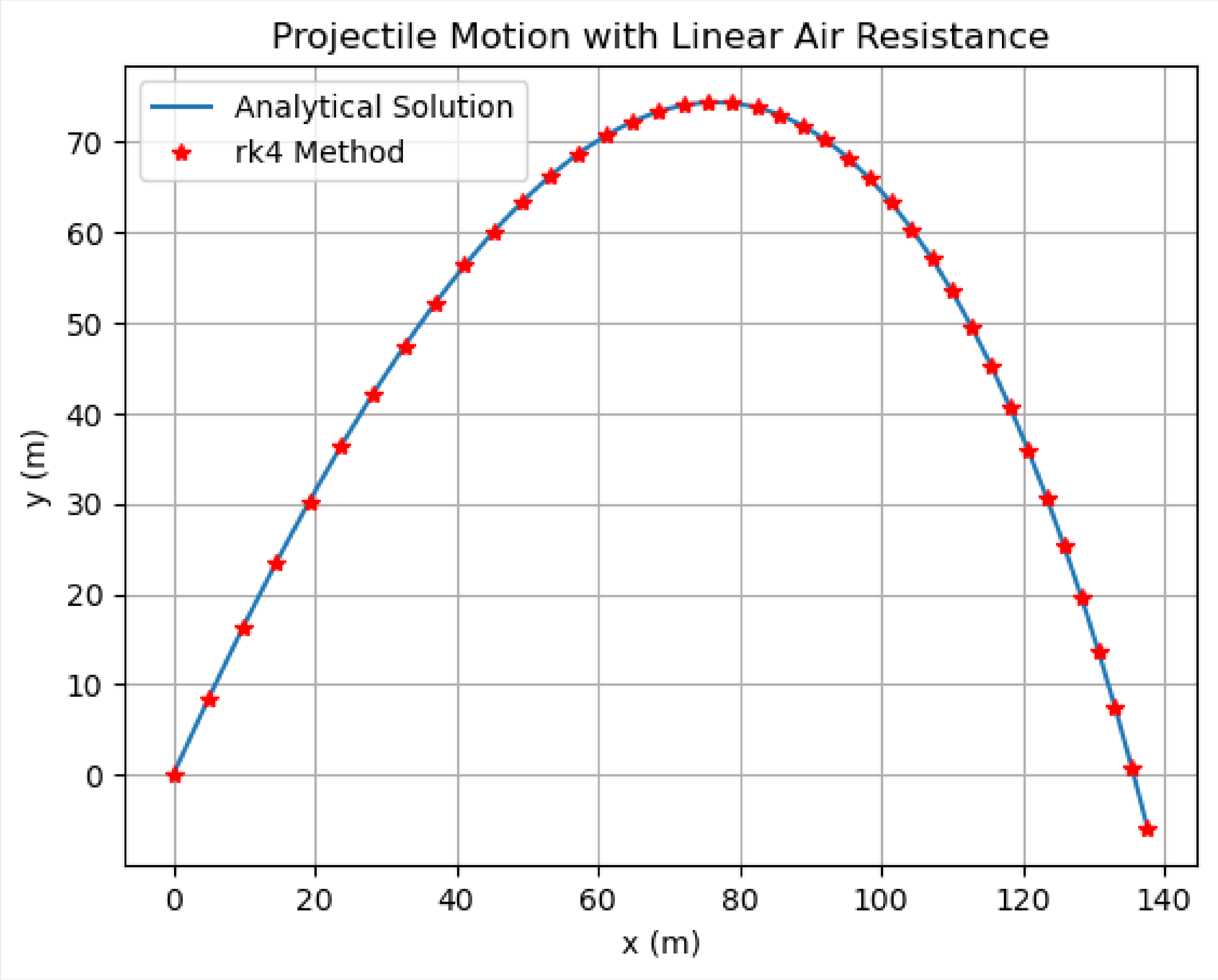
Range is obtained by setting $y(t) = 0$

How do we obtain max range?

System of 1st Order Equations

- $\boldsymbol{y_1 = x}$, $\boldsymbol{y_2 = \dot{x}}$, $\boldsymbol{y_3 = y}$, $\boldsymbol{y_4 = \dot{y}}$
- Equations are
- $\frac{dy_1}{dt} = y_2$ $\frac{dy_2}{dt} = -ky_2$
- $\frac{dy_3}{dt} = y_4$ $\frac{dy_4}{dt} = -g - ky_4$
- $\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \\ \dot{y}_4 \end{bmatrix} = \begin{bmatrix} yp(1) \\ yp(2) \\ yp(3) \\ yp(4) \end{bmatrix} = \begin{bmatrix} y_2 \\ -ky_2 \\ y_4 \\ -g - ky_4 \end{bmatrix}$

```python
def f(t, y, p):
    k, g = p
    yp = np.zeros(4)
    yp[0] = y[1]
    yp[1] = -k * y[1]
    yp[2] = y[3]
    yp[3] = -g - k * y[3]
    return yp

# Main script
v0 = 50
theta = 60
k = 0.1
g = 9.8
p = [k, g]
tspan = [0, 8]
y0 = [0, v0 * np.cos(np.radians(theta)), 0, v0 * np.sin(np.radians(theta))]
step = 0.2

# Solve using rk4 method
T, YY = rk4sys(f, tspan, y0, step, p)
T = np.array(T)  # Ensure T is a NumPy array

# Analytical solution for comparison
xe = (v0 * np.cos(np.radians(theta)) * (1 - np.exp(-k * T))) / k
ye = (-g * T / k) + ((v0 * np.sin(np.radians(theta)) + g / k) * (1 - np.exp(-k * T))) / k
# Plot results
plt.plot(xe, ye, '-', label='Analytical Solution')
plt.plot(YY[:, 0], YY[:, 2], '*r', label=‘rk4 Method')
plt.xlabel('x (m)')
plt.ylabel('y (m)')
plt.legend()
plt.title('Projectile Motion with Linear Air Resistance')
plt.grid()
plt.show()
```


Projectile Motion with Linear Air Resistance
Analytical Solution
rk4 Method
y (m)
x (m)
0
10
20
30
40
50
60
70
0
20
40
60
80
100
120
140

Assignment

a. Use incsearch and fsolve to find the range for the example of linear resistance analytic solution. Also calculate the maximum height.

b. Find maximum height and range for the numerical solution of linear air resistance

c. Compare the two solutions