Question 4

Given ODE

$\frac{dy}{dx} = 1 + \frac{y}{x}$

$x > 0$

Initial Condition

$y(1) = 1$

Analytical solution of ODE is

$y_{analytical} = x(1 + \ln x)$

Part (a)

Trying 2nd Order Runge-Kutta (*Heun*'s *Method*) by hand. Taking $h = 0.5$.

We will retain 6 digits. Two steps will take $x = 1$ to $x = 2$

$x_1 = 1$

$y_1 = 1$

$h = 0.5$

*(In code our array and ilteration will start from **0th** index in hand calculation it will start with **1st** index)*

$\frac{dy}{dx} = f(x, y) = 1 + \frac{y}{x}$

<u>**First Step**</u>

$x_0 = 1$

$y_0 = 1$

$h = 0.5$

$x_1 = x_0 + h = 1 + 0.5 = 1.5$

$k_1 = hf(x_0, y_0) = .5(1 + \frac{1}{1}) = 1$

$k_2 = hf(x_1, y_0 + k_1) = 0.5(1 + \frac{2}{1.5}) = 1.166666$

$y_1 = y_0 + 0.5(k_1 + k_2) = 1 + 0.5(1 + 1.166666) = 2.083333$

Second Step

$x_1 = 1.5$

$y_1 = 2.083333$

$h = 0.5$

$x_2 = x_1 + h = 1 + 1.5 = 2$

$k_1 = hf(x_1, y_1) = .5(1 + \frac{2.083333}{1.5}) = 1.194444$

$k_2 = hf(x_2, y_1 + k_1) = 0.5(1 + \frac{3.277777}{2}) = 1.319444$

$y_2 = y_1 + 0.5(k_1 + k_2) = 2.083333 + 0.5(1.94444 + 1.319444) = 3.340277$

This after two steps we have

$\mathbf{y_2 = 3.340277}$ at $x = 2$

In [64]:

```
import numpy as np

def WriteDataToFile(list,myfile):
    myfile.write(', '.join(str(item) for item in list)+'\n')


fxy = lambda x,y : 1 + y/x
F = lambda x : x*(1 + np.log(x))

xi = 1
xf = 6
h = 0.5
y0 = 1

x = np.arange(xi,xf,h , dtype=float)
##Numpy do not include xf so adding it manually
x = np.append(x,xf)
y = np.zeros(len(x))
y[0] = y0

myfile = open("output_data\\rk.csv", 'a')
myfile.seek(0)
myfile.truncate()
WriteDataToFile(["itr","x","y_numerical","y_analytical","Error"],myfile)
WriteDataToFile([0,x[0],y0,F(x[0]),abs(y0-F(x[0]))],myfile)

for i in range(1,len(x)):
    x_0 = x[i-1]
    y_0 = y[i-1]
    ##This is necessary cuz H changes in last ilteration(since we appended xf)
    hint = x[i] - x_0
    k1 = hint*fxy(x_0,y_0)
    k2 = hint*fxy(x_0 + hint,y_0 + k1)
    y[i] = y_0 + k2/2+k1/2
    actual_y = F(x[i])
    WriteDataToFile([i,x[i],y[i],actual_y,abs(y[i] - actual_y)],myfile)
myfile.close()
```

Part (b)

Our numerical answers does mathc with the answers for x_2 we calculated with hand. On x_2 our hand calculation yielded $y_2 = 3.340277$ which is exactly equal to y_2 from rk method **within 6 decimal place**

Also;

$y_{num}(6) = 16.564193984526778$

Absolute error;

$|y_{num}(6) - y_{actual}| = 0.1863628308415528$

Part (c)

In [65]:

```
from scipy.integrate import solve_ivp

def WriteDataToFile(list,myfile):
    myfile.write(', '.join(str(item) for item in list)+'\n')


fxy = lambda x,y : 1 + y/x
F = lambda x : x*(1 + np.log(x))

def rk2(xi,xf,h,y0):
    x = np.arange(xi,xf,h , dtype=float)
    x = np.append(x,xf)
    y = np.zeros(len(x))
    y[0] = y0
    for i in range(1,len(x)):
        ##This seems unnecessary but it is necessary cuz last xf DO NOT HAVE interv
        x_0 = x[i-1]
        y_0 = y[i-1]
        hint = x[i] - x_0
        k1 = hint*fxy(x_0,y_0)
        k2 = hint*fxy(x_0 + hint,y_0 + k1)
        y[i] = y_0 + k2/2 + k1/2
    return y[-1]



hv = np.flip(np.arange(.001,.05,0.005,dtype=float))


myfile = open("output_data\\rk_h.csv", 'a')
myfile.seek(0)
myfile.truncate()
WriteDataToFile(["h","n","error_in_RK2","e2h/eh"],myfile)
xinit = 1
xfinal = 6

y_actual_at_6 = F(xfinal)
for i in hv:
    rktwo = rk2(xinit,xfinal,i,1)
    rktwo2h = rk2(xinit,xfinal,i*2,1)
    eh = abs(rktwo - y_actual_at_6)
    e2h = abs(rktwo2h - y_actual_at_6)
```

```
    WriteDataToFile([i,np.ceil((xfinal-xinit)/i),eh,e2h/eh],myfile)

myfile.close()

rkfour = solve_ivp(fxy, [1,6],[1],t_eval=np.linspace(1, 6, 6),method='RK45')
rktwo = rk2(xinit,xfinal,0.1,1)
print("RK2 Error : {e}".format(e = abs(rktwo - F(xfinal))))
print("RK4 Error : {e}".format(e = abs(rkfour.y[0][-1]-F(xfinal))))
```

```
RK2 Error : 0.009230502441933908
RK4 Error : 0.0005262026957986166
```

Part (c)

We will tabulate errors for different h and see ration of convergance.

h	n	error_in_RK2	e2h/eh
0.046	109.0	0.002008699826330229	3.9047013945386224
0.041	122.0	0.001600099323837867	3.9155031797401767
0.036000000000000004	139.0	0.0012367891819131671	3.9251170558936903
0.031	162.0	0.0009194548008650827	3.9356005589363883
0.026000000000000002	193.0	0.0006484536527970874	3.946494067496399
0.021	239.0	0.00042414298501469716	3.9568341506074023
0.016	313.0	0.00024683953161996897	3.967079997330838
0.011	455.0	0.00011697249590625347	3.9773757291591854
0.006	834.0	3.489197315076353e-05	3.987597195245042
0.001	5000.0	9.717245141871445e-07	3.997952281681664

As you can see as we become more precise our error decreases but also $\frac{e_{2h}}{e_h}$ tends to **4**

Which actually makes sense, well as the name suggest its 2nd Order Runga Kutta method nothing less expected right !!!

Part (d)

We used python inbuilt solver that implemented **rk4** method. `scipy.integrate.solve ivp`

Their solution was fairly more accurate than our OG **rk2** method. Errors for few test cases are written below code in o/p cell