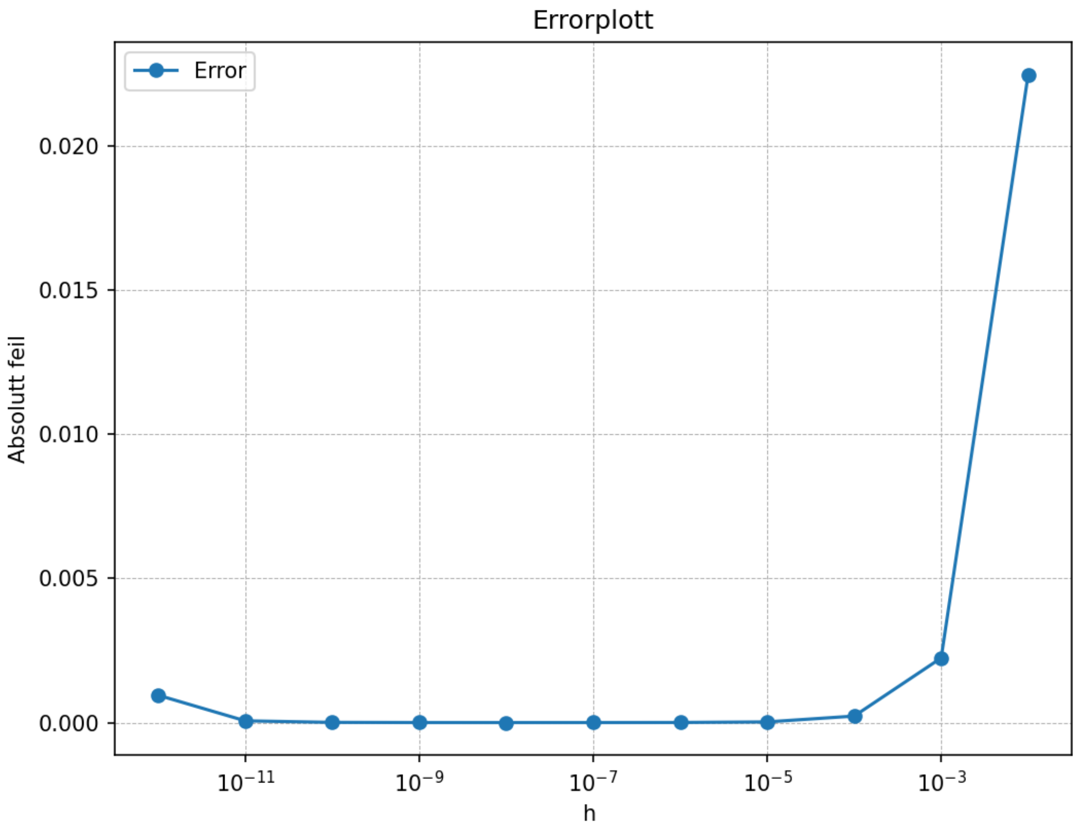
Standardprosjekt

1) For å løse denne oppgaven lagde jeg et pythonprogram som finner feilen for ulike h og plotter den.

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

h_values=[0.01,0.001,0.0001,0.00001,0.000001,0.0000001,0.00000001,0.000000001,0.0000000001,0.00000000001,0.000000000001]

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

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

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

calc_list=[]

for h in h_values:
    calc_list.append(f_prime_numerical(1.5,h))


for i in range(len(calc_list)-1):
    print("h: ",h_values[i]," Error: ",abs(calc_list[i]-f_prime_analytic(1.5)))

plt.figure(figsize=(8,6))
plt.plot(h_values,abs(calc_list-f_prime_analytic(1.5)),'-o',label='Error')
plt.xscale('log')

plt.xlabel("h ")
plt.ylabel("Absolutt feil")
plt.title("Errorplott")



plt.legend()
plt.grid(True, linestyle="--", linewidth=0.5)
plt.show()
```



Her ser man at for store h blir feilen veldig stor, for veldig små h vil feilen begynne å øke. Så svaret er da at for h mindre enn 1E-11 vil det oppstå problemer.

2)

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

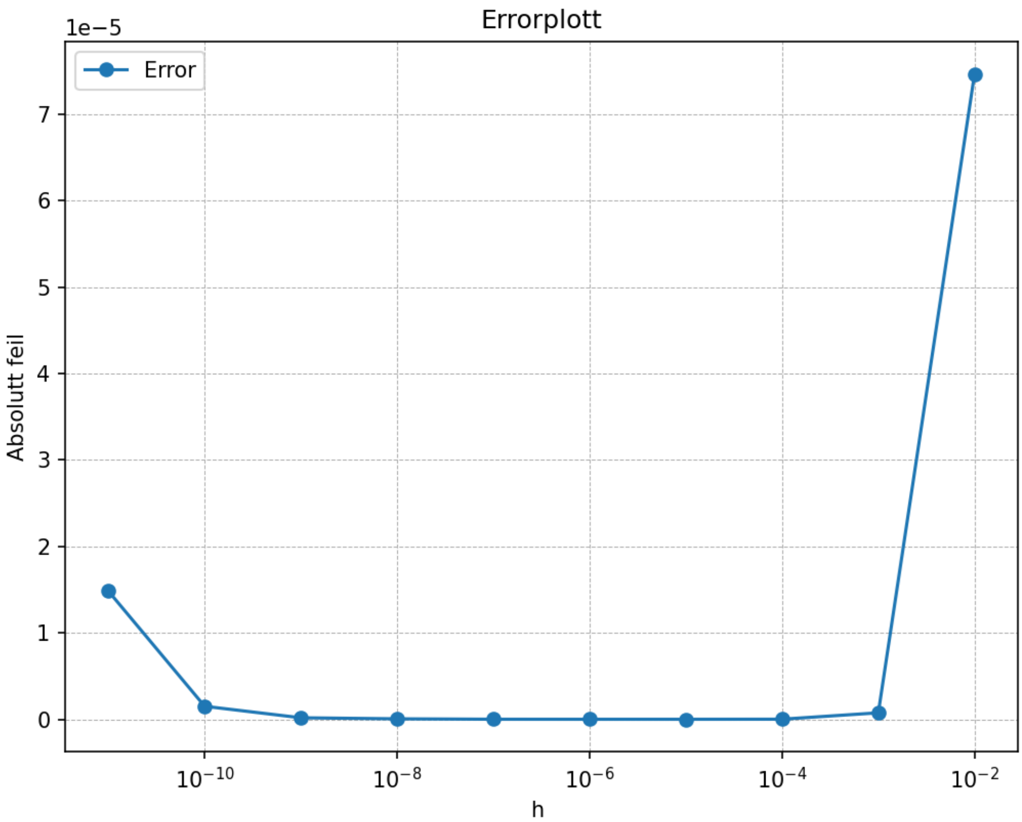
$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + O(h^4)$$

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

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

$$\text{feil} = \frac{h^2}{6}e^x$$

Her har jeg beskrevet taylorutviklingen for f(x+h) og f(x-h) og delt på 2h for å få en taylorutvikling for gitt formel. Feilen blir da det siste leddet. Antakelsen er at for store h vil feilen så klart bli veldig stor, og bli lavere for mindre h, helt til h blir veldig liten og feilen blir større.

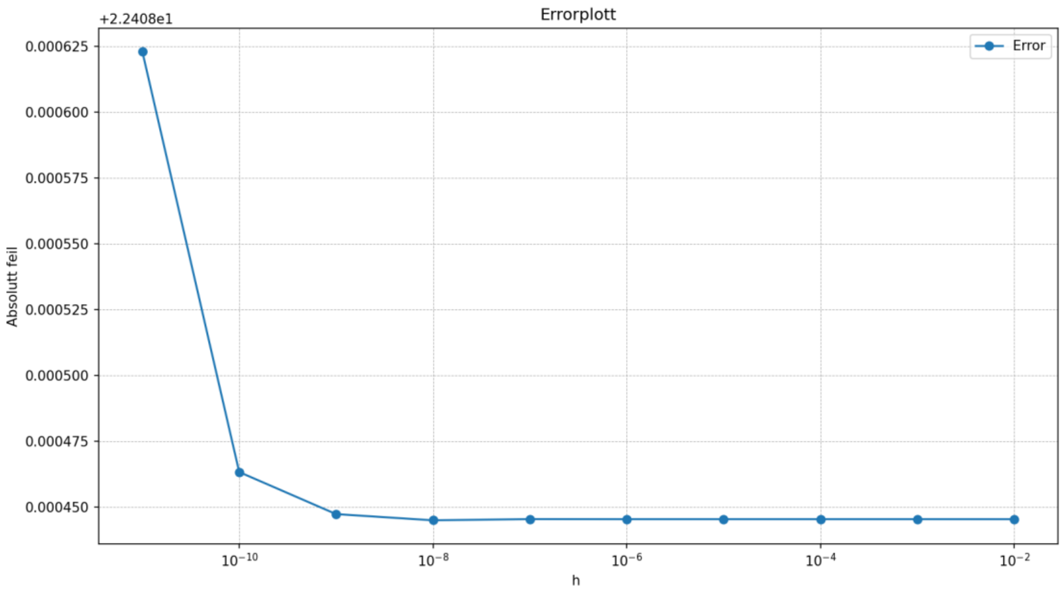


Dette er plottet for formelen over. Her ser man at antakelsen stemmer. Feilen er veldig stor for h>1E-2 og blir større igjen for h<1E-10.

Koden er så si identisk som forrige.

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

h_values=[0.01,0.001,0.0001,0.00001,0.000001,0.0000001,0.00000001,0.000000001,0.0000000001,0.00000000001]

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

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

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



calc_list=[]

for h in h_values:
    calc_list.append(f_prime_numerical(1.5,h))


for i in range(len(calc_list)):
    print("h: ",h_values[i]," Error: ",abs(calc_list[i]-f_prime_analytic(1.5)))

plt.figure(figsize=(8,6))
plt.plot(h_values,abs(calc_list-f_prime_analytic(1.5)),'-o',label='Error')
plt.xscale('log')

plt.xlabel("h ")
plt.ylabel("Absolutt feil")
plt.title("Errorplott")



plt.legend()
plt.grid(True, linestyle="--", linewidth=0.5)
plt.show()
```

3) Samme prosedyre, så si samme kode her også.

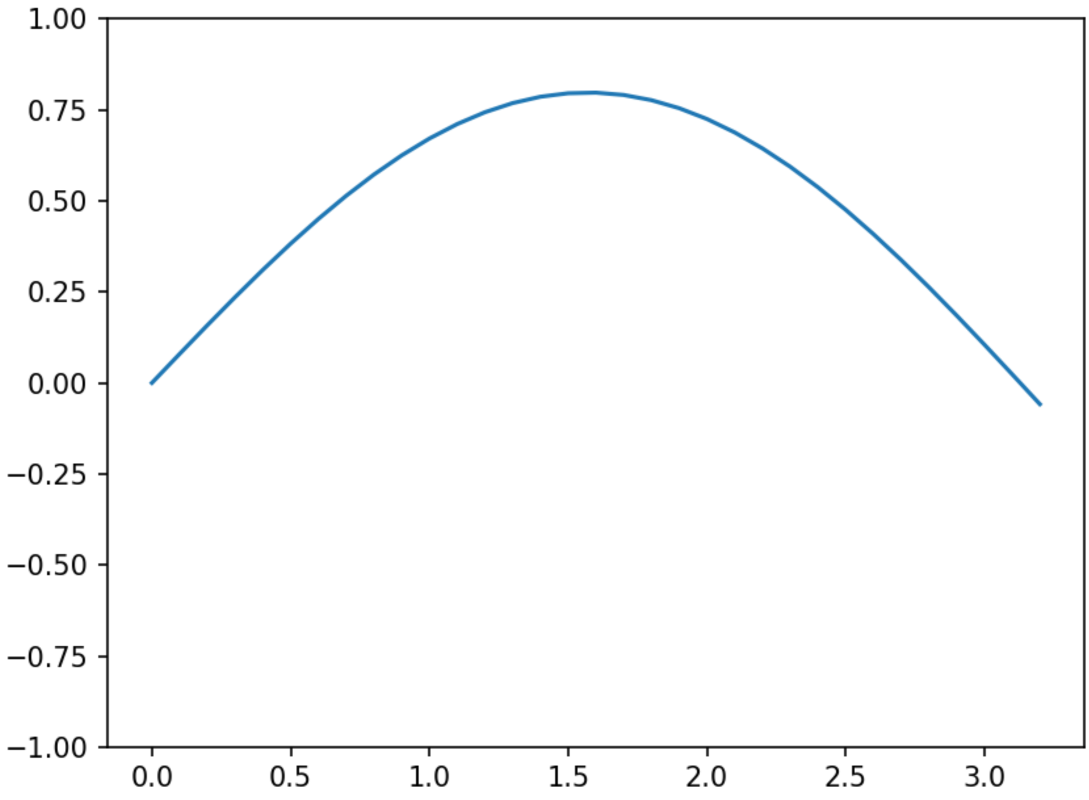


Feilen vokser for ekstremt små h, her begynner feilen å vokse for h<1E-8.

4)

```python
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import matplotlib.animation as animation
4
5  def solve_heat_equation(h, k, T, L):
6      x = np.arange(0, L+h, h)
7      t = np.arange(0, T+k, k)
8
9      u = np.sin(x)
10     U = [u.copy()]
11
12     lambd = k / h**2
13     if lambd > 0.5:
14         print(f"Ustabil for lambda={lambd:.2f}, velg annen h og/eller k")
15         return None
16
17     for _ in t[1:]:
18         u_new = u.copy()
19         for j in range(1, len(x)-1):
20             u_new[j] = u[j] + lambd * (u[j+1] - 2*u[j] + u[j-1])
21         u = u_new
22         U.append(u.copy())
23
24     return x, t, U
25
26 def animate_solution(x, t, U):
27     fig, ax = plt.subplots()
28     line, = ax.plot(x, U[0])
29     ax.set_ylim(-1, 1)
30
31     def update(n):
32         line.set_ydata(U[n])
33         return line,
34
35     ani = animation.FuncAnimation(fig, update, frames=len(t), interval=50)
36     plt.show()
37
38
39 h_values = [1000,0.1, 0.05,0.01]
40 k_values = [0.01,0.01, 0.001,0.01]
41 T, L = 1, np.pi
42
43 for h in h_values:
44     for k in k_values:
45         print(f"Kjører for h={h}, k={k}")
46         result = solve_heat_equation(h, k, T, L)
47         if result:
48             x, t, U = result
49             animate_solution(x, t, U)
50
```

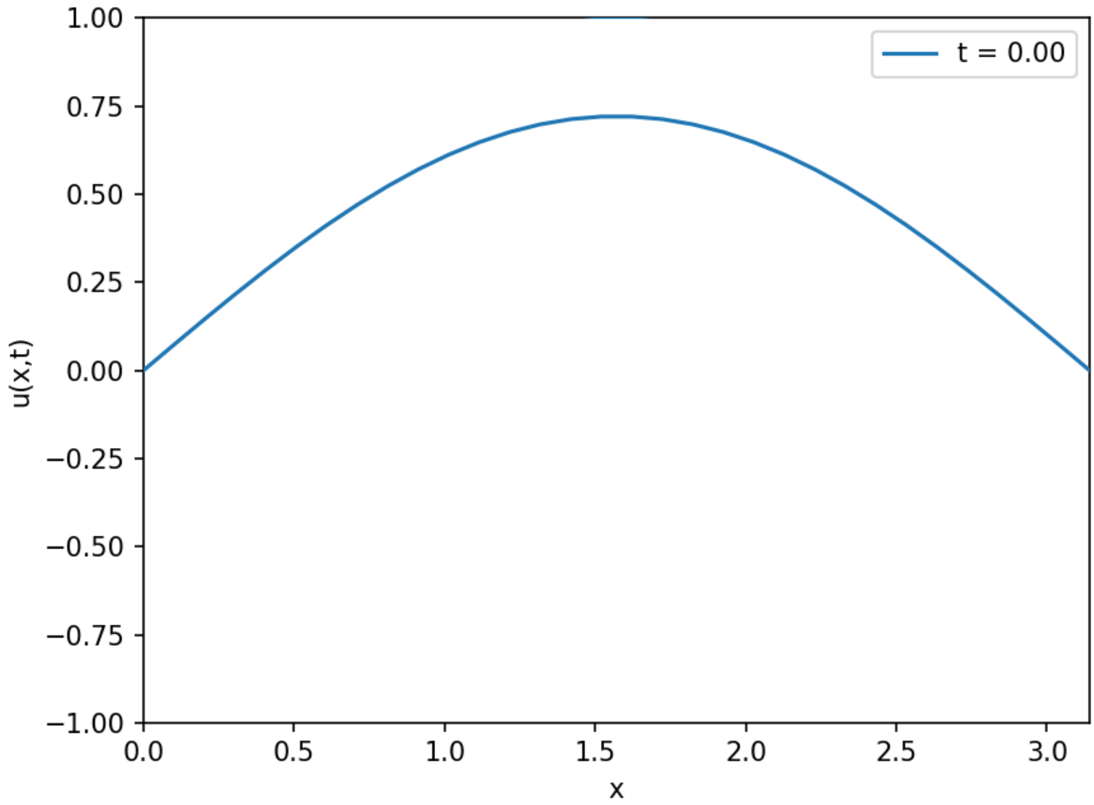
Her har jeg definert variabelen k/h**2. Linje 39 og 40 er to lister for ulike verdier for h og k. Her itererer jeg gjennom begge listene og animerer resultatene.




```
Kjører for h=0.1, k=0.01
Ustabil for lambda=1.00, velg annen h og/eller k
Kjører for h=0.05, k=0.01
Ustabil for lambda=4.00, velg annen h og/eller k
Kjører for h=0.05, k=0.01
Ustabil for lambda=4.00, velg annen h og/eller k
Kjører for h=0.05, k=0.001
Kjører for h=0.05, k=0.01
Ustabil for lambda=4.00, velg annen h og/eller k
Kjører for h=0.01, k=0.01
Ustabil for lambda=100.00, velg annen h og/eller k
Kjører for h=0.01, k=0.01
Ustabil for lambda=100.00, velg annen h og/eller k
Kjører for h=0.01, k=0.001
Ustabil for lambda=10.00, velg annen h og/eller k
Kjører for h=0.01, k=0.01
Ustabil for lambda=100.00, velg annen h og/eller k
```

5)

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

def solve_heat_eq(h, k, L, T):
    Nx = int(L / h)
    Nt = int(T / k)
    x = np.linspace(0, L, Nx+1)
    t = np.linspace(0, T, Nt+1)
    r = k / h**2

    U = np.zeros((Nx+1, Nt+1))
    U[:, 0] = np.sin(x)

    A = np.zeros((Nx-1, Nx-1))
    for i in range(Nx-1):
        A[i, i] = 1 + 2 * r
        if i > 0:
            A[i, i-1] = -r
        if i < Nx-2:
            A[i, i+1] = -r

    for j in range(Nt):
        b = U[1:Nx, j]
        U[1:Nx, j+1] = np.linalg.solve(A, b)

    return x, t, U

def animate(x, t, U):
    fig, ax = plt.subplots()
    line, = ax.plot(x, U[:, 0], label="t = 0")
    ax.set_xlim([x[0], x[-1]])
    ax.set_ylim([-1, 1])
    ax.set_xlabel("x")
    ax.set_ylabel("u(x,t)")
    ax.legend()

    def update(frame):
        line.set_ydata(U[:, frame])
        ax.legend([f"t = {t[frame]:.2f}"])
        return line,

    ani = animation.FuncAnimation(fig, update, frames=len(t), interval=50, blit=True)
    plt.show()

def main():
    h_values=[0.1,0.001,0.0001]
    k_values=[0.01,0.001,0.000001]
    for h in h_values:
        for k in k_values:
            print(f"Kjører for h={h},k={k}")
            L,T=np.pi,1
            x,t,U=solve_heat_eq(h,k,L,T)
            animate(x,t,U)

if __name__ == "__main__":
    main()
```



6)

```python
def solve_heat_eq(h, k, L, T):
    Nx = int(L / h)
    Nt = int(T / k)
    x = np.linspace(0, L, Nx+1)
    t = np.linspace(0, T, Nt+1)
    r = k / h**2
    U = np.zeros((Nx+1, Nt+1))
    U[:, 0] = np.sin(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 * r
        B[i,i]=1-2*r
        if i > 0:
            A[i, i-1] = -r
            B[i,i-1]=r
        if i < Nx-2:
            A[i, i+1] = -r
            B[i,i+1]=r

    for j in range(Nt):
        b = B.dot(U[1:Nx,j])
        U[1:Nx, j+1] = np.linalg.solve(A, b)

    return x, t, U

def animate(x, t, U):
    fig, ax = plt.subplots()
    line, = ax.plot(x, U[:, 0], label="t = 0")
    ax.set_xlim([x[0], x[-1]])
    ax.set_ylim([-1, 1])
    ax.set_xlabel("x")
    ax.set_ylabel("u(x,t)")
    ax.legend()

    def update(frame):
        line.set_ydata(U[:, frame])
        ax.legend([f"t = {t[frame]:.2f}"])
        return line,

    ani = animation.FuncAnimation(fig, update, frames=len(t), interval=50, blit=True)
    plt.show()

def main():
    h_values=[0.1,0.001,0.0001]
    k_values=[0.01,0.001,0.000001]
    for h in h_values:
        for k in k_values:
            print(f"Kjører for h={h},k={k}")
            L,T=np.pi,1
            x,t,U=solve_heat_eq(h,k,L,T)
            animate(x,t,U)

if __name__ == "__main__":
    main()
```

Sammenlikningen forteller at for store h og store k foretrekkes Crank Nickolson og implisitt metode. Dette er på grunn av at λ≤0.5 for stabilitet. Ellers vil feilen bli for stor. Eksplisitt metode er da bedre egnet for enkle beregninger gitt λ≤0.5.