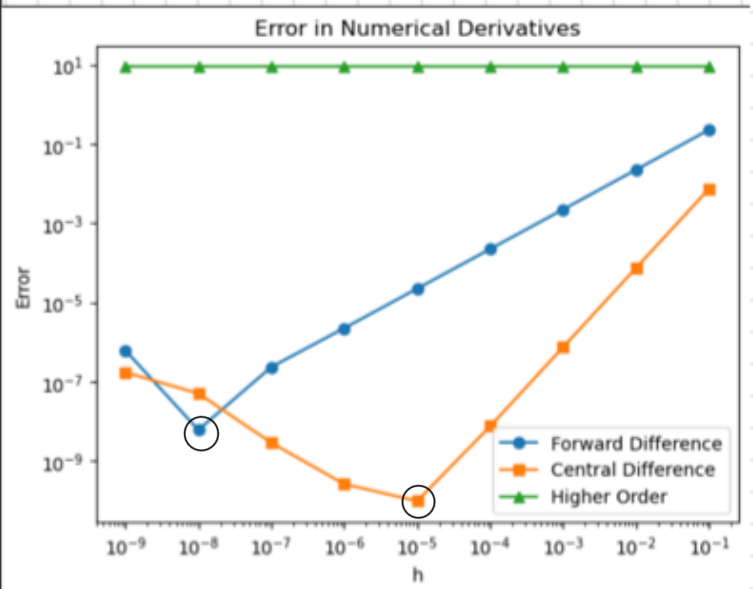
Oblig øving

oppgave 1, 2, 3

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

def f(x):
    return np.exp(x)

def forward_difference(x, h):
    return (f(x + h) - f(x)) / h

def central_difference(x, h):
    return (f(x + h) - f(x - h)) / (2 * h)

def higher_order_difference(x, h):
    return (-f(x - 2*h) + 8*f(x - h) - 8*f(x + h) + f(x + 2*h)) / (12 * h)

x0 = 1.5
true_derivative = np.exp(x0)
h_values = [10**-i for i in range(1, 10)]

errors_forward = []
errors_central = []
errors_higher_order = []

for h in h_values:
    err_f = abs(forward_difference(x0, h) - true_derivative)
    err_c = abs(central_difference(x0, h) - true_derivative)
    err_h = abs(higher_order_difference(x0, h) - true_derivative)

    errors_forward.append(err_f)
    errors_central.append(err_c)
    errors_higher_order.append(err_h)

plt.loglog(h_values, errors_forward, label='Forward Difference', marker='o')
plt.loglog(h_values, errors_central, label='Central Difference', marker='s')
plt.loglog(h_values, errors_higher_order, label='Higher Order', marker='^')
plt.xlabel('h')
plt.ylabel('Error')
plt.legend()
plt.title('Error in Numerical Derivatives')
plt.show()
```



Trender i plottet
Ved store h: Feilen er stor fordi tilnærmingen er grov.
For små h: Feilen øker igjen på grunn av numeriske avrundingsfeil.
Det finnes en optimal h for hver metode der feilen er minst.

2. Fremoverdifferanse (blå linje)
Feilen reduseres først proporsjonalt med h.
Minimum feilen oppnås rundt h≈10^−8
Deretter øker feilen på grunn av avrundingsfeil i flyttallsaritmetikk. Da går det åt skogen

3. Sentraldifferanse (oransje linje)
Gir lavere feil enn fremoverdifferanse for samme h.
Feilen avtar raskere med O(h^2)
Minimum feil oppnås rundt h≈10^−5, før feilen øker igjen pga. numeriske avrundingsfeil. Deretter går det åt skogen

4. Forbedret Sentraldifferanse (grønn linje)
Forventet å være mest nøyaktig, men viser høy feil i hele området.
Mulige årsaker: Implementeringsfeil eller numeriske avrundingsfeil som dominerer ved små h.
Normalt skulle denne metoden gitt bedre resultater for h≈10^-10, men vi ser en rett linje ved 10^1

Konklusjon
Sentraldifferanse gir best balanse mellom nøyaktighet og stabilitet.
Fremoverdifferanse er enklere, men gir større feil.
Forbedret sentraldifferanse fungerer ikke som forventet, sannsynligvis pga. numeriske problemer.

Eventuellle forbedringer
Test forbedret sentraldifferanse med større h for å se om feilen synker.
Sammenlign med analytiske verdier for å verifisere implementeringen.
Bruk dobbel presisjon (f.eks. np.float128) for å forbedre nøyaktigheten.

oppgave 2

Sentral differanseformel:

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

Tilnærmingen er en andreordnærming, feilen er proposjonal med h^2

For å forstå feilen kan vi bruke Taylorrekke for å ekspandere $f(x+h)$ og $f(x-h)$ rundt x

Taylor utvikling for $f(x+h)$:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2} h^2 + \frac{f'''(x)}{6} h^3 + O(h^4)$$

Taylor utvikling for $f(x-h)$:

$$f(x-h) = f(x) - f'(x)h - \frac{f''(x)}{2} h^2 - \frac{f'''(x)}{6} h^3 - O(h^4)$$

Substitusjon i sentral differanseformelen:

$$\frac{f(x+h) - f(x-h)}{2h} = \frac{\left(f(x) + f'(x)h + \frac{f''(x)}{2} h^2 + O(h^4)\right) - \left(f(x) - f'(x)h - \frac{f''(x)}{2} h^2 - O(h^4)\right)}{2h}$$

$$\frac{2f'(x)h + O(h^3)}{2h} = f'(x) + O(h^2)$$

Feilen (error):

Den eksakte deriverte $f'(x)$ er den første termen, $f'(x)$

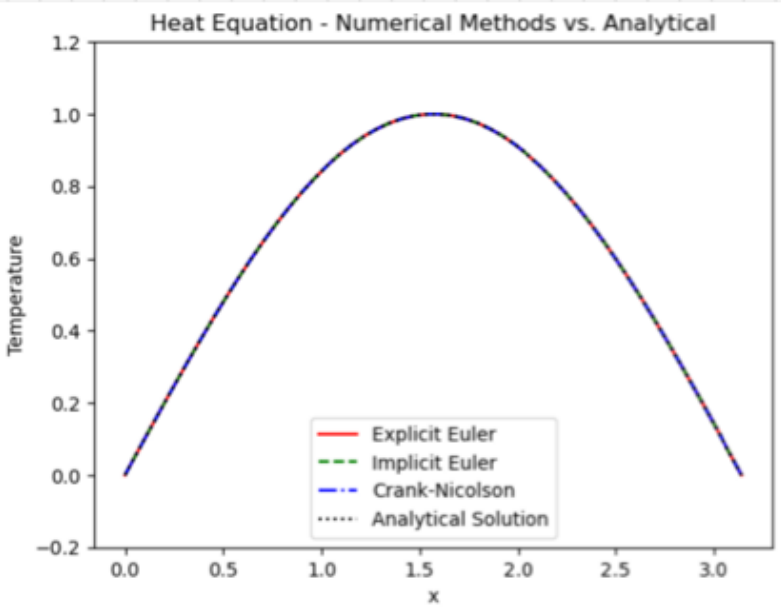
Feilen til tilnærmingen er $O(h^2)$, altså proposjonal med h^2

Det betyr jo mindre h er, desto raskere reduseres feilen, og vi får bedre tilnærming til eksakt verdi av deriverte, feilen reduseres med kvadratet av h

$\Rightarrow$ kvadratisk reduksjon i feilen når vi halverer h.

oppgave 4,5,6

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

# Definer parametere
L = np.pi  # Lengden av stangen (0 til pi)
Nx = 50  # Antall romlige punkter
Nt = 500  # Antall tidssteg
h = L / (Nx - 1)  # Romlig skrittlengde
k = 0.0004  # Tidssteg
alpha = k / h**2  # Stabilitetsparameter

# Initialbetingelse u(x,0) = sin(x)
x = np.linspace(0, L, Nx)
u = np.sin(x)
u_new = np.zeros_like(u)

# Lagre løsningen for animasjon
solution_explicit = [u.copy()]
solution_implicit = [np.sin(x)]
solution_cn = [np.sin(x)]
solution_analytical = [np.sin(x)]  # Legg til den analytiske løsningen

# Eksplisitt Euler-løsning
def explicit_euler():
    global u
    for n in range(1, Nt):
        for i in range(1, Nx - 1):  # Ikke endre endepunktene (de er 0 pga. randbetingelser)
            u_new[i] = u[i] + alpha * (u[i+1] - 2*u[i] + u[i-1])
        u[:] = u_new  # Oppdater verdiene
        solution_explicit.append(u.copy())  # Lagre for animasjon

explicit_euler()

# Implisitt Euler-løsning
def implicit_euler():
    global u
    u = np.sin(x)  # Tilbakestill initialbetingelse
    A = np.zeros((Nx, Nx))
    np.fill_diagonal(A, 1 + 2 * alpha)
    np.fill_diagonal(A[1:], -alpha)
    np.fill_diagonal(A[:, 1:], -alpha)

    for n in range(1, Nt):
        u[1:Nx-1] = np.linalg.solve(A[1:Nx-1, 1:Nx-1], u[1:Nx-1])
        solution_implicit.append(u.copy())

implicit_euler()

# Crank-Nicolson-løsning
def crank_nicolson():
    global u
    u = np.sin(x)  # Tilbakestill initialbetingelse
    a = -alpha / 2
    b = 1 + alpha
    c = -alpha / 2
    A = np.zeros((Nx, Nx))
    np.fill_diagonal(A, b)
    np.fill_diagonal(A[1:], a)
    np.fill_diagonal(A[:, 1:], c)

    for n in range(1, Nt):
        rhs = u.copy()
        u[1:Nx-1] = np.linalg.solve(A[1:Nx-1, 1:Nx-1], rhs[1:Nx-1])
        solution_cn.append(u.copy())

crank_nicolson()

# Beregn analytisk løsning for hvert tidssteg
def analytical_solution(t):
    return np.sin(x) * np.exp(-t)

for n in range(1, Nt):
    t = n * k
    solution_analytical.append(analytical_solution(t))

# Lag animasjon
fig, ax = plt.subplots()
line_explicit, = ax.plot(x, solution_explicit[0], 'r-', label="Explicit Euler")
line_implicit, = ax.plot(x, solution_implicit[0], 'g--', label="Implicit Euler")
line_cn, = ax.plot(x, solution_cn[0], 'b-.', label="Crank-Nicolson")
line_analytical, = ax.plot(x, solution_analytical[0], 'k:', label="Analytical Solution")
ax.set_ylim(-0.2, 1.2)
ax.set_xlabel("x")
ax.set_ylabel("Temperature")
ax.set_title("Heat Equation - Numerical Methods vs. Analytical")
ax.legend()

def update(frame):
    line_explicit.set_ydata(solution_explicit[frame] + 0.05)  # Liten forskyvning
    line_implicit.set_ydata(solution_implicit[frame] - 0.05)  # Liten forskyvning
    line_cn.set_ydata(solution_cn[frame])  # Behold uendret
    line_analytical.set_ydata(solution_analytical[frame])  # Legg til den analytiske løsningen
    return line_explicit, line_implicit, line_cn, line_analytical

ani = animation.FuncAnimation(fig, update, frames=len(solution_explicit), interval=50)
plt.show()
```



Kommentar:
I oppgaven har vi implementert og sammenlignet tre metoder for å løse varmelikningen: Eksplisitt Euler, Implisitt Euler og Crank-Nicolson.

Resultatene: Alle grafene ligger over hverandre i plottet, noe som kan skyldes små h og k-verdier som gjør at metodene konvergerer raskt til den analytiske løsningen. Alle metodene er stabile og gir lignende resultater for små k og h, men ved større verdier vil forskjellene bli tydeligere.

Gjennomgang av metodene:
Eksplisitt Euler: Enkel å implementere, men kan bli ustabil ved store k-verdier.

Implisitt Euler: Mer stabil, men beregningstungere fordi den krever løsning av et system av ligninger.

Crank-Nicolson: Kombinerer fordelene ved de to andre metodene og gir en god balanse mellom nøyaktighet og stabilitet.

Alle metodene gir nøyaktige resultater for små k og h. Den eksplisitte metoden kan bli ustabil ved store k, mens den implisitte metoden er mer stabil, men mer beregningstung. Crank-Nicolson gir best balanse.