VI. Ordinary Differential Equations

The numerical methods to be discussed in this section are applied to solve ordinary differential equations (ODE) to obtain particular solutions at given initial conditions. This type of problems is called *initial value problems (IVP)*. The solution methods are based on a similar principle of calculation of approximate solution by constant increment, i.e. step size. The order of magnitude of the error is related to the step size.

Euler's Method

The Taylor series can be written as

$$y(x+h) = y(x) + y'(x)h + \frac{y''(x)}{2!}h^2 + \frac{y'''(x)}{3!}h^3 + \cdots$$

By truncation the series at the first derivative term, the approximate solution of Euler's method is obtained. Thus

$$y(x+h) = y(x) + y'(x)h$$

The initial conditions in this case should be the value of $y(x)$ at initial x. This method is known as point-slope method because it predicts the next point using the slope $y'(x)$.

Example 1: Find the numerical solution of the following differential equation over the domain [0, 2].

$$y' = xy, \qquad y(0) = 1$$

Analytical Solution: $y = e^{x^2/2}$

Solution: The numerical solution can be programmed by a single for-loop of x with increment of h. In the beginning, define the given equation in an inline function and suppose h = 0.5.

```
dy = lambda x,y: x*y    # dy = function(x, y)
x = 0                   # initial value of x
xn = 2                  # final value of x
y = 1                   # value of y(x0)
h = 0.5                 # step size
n = int((xn-x)/h) # total number of steps
print ('x  \t\t y (Euler)')   # data table header
print ('%f \t %f'% (x,y))     # tabular format output
for i in range(n):
    y += dy(x, y)*h           # calculate next y
    x += h                    # x increment
    print ('%f \t %f'% (x,y))
```

Since the initial values are known, the loop starts form the next x value, and the derivative in this example is function of y only.

The output of the program:

```
x               y (Euler)
0.000000        1.000000
0.500000        1.000000
1.000000        1.250000
1.500000        1.875000
2.000000        3.281250
```

In order to compare the numerical values with the corresponding analytical ones for this example, the analytical solution can be added to the code as following:

```
from math import exp  # exponential function
dy = lambda x,y: x*y
f  = lambda x: exp(x**2/2)   # analytical solution function
x = 0
xn = 2
y = 1
h = 0.5
n = int((xn-x)/h)
print ('x  \t\t y (Euler) \t y (analytical)')
print ('%f \t %f \t %f'% (x,y,f(x)))
for i in range(n):
    y += dy(x, y)*h
    x += h
    print ('%f \t %f \t %f'% (x,y,f(x)))
```

So, the output becomes

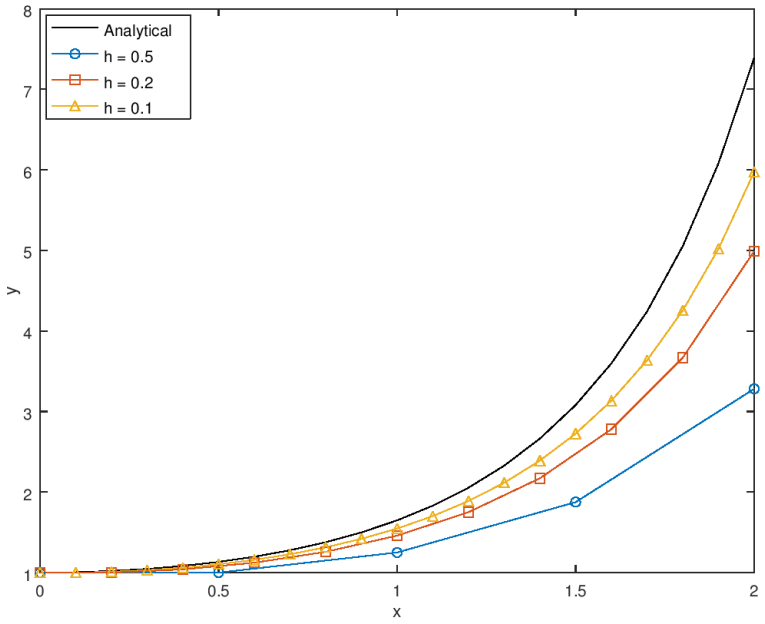
```
x               y (Euler)       y (analytical)
0.000000        1.000000        1.000000
0.500000        1.000000        1.133148
1.000000        1.250000        1.648721
1.500000        1.875000        3.080217
2.000000        3.281250        7.389056
```

It is obvious that there is a large error at each computed numerical value varies from 24.18% at x = 1 to 55.59% at x = 2. If the program is run with h = 0.2, the output will be

```
x               y (Euler)       y (analytical)
0.000000        1.000000        1.000000
0.200000        1.000000        1.020201
0.400000        1.040000        1.083287
0.600000        1.123200        1.197217
0.800000        1.257984        1.377128
1.000000        1.459261        1.648721
1.200000        1.751114        2.054433
1.400000        2.171381        2.664456
1.600000        2.779368        3.596640
1.800000        3.668765        5.053090
2.000000        4.989521        7.389056
```

The error is remarkably reduced to be from 11.49% to 32.47% at the same x- values, respectively. This is visualized in the plot shown below, in addition to the result of h = 0.1, which

shows that the error is still considerably large and more accurate solution will require smaller step size which will normally increase the run time and memory allocation if the computed values are to be saved in arrays (for further calculations or to plot).



Second-Order Runge-Kutta Method

Runge-Kutta (RK) methods are derived from the Taylor series with including the higher derivatives in approximation. The second-order RK method include the second derivative approximation in the following manner.

$$y(x+h) = y(x) + y'\left(x + \frac{h}{2}, y + \frac{h}{2}y'(x,y)\right)h$$

To simplify calculation procedure, the equation can be divided to three steps as

$$K_1 = hy'(x,y)$$

$$K_2 = hy'\left(x + \frac{h}{2}, y + \frac{1}{2}K_1\right)$$

$$y(x+h) = y(x) + K_2$$

The following program solves the previous example at h = 0.5:

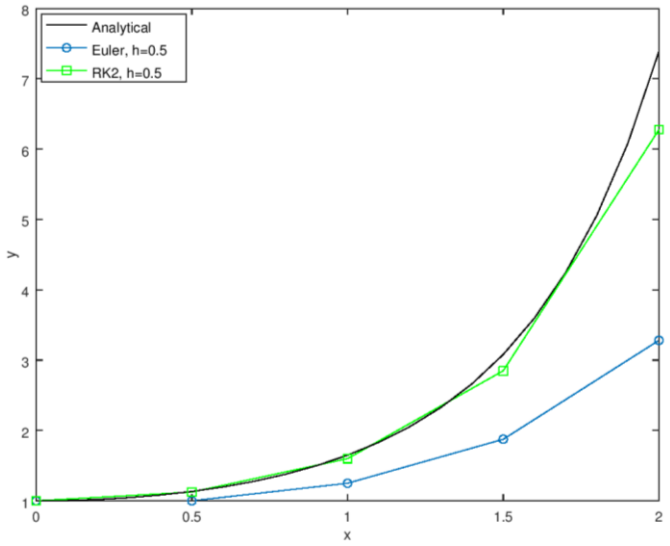
```
from math import exp
dy = lambda x,y: x*y
f  = lambda x: exp(x**2/2)
x = 0
xn = 2
y = 1
h = 0.5
n = int((xn-x)/h)
```

```
print ('x  \t\t y (RK2) \t y (analytical)')
print ('%f \t %f \t %f'% (x,y,f(x)))
for i in range(n):
    K1 = h*dy(x, y)
    K2 = h*dy(x + h/2, y + K1/2)
    y += K2
    x += h
    print ('%f \t %f \t %f'% (x,y,f(x)))
```

Nothing has been changed except the equations of 2nd order RK method inside the for-loop. The results are displayed as

```
x                y (RK2)          y (analytical)
0.000000         1.000000         1.000000
0.500000         1.125000         1.133148
1.000000         1.599609         1.648721
1.500000         2.849304         3.080217
2.000000         6.277373         7.389056
```

The following graph shows a comparison between this solution and Euler's method at the same step size.



Fourth-Order Runge-Kutta Method

This version of RK methods is widely used in solving ordinary differential equations due its relative high accuracy since it takes more higher- derivative terms of the Taylor series into consideration. It is programmable form is as following

$$K_1 = hy'(x, y)$$

$$K_2 = hy'\left(x + \frac{h}{2}, y + \frac{1}{2}K_1\right)$$

$$K_3 = hy'\left(x + \frac{h}{2}, y + \frac{1}{2}K_2\right)$$

$$K_4 = hy'(x + h, y + K_3)$$

$$y(x+h) = y(x) + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4)$$

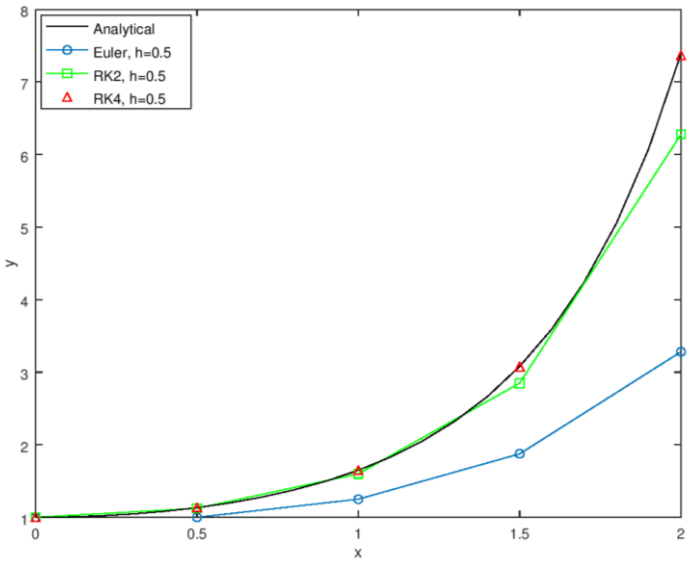
The following program solves the previous example at h = 0.5:

```
from math import exp
dy = lambda x,y: x*y
f  = lambda x: exp(x**2/2)
x = 0
xn = 2
y = 1
h = 0.5
n = int((xn-x)/h)
print ('x  \t\t y (RK4) \t y (analytical)')
print ('%f \t %f \t %f'% (x,y,f(x)))
# main loop of x values
for i in range(n):
    K1 = h*dy(x, y)
    K2 = h*dy(x + h/2, y + K1/2)
    K3 = h*dy(x + h/2, y + K2/2)
    K4 = h*dy(x +h, y + K3)
    y += (K1 + 2*K2 + 2*K3 + K4)/6
    x += h
    print ('%f \t %f \t %f'% (x,y,f(x)))
```

The output of the program:

```
x                y (RK4)          y (analytical)
0.000000         1.000000         1.000000
0.500000         1.133138         1.133148
1.000000         1.648528         1.648721
1.500000         3.077976         3.080217
2.000000         7.366803         7.389056
```

The accuracy can be simply noticed by comparing the numerical values of RK4 solution with the analytical values. The maximum error percentage is 0.3% at x = 2. Graphically, the points of 4th order RK solution are coinciding with the curve of the analytical solution.



Example 2: By using RK4, find the numerical solution of the following differential equation for the domain [2, 4] and plot the both solutions. Use step size h = 0.1.

$$y' = y + 3, \qquad y(2) = -2$$

Analytical Solution: $y = e^{x-2} - 3$

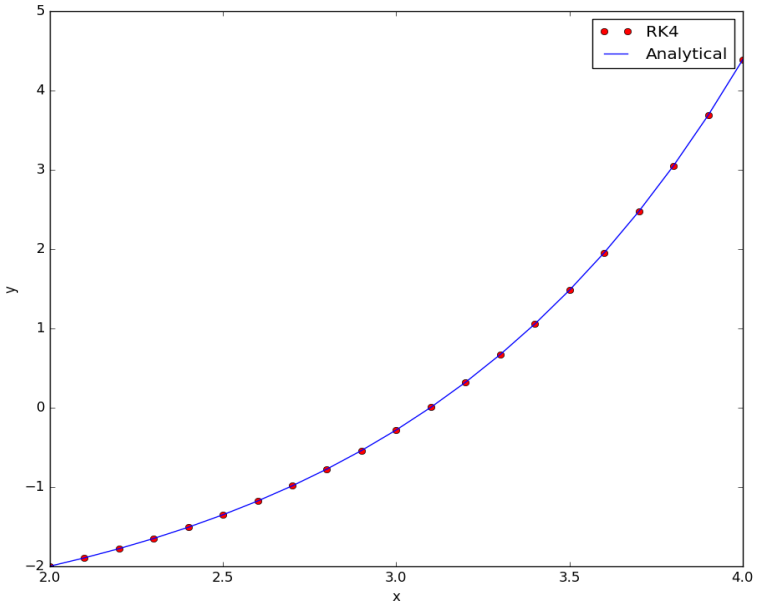
Solution: Although the equation does not contain x in its left hand side, there are no changes required in the RK4 calculations part. To plot the results, arrays x1, y1 and f1 are used to save the solutions for plot functions. Lines added for plotting purpose are those in blue.

```
import numpy as np      # array module
import matplotlib.pyplot as plt # plot module
dy = lambda x,y: y+3
f = lambda x: np.exp(x-2) - 3
x = 2
xn = 4
y = -2
h = 0.1
n = int((xn-x)/h)
xp = np.linspace(x,xn,n+1)    # x array for plot
yp = np.empty(n+1, float)     # y array for plot
yp[0] = y
print ('x  \t\t y (RK4) \t y (analytical)')
print ('%f \t %f \t %f'% (x,y,f(x)))
# main loop
for i in range(1,n+1):
    K1 = h*dy(x, y)
    K2 = h*dy(x + h/2, y + K1/2)
    K3 = h*dy(x + h/2, y + K2/2)
    K4 = h*dy(x +h, y + K3)
    y += (K1 + 2*K2 + 2*K3 + K4)/6
    yp[i] = y
    x += h
    print ('%f \t %f \t %f'% (x,y,f(xp)))
# Plot fuctions
plt.plot(xp,yp,'ro', xp,f(xp))
plt.xlabel('x')
plt.ylabel('y')
plt.legend(['RK4', 'Analytical'])
plt.show()
```

The output:

```
x               y (RK4)         y (analytical)
2.000000        -2.000000       -2.000000
2.100000        -1.894829       -1.894829
2.200000        -1.778597       -1.778597
2.300000        -1.650142       -1.650141
2.400000        -1.508176       -1.508175
2.500000        -1.351279       -1.351279
2.600000        -1.177882       -1.177881
```

```
2.700000        -0.986248       -0.986247
2.800000        -0.774460       -0.774459
2.900000        -0.540399       -0.540397
3.000000        -0.281720       -0.281718
3.100000        0.004163        0.004166
3.200000        0.320114        0.320117
3.300000        0.669293        0.669297
3.400000        1.055196        1.055200
3.500000        1.481684        1.481689
3.600000        1.953026        1.953032
3.700000        2.473940        2.473947
3.800000        3.049639        3.049647
3.900000        3.685885        3.685894
4.000000        4.389045        4.389056
```



Higher-Order Ordinary Differential Equations

The solution of higher-order ODE's (2nd and more) is conducted by reduction to a system of first order equations that are solved simultaneously by using the methods discussed above. It should be noticed that the number of initial conditions should equal to the order of the given equation. In this section, the solution of second order ODE by using RK4 is to be discussed, but the same procedure can be applied for higher orders.

To solve a second order ODE in the form $y'' = f(x, y, y')$, it should be divided into two first order ODE's as following

$$y' = u$$

$$u' = f(x, y, u)$$

These two equations can be solved simultaneously by using RK4 by setting the calculation procedure in the following sequence:

$$L_1 = hu'(x, y, u)$$

$$K_1 = hy'(x, y, u)$$

$$L_2 = hu'\left(x + \frac{h}{2}, y + \frac{1}{2}K_1, u + \frac{1}{2}L_1\right)$$
$$K_2 = hy'\left(x + \frac{h}{2}, y + \frac{1}{2}K_1, u + \frac{1}{2}L_1\right)$$

$$L_3 = hu'\left(x + \frac{h}{2}, y + \frac{1}{2}K_2, u + \frac{1}{2}L_2\right)$$
$$K_3 = hy'\left(x + \frac{h}{2}, y + \frac{1}{2}K_2, u + \frac{1}{2}L_2\right)$$

$$L_4 = hu'(x + h, y + K_3, u + L_3)$$
$$K_4 = hy'(x + h, y + K_3, u + L_3)$$

$$u(x + h) = u(x) + \frac{1}{6}(L_1 + 2L_2 + 2L_3 + L_4)$$
$$y(x + h) = y(x) + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4)$$

Example 3: find the solution of the following 2nd order ODE over the domain [π, 2 π]

$$y'' + y = 4x + 10\sin x, \qquad y(\pi) = 0, y'(\pi) = 2$$

Analytical Solution: $y = 9\pi\cos x + 7\sin x + 4x - 5x\cos x$

Solution: First, the second order equation is expressed as a system of two first order equations

$$y' = u$$
$$u' = 4x + 10\sin x - y$$

Second, the system is to be solved simultaneously by setting the RK4 equations inside the for-loop as shown in the program:

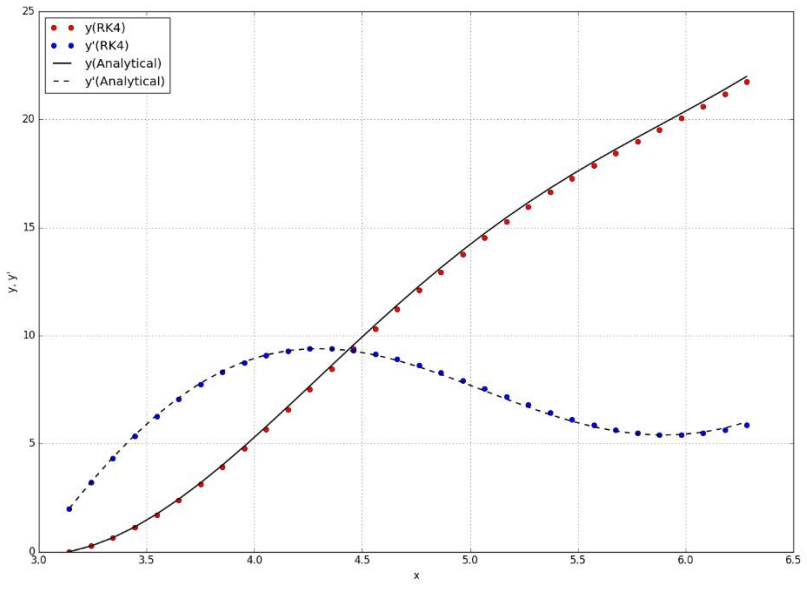
```
import numpy as np
import matplotlib.pyplot as plt
# system of the first order ODE's
dy = lambda x, y, u: u
du = lambda x, y, u: 4*x + 10 * np.sin(x) - y
# the analytical solution
f = lambda x: 9*np.pi*np.cos(x) + 7*np.sin(x) + 4*x - 5*x*np.cos(x)
df = lambda x: -9*np.pi*np.sin(x) + 7 * np.cos(x) + 4 - 5*(np.cos(x) -
x*np.sin(x))
```

```
# initial values
x = np.pi
xn = 2*np.pi
y = 0.0
u = 2
h = 0.1
n = int((xn-x)/h)
# plot arrays
xp = np.linspace(x, xn, n+1)
yp = np.empty(n+1, float)
up = np.empty(n+1, float)
yp[0] = y
up[0] = u
# the header of the output table
print('x \t\t y\'(RK4) \t y(RK4) \t y\'(Exact) \t y(Exact)')
print('%f \t %f \t %f \t %f \t %f'%(x, u, y, df(x), f(x)))
for i in range(1, n+1):
    L1 = h*du(x, y, u)
    K1 = h*dy(x, y, u)
    L2 = h*du(x+h/2, y+K1/2, u+L1/2)
    K2 = h*dy(x+h/2, y+K1/2, u+L1/2)
    L3 = h*du(x+h/2, y+K2/2, u+L2/2)
    K3 = h*dy(x+h/2, y+K2/2, u+L2/2)
    L4 = h*du(x+h, y+K3, u+L3)
    K4 = h*dy(x+h, y+K3, u+L3)
    u += (L1 + 2*L2 + 2*L3 + L4)/6
    up[i] = u
    y += (K1 + 2*K2 + 2*K3 + K4)/6
    yp[i] = y
    x += h
    print('%f \t %f \t %f \t %f \t %f'%(x, u, y, df(x), f(x)))

# Plot
plt.plot(xp, yp, color = 'r', marker = 'o', ls = '', label='y (RK4)')
plt.plot(xp, up, color = 'b', marker = 's', ls = '', label='y\' (RK4)')
plt.plot(xp, f(xp), color = 'k', lw = 1.5, ls = '-', label='y (Exact)')
plt.plot(xp, df(xp), color = 'k', lw = 1.5, ls = '--', label='y\'
(Exact)')
plt.xlabel('x')
plt.ylabel('y, y\'')
plt.legend(loc = 'upper left')
plt.grid()
plt.axis([np.pi, 2*np.pi, None, None])
plt.show()
```

The output:

x	y'(RK4)	y'(analyt.)	y(RK4)	y(analyt.)
3.141593	2.000000	2.000000	0.000000	0.000000
3.241593	3.214618	3.214619	0.261447	0.261448
3.341593	4.337748	4.337750	0.639870	0.639872
3.441593	5.359660	5.359663	1.125619	1.125622
3.541593	6.272613	6.272617	1.708166	1.708170
3.641593	7.070906	7.070910	2.376315	2.376321
3.741593	7.750904	7.750908	3.118400	3.118407
3.841593	8.311028	8.311032	3.922495	3.922504
3.941593	8.751721	8.751725	4.776620	4.776630
4.041593	9.075382	9.075385	5.668935	5.668946
4.141593	9.286275	9.286277	6.587934	6.587946
4.241593	9.390408	9.390409	7.522627	7.522641
4.341593	9.395399	9.395399	8.462707	8.462722
4.441593	9.310305	9.310303	9.398701	9.398716
4.541593	9.145447	9.145444	10.322107	10.322122
4.641593	8.912209	8.912205	11.225509	11.225525
4.741593	8.622828	8.622823	12.102675	12.102691
4.841593	8.290173	8.290166	12.948631	12.948646
4.941593	7.927515	7.927506	13.759709	13.759724
5.041593	7.548296	7.548286	14.533581	14.533596
5.141593	7.165900	7.165888	15.269263	15.269276
5.241593	6.793416	6.793403	15.967093	15.967104
5.341593	6.443422	6.443407	16.628697	16.628707
5.441593	6.127766	6.127750	17.256923	17.256931
5.541593	5.857366	5.857350	17.855761	17.855766
5.641593	5.642025	5.642008	18.430234	18.430238
5.741593	5.490258	5.490241	18.986287	18.986289
5.841593	5.409147	5.409130	19.530644	19.530643
5.941593	5.404213	5.404196	20.070659	20.070656
6.041593	5.479313	5.479297	20.614158	20.614152
6.141593	5.636567	5.636551	21.169264	21.169256
6.241593	5.876302	5.876288	21.744224	21.744214



The numerical values and the plot show the accuracy of both y and y' values obtained by RK4.

ODE solution in SciPy

The main ODE solver of the module `scipy.integrate` is `odeint()`. It is used in solving a system of equations as well as first order equation. The examples discussed above are solved here by using `odeint()`. The same variable names are used.

```
>>> from scipy.integrate import odeint
>>> from numpy import linspace
>>> dy = lambda y, x: x*y     # Example 1
```

The sequence of the argument list in the function definition should be $f(y, x, \ldots)$.

```
>>> y0 = 1
>>> x = linspace(0, 2, 5)     # generates an array of 5 elements, h=0.5
>>> print(x)
[ 0.   0.5  1.   1.5  2. ]
>>> y = odeint(dy, y0, x)
>>> print(y)
[[ 1.        ]
 [ 1.13314865]
 [ 1.64872162]
 [ 3.08021781]
 [ 7.3890592 ]]
```

In comparison with the results of RK4 in page 5, odeint() returned remarkably more accurate results.

```
>>> dy = lambda y,x: y+3    # Example 2
>>> y0 = -2
>>> x = linspace (2, 4, 21) # generates an array of 21 elements, h=0.1
>>> print(x)
[ 2.   2.1  2.2 ...,  3.8  3.9  4. ]
>>> y = odeint(dy, y0, x)
>>> print(y)
[[-2.        ]
 [-1.89482904]
 [-1.7785972 ]
 ...,
 [ 3.04964786]
 [ 3.68589488]
 [ 4.38905658]]
```

In comparison with the results of RK4 in pages 6 and 7, odeint() solution is still more accurate.

(*Note: Python console may print the first and the last three elements of the arrays. In order to print the whole array set the print option of threshold to nan or Inf as following*

```
set_printoptions(threshold = nan)
```

This function should be imported from numpy module)

To solve a higher order ODE, the equation should be divided to a system of first order equation. The equations should be included in the function to be passed to odeint(). The initial values are given as an array.

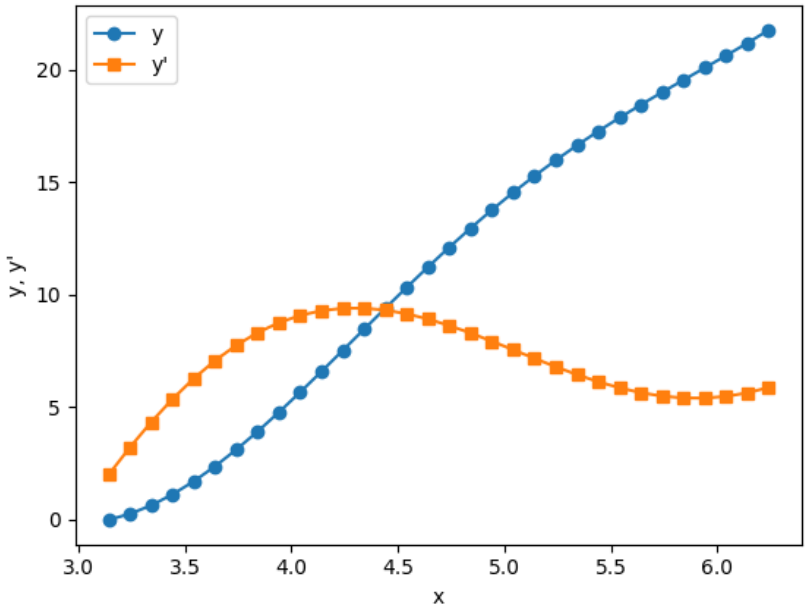
```
from numpy import sin, pi
>>> def dy(y, x):                          # Example 3
...     y, u = y                           # passing initial values
...     dydx = [u, 4*x +10*sin(x)-y]       # equations of the system
...     return dydx
...
>>> y0 = [0, 2]
>>> x = arange(pi, 2*pi, 0.1)             # h = 0.1
>>> print(x)
[ 3.14159265  3.24159265  3.34159265 ...,  6.04159265  6.14159265
  6.24159265]
>>> sol = odeint(dy, y0, x)
>>> print(sol)
[[  0.           2.        ]
 [  0.2614477    3.21461869]
 [  0.63987206   4.33774996]
 ...,
 [ 20.61415227   5.47929668]
 [ 21.16925591   5.63655105]
 [ 21.74421392   5.87628786]]
```

The first column represents the values of y and the second represents the values of y'. This result can be compared to those tabulated in page 10. In this problem x values are generated by using numpy function `arange()` to ensure a constant step size.

The results can be plotted as following

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, sol[:,0],'o-', x,sol[:,1],'s-')
>>> plt.xlabel('x')
>>> plt.ylabel('y, y\'')
>>> plt.legend(['y', 'y\''])
```

For more information about `odeint()`:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html


y
y'
20
15
10
5
0
y, y'
3.0
3.5
4.0
4.5
5.0
5.5
6.0
x