Mattematikk 2 Oblig – Aksel Eriksen

1

```python
import numpy as np


for i in range(20):

    h = 10**(-i)
    x = (np.exp(1.5+h)-np.exp(1.5))/h
    print(f"For h = {h}: f'(1.5) = {round(x, 5)} og feilen blir {round(np.exp(1.5)-x, 5)}")

#Her ser vi at det begynner å gå åt skogen cirka når h = 1e -10, da begynner verdien å øke pga avrundingsfeil
```

✓ 0.0s

```
For h = 1: f'(1.5) = 7.7008 og feilen blir -3.21912
For h = 0.1: f'(1.5) = 4.71343 og feilen blir -0.23174
For h = 0.01: f'(1.5) = 4.50417 og feilen blir -0.02248
For h = 0.001: f'(1.5) = 4.48393 og feilen blir -0.00224
For h = 0.0001: f'(1.5) = 4.48191 og feilen blir -0.00022
For h = 1e-05: f'(1.5) = 4.48171 og feilen blir -2e-05
For h = 1e-06: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-07: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-08: f'(1.5) = 4.48169 og feilen blir 0.0
For h = 1e-09: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-10: f'(1.5) = 4.4817 og feilen blir -1e-05
For h = 1e-11: f'(1.5) = 4.48175 og feilen blir -6e-05
For h = 1e-12: f'(1.5) = 4.48264 og feilen blir -0.00095
For h = 1e-13: f'(1.5) = 4.4853 og feilen blir -0.00361
For h = 1e-14: f'(1.5) = 4.52971 og feilen blir -0.04802
For h = 1e-15: f'(1.5) = 5.32907 og feilen blir -0.84738
For h = 1e-16: f'(1.5) = 0.0 og feilen blir 4.48169
For h = 1e-17: f'(1.5) = 0.0 og feilen blir 4.48169
For h = 1e-18: f'(1.5) = 0.0 og feilen blir 4.48169
For h = 1e-19: f'(1.5) = 0.0 og feilen blir 4.48169
```

2

```python
import numpy as np


for i in range(20):

    h = 10**(-i)
    x = (np.exp(1.5+h)-np.exp(1.5-h))/(2*h)
    print(f"For h = {h}: f'(1.5) = {round(x, 5)} og feilen blir {round(np.exp(1.5)-x, 5)}")


#I det første tilfellet fikk vi at feilen er proporsjonal med h
#I det andre tilfellet får vi at feilen er proporsjonal med h**2 som betyr at den minker kvadratisk
```

✓ 0.0s

```
For h = 1: f'(1.5) = 5.26689 og feilen blir -0.7852
For h = 0.1: f'(1.5) = 4.48916 og feilen blir -0.00747
For h = 0.01: f'(1.5) = 4.48176 og feilen blir -7e-05
For h = 0.001: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 0.0001: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-05: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-06: f'(1.5) = 4.48169 og feilen blir 0.0
For h = 1e-07: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-08: f'(1.5) = 4.48169 og feilen blir 0.0
For h = 1e-09: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-10: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-11: f'(1.5) = 4.4817 og feilen blir -1e-05
For h = 1e-12: f'(1.5) = 4.48219 og feilen blir -0.0005
For h = 1e-13: f'(1.5) = 4.48086 og feilen blir 0.00083
For h = 1e-14: f'(1.5) = 4.4853 og feilen blir -0.00361
For h = 1e-15: f'(1.5) = 4.88498 og feilen blir -0.40329
For h = 1e-16: f'(1.5) = 0.0 og feilen blir 4.48169
For h = 1e-17: f'(1.5) = 0.0 og feilen blir 4.48169
For h = 1e-18: f'(1.5) = 0.0 og feilen blir 4.48169
For h = 1e-19: f'(1.5) = 0.0 og feilen blir 4.48169
```

3

```python
import numpy as np


for i in range(20):

    h = 10**(-i)
    x = (np.exp(1.5- 2*h)-8*np.exp(1.5-h)+8*np.exp(1.5+h)-np.exp(1.5+2*h))/(12*h)
    print(f"For h = {h}: f'(1.5) = {round(x, 5)} og feilen blir {round(np.exp(1.5)-x, 5)}")
```

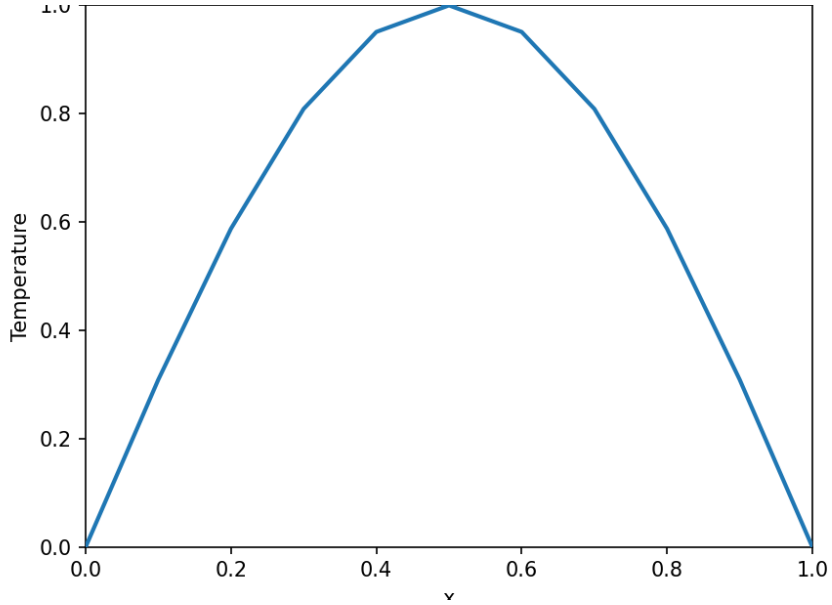
✓ 0.0s

```
For h = 1: f'(1.5) = 4.31344 og feilen blir 0.16825
For h = 0.1: f'(1.5) = 4.48167 og feilen blir 1e-05
For h = 0.01: f'(1.5) = 4.48169 og feilen blir 0.0
For h = 0.001: f'(1.5) = 4.48169 og feilen blir 0.0
For h = 0.0001: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-05: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-06: f'(1.5) = 4.48169 og feilen blir 0.0
For h = 1e-07: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-08: f'(1.5) = 4.48169 og feilen blir 0.0
For h = 1e-09: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-10: f'(1.5) = 4.48169 og feilen blir -0.0
For h = 1e-11: f'(1.5) = 4.48172 og feilen blir -3e-05
For h = 1e-12: f'(1.5) = 4.48256 og feilen blir -0.00087
For h = 1e-13: f'(1.5) = 4.48012 og feilen blir 0.00157
For h = 1e-14: f'(1.5) = 4.4853 og feilen blir -0.00361
For h = 1e-15: f'(1.5) = 5.10703 og feilen blir -0.62534
For h = 1e-16: f'(1.5) = -1.4803 og feilen blir 5.96199
For h = 1e-17: f'(1.5) = -7.40149 og feilen blir 11.88318
For h = 1e-18: f'(1.5) = -74.01487 og feilen blir 78.49656
For h = 1e-19: f'(1.5) = -740.14868 og feilen blir 744.63037
```

Denne metoden gir svært liten feil også for store verdier av h, men for svært små verdien av h blir feilen stor.

4

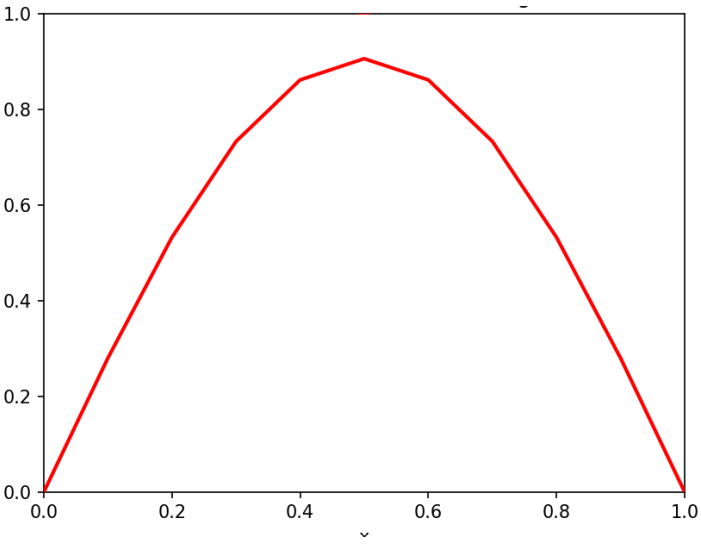
```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

alpha = 1
L = 1.0
Nx = 10
Nt = 100
h = L / Nx
k = 0.001
lambda_ = alpha * k / h**2

x = np.linspace(0, L, Nx+1)
u = np.sin(np.pi * x)
solution = [u.copy()]

for n in range(Nt):
    u_new = u.copy()
    for j in range(1, Nx):
        u_new[j] = u[j] + lambda_ * (u[j+1] - 2*u[j] + u[j-1])
    u = u_new.copy()
    solution.append(u)

fig, ax = plt.subplots()
ax.set_xlim(0, L)
ax.set_ylim(0, 1)
line, = ax.plot(x, solution[0], lw=2)

def update(frame):
    line.set_ydata(solution[frame])
    ax.set_title(f"t = {frame * k:.4f}")
    return line,

ani = animation.FuncAnimation(fig, update, frames=range(0, Nt, Nt//10), blit=True)
plt.xlabel("x")
plt.ylabel("Temperature")
plt.show()
```



Her er en av figurene fra eksplisitt metoden. Simulasjonen ble ustabil både når k var større en h og når k var lik h. Også når k bare var litt mindre enn h. Først når k var ca. 100 ganger mindre enn h ble simulasjonen stabil. Dette er fordi k må være mindre eller lik $h^2/2$. Ser også at h^{-1} tilsvarer antall linjer grafen består av.

5

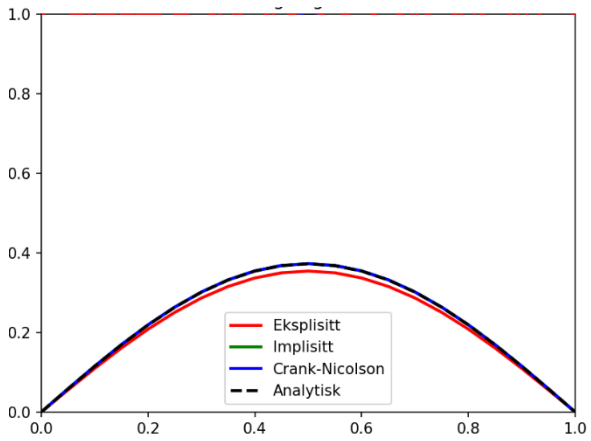
```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

alpha = 1
L = 1.0
Nx = 10
Nt = 100
h = L / Nx
k = 0.001
lambda_ = alpha * k / (2 * h**2)

x = np.linspace(0, L, Nx+1)
u = np.sin(np.pi * x)

A = np.zeros((Nx-1, Nx-1))
B = np.zeros((Nx-1, Nx-1))

for i in range(Nx-1):
    A[i, i] = 1 + 2 * lambda_
    B[i, i] = 1 - 2 * lambda_

    if i > 0:
        A[i, i-1] = -lambda_
        B[i, i-1] = lambda_

    if i < Nx-2:
        A[i, i+1] = -lambda_
        B[i, i+1] = lambda_

solution = [u.copy()]

for n in range(Nt):
    b = B @ u[1:-1]
    u_new_inner = np.linalg.solve(A, b)
    u[1:-1] = u_new_inner
    solution.append(u.copy())

fig, ax = plt.subplots()
ax.set_xlim(0, L)
ax.set_ylim(0, 1)
line, = ax.plot([], [], 'r-', lw=2)
ax.set_xlabel("x")
ax.set_ylabel("Temperatur")
ax.set_title("Crank-Nicolson - Varmeledning")

def init():
    line.set_data(x, solution[0])
    return line,

def update(frame):
    line.set_data(x, solution[frame])
    ax.set_title(f"t = {frame * k:.4f}")
    return line,

ani = animation.FuncAnimation(fig, update, frames=len(solution), init_func=init, blit=True, interval=50)

plt.show()
```

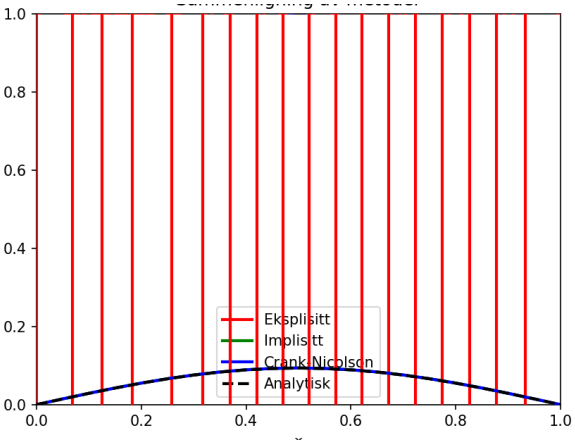


Implisitt metode ga en mye mer robust simulering som fungerte både når k var større enn, mindre enn og lik h. h justerer stegene i romdimensjonen, mens k justerer stegene i tidsdimensjonen.

6

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def analytical_solution(x, t, alpha=1):
    return np.exp(-np.pi**2 * alpha * t) * np.sin(np.pi * x)

alpha = 1
L = 1.0
Nx = 100
Nt = 100
h = L / Nx
k = 0.003
lambda_ = alpha * k / (h**2)

x = np.linspace(0, L, Nx+1)
u_explicit = np.sin(np.pi * x)
u_implicit = u_explicit.copy()
u_cn = u_explicit.copy()

# Eksplisitt metode
solution_explicit = [u_explicit.copy()]
for n in range(Nt):
    u_new = u_explicit.copy()
    for i in range(1, Nx):
        u_new[i] = u_explicit[i] + lambda_ * (u_explicit[i-1] - 2*u_explicit[i] + u_explicit[i+1])
    u_explicit[:] = u_new
    solution_explicit.append(u_explicit.copy())

# Implisitt metode
A = np.zeros((Nx-1, Nx-1))
B = np.zeros((Nx-1, Nx-1))
lambda_implicit = alpha * k / (2 * h**2)
for i in range(Nx-1):
    A[i, i] = 1 + 2 * lambda_implicit
    B[i, i] = 1 - 2 * lambda_implicit
    if i > 0:
        A[i, i-1] = -lambda_implicit
        B[i, i-1] = lambda_implicit
    if i < Nx-2:
        A[i, i+1] = -lambda_implicit
        B[i, i+1] = lambda_implicit
solution_implicit = [u_implicit.copy()]
for n in range(Nt):
    b = B @ u_implicit[1:-1]
    u_new_inner = np.linalg.solve(A, b)
    u_implicit[1:-1] = u_new_inner
    solution_implicit.append(u_implicit.copy())

# Crank-Nicolson metode
solution_cn = [u_cn.copy()]
for n in range(Nt):
    b = B @ u_cn[1:-1]
    u_new_inner = np.linalg.solve(A, b)
    u_cn[1:-1] = u_new_inner
    solution_cn.append(u_cn.copy())

# Sammenligning med analytisk løsning
solution_analytical = [analytical_solution(x, n*k) for n in range(Nt)]

fig, ax = plt.subplots()
ax.set_xlim(0, L)
ax.set_ylim(0, 1)
line_explicit, = ax.plot([], [], 'r-', lw=2, label='Eksplisitt')
line_implicit, = ax.plot([], [], 'g-', lw=2, label='Implisitt')
line_cn, = ax.plot([], [], 'b-', lw=2, label='Crank-Nicolson')
line_analytical, = ax.plot([], [], 'k--', lw=2, label='Analytisk')
ax.set_xlabel("x")
ax.set_ylabel("Temperatur")
ax.set_title("Sammenligning av metoder")
ax.legend()

def init():
    line_explicit.set_data(x, solution_explicit[0])
    line_implicit.set_data(x, solution_implicit[0])
    line_cn.set_data(x, solution_cn[0])
    line_analytical.set_data(x, solution_analytical[0])
    return line_explicit, line_implicit, line_cn, line_analytical

def update(frame):
    line_explicit.set_data(x, solution_explicit[frame])
    line_implicit.set_data(x, solution_implicit[frame])
    line_cn.set_data(x, solution_cn[frame])
    line_analytical.set_data(x, solution_analytical[frame])
    ax.set_title(f"t = {frame * k:.4f}")
    return line_explicit, line_implicit, line_cn, line_analytical

ani = animation.FuncAnimation(fig, update, frames=Nt, init_func=init, blit=True, interval=50)
plt.show()
```





Her ser vi at i begynnelsen følger alle metodene greit på den analytiske, med unntak av den eksplisitte som ligger litt under. Men ettersom tiden går blir eksplisitt metode ustabil. Crank Nicolsen er en kombinasjon av de andre to som fungerer svært godt.