Øving 13 - Numeriske metoder for differensiallikninger II - LF

Obligatoriske oppgaver

In these exercises we study the heat equation

$$u_t = u_{xx}, \qquad \text{in } (0,1)\times(0,T)$$

where $T > 0$ is some positive number. We have boundary conditions

$$u(0,t) = g_1(t), \quad u(1,t) = g_2(t),$$

and initial conditions

$$u(x,0) = f(x)$$

for some nice functions g and f. We keep it a bit general for now, so that we can just apply what we discover later. We will construct numerical methods to solve this equation.

Definitions.

Define

$$k = T/M \qquad \text{(time step)},$$

$$h = 1/N \qquad \text{(space step)}$$

and the grid

$$x_i = ih, \qquad t_j = jk,$$

where $j \in \{0,1,2,\dots,M\}$ and $i \in \{0,1,2,\dots,N\}$. Further, let

$$\begin{aligned} f^i &:= f(x_i), \\ g_1^j &:= g_1(t_j), \\ g_2^j &:= g_2(t_j). \end{aligned}$$

Now the story begins.

$\boxed{1}$ **Forward Euler method**

We write the discrete approximation as

$$\frac{u_i^{j+1} - u_i^j}{k} = \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{h^2},$$

which we rearrange to

$$u_i^{j+1} = u_i^j + \frac{k}{h^2}\Big(u_{i+1}^j - 2u_i^j + u_{i-1}^j\Big).$$

For the points u_1^{j+1} and u_{N-1}^{j+1} we note that we get

$$u_1^{j+1} = u_i^j + \frac{k}{h^2}\Big(u_2^j - 2u_1^j + g_1^j\Big),$$

and

$$u_{N-1}^{j+1} = u_{N-1}^j + \frac{k}{h^2}\Big(g_2^j - 2u_{N-1}^j + u_{N-2}^j\Big).$$

If you think about it for a while, you will realize that we can write everything in neat vector notation as

$$\mathbf{u}^{j+1} = (I - \frac{k}{h^2}A)\mathbf{u}^j + \frac{k}{h^2}\mathbf{g}^j \tag{1}$$

where

$$\mathbf{u}^{j+1} = \begin{bmatrix} u_1^j \\ \vdots \\ u_{N-1}^j \end{bmatrix},$$

I is the $(N-1)\times(N-1)$ identity matrix, A is an $(N-1)\times(N-1)$-matrix of the form

$$A = \begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & \ddots & \ddots & & \\ & & \ddots & \ddots & -1 & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{bmatrix},$$

and

$$\mathbf{g} = \begin{bmatrix} g_1^j \\ 0 \\ \vdots \\ 0 \\ g_2^j \end{bmatrix}.$$

We implemented the scheme (1) with initial conditions $u_i^0 = f(x_i)$ and boundary conditions g_1, g_2 in Python. See the code below.

$\boxed{2}$ **Implicit Euler.**

The Implicit Euler method can be written in vector notation as

$$(I + \frac{k}{h^2}A)\mathbf{u}^{j+1} = \mathbf{u}^j + \frac{k}{h^2}\mathbf{g}^{j+1} \tag{2}$$

where all the quantities are defined as before. Note that in each time-step we need to solve a system of equations of the form $Ax = b$. We implemented this code in Python as well. See the code below.

$\boxed{3}$ Finally, for Crank-Nicholson we have

$$(2I + \frac{k}{h^2}A)\mathbf{u}^{j+1} = (2I - \frac{k}{h^2}A)\mathbf{u}^j + \frac{k}{h^2}(\mathbf{g}^j + \mathbf{g}^{j+1}) \tag{3}$$

We also implemented this in Python.

$\boxed{\text{Code}}$ **Python:**

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation


def tridiag(a,b,c, k1=-1, k2=0, k3=1):
    #Lager tridiagonal matrise med vektorene a og b paa under- og
    #overdiagonalen og b paa diagonalen.
    return np.diag(a,k1) + np.diag(b,k2) + np.diag(c,k3)


def forward_euler(f,g1,g2,n,m,T):

    #Space step
    h = 1.0/n

    #Time step
    k = T/m

    #factor
    r = k/h**2

    #x-grid
    x = np.linspace(0,1,n+1)

    #t-grid
    t = np.linspace(0,T,m+1)

    #Constructing the matrix A
    a = 2*np.ones(n-1)
    b = -1*np.ones(n-2)

    A = tridiag(b,a,b)

    #We will use this matrix over and over again
    B = np.diag(np.ones(n-1)) - r*A

    #The solution
    u = np.zeros((m+1,n+1))

    #Initial conditions
    u[0,:] = f(x)

    #Boundary conditions
    u[:,0] = g1(t)
    u[:,-1] = g2(t)

    for j in range(m):

        #The scheme
```

```
        u[j+1,1:-1] = np.dot(B,u[j,1:-1])

        #Dealing with boundary conditions
        u[j+1,1] += r*u[j,0]
        u[j+1,-2] += r*u[j,-1]

    return u

def implicit_euler(f,g1,g2,n,m,T):


    #Space step
    h = 1.0/n

    #Time step
    k = T/m

    #factor
    r = k/h**2

    #x-grid
    x = np.linspace(0,1,n+1)

    #t-grid
    t = np.linspace(0,T,m+1)

    #Constructing the matrix A
    a = 2*np.ones(n-1)
    b = -1*np.ones(n-2)

    A = tridiag(b,a,b)

    #We will use this matrix over and over again
    B = np.diag(np.ones(n-1)) + r*A

    #The solution
    u = np.zeros((m+1,n+1))

    #Initial conditions
    u[0,:] = f(x)

    #Boundary conditions
    u[:,0] = g1(t)
    u[:,-1] = g2(t)

    for j in range(m):

        #We first calculate the right-hand side
        #We need to take a copy of the array.
        tmp = u[j,1:-1].copy()

        #Adding the boundary conditions
        tmp[0] += r*u[j+1,0]
        tmp[-1] += r*u[j+1,-1]

        #Finally, we do the linear algebra to compute
        #the next time step.

        u[j+1,1:-1] = np.linalg.solve(B, tmp)

    return u
```

```python
def crank_nicholson(f,g1,g2,n,m,T):

    #Space step
    h = 1.0/n

    #Time step
    k = T/m

    #factor
    r = k/h**2

    #x-grid
    x = np.linspace(0,1,n+1)

    #t-grid
    t = np.linspace(0,T,m+1)

    #Constructing the matrix A
    a = 2*np.ones(n-1)
    b = -1*np.ones(n-2)

    A = tridiag(b,a,b)

    #We will use these matrices over and over again
    B1 = np.diag(np.ones(n-1)) + r*A
    B2 = np.diag(np.ones(n-1)) - r*A

    #The solution
    u = np.zeros((m+1,n+1))

    #Initial conditions
    u[0,:] = f(x)

    #Boundary conditions
    u[:,0] = g1(t)
    u[:,-1] = g2(t)

    for j in range(m):

        #We first calculate the right-hand side
        tmp = np.dot(B2, u[j,1:-1])

        #Adding the boundary conditions
        tmp[0] += r*(u[j+1,0] + u[j,0])
        tmp[-1] += r*(u[j+1,-1] + u[j,-1])

        #Finally, we do the linear algebra
        #to compute the next time step.
        u[j+1,1:-1] = np.linalg.solve(B1, tmp)

    return u



if __name__ == "__main__":


    #Initial- and boundary conditions

    #Obligatoriske oppgaver
```

```
f = lambda x : np.sin(np.pi*x)
g1 = lambda x : np.zeros(len(x))
g2 = lambda x : np.zeros(len(x))


#Anbefalte oppgaver/for fun
#f = lambda x : np.sin(np.pi*x/2)
#g1 = lambda x : np.zeros(len(x))
#g2 = lambda x : np.cos(np.pi*x)

#Grid data
m=1000
n=10
T=3

#Exercise 1-3: Remove comments to use different methods.

#Forward Euler:
u = forward_euler(f,g1,g2,n,m,T)

#Implicit Euler:
#u = implicit_euler(f,g1,g2,n,m,T)

#Crank-Nicholson:
#u = crank_nicholson(f,g1,g2,n,m,T)


#samme kommoentar som oving 5 og 6
fig, ax = plt.subplots()
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')

xdata, ydata = [], []
ln, = plt.plot([], [], animated=True,label=r'$u\, (x,t)$') # skriv inn ...
    'ro' som tredje argument hvis du vil ha tilbake punkter i stedet ...
    for linjer.

def init():
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1.1)
    return ln,

def update(frame):
    xdata=np.linspace(0, 1, n+1)
    ydata=u[frame]
    ln.set_data(xdata, ydata)
    return ln,
# Man kan endre hastigheten ved aa endre paa interval-parameteren
ani = FuncAnimation(fig, update, frames=np.linspace(0, m, m+1, ...
    dtype=np.int32),interval=30,
                    init_func=init, blit=True)
plt.legend()
plt.show()
```

Anbefalte oppgaver

1. The code presented above tackles all our problems. See the lambda functions that are commented out for an example.

$\boxed{\text{2 and 3}}$ See the section on the Laplace equation in the Lecture notes. You have to set up a system of equations, using matrix notation, and solve it with Python or Matlab. The system of equations you get is different from those of the 'obligatorisk del'.