Numerical Derivatives and Heat Equation
TMA4106 Assignment Report

Nicole Maria Powell

April 2025

Introduction

This report presents the solution of selected tasks from the TMA4106 assignment related to numerical differentiation and the numerical solution of the heat equation.

1 Numerical Differentiation

I use Python to compute the derivative of $f(x) = e^x$ at $x = 1.5$.

Python Code Snippet

Below is a snippet of how I implemented this in my notebook:

```
# Explicit method (heat equation)
for j in range(0, nt):
    for i in range(1, nx):
        u_new = alpha * (u[j, i+1] - 2*u[j, i] + u[j, i-1]) + u[j, i]
        u[j+1, i] = u_new
```

```
f = lambda x: np.exp(x)
x0 = 1.5
true_deriv = np.exp(x0)

D1 = (f(x0 + h) - f(x0)) / h
D2 = (f(x0 + h) - f(x0 - h)) / (2 * h)
D4 = (f(x0 - 2*h) - 8*f(x0 - h) + 8*f(x0 + h) - f(x0 + 2*h)) / (12 * h)
```

Results

For $h = 10^{-5}$, I obtained:

- Forward difference: $f'(x) \approx 4.48179$
- Centered difference: $f'(x) \approx 4.48169$
- Richardson 4-point: $f'(x) \approx 4.48169$
- True derivative: $f'(x) = e^{1.5} \approx 4.48169$

I wanted to see how accurate different methods for computing the derivative are of $f(x) = e^x$ at $x = 1.5$ using three methods:

- Forward difference
- Centered difference
- Richardson's 4-point formula

Plot of the absolute error for decreasing values of h.

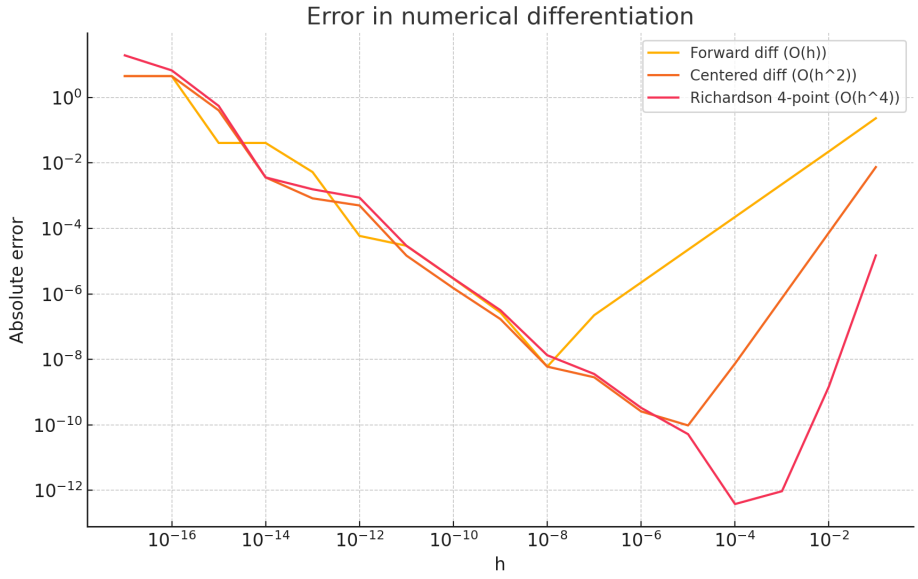
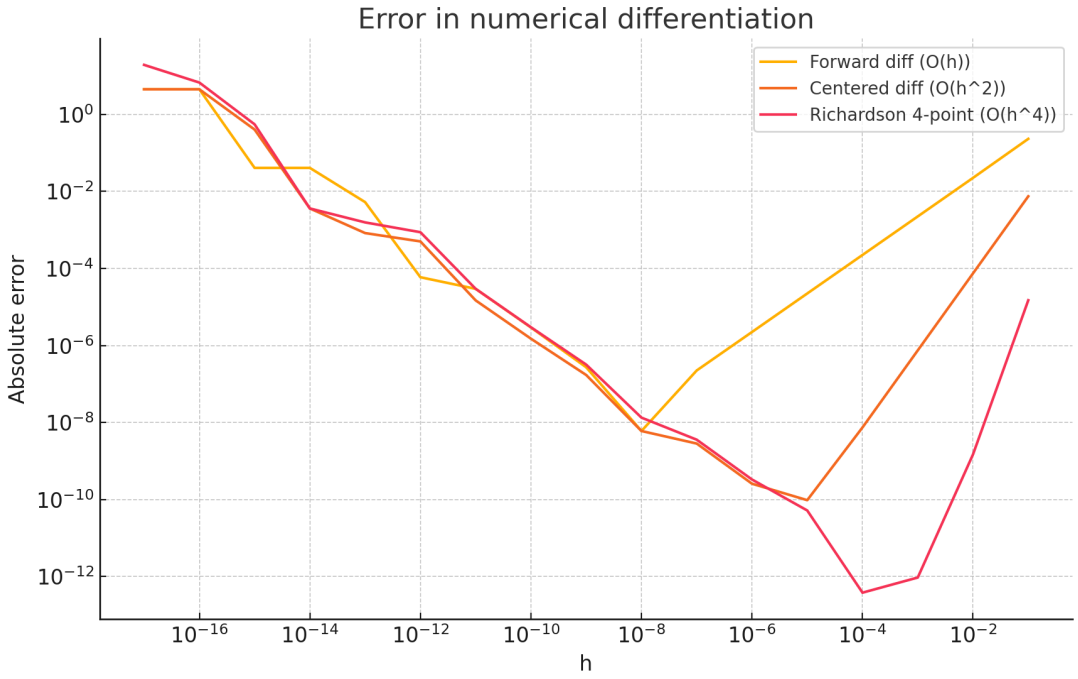


Figure 1: Log-log plot of absolute error vs. step size h for different derivative formulas.



The forward method shows linear error in h, while centered and Richardson's methods show quadratic and quartic behavior respectively.

2 Heat Equation - Explicit Method

Python Code Snippet

Below is a snippet of how I implemented this in my notebook:

```
# Explicit method (heat equation)
for j in range(0, nt):
    for i in range(1, nx):
        u_new = alpha * (u[j, i+1] - 2*u[j, i] + u[j, i-1]) + u[j, i]
        u[j+1, i] = u_new
```

```
u[i, j+1] = alpha * (u[i+1,j] - 2*u[i,j] + u[i-1,j]) + u[i,j]
```

Sample Output

The heat distribution decreases smoothly toward zero at boundaries and stabilizes as time progresses.

To simulate heat distribution, I used different numerical methods to solve the heat equation:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

on the domain $x \in [0, 1]$, $t \in [0, 1]$ with initial condition $u(x, 0) = \sin(\pi x)$ and boundary conditions $u(0, t) = u(1, t) = 0$.

Using the explicit scheme:

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

An animation shows the diffusion of heat over time.

3 Heat Equation - Implicit Method

Python Code Snippet

Below is a snippet of how I implemented this in my notebook:

```
# Explicit method (heat equation)
for j in range(0, nt):
    for i in range(1, nx):
        u_new = alpha * (u[j, i+1] - 2*u[j, i] + u[j, i-1]) + u[j, i]
        u[j+1, i] = u_new
```

```
from scipy.linalg import solve_banded
A_diag = (1 + 2*alpha) * np.ones(nx-1)
A_off = -alpha * np.ones(nx-2)
ab = np.zeros((3, nx-1))
ab[0, 1:] = A_off
ab[1, :] = A_diag
ab[2, :-1] = A_off
u_new = solve_banded((1,1), ab, u_prev)
```

Results

Solution remained stable even for higher time step sizes compared to the explicit method.

I tried out the implicit Euler method, which required solving a linear system at each time step backward Euler scheme:

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

This method is unconditionally stable, allowing larger time steps.

4 Heat Equation - Crank-Nicolson Method

Python Code Snippet

Below is a snippet of how I implemented this in my notebook:

```
# Explicit method (heat equation)
for j in range(0, nt):
    for i in range(1, nx):
        u_new = alpha * (u[j, i+1] - 2*u[j, i] + u[j, i-1]) + u[j, i]
        u[j+1, i] = u_new
```

```
rhs = (alpha/2) * u[j, :-2] + (1 - alpha) * u[j, 1:-1] + (alpha/2) * u[j, 2:]
u_new = solve_banded((1, 1), ab_CN, rhs)
```

Observation

The Crank-Nicolson scheme balances accuracy and stability, making it suitable for most practical scenarios.

This method mixes the explicit and implicit approaches, and gave me good results without instability:

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

This method offers higher accuracy and good stability.

Conclusion

From the experiments, I could clearly see how the error behaves for different derivative formulas. I also got working simulations for heat diffusion with all three methods. Crank-Nicolson seemed to be the most robust. of accuracy of various derivative formulas, and successfully implemented three different numerical schemes to solve the heat equation. The Crank-Nicolson scheme provided a good balance between stability and accuracy.