3ο Σετ Ασκήσεων

Υπολογιστικά Μαθηματικά

Ψαρράς Δημήτριος

ΑΕΜ: 4407

Επίλυση σε Python 3

Άσκηση 1η

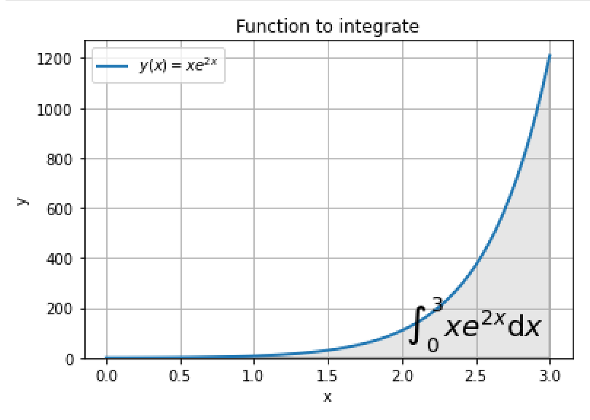
Εκφώνηση

Use the simple Simpson's h/3 rule and the same rule with n=8 to calculate the integral:

$$\int_0^3 xexp(2x)\,dx$$

Λύση

In [3]:
```python
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
import math

def function(x):
    return x*math.exp(2*x)

def plot_integrated_func(y,r):
    x = sp.symbols('x')
    left, width = .3, .5
    bottom = .2
    right = left + width

    p1 = sp.plot(y,(x,r[0],r[1]), ylabel='y', xlabel='x', show = False)
    x1, y1 = p1[0].get_points()

    fig, ax = plt.subplots()
    ax.plot(x1,y1,label=r'$y(x)=xe^{2x}$', linewidth=2)
    ax.set_ylim(bottom=0)
    plt.title('Function to integrate')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.grid()

    ix = np.linspace(r[0], r[1])
    iy = ix*np.exp(2*ix)
    verts = [(r[0], 0), *zip(ix, iy), (r[1], 0)]
    poly = Polygon(verts, facecolor='0.9', edgecolor='0.5')
    ax.add_patch(poly)

    ax.text(right, bottom, r"$\int_0^3 xe^{2x}\mathrm{d}x$",
            horizontalalignment='center',verticalalignment='top',transform=ax.transAxes, fontsize=20)

    plt.show()
    return

def simpson_h3(r,n):
    h = (r[1]-r[0])/n
    m = r[0]+h
    In = (h/3)*(function(r[0])+4*function(m)+function(r[1]))
    return In

def simpson_h3_8(r,n):
    if n%2 != 0:
        raise ValueError('Simpson\'s h/3 rule only works for an even number of segments (n).')
    sum_even = 0.0
    sum_odd = 0.0
    h = (r[1]-r[0])/n
    dx = np.linspace(r[0],r[1],n+1)
    for i in range(1,n):
        if (i%2 != 1):
           sum_even += function(dx[i])
        else:
           sum_odd += function(dx[i])

    In = (h/3)*(function(r[0])+4*sum_odd+2*sum_even+function(r[1]))
    return In


x = sp.symbols('x')
f = x*sp.exp(2*x)

x = np.array([0, 3])
n = np.array([2, 8])

plot_integrated_func(f,x)

I1 = simpson_h3(x,n[0])
I2 = simpson_h3_8(x,n[1])

print(f'Using the simple Simpson\'s h/3 rule the integral is calculated,\nI = {I1}\n')
print(f'Using the same rule with n = 8 the integral is calculated,\nI = {I2}')
```




```
Using the simple Simpson's h/3 rule the integral is calculated,
I = 665.3998010086657

Using the same rule with n = 8 the integral is calculated,
I = 506.0083576012802
```

Συμπεράσματα

Προκειμένου να εξαχθούν συμπεράσματα για τους παραπάνω υπολογισμούς θα υπολογισθούν τα σφάλματα. Ειδικότερα, η πραγματική τιμή του ολοκληρώματος υπολογίσθηκε ίση με 504.53599. Έτσι, για τον απλό κανόνα του Simpson h/3, n=2, το σφάλμα δίνεται από τη σχέση:

$$\mathrm{E} = -\frac{b-a}{180}h^4 f^{(4)}(\xi)$$

όπου ξ βρίσκεται ανάμεσα στις τιμές a=0 και b=3 και έτσι επιλέγεται ξ=a+$\frac{b-a}{2}$=1.5. Επιπλέον $h = \frac{b-a}{n}$=1.5. Έτσι υπολογίζοντας την τέταρτη παράγωγο $f^{(4)}(\xi) = 32e^{2\xi} + 16\xi e^{2\xi}$ το σφάλμα είναι ίσο με:

$$\mathrm{E} = -\frac{3}{180}1.5^4(1124.79) = -94.90$$

και το σχετικό σφάλμα είναι ίσο

$$\varepsilon_t = \frac{measured - real}{real} * 100 = 31.88\%$$

Στη συνέχεια υπολογίζεται το σφάλμα για την περίπτωση όπου χρησιμοποιήθηκε ξανά ο κανόνας του Simpson h/3 αλλά αυτή τη φορά για n=8 υποδιαιρέσεις, όπου $h = \frac{b-a}{n}$=0.375 . Έτσι το σφάλμα υπολογίσθηκε ίσο με:

$$\mathrm{E} = -\frac{3}{180}0.375^4(1124.79) = -0.3707$$

και το σχετικό σφάλμα είναι ίσο

$$\varepsilon_t = \frac{measured - real}{real} * 100 = 0.29\%$$

Επομένως, γίνεται σαφές ότι είναι προτιμότερο να λαμβάνεται το δυνατό μεγαλύτερος αριθμός υποδιαιρέσεων όταν εφαρμόζεται ο κανόνας του Simpson h/3, καθώς με την χρήση μόλις 8 υποδιαιρέσεων επιτεύχθει σχετικό σφάλμα μικρότερο του 0.5%.

Άσκηση 2η

Εκφώνηση

Solve the following differential equation from t=0 until t=2 with y(0)=1

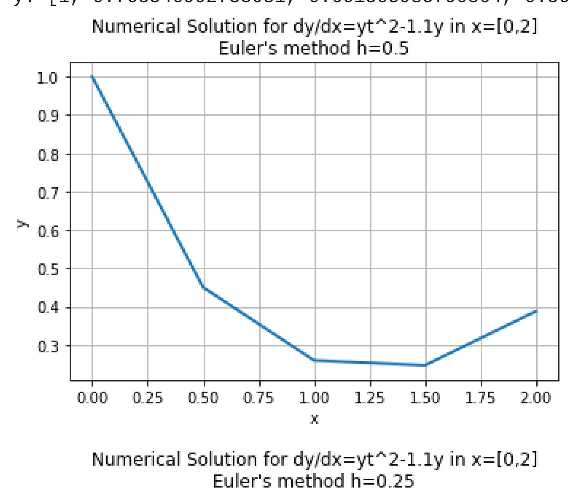
$$\frac{dy}{dt} = yt^2 - 1.1y$$

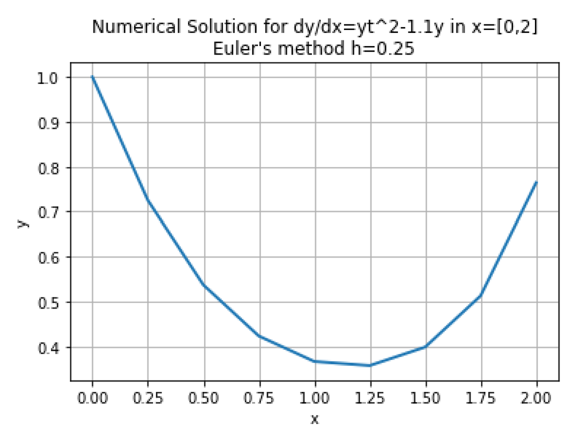
using (a) Euler's method with h=0.5 and h=0.25 (b) Runge Kutta 4th order with h=0.5 and h=0.25

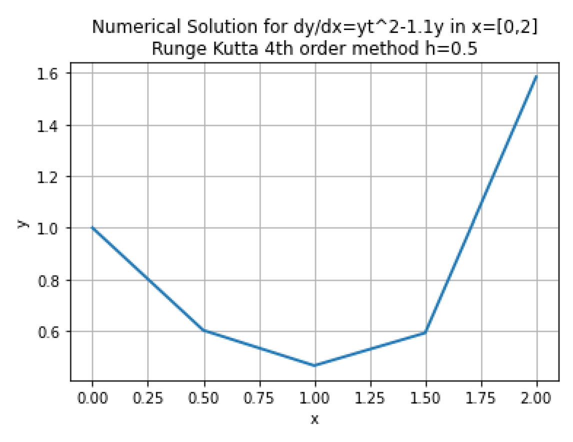
Λύση

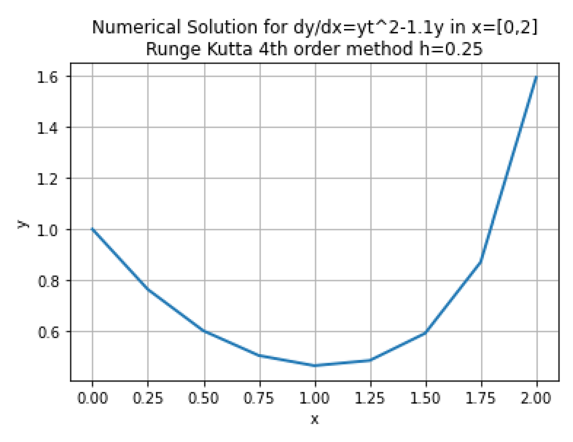
In [4]:
```python
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt

def Euler(yd,h_a,t_a,y0):
    euler = 'Euler\'s method'
    y, t = sp.symbols('y, t')
    n = len(h_a)
    points = []
    yn = y0
    for i in range(n):
        h = h_a[i]
        t_all = np.linspace(t_a[0],t_a[1],int((t_a[1]-t_a[0])/h)+1)
        for xn in t_all:
            yn1 = yn + h*(yd.subs([(t,xn),(y,yn)]))
            points.append(yn)
            yn = yn1

        print_function(points,t_all,h,euler)
        print(f'The points of x and y are:\nx: {t_all}\ny: {points}')

        yn1 = 0
        yn = y0
        points = []
    return

def Runge_Kutta(yd,h_a,t_a,y0):
    RK4 = 'Runge Kutta 4th order method'
    y, t = sp.symbols('y, t')
    n = len(h_a)
    points = []
    yn = y0
    for i in range(n):
        h = h_a[i]
        t_all = np.linspace(t_a[0],t_a[1],int((t_a[1]-t_a[0])/h)+1)
        for xn in t_all:
            k1 = h*yd.subs([(t,xn),(y,yn)])
            k2 = h*yd.subs([(t,xn+(h/2)),(y,yn+(k1/2))])
            k3 = h*yd.subs([(t,xn+(h/2)),(y,yn+(k2/2))])
            k4 = h*yd.subs([(t,xn+h),(y,yn+k3)])
            yn1 = yn + (1/6)*(k1 + 2*k2 + 2*k3 +k4)
            points.append(yn)
            yn = yn1

        print_function(points,t_all,h,RK4)
        print(f'The points of x and y are:\nx: {t_all}\ny: {points}')

        yn1 = 0
        yn = y0
        points = []
    return

def print_function(y,x,h,s):
    plt.figure()
    plt.plot(x,y, linewidth=2)
    plt.title(f'Numerical Solution for dy/dx=yt^2-1.1y in x=[0,2]\n{s} h={h}')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid()
    return

y, t = sp.symbols('y, t')
y_dot = y*t**2 - 1.1*y

y0 = 1
t = np.array([0.0, 2.0])
h = np.array([0.5, 0.25])

Euler(y_dot,h,t,y0)
Runge_Kutta(y_dot,h,t,y0)
```

```
The points of x and y are:
x: [0.  0.5 1.  1.5 2. ]
y: [1, 0.450000000000000, 0.258750000000000, 0.245812500000000, 0.387154687500000]
The points of x and y are:
x: [0.   0.25 0.5  0.75 1.   1.25 1.5  1.75 2.  ]
y: [1, 0.725000000000000, 0.536953125000000, 0.422850585937500, 0.366030038452148, 0.356879287490845, 0.398143455106974, 0.512609698450228, 0.764108831752372]
The points of x and y are:
x: [0.  0.5 1.  1.5 2. ]
y: [1, 0.601570237223307, 0.464523785090807, 0.591380279545628, 1.58445210433665]
The points of x and y are:
x: [0.   0.25 0.5  0.75 1.   1.25 1.5  1.75 2.  ]
y: [1, 0.763546992733081, 0.601505985790864, 0.504411559924858, 0.464563861166024, 0.484833817417430, 0.591553618187111, 0.870544290120920, 1.59370233674348]
```









Συμπεράσματα

Για την μέθοδο του Euler το global trunctuation error είναι της τάξης του O(h) και επομένως παρατηρείται από τα αποτελέσματα του προγράμματος ότι μειώνοντας το βήμα h από 0.5 σε 0.25 το σφάλμα μειώνεται επίσης. Έτσι η αριθμητική επίλυση είναι ακριβέστερη για το μειωμένο βήμα h. Ωστόσο, παρατηρείται σημαντική βελτίωση με την χρήση της μεθόδου Runge Kutta 4ης τάξης, η οποία έχει global trunctuation error της τάξης O(h^4) και επομένως είναι καλύτερη από την μέθοδο του Euler ακόμα και για μεγάλο βήμα h=0.5. Επίσης, και για την μέθοδο Runge Kutta 4ης τάξης το σφάλμα μειώνεται και η αριθμητική λύση είναι ακριβέστερη για μείωση του βήματος h.