Компьютерные технологии в науке и образовании

Домашнее задание: week2

НПМмд-01-19

Преподаватель: Попов Владимир Алексеевич ([popov-va@rudn.ru (mailto:popov-va@rudn.ru)](mailto:popov-va@rudn.ru))

Студент: Мухамаду Хади Диалло ([1032195419@rudn.ru (mailto:1032195419@rudn.ru)](mailto:1032195419@rudn.ru))

---

1. Реализовать метод Рунге-Кутты 4-го порядка и модифицированный метод Эйлера для решения задачи Коши. Проверить на задаче из задачника ОДУ (Филиппов, например)

Решим дифференциальное уравнение $y' + y = x$, с начальным условием $y(0) = 1$.

Точное решение: $y(x) = x - 1 + 2e^{-x}$

Entrée [18]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook
```

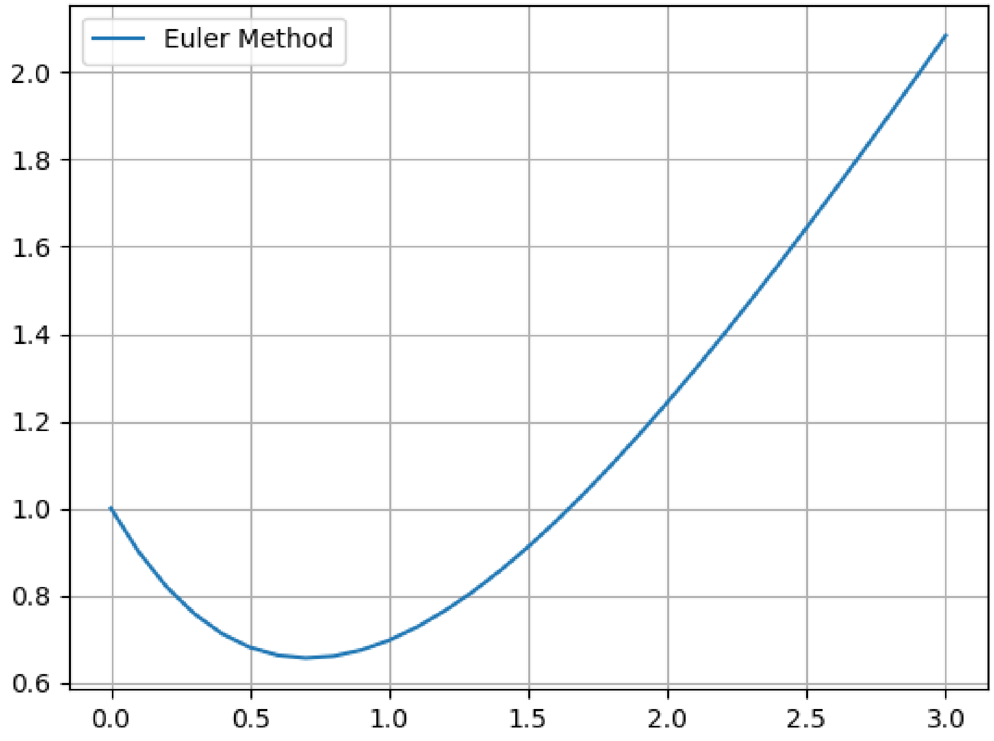
Euler method

Entrée [16]:

```
def y_exact(x) :
    return x-1+2*np.exp(-x)
# Euler Method

def f(x,y):
    return x-y

def Euler(f,x0,b,y0,n) :
    x=np.linspace(x0,b,n)
    y=np.zeros(n)
    y[0]=y0
    for i in range(1,n):
        y[i]=y[i-1]+(x[i]-x[i-1])*f(x[i-1],y[i-1])
    return x, y
def Euler_app(f,x0=0,y0=1,b=3,n=31) :
    X=Euler(f,x0,b, y0,n)[0]
    Y=Euler(f,x0,b, y0,n)[1]
    fig = plt.figure("Euler method",facecolor='white')
    ax = fig.gca()
    ax.plot(X, Y, label='Euler Method')
    ax.legend()
    ax.grid(True)
```

Entrée [17]:

```
Euler_app(f)
```



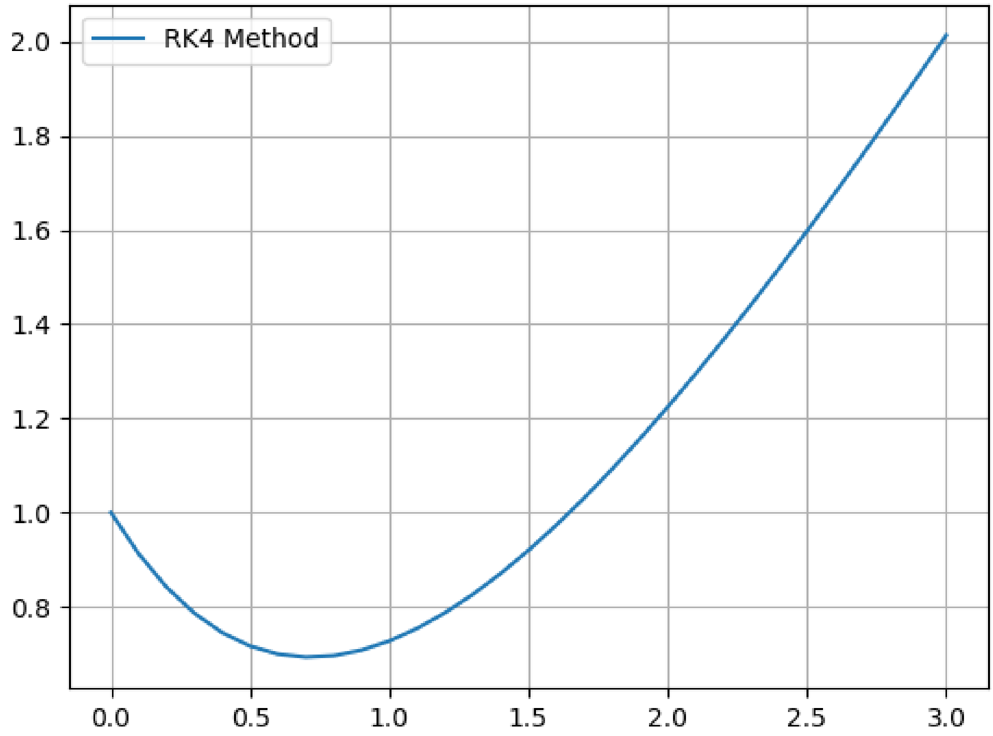
RK4

Entrée [18]:

```
# define  the >RK4 function
def RangeKuta4(f,x0, y0, b, n):
    # Count number of iterations using step size or
    # step height h
    h = (float)((b - x0)/n)
    y = y0
    res=[]
    for i in range(1, n + 1): # Iterate for number of iterations
        res.append(y)
        k1 = h * f(x0, y)
        k2 = h * f(x0 + 0.5 * h, y + 0.5 * k1)
        k3 = h * f(x0 + 0.5 * h, y + 0.5 * k2)
        k4 = h * f(x0 + h, y + k3)

        # Update next value of y
        y = y + (1.0 / 6.0)*(k1 + 2 * k2 + 2 * k3 + k4)
        # Update next value of x
        x0 = x0 + h
    return res

def RangeKuta4_app(f,x0=0, y0=1, b=3, n=31) :
    X=np.linspace(x0,b,n)
    Y=RangeKuta4(f,x0, y0, b, n)
    fig = plt.figure("Range Kutta",facecolor='white')
    ax = fig.gca()
    ax.plot(X, Y, label='RK4 Method')
    ax.legend()
    ax.grid(True)
```

Entrée [19]:

```
RangeKuta4_app(f)
```



Entrée [7]:

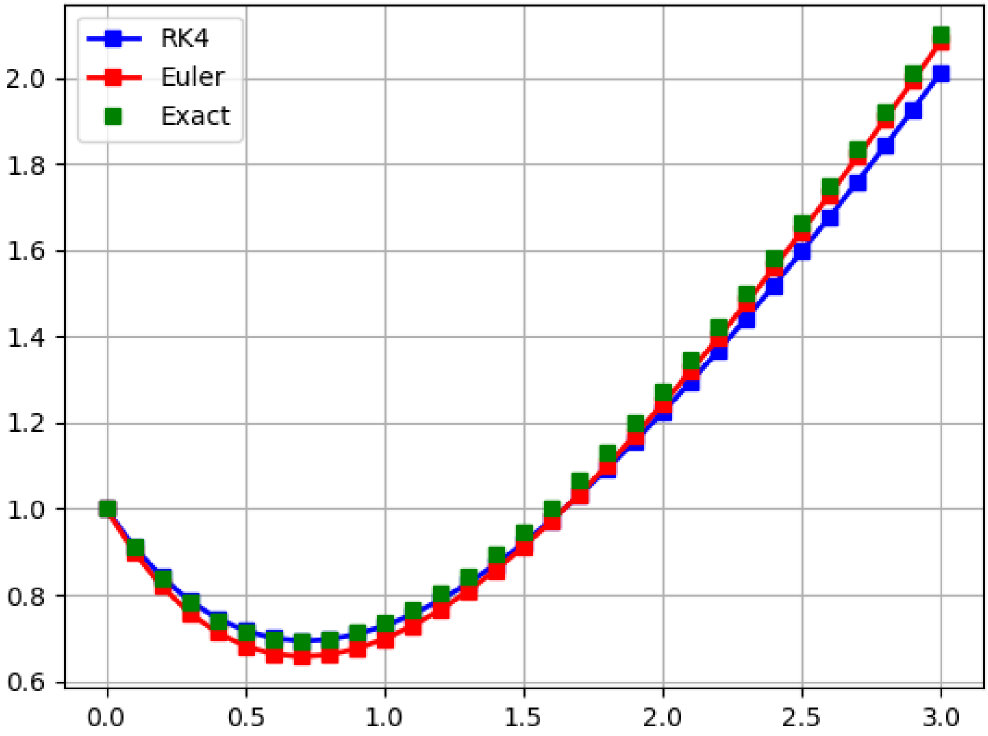
```
x0,y0,b, n=(0,1,3,31)
X=np.linspace(x0,b,n)
Y_RK4=RangeKuta4(f,x0=0, y0=1, b=3, n=31)
Y_Eu=Euler(f,x0=0,b=3,y0=1,n=31)[1]
Y_exa=y_exact(X)
# lenph verification
(len(X), len(Y_RK4),len(Y_exa))
```

Out[7]:

```
(31, 31, 31)
```

Entrée [13]:

```
# graphics
plt.figure("Solve using Euler and Runge Kutta methods")
plt.plot(X,Y_RK4,'-sb',linewidth = 2,label="RK4")
plt.plot(X,Y_Eu,'-sr', linewidth = 2,label="Euler")
plt.plot(X,Y_exa,"sg",linewidth = 2,label="Exact")

plt.grid()
plt.legend()
plt.show()
```



2. Решить с помощью библиотеки scipy (odeint( )) и с помощью библиотеки Sympy (dsolve).

using scipy (odeint( ))

Entrée [21]:

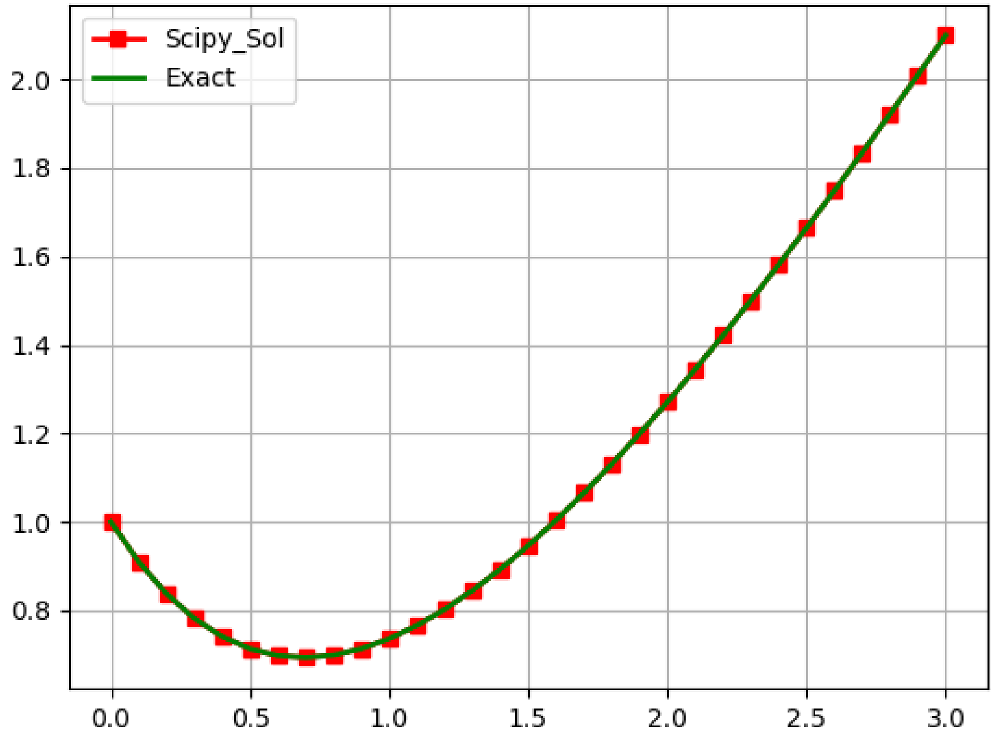
```
import sympy as sp
from scipy.integrate import odeint

# solution with scipy
def f(y, x): # second part of the equation
    return x -y
x0,y0,b, n=(0,1,3,31)
x=np.linspace(x0,b,n)
y_scipy=odeint(f,float(y0),x)
# transformation
y_scipy=np.array(y_scipy.T).flatten()
(len(x),len(y_scipy))
```

Out[21]:

(31, 31)

Entrée [22]:

```
# graphics
plt.figure("Solve using scipy")
plt.plot(x,y_scipy,'-sr', linewidth = 2,label="Scipy_Sol")
plt.plot(x,Y_exa,"g",linewidth = 2,label="Exact")
plt.grid()
plt.legend()
plt.show()
```



using sympy

Entrée [23]:

```
from sympy import symbols,diff, Eq, Function, dsolve, simplify,init_printing
init_printing()
x,y=symbols('x,y')
f=Function('f')
eqq=Eq(f(x).diff(x)+f(x),x)
eqq
```

Out[23]:

$$f(x) + \frac{d}{dx} f(x) = x$$

Entrée [24]:

```
#print(help(dsolve))
y_sy=dsolve(eqq,f(x),ics={f(0):1}) # ``ics`` is the set of initial/boundary conditions for
simplify(y_sy)
```

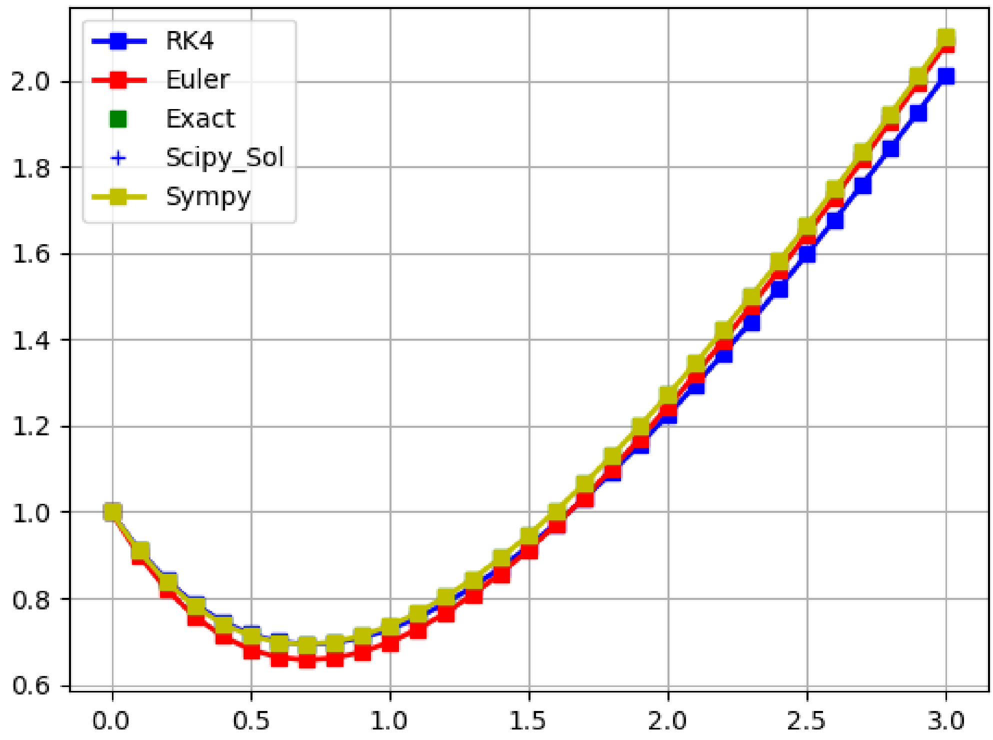
Out[24]:

$$f(x) = x - 1 + 2e^{-x}$$

3. На примере с известным решением построить графики решений пунктов 1 и 2 , а также точного решения

Entrée [25]:

```
# graphics
plt.figure("Solve using Euler, Runge Kutta methods, odeint, dsolve")

plt.plot(X,Y_RK4,'-sb',linewidth = 2,label="RK4")
plt.plot(X,Y_Eu,'-sr', linewidth = 2,label="Euler")
plt.plot(X,Y_exa,"sg",linewidth = 2,label="Exact")
plt.plot(X,y_scipy,'+b', linewidth = 2,label="Scipy_Sol")
plt.plot(X,Y_exa,"-sy",linewidth = 2,label="Sympy")

plt.grid()
plt.legend()
plt.show()
```



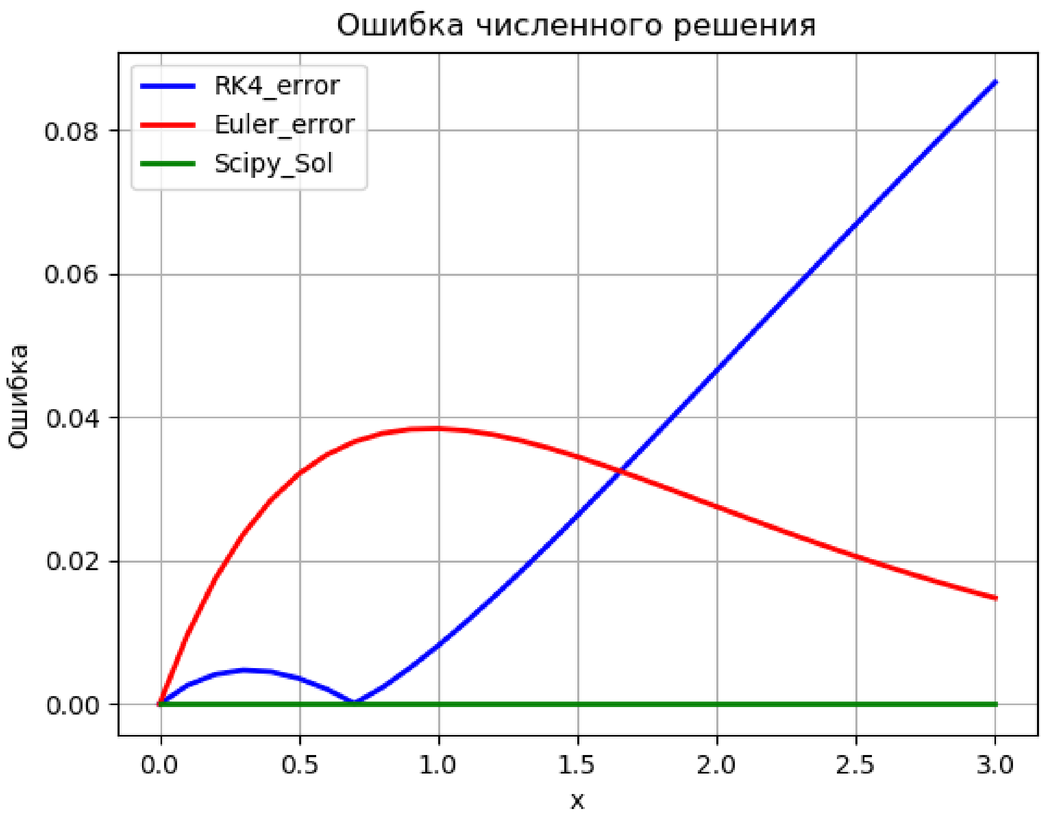
4. Построить график абсолютной погрешности решений из п.1 и 2.

Entrée [26]:

```
error_RK4 = np.abs(Y_exa - Y_RK4)
error_Euler = np.abs(Y_exa - Y_Eu)
error_Odeint = np.abs(Y_exa - y_scipy)
```

Entrée [27]:

```
plt.figure("Absolute Errors terms",facecolor='white')
plt.plot(X,error_RK4,'-b',linewidth = 2,label="RK4_error")
plt.plot(X,error_Euler,'-r', linewidth = 2,label="Euler_error")
plt.plot(X,error_Odeint,'g', linewidth = 2,label="Scipy_Sol")
plt.ylabel('Ошибка')
plt.xlabel('x')
plt.title("Ошибка численного решения")
plt.grid()
plt.legend()
plt.show()
```



5. Самостоятельно реализовать метод прямоугольников, трапеций и Симпсона для вычисления интеграла и сравнить с результатами работы функции simps и quad.

let consider this integarl $I = \int_a^b f(x)dx$.

Numerical method divide the interval $[a, b]$ in $n - time$ with $x_i = a + ih, i = 0, 1, \dots, n$, therefore $h = \frac{b-a}{n}$

Suppose that : $f(x) = x - 1 + 2e^{-x}$ and $[a, b] = [0, 3]$

Entrée [28]:

```
from sympy import Integral,lambdify
x=symbols('x')
f_x=x-1+2*sp.exp(-x)
I = Integral(f_x, (x, 0, 3))
I
```

Out[28]:

$$\int_{0}^{3} \left(x - 1 + 2e^{-x}\right)\, dx$$

Entrée [29]:

```
I_exact=I.doit()
I_exact
```

Out[29]:

$$\frac{7}{2} - \frac{2}{e^{3}}$$

Entrée [30]:

```
I_exact.evalf()
```

Out[30]:

3.40042586326427

метод прямоугольников

$I_{rect} = \sum_{i=0}^{n-1} f(x_i)h$

Entrée [31]:

```
# Implementation
f = lambdify(x, f_x, "numpy")
# algorithm

def MethRect(f,a=0,b=3,n=100):
    I_rect = 0
    h=float((b-a)/n)
    for i in range(0,n-1):
        I_rect = I_rect + f(a+i*h)*h
    return I_rect

print("Rectangle Method =",MethRect(f))
```

```
Rectangle Method = 3.3218965823951323
```

метод трапеций

$$I_{trap} = h[\frac{1}{2}f(x_0) + \sum_{i=1}^{n-1} f(x_i) + \frac{1}{2}f(x_n)$$

Entrée [32]:

```
# algorthm
def MethTrap(f,a=0,b=3,n=100) :
    h=float((b-a)/n)
    Sum=0
    res=0
    for i in range(1,n-1):
        Sum+=f(a+i*h)
    res=h*(0.5*f(a)+Sum+0.5*f(b))
    return res

print("Trapeze Method =",MethTrap(f))
```

```
Trapeze Method = 3.338390194446169
```

метод Симпсона

$$I_{Simpson} = \frac{h}{3}[f(a) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(b)]$$

Entrée [33]:

```
# algorthm
def MethSymp(f,a=0,b=3,n=100) :
    h=float((b-a)/n)
    Sum1=0;Sum2=0

    res=0
    for j in range(1,int(n/2)-1) :
        Sum1=Sum1+f(a+2*j*h)
    for j in range(1,int(n/2)) :
        Sum2=Sum2+f(a+(2*j-1)*h)
    return h*(f(a)+Sum1+Sum2+f(b))

print("Simpson Method =",MethSymp(f))
```

```
Simpson Method = 3.323511862772903
```

Comparaison

Entrée [34]:

```
import scipy.integrate as integrate
a,b=0,3
I_quad=integrate.quad(f,a,b)
print('scipy method quad =',I_quad[0])
```

```
scipy method quad = 3.400425863264272
```

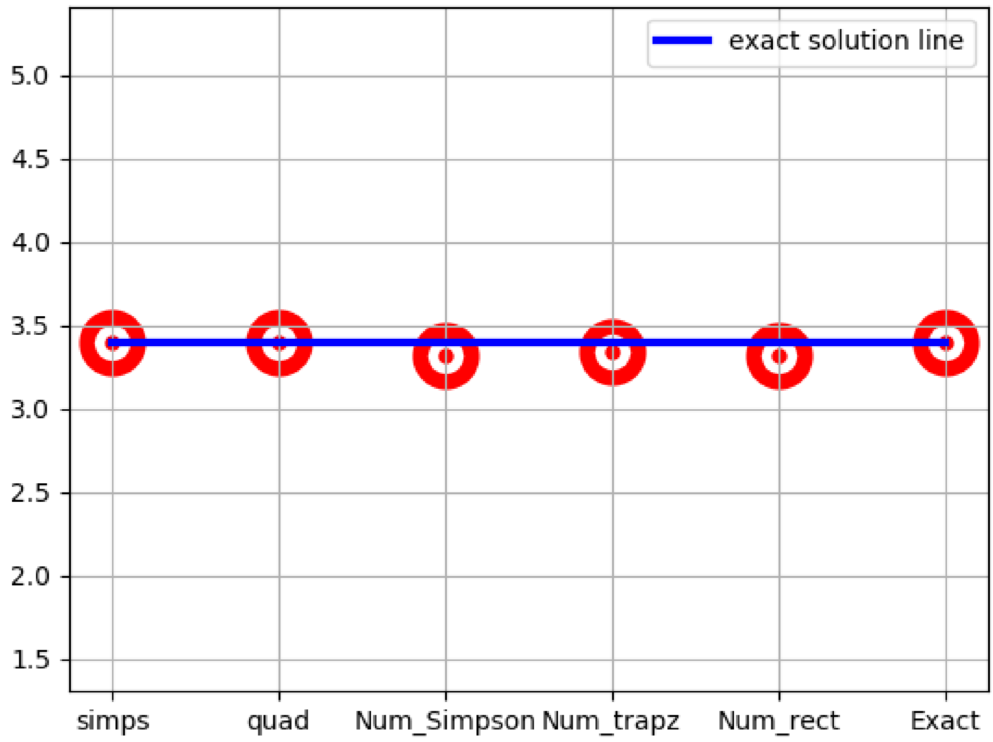
Entrée [35]:

```
#help(integrate.simps)
a,b,n=(0,3,100)
t=np.linspace(a,b,n)
I_simp=integrate.simps(f(t),t)
print('scipy method simps =',I_simp)
```

```
scipy method simps = 3.400428273289831
```

Entrée [38]:

```
Y1=["simps","quad","Num_Simpson","Num_trapz","Num_rect","Exact"]
Y2=[I_simp,I_quad[0],MethSymp(f),MethTrap(f),MethRect(f),I_exact.evalf()]
Y3=[I_exact.evalf(),I_exact.evalf(),I_exact.evalf(),I_exact.evalf(),I_exact.evalf(),I_exact
plt.figure("comparaison")
#plt.pie(Y2, labels=Y1, autopct='%1.1f%%', startangle=90, shadow=True)
width = 0.8
plt.scatter(Y1, Y2, color='r',lw=20)
plt.plot(Y1,Y3,'b', lw=3, label="exact solution line")
plt.show()
plt.grid()
plt.legend()
plt.axis('equal')
plt.show()
```



6. Реализовать метод прогонки (Метод Томаса) решения 3-х диагональных систем. Повторить метод конечных разностей для ОДУ и УРЧП.

[https://pro-prof.com/forums/topic/sweep-method-for-solving-systems-of-linear-algebraic-equations (https://pro-prof.com/forums/topic/sweep-method-for-solving-systems-of-linear-algebraic-equations)](https://pro-prof.com/forums/topic/sweep-method-for-solving-systems-of-linear-algebraic-equations)

$a_i X_{i-1} + b_i X_i + c_i X_{i+1} = d_i$, with $i = 1, .., N$, $a_1 = 0$

The Thomas algorithm

- Step1 : $i = 1$, $y_1 = b_1$, $\alpha_1 = \frac{-c_1}{y_1}$,$\beta_1 = \frac{d_1}{y_1}$
- Step2 : $i = 2, .., n-1$, $y_i = b_i + a_i \alpha_{i-1}$, $\alpha_i = \frac{-c_i}{y_i}$,$\beta_i = \frac{d_i - a_i \beta_{i-1}}{y_i}$
- Step3 : $i = n$, $y_n = b_n + a_n \alpha_{n-1}$, $\beta_n = \frac{d_n - a_n \beta_{n-1}}{y_n}$
- Step4 : $i = n$, $x_n = \beta_n$
- Step5 : $i = n-1, ..., 1$, $x_i = \alpha_i x_{i+1} + \beta_i$

Applicatioon in the third equational system

$b_1 X_1 + c_1 X_2 + 0 = d_1, i = 1$

$a_2 X_1 + b_2 X_2 + c_2 X_3 = d_2, i = 2$

$0 + a_3 X_2 + b_3 X_3 = d_3, i = 3$

Entrée [39]:

```
import sympy as sp
# Implementation
a1,a2,a3=sp.symbols("a1,a2,a3")
b1,b2,b3=sp.symbols("b1,b2,b3")
c1,c2,c3=sp.symbols("c1,c2,c3")
d1,d2,d3=sp.symbols("d1,d2,d3")
X1,X2,X3=sp.symbols("X1,X2,X3")

#Step 1
y1=b1; alfa_1=-c1/y1; beta_1=d1/y1;
#Step 2
y2=b2+a2*alfa_1; alfa_2=-c2/y2; beta_2=(d2-a2*beta_1)/y2;
#Step 3
y3=b3+a3*alfa_2; beta_3=(d3-a3*beta_2)/y3;

#Step 4
X3=beta_3
#Step 5
X2=alfa_2*X3+beta_2
X1=alfa_1*X2+beta_1
print("X1:",X1," X2:",X2," X3:",X3)
```

```
X1: -c1*(-c2*(-a3*(-a2*d1/b1 + d2)/(-a2*c1/b1 + b2) + d3)/((-a2*c1/b1 + b2)*
(-a3*c2/(-a2*c1/b1 + b2) + b3)) + (-a2*d1/b1 + d2)/(-a2*c1/b1 + b2))/b1 + d
1/b1  X2: -c2*(-a3*(-a2*d1/b1 + d2)/(-a2*c1/b1 + b2) + d3)/((-a2*c1/b1 + b2)
*(-a3*c2/(-a2*c1/b1 + b2) + b3)) + (-a2*d1/b1 + d2)/(-a2*c1/b1 + b2)  X3: (-
a3*(-a2*d1/b1 + d2)/(-a2*c1/b1 + b2) + d3)/(-a3*c2/(-a2*c1/b1 + b2) + b3)
```

Entrée [40]:

```
# assignement values
a1,a2,a3=(0,5,1)
b1,b2,b3=(2,4,-3)
c1,c2,c3=(-1,2,0)
d1,d2,d3=(3,6,2)

#Step 1
y1=b1; alfa_1=-c1/y1; beta_1=d1/y1;
#Step 2
y2=b2+a2*alfa_1; alfa_2=-c2/y2; beta_2=(d2-a2*beta_1)/y2;
#Step 3
y3=b3+a3*alfa_2; beta_3=(d3-a3*beta_2)/y3;

#Step 4
X3=beta_3
#Step 5
X2=alfa_2*X3+beta_2
X1=alfa_1*X2+beta_1
print("X1:",X1," X2:",X2," X3:",X3)
solut_thomas=[X1,X2,X3]
```

```
X1: 1.4883720930232558  X2: -0.023255813953488358  X3: -0.6744186046511629
```

Entrée [41]:

```
# solution with sympy
X1,X2,X3=sp.symbols("X1,X2,X3")
syst_eq=[ sp.Eq(b1*X1+c1*X2,d1),
         sp.Eq(a2*X1+b2*X2+c2*X3,d2),
         sp.Eq(a3*X2+b3*X3,d3),

]
solution=sp.solve(syst_eq,[X1,X2,X3])
print("System : ",syst_eq)
print("Solution : ",solution)
```

```
System :  [Eq(2*X1 - X2, 3), Eq(5*X1 + 4*X2 + 2*X3, 6), Eq(X2 - 3*X3, 2)]
Solution :  {X1: 64/43, X2: -1/43, X3: -29/43}
```

Entrée [42]:

```
Solu_sp=[64/43,-1/43,-29/43]
Solu_sp
```

Out[42]:

$$[1.4883720930232558, \ -0.023255813953488372, \ -0.6744186046511628]$$

Entrée [43]:

```
print("Thomas Algo solution",solut_thomas)
print("Solution sympy : ",Solu_sp)
print("Erros=r",np.asarray(Solu_sp)-np.asarray(solut_thomas))
```

```
Thomas Algo solution [1.4883720930232558, -0.023255813953488358, -0.67441860
46511629]
Solution sympy :  [1.4883720930232558, -0.023255813953488372, -0.67441860465
11628]
Erros=r [ 0.00000000e+00 -1.38777878e-17  1.11022302e-16]
```

Applicatioon in ODE

Entrée [44]:

```
x,y=sp.symbols('x,y')
f=sp.Function('f')
eqq=sp.Eq(f(x).diff(x,x)+f(x),x)
eqq
```

Out[44]:

$$f(x) + \frac{d^2}{dx^2} f(x) = x$$

Entrée [46]:

```
#u_exact=sp.solve(eqq,f(x),)
res = sp.dsolve(eqq, f(x), ics={f(0):0.0,f(x).diff(x, 1).subs(x, 0): -4.0})
res
```

Out[46]:

$$f(x) = x - 5.0 \sin(x)$$

Entrée [47]:

```
f_exact=sp.lambdify(x,res.rhs,"numpy")
f_exact(0)
```

Out[47]:

0.0

Numerical solution

$\frac{\partial^2 f}{\partial x^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}$

$u_{i+1} + (h^2 - 2)u_i + u_{i-1} = h^2 F(x_i)$

- $X_i = u_i$,
- $X_{i+1} = u_{i+1}$
- $X_{i-1} = u_{i-1}$

$a_i X_{i-1} + b_i X_i + c_i X_{i+1} = d_i$, with $i = 1, .., N$, $a_1 = 0$

- $a_i = 1, a_1 = 0$
- $b_i = h^2 - 2$
- $c_i = 1, c_n = 0$
- $d_i = h^2 F(x_i)$

Entrée [48]:

```
#Step 1
def F(x):
    return x
n=10
h=float((3-0)/n)
x=np.linspace(0,3,n)
y1=1; alfa_1=-1/y1; beta_1=F(x[0])*h**2/y1;

y=np.zeros(n)
X=np.zeros(n)
alfa=np.zeros(n)
beta=np.zeros(n)
y[0]=y1
alfa[0]=alfa_1
beta[0]=beta_1

#Step 2
for i in range(1,n-1) :
    y[i]=h**2-2+alfa[i-1]
    alfa[i]=-1/y[i]
    beta[i]=(F(x[i])*h**2-beta[i-1])/y[i]

#Step 3
y[-1]=h**2-2+alfa[n-1]
beta[-1]=(F(x[-1])*h**2-beta[n-1])/y[-1]

#Step 4
X[-1]=beta[-1]

#Step 5
for i in reversed(range(1, n-1)):
    X[i]=alfa[i]*X[i+1]+beta[i]
X
```

Out[48]:

```
array([ 0.        , -0.72906356, -2.09157497, -3.20584463, -3.94158827,
       -4.20258897, -3.93535666, -3.13394225, -1.84047304, -0.14136126])
```

Entrée [49]:

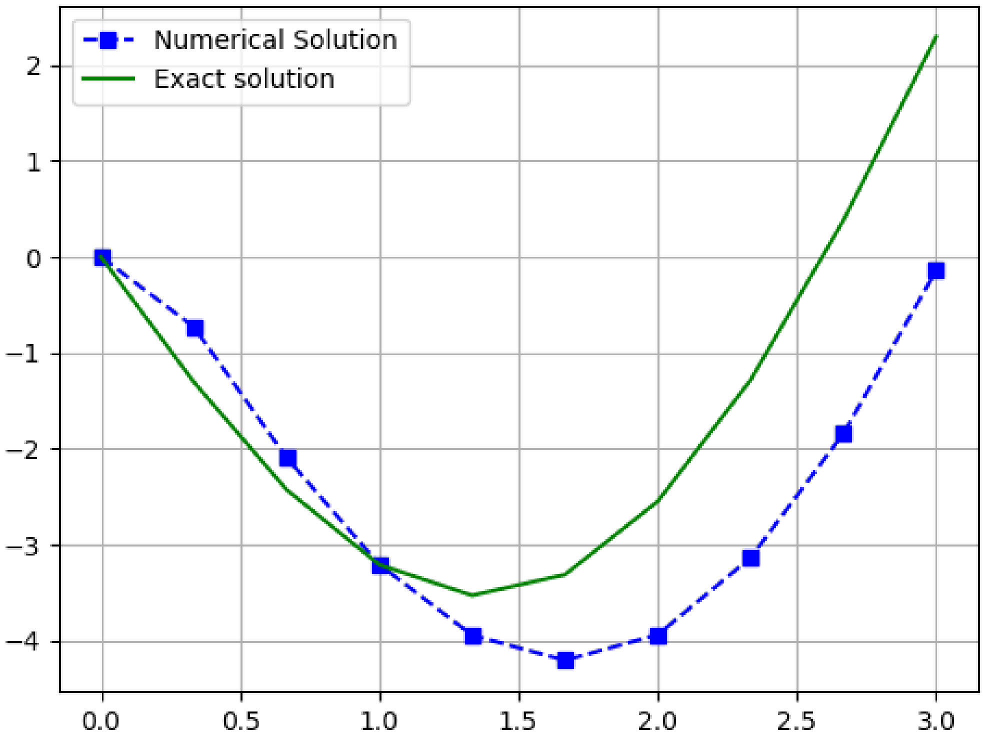
```
f_exact(x)
```

Out[49]:

```
array([ 0.        , -1.30264015, -2.42518235, -3.20735492, -3.52635617,
       -3.31037312, -2.54648713, -1.28209608,  0.38030353,  2.29439996])
```

Entrée [50]:

```
# graphics
plt.figure("Thomas algorithm")
plt.plot(x,X, "--sb",label="Numerical Solution")
plt.plot(x,f_exact(x),"-g",label="Exact solution")

plt.grid()
plt.legend()
plt.show()
```



Application in PDE

Entrée [5]:

```
import sympy as sp
from sympy import diff,Derivative as D
x, t,a,s=sp.symbols("x,t,a,s")
u=sp.Function('u')
F=sp.Function('F')
IC=sp.Function('IC')
diffision=sp.Eq(D(u(x,t),t),a*D(u(x,t),x,x)+F(x,t))
IC=lambda x: sp.exp(-x)
```

Entrée [ ]:

```
#conda install pysde
#diff_sol=sp.dsolve(diffision,u(x,t), ics={u(x,0):IC(x)}) #f(x).diff(x, 1).subs(x, 0): -5.6
#from sympy.solvers import pdsolve #from sympy import pprint
#ut=u(x,t).diff(t) #uxx=u(x,t).diff(x,x) #dif_eq=ut-uxx #pprint(dif_eq) #pprint(pdsolve(dif
```

Entrée [6]:

```
print("The diffision recation equation : ")
diffision
```

```
The diffision recation equation :
```

Out[6]:

$$\frac{\partial}{\partial t}u(x,t) = a\frac{\partial^2}{\partial x^2}u(x,t) + F(x,t)$$

Approximation by the implicite scheme

$u(x_i, t_n) \approx U_i^n$

$-\lambda U_{i-1}^{n+1} + (1+2\lambda)U_i^{n+1} - \lambda U_{i+1}^{n+1} = U_i^n + \tau F_i^n$, with $\lambda = \frac{a\tau}{h^2}$

$a_i U_{i-1}^{n+1} + b_i U_i^n + c_i U_{i+1}^{n+1} = d_i^n$, with $i = 1, .., N$

- $X_i = U_i^{n+1}$,
- $X_{i+1} = U_{i+1}^{n+1}$
- $X_{i-1} = U_{i-1}^{n+1}$
- $a_i = -\lambda$
- $b_i = 1 + 2\lambda$
- $c_i = -\lambda$
- $d_i = U_i^n + \tau F_i^n$
- $F(x,t) = 0$
- Initial condition : $IC = U(x, 0) = \exp -x$
- Boundary condion : $BC = U(0,t) = 1, U(L,t) = 0$

Entrée [29]:

```
# Algorithm

# Step Zeros
import numpy as np
M=10
N=5
x0,t0=(0,0)
L=3
a=1
x=np.linspace(x0,L,M)
t=np.linspace(t0,L,N)
h=float((L-x0)/N)
lamda=0.5
tau=lamda*h**2/a

U=np.zeros((M,N))
X=np.zeros(M)

for n in range(0,N-1) :
    U[0][n]=1.0
    U[-1][n]=0.0
    #Step 1
    U[0][0]=IC(x[0]) # initial condition
    y1=1+2*lamda; alfa_1=-lamda/y1; beta_1=(U[1][n])/y1;

    y=np.zeros(M)
    alfa=np.zeros(M)
    beta=np.zeros(M)
    y[0]=y1
    alfa[0]=alfa_1
    beta[0]=beta_1

    #Step 2
    for i in range(1,M-1) :
        U[i][0]=IC(x[i]) # int condition
        y[i]=1+2*lamda-lamda*alfa[i-1]
        alfa[i]=lamda/y[i]
        beta[i]=(U[i][n]+lamda*beta[i-1])/y[i]

    #Step 3
    U[-1][0]=IC(x[-1])
    y[-1]=1+2*lamda-lamda*alfa[M-1]
    beta[-1]=(U[-1][n]+lamda*beta[M-1])/y[i]

    #Step 4
    X[-1]=beta[-1]

    #Step 5
    for i in reversed(range(1, M-1)):
        X[i]=alfa[i]*X[i+1]+beta[i]
        # evalution U
        U[i][n+1]=X[i]
        U[i-1][n+1]=X[i-1]
        U[i+1][n+1]=X[i+1]
    U[0][-1]=1.0
U
```

Out[29]:

```
array([[1.        , 1.        , 1.        , 1.        , 1.        ],
       [0.71653131, 0.4453928 , 0.35802697, 0.29582393, 0.24992982],
       [0.51341712, 0.45985677, 0.40813262, 0.36218428, 0.32264189],
       [0.36787944, 0.36720006, 0.35478996, 0.33664797, 0.31626918],
       [0.26359714, 0.27318457, 0.2766271 , 0.27482767, 0.26913888],
       [0.1888756 , 0.19834395, 0.20534929, 0.2094085 , 0.21063101],
       [0.13533528, 0.14244002, 0.14808216, 0.15210777, 0.15456817],
       [0.09697197, 0.10074557, 0.10209932, 0.10285824, 0.10342611],
       [0.06948345, 0.06659832, 0.05882399, 0.05512656, 0.05341981],
       [0.04978707, 0.        , 0.        , 0.        , 0.        ]])
```

Entrée [3]:

```
#pip install request
```

Entrée [30]:

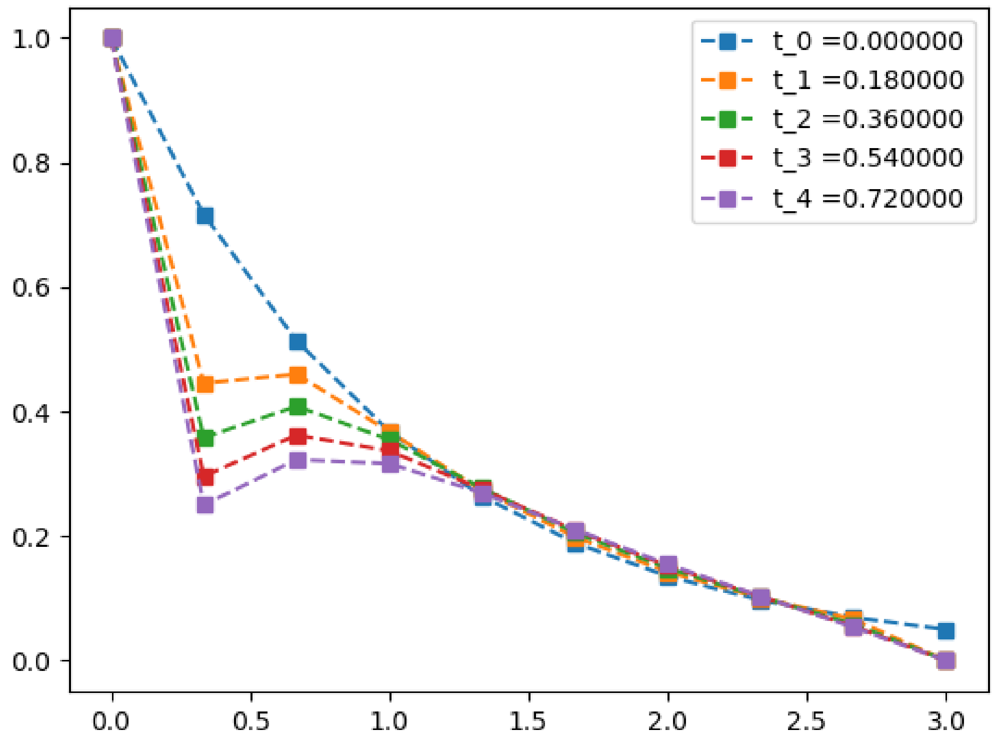
```
import pandas as pd
u_exp = np.round(U, 3)
u_exp = pd.DataFrame(U)
u_exp
```

Out[30]:

	0	1	2	3	4
0	1.000000	1.000000	1.000000	1.000000	1.000000
1	0.716531	0.445393	0.358027	0.295824	0.249930
2	0.513417	0.459857	0.408133	0.362184	0.322642
3	0.367879	0.367200	0.354790	0.336648	0.316269
4	0.263597	0.273185	0.276627	0.274828	0.269139
5	0.188876	0.198344	0.205349	0.209409	0.210631
6	0.135335	0.142440	0.148082	0.152108	0.154568
7	0.096972	0.100746	0.102099	0.102858	0.103426
8	0.069483	0.066598	0.058824	0.055127	0.053420
9	0.049787	0.000000	0.000000	0.000000	0.000000

Entrée [35]:

```
# extraction
u_exp = np.round(U, 3)
u_exp = pd.DataFrame(U)
    #u_exp
    # Rename time and columns
a=[]
    # time
for i in range(N):
    b=['t_'+str(i)]
    a=a+b
u_exp.columns = a
# space

c=[]
for i in range(M):
    b=['x_'+str(i)]
    c=c+b
u_exp.index = c

    # Graphics

for i in range(N) :
    a='t_'+str(i)
    b=str(a)
    plt.figure("U diffusion")
    plt.subplot()
    plt.plot(x,u_exp[b],"--s", label=b+" =%f"%(tau*i))
    plt.legend()
    plt.show()
```



```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:26: Matplot
libDeprecationWarning: Adding an axes using the same arguments as a previous
axes currently reuses the earlier instance.  In a future version, a new inst
ance will always be created and returned.  Meanwhile, this warning can be su
ppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
```

Entrée [ ]:

```
#pip install nbconvert
# jupyter Nbconvert --to pdf homework_week_2.ipynb
```

Entrée [ ]:

```

```