OBLIG Oppgave TMA4106

Ruben Johnsen

April 2025

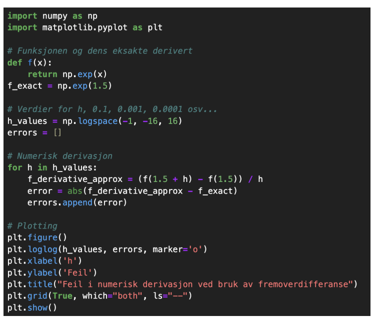
1 Introduction

I denne oppgaven jobber vi med numeriske metoder for derivasjon og løsning av partielle differensiallikninger, med spesielt fokus på varmelikningen.

Første del av oppgaven handler om numerisk derivasjon. Her undersøker vi hvordan feilen utvikler seg når man bruker forskjellige metoder. Ved å variere steglengden h og analysere feilene, får man innsikt i hvordan Taylorutvikling og maskinpresisjon påvirker resultatet.

Andre del av oppgaven handler om å løse varmelikningen numerisk. Jeg bruker tre forskjellige metoder, eksplisitt Euler, implisitt Euler, og Crank-Nicolson. Til slutt sammenligner jeg resultatene fra de numeriske metodene med den analytiske løsningen for en enkel initialverdi, u(x,0) = $\sin(\pi x)$

Oppgave 1



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

# Funksjonen og dens eksakte derivert
def f(x):
    return np.exp(x)
f_exact = np.exp(1.5)

# Verdier for h, 0.1, 0.001, 0.0001 osv...
h_values = np.logspace(-1, -16, 16)
errors = []

# Numerisk derivasjon
for h in h_values:
    f_derivative_approx = (f(1.5 + h) - f(1.5)) / h
    error = abs(f_derivative_approx - f_exact)
    errors.append(error)

# Plotting
plt.figure()
plt.loglog(h_values, errors, marker='o')
plt.xlabel('h')
plt.ylabel('Feil')
plt.title("Feil i numerisk derivasjon ved bruk av fremoverdifferanse")
plt.grid(True, which="both", ls="--")
plt.show()
```

Figure 1: Kode til oppgave 1.

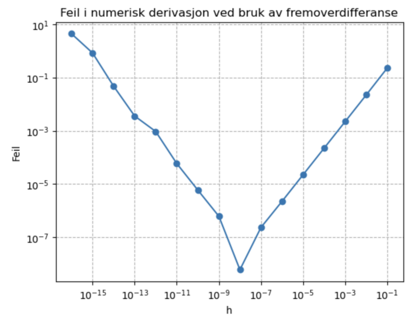


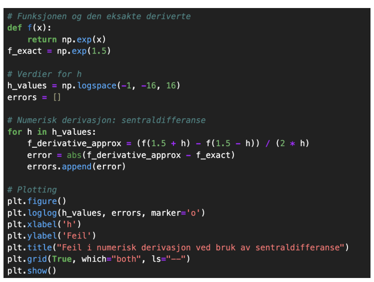
Figure 2: Plott til oppgave 1.

Figur 1 og 2 viser nøyaktigheten til fremoverdifferanse som metode for numerisk derivasjon. Metoden er basert på formelen

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$

og benyttes til å tilnærme den deriverte av funksjonen $f(x) = e^x$ i punktet $x = 1.5$. Den eksakte verdien er kjent, $f'(1.5) = e^{1.5}$, og dette gjør det mulig å beregne feilen ved ulike valg av h. Resultatene viser at feilen i starten avtar lineært når h reduseres, i tråd med forventet teori. Likevel kan vi se øker feilen igjen når h blir svært liten. Dette skyldes numeriske avrundingsfeil som oppstår når datamaskinen prøver å representere små tall.

Oppgave 2



```python
# Funksjonen og den eksakte deriverte
def f(x):
    return np.exp(x)
f_exact = np.exp(1.5)

# Verdier for h
h_values = np.logspace(-1, -16, 16)
errors = []

# Numerisk derivasjon: sentraldifferanse
for h in h_values:
    f_derivative_approx = (f(1.5 + h) - f(1.5 - h)) / (2 * h)
    error = abs(f_derivative_approx - f_exact)
    errors.append(error)

# Plotting
plt.figure()
plt.loglog(h_values, errors, marker='o')
plt.xlabel('h')
plt.ylabel('Feil')
plt.title("Feil i numerisk derivasjon ved bruk av sentraldifferanse")
plt.grid(True, which="both", ls="--")
plt.show()
```

Figure 3: Kode til oppgave 2.

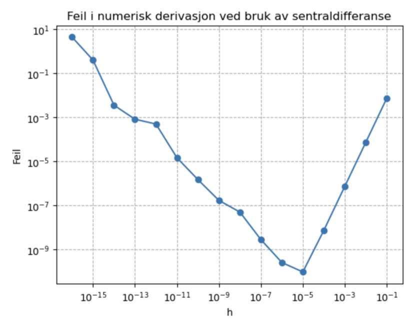


Figure 4: Plott til oppgave 2.

Her benyttes sentraldifferanse til numerisk derivasjon, med formelen

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

Denne metoden gir en mer nøyaktig tilnærming enn fremoverdifferansen, ettersom feilen her er proporsjonal med h^2. Beregningene viser at feilen reduseres betydelig raskere når h blir mindre, og sentraldifferanse tillater bruk av mindre steglengder før rundingsfeil blir merkbare. Metoden demonstrerer dermed bedre numerisk stabilitet og høyere presisjon innenfor rimelige verdier av h.

Oppgave 3

I figur 5 og 6 benyttes en fjerdeordens differanseformel som kombinerer flere funksjonsverdier rundt punktet x, og som gir en veldig presis tilnærming til den deriverte. Formelen er gitt som

$$f'(x) \approx \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h},$$

```python
# Funksjonen og eksakt verdi
def f(x):
    return np.exp(x)
f_exact = np.exp(1.5)

# h-verdier og feilliste
h_values = np.logspace(-1, -16, 16)
errors = []

# 4-punkts differanse
for h in h_values:
    f_derivative_approx = (f(1.5 - 2*h) - 8*f(1.5 - h) + 8*f(1.5 + h)
                           - f(1.5 + 2*h)) / (12 * h)
    error = abs(f_derivative_approx - f_exact)
    errors.append(error)

# Plotting
plt.figure()
plt.loglog(h_values, errors, marker='o')
plt.xlabel('h')
plt.ylabel('Feil')
plt.title("Feil i numerisk derivasjon med 4-punkts formel")
plt.grid(True, which="both", ls="--")
plt.show()
```

Figure 5: Kode til oppgave 3.

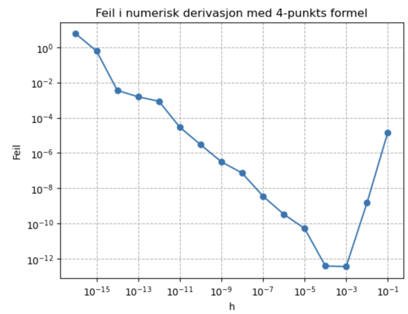
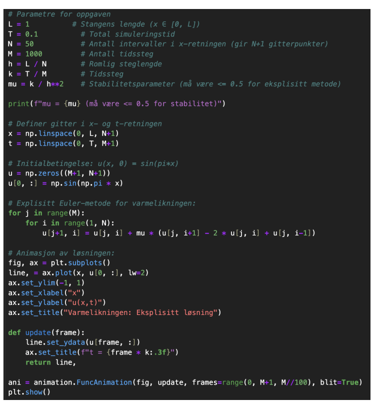


Figure 6: Plott til oppgave 3.

og har en feil som er proporsjonal med h^4. Plottet bekrefter at feilen avtar raskt når h reduseres, og metoden gir veldig høy presisjon sammenlignet med de tidligere metodene. Avrundingsfeil oppstår også her for små h.

Oppgave 4



```python
# Parametre for oppgaven
L = 1          # Stangens lengde (x ∈ [0, L])
T = 0.1          # Total simuleringstid
N = 50           # Antall intervaller i x-retningen (gir N+1 gitterpunkter)
M = 1000         # Antall tidssteg
h = L / N        # Romlig steglengde
k = T / M        # Tidssteg
mu = k / h**2    # Stabilitetsparameter (må være <= 0.5 for eksplisitt metode)

print(f"mu = {mu} (må være <= 0.5 for stabilitet)")

# Definer gitter i x- og t-retningen
x = np.linspace(0, L, N+1)
t = np.linspace(0, T, M+1)

# Initialbetingelse: u(x, 0) = sin(pi*x)
u = np.zeros((M+1, N+1))
u[0, :] = np.sin(np.pi * x)

# Explisitt Euler-metode for varmelikningen:
for j in range(M):
    for i in range(1, N):
        u[j+1, i] = u[j, i] + mu * (u[j, i+1] - 2 * u[j, i] + u[j, i-1])

# Animasjon av løsningen:
fig, ax = plt.subplots()
line, = ax.plot(x, u[0, :], lw=2)
ax.set_ylim(-1, 1)
ax.set_xlabel("x")
ax.set_ylabel("u(x,t)")
ax.set_title("Varmelikningen: Eksplisitt løsning")

def update(frame):
    line.set_ydata(u[frame, :])
    ax.set_title(f"t = {frame * k:.3f}")
    return line,

ani = animation.FuncAnimation(fig, update, frames=range(0, M+1, M//100), blit=True)
plt.show()
```

Figure 7: Kode til oppgave 4.

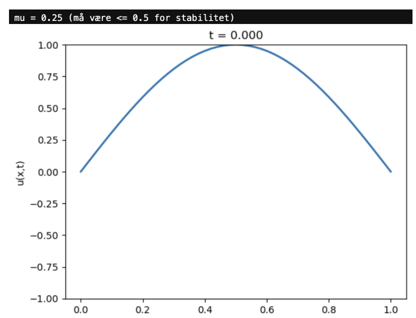


Figure 8: Plott til oppgave 4.

Figuren viser den numeriske løsningen $u(x, T)$ for varmelikningen, der initialbetingelsen er

$$u(x, 0) = \sin(\pi x)$$

og randbetingelsene er $u(0, t) = u(1, t) = 0$. Løsningen er oppnådd med eksplisitt Euler-metode, som forutsetter at stabilitetsparameteren

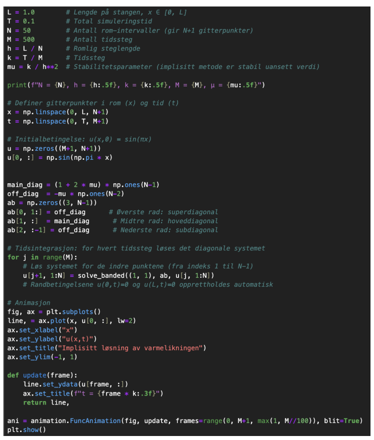
$$\mu = \frac{k}{h^2}$$

er mindre enn eller lik 0.5.

Når man reduserer h (øker antallet romintervaller) oppnås en finere romlig oppløsning, men h^2 minker raskt. Hvis tidssteget k holdes konstant, vil μ øke, noe som kan medføre numerisk ustabilitet dersom $\mu > 0.5$. Tilsvarende vil et for stort k (få tidssteg) også øke μ og føre til ustabile løsninger.

Ved korrekte valg av h og k (slik at $\mu \leq 0.5$) følger den numeriske løsningen den analytiske utviklingen, med diffusjon som gradvis flater ut initialtilstanden. På den måten illustrerer plottet både effekten av oppløsningen og viktigheten av å oppfylle stabilitetsbetingelsene for en nøyaktig og stabil simulering.

Oppgave 5



```python
L = 1.0        # Lengde på stangen, x ∈ [0, L]
T = 0.1        # Total simuleringstid
N = 50         # Antall rom-intervaller (gir N+1 gitterpunkter)
M = 500        # Antall tidssteg
h = L / N      # Romlig steglengde
k = T / M      # Tidssteg
mu = k / h**2  # Stabilitetsparameter (implisitt metode er stabil uansett verdi)

print(f"N = {N}, h = {h:.5f}, k = {k:.5f}, M = {M}, μ = {mu:.5f}")

# Definer gitterpunkter i rom (x) og tid (t)
x = np.linspace(0, L, N+1)
t = np.linspace(0, T, M+1)

# Initialbetingelse: u(x,0) = sin(πx)
u = np.zeros((M+1, N+1))
u[0, :] = np.sin(np.pi * x)


main_diag = (1 + 2 * mu) * np.ones(N-1)
off_diag  = -mu * np.ones(N-2)
ab = np.zeros((3, N-1))
ab[0, 1:] = off_diag      # Øverste rad: superdiagonal
ab[1, :]  = main_diag      # Midtre rad: hoveddiagonal
ab[2, :-1] = off_diag      # Nederste rad: subdiagonal

# Tidsintegrasjon: for hvert tidssteg løses det diagonale systemet
for j in range(M):
    # Løs systemet for de indre punktene (fra indeks 1 til N-1)
    u[j+1, 1:N] = solve_banded((1, 1), ab, u[j, 1:N])
    # Randbetingelsene u(0,t)=0 og u(L,t)=0 opprettholdes automatisk

# Animasjon
fig, ax = plt.subplots()
line, = ax.plot(x, u[0, :], lw=2)
ax.set_xlabel("x")
ax.set_ylabel("u(x,t)")
ax.set_title("Implisitt løsning av varmelikningen")
ax.set_ylim(-1, 1)

def update(frame):
    line.set_ydata(u[frame, :])
    ax.set_title(f"t = {frame * k:.3f}")
    return line,

ani = animation.FuncAnimation(fig, update, frames=range(0, M+1, max(1, M//100)), blit=True)
plt.show()
```

Figure 9: Kode til oppgave 5.

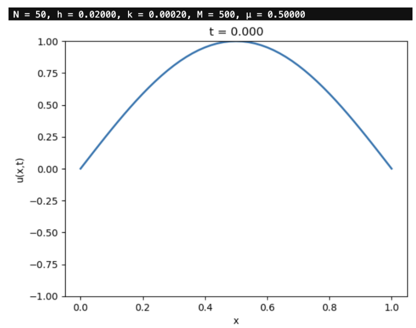


Figure 10: Plott til oppgave 5.

Vi løser varmelikningen

$$u_t = u_{xx}, \quad 0 \leq x \leq 1, \ t > 0,$$

med randbetingelser $u(0,t) = u(1,t) = 0$ og initialbetingelsen

$$u(x,0) = \sin(\pi x).$$

Vi diskretiserer rom med $x_i = ih$ og tid med $t_j = jk$, der $h = 1/N$ og $k = T/M$. Tidsderivert tilnærmes med den implisitte Euler-metoden:

$$\frac{u_i^{j+1} - u_i^j}{k} \approx u_{xx}(x_i, t_{j+1}),$$

og rom-derivert med en sentral differanse:

$$u_{xx}(x_i, t_{j+1}) \approx \frac{u_{i+1}^{j+1} - 2u_i^{j+1} + u_{i-1}^{j+1}}{h^2}.$$

Dette gir det tridiagonale systemet for de indre punktene ($i = 1, \ldots, N-1$):

$$-\mu\, u_{i-1}^{j+1} + (1+2\mu)\, u_i^{j+1} - \mu\, u_{i+1}^{j+1} = u_i^j,$$

med $\mu = \frac{k}{h^2}$ og $u_0^{j+1} = u_N^{j+1} = 0$. Metoden er ubetinget stabil, noe som betyr at vi kan velge k og h uten å bekymre oss for ustabilitet.

Oppgave 6

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import solve_banded

# Parametre
L = 1.0             # Domenelengde
T = 0.2             # Simuleringstid (økt slik at feil akkumuleres)
N = 50              # Antall romintervaller; dermed h = L/N
h = L / N
k = 0.0002          # Tidssteg valgt slik at μ = k/h^2 = 0.0002/(0.02^2) = 0.5
M = int(T / k)      # Antall tidssteg
mu = k / h**2

print(f"N = {N}, h = {h:.5f}, k = {k:.5f}, M = {M}, μ = {mu:.5f}")

# Diskretiseringspunkter i x
x = np.linspace(0, L, N+1)
# Analytisk løsning ved t = T: u(x,T) = sin(pi*x)*exp(-pi^2*T)
u_exact = np.sin(np.pi * x) * np.exp(-np.pi**2 * T)

# Initialbetingelse: u(x, 0) = sin(pi*x)
def initial(x):
    return np.sin(np.pi*x)

# Lagring av løsninger for hver metode
u_exp = np.zeros((M+1, N+1))   # Eksplisitt Euler
u_impl = np.zeros((M+1, N+1))  # Implisitt Euler
u_cn = np.zeros((M+1, N+1))    # Crank-Nicolson

u_exp[0, :] = initial(x)
u_impl[0, :] = initial(x)
u_cn[0, :] = initial(x)

# --- Eksplisitt Euler ---
for j in range(M):
    for i in range(1, N):
        u_exp[j+1, i] = u_exp[j, i] + mu * (u_exp[j, i+1] - 2*u_exp[j, i] + u_exp[j, i-1])
    # Randbetingelser: u(0,t) og u(L,t) er 0

# --- Implisitt Euler ---
main_diag = (1 + 2*mu) * np.ones(N-1)
off_diag  = -mu * np.ones(N-2)
ab_impl = np.zeros((3, N-1))
ab_impl[0, 1:] = off_diag     # Superdiagonalen
ab_impl[1, :]  = main_diag     # Hoveddiagonalen
ab_impl[2, :-1] = off_diag     # Subdiagonalen

for j in range(M):
    u_impl[j+1, 1:N] = solve_banded((1, 1), ab_impl, u_impl[j, 1:N])
```

Figure 11: Kode til oppgave 6. Trengt litt hjelp fra AI med denne.

```python
# --- Crank-Nicolson ---
B = np.zeros((N-1, N-1))
np.fill_diagonal(B, 1 - mu)
np.fill_diagonal(B[1:], mu/2)
np.fill_diagonal(B[:, 1:], mu/2)

ab_A = np.zeros((3, N-1))
ab_A[0, 1:] = -mu/2         # Superdiagonalen
ab_A[1, :]  = 1 + mu         # Hoveddiagonalen
ab_A[2, :-1] = -mu/2         # Subdiagonalen

for j in range(M):
    rhs = B @ u_cn[j, 1:N]
    u_cn[j+1, 1:N] = solve_banded((1, 1), ab_A, rhs)

# --- Plotting ---
plt.figure(figsize=(10, 6))
plt.plot(x, u_exp[-1, :], label='Eksplisitt Euler', linewidth=2)
plt.plot(x, u_impl[-1, :], label='Implisitt Euler', linewidth=2)
plt.plot(x, u_cn[-1, :], label='Crank-Nicolson', linewidth=2)
plt.plot(x, u_exact, 'k--', label='Analytisk løsning', linewidth=2)
plt.xlabel('x')
plt.ylabel('u(x,T)')
plt.title(f"Sammenligning av numeriske metoder ved T = {T}")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# --- Feilmåling ---
error_exp = np.linalg.norm(u_exp[-1, :] - u_exact, np.inf)
error_impl = np.linalg.norm(u_impl[-1, :] - u_exact, np.inf)
error_cn = np.linalg.norm(u_cn[-1, :] - u_exact, np.inf)
print("Feil (maksimum feil):")
print("Eksplisitt Euler: ", error_exp)
print("Implisitt Euler: ", error_impl)
print("Crank-Nicolson: ", error_cn)
```

Figure 12: Kode til oppgave 6. Trengt litt hjelp fra AI med denne.

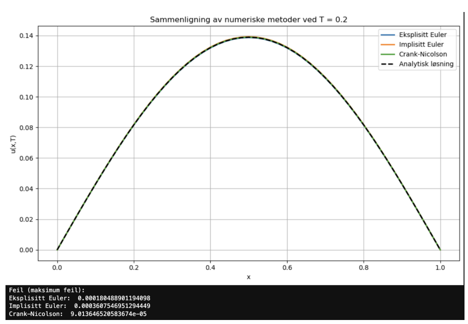


Figure 13: Plott til oppgave 6.

Vi løser varmelikningen

$$u_t = u_{xx}, \quad 0 \le x \le 1,\ t > 0,$$

med randbetingelser

$$u(0, t) = u(1, t) = 0,$$

og initialbetingelsen

$$u(x, 0) = \sin(\pi x).$$

Den analytiske løsningen for dette problemet er

$$u(x, t) = \sin(\pi x)e^{-\pi^2 t}.$$

Crank-Nicolson-metoden er en tidssentrert, implisitt metode som kombinerer fordelene til både eksplisitt og implisitt Euler. Den tilnærmer tidsderivert ved

$$\frac{u_i^{j+1} - u_i^j}{k} \approx \frac{1}{2}\left(\frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{h^2} + \frac{u_{i+1}^{j+1} - 2u_i^{j+1} + u_{i-1}^{j+1}}{h^2}\right),$$

hvor h er romsteget og k er tidssteget, med $\mu = \frac{k}{h^2}$.

Dette fører til det tridiagonale systemet for de indre punktene ($i = 1, \ldots, N-1$):

$$-\frac{\mu}{2}\, u_{i-1}^{j+1} + (1+\mu)\, u_i^{j+1} - \frac{\mu}{2}\, u_{i+1}^{j+1} = \frac{\mu}{2}\, u_{i-1}^{j} + (1-\mu)\, u_i^{j} + \frac{\mu}{2}\, u_{i+1}^{j}.$$

Randbetingelsene $u_0^{j+1} = u_N^{j+1} = 0$ benyttes for å fullføre systemet.

Metoden er numerisk stabil og oppnår andreordens nøyaktighet i både tid og rom. Ved å løse systemet for hvert tidssteg kan vi sammenligne den numeriske løsningen med den analytiske løsningen for å verifisere presisjonen til metoden, noe vi ser i figur 13.