Final Numerical Analysis

Lena Katharina Siemer
Student ID: 11864551

October 2023

1 Problem 1

Consider the partial differential (diffusion) equation

$$\delta_t u = D\delta_x^2 u, \tag{1}$$

for $x \in [0,1]$, and $t > 0$. The initial conditions are given by

$$u(x,0) = \begin{cases} x & if \quad 0 \leq x \leq 1/2, \\ 1-x & if \quad 1/2 < x \leq 1. \end{cases} \tag{2}$$

The boundary conditions are

$$u(0,t) = u(1,t) = 0. \tag{3}$$

Solve the problem (1)-(3) with $D = 0.5$ numerically using the explicit difference scheme

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = D\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2} \tag{4}$$

Take $\Delta x = 0.1$ and compute the numerical solutions for three values of Δt : $\Delta t = \frac{1}{50}, \frac{1}{100}, \frac{1}{200}$. Note that the use of the finite difference scheme for discretizing partial derivatives reduces the problem to vector-matrix multiplication. Here the unknown vector $U^n = (u_1^n, u_2^n, ..., u_N^n)^T$ refers to the sample of the grid values of the solution function $u(x,t)$ evaluated at $t = t_n$. Equivalently, $u_j^n = u(x_j, t_n)$. The purpose of Eq. (4) is to update U^n to U^{n+1} (marching forward in time). If more background is needed, please consult our textbook, Section 9.1

1. Check numerically the order of accuracy in time by plotting the error (the difference between the exact and numerical solutions) for the above values of Δt.

2. Plot the exact and numerical solutions for the above values of Δt. (You should have one plot for each Δt showing the exact and numerical solutions on the same set of axis).

3. Comment on the results of the computations.

The exact solution of (1)-(3) can be obtained by separation of variables. The description of the method is available in many introductory PDE textbooks. For example, you can consult "Advanced Engineering Mathematics" by Erwin Kreyszig, Ch. 11. The exact solution $v(x,t)$ has the series representation

$$v(x,t) = \sum_{k=1}^{\infty} \frac{4}{(k\pi)^2} \sin(\frac{k\pi}{2}) \sin(k\pi x) e^{-D(k\pi)^2 t}. \tag{5}$$

For plotting, use the truncated series with 14 terms.

Solution:

We first want to check if the given exact solution is actually a solution of the given problem.

$$\delta_t u = -\sum_{k=1}^{\infty} 4D \sin(\frac{k\pi}{2}) \sin(k\pi x) e^{-D(k\pi)^2 t}$$

$$\delta_x u = \sum_{k=1}^{\infty} \frac{4}{k\pi} \sin(\frac{k\pi}{2}) \cos(k\pi x) e^{-D(k\pi)^2 t}.$$

$$\delta_x^2 = -\sum_{k=1}^{\infty} 4 \sin(\frac{k\pi}{2}) \sin(k\pi x) e^{-D(k\pi)^2 t}.$$

We can see that $\delta_t u = D\delta_x^2 u$.

If we take

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = D \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}$$

we get

$$u_j^{n+1} = D \frac{\Delta t}{(\Delta x)^2} (u_{j+1}^n - 2u_j^n + u_{j-1}^n) + u_j^n$$

We start with defining the necessary functions to compute the numerical and exact solutions.

```python
import numpy as np
import math
from matplotlib import pyplot as plt
def u_initial(x, t):
    if t == 0 and 0 <= x <= 0.5:
        return x
    elif t == 0 and 0.5 < x <= 1:
        return 1 - x
    elif x == 0 or x == 1:
        return 0
def explicit_difference_scheme(x, t, delta_x, delta_t, D):
    # Initialization using the boudary and initial conditions
    U = np.zeros((len(t), len(x)))
    for k in range(len(x)):
        U[0, k] = u_initial(x[k], 0)
    # Calculate other values of the matrix
    S = D * (delta_t) / ((delta_x) ** 2)
    for i in range(1, len(t)):
        for j in range(1, len(x) - 1):
        #calculate the new value using the formular from above
```

```
            U[i, j] = S * (U[i - 1, j + 1] - 2 * U[i - 1, j] + U[i - 1, j - 1]) + U[i - 1, j]
    return U
def exact_solution(x, t, D):
    #create the matrix
    U = np.zeros((len(t), len(x)))
    #go through all entries of the matrix to compute the value
    for i in range(len(t)):
        for j in range(len(x)):
        #compute the new value
            s = 0
            for k in range(1, 14):
                a= math.sin(k*math.pi/2)*math.sin(k*math.pi*x[j])*math.exp(-D*(k * math.pi)**2*t[i])
                s += (4/(k*math.pi)**2)*a
            U[i, j] = s
    return U

#we calculate our solutions
D = 0.5
delta_x = 0.1
x=[]
i=0
while i<1:
    x.append(i)
    i=i+delta_x
```

1. First we want to look at the errors.

```
    #plot the errors
delta_t = 1 / 50
t = np.arange(0, 4*delta_t, delta_t)
#calculate the numerical solution
solution = explicit_difference_scheme(x, t, delta_x, delta_t, D)
#calculate the exact solution
exact = exact_solution(x,t,D)

plt.figure(figsize=(8, 5))
for target_time in t:
    #plot the numerical solution
    t_index = np.abs(t - target_time).argmin()
    plt.plot(x, np.abs(solution[t_index, :]-exact[t_index, :]), label=f't = {target_time}')

plt.xlabel('X')
plt.ylabel('Error')
plt.title('Error for Delta_t=1/50')
plt.legend()
plt.grid(True)
plt.show()
```

```python


delta_t = 1 / 100
t = np.arange(0, 4*delta_t, delta_t)
#calculate the numerical solution
solution = explicit_difference_scheme(x, t, delta_x, delta_t, D)
#calculate the exact solution
exact = exact_solution(x,t,D)

plt.figure(figsize=(8, 5))
for target_time in t:
    #plot the numerical solution
    t_index = np.abs(t - target_time).argmin()
    plt.plot(x, np.abs(solution[t_index, :]-exact[t_index, :]), label=f't = {target_time}')

plt.xlabel('X')
plt.ylabel('Error')
plt.title('Error for Delta_t=1/100')
plt.legend()
plt.grid(True)
plt.show()


delta_t = 1 / 200
t = np.arange(0, 4*delta_t, delta_t)
#calculate the numerical solution
solution = explicit_difference_scheme(x, t, delta_x, delta_t, D)
#calculate the exact solution
exact = exact_solution(x,t,D)

plt.figure(figsize=(8, 5))
for target_time in t:
    #plot the numerical solution
    t_index = np.abs(t - target_time).argmin()
    plt.plot(x, np.abs(solution[t_index, :]-exact[t_index, :]), label=f't = {target_time}')

plt.xlabel('X')
plt.ylabel('Error')
plt.title('Error for Delta_t=1/200')
plt.legend()
plt.grid(True)
plt.show()
```

For the different Δt values, we get

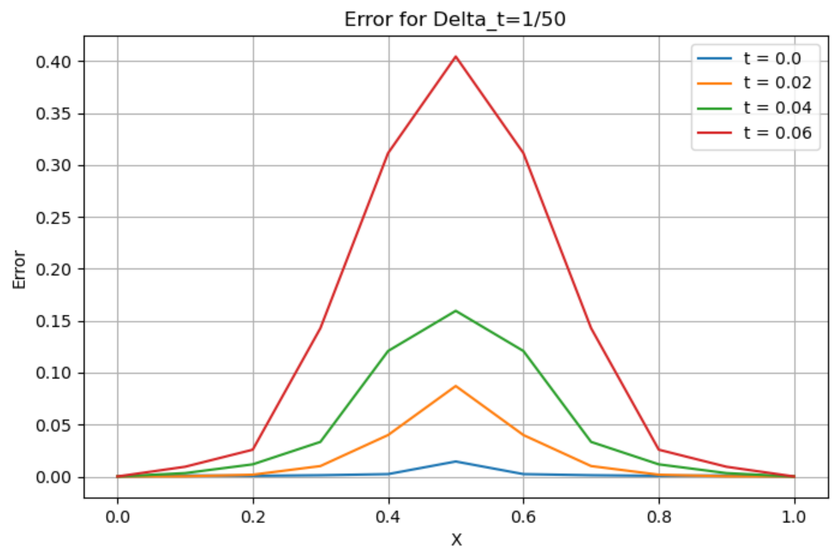

Figure 1: Errors for $\Delta t = \frac{1}{50}$

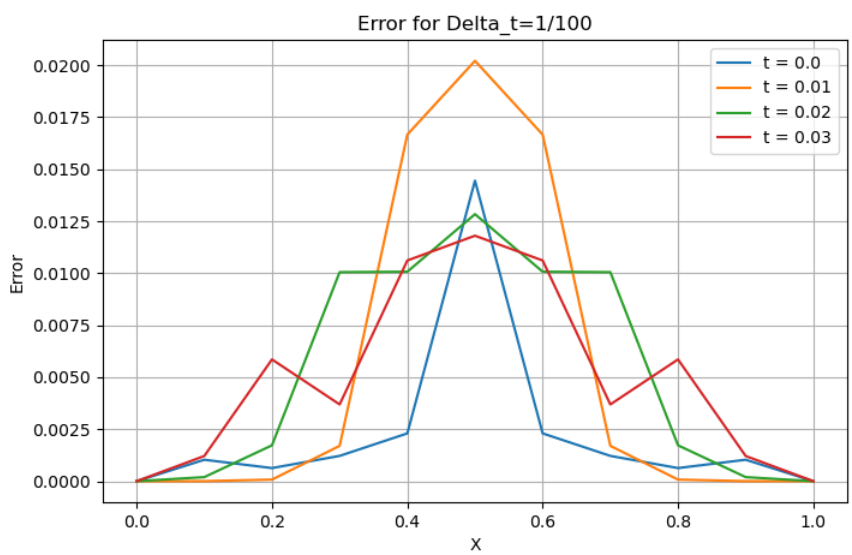

Figure 2: Errors for $\Delta t = \frac{1}{100}$

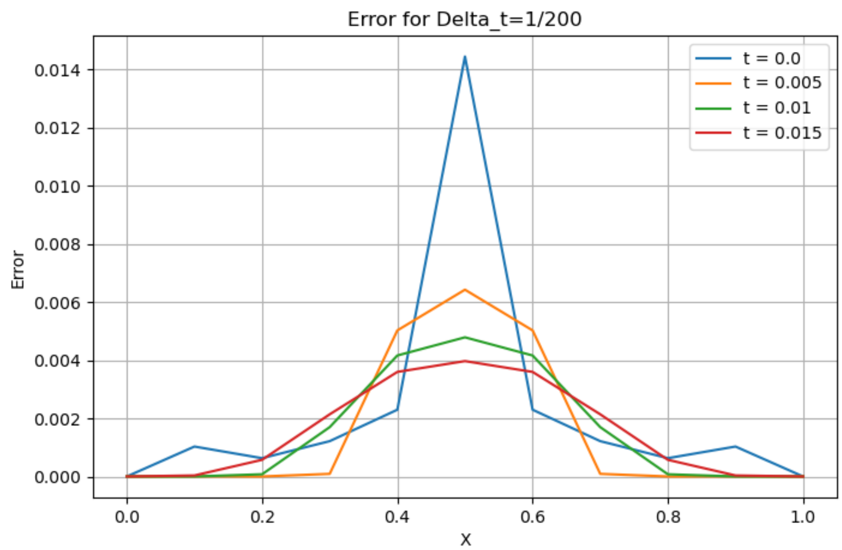

Figure 3: Errors for $\Delta t = \frac{1}{200}$

2. Next we want to take a look at the numerical and exact solutions separately.

```python
#we calculate our solutions
D = 0.5
delta_x = 0.1
x = np.arange(0, 1, delta_x)


#for delta_t=1/50
delta_t = 1 / 50
t = np.arange(0, 4*delta_t, delta_t)
#calculate the numerical solution
solution = explicit_difference_scheme(x, t, delta_x, delta_t, D)
#calculate the exact solution
exact = exact_solution(x,t,D)

plt.figure(figsize=(8, 5))
for target_time in t:
    #plot the numerical solution
    t_index = np.abs(t - target_time).argmin()
    plt.plot(x, solution[t_index, :], label=f'NS: t = {target_time}')
    plt.plot(x, exact[t_index, :], label=f'ES: t = {target_time}')

```

```python
plt.xlabel('X')
plt.ylabel('U')
plt.title('Solution for Delta_t=1/50')
plt.legend()
plt.grid(True)
plt.show()

################################
#for delta_t=1/100
delta_t = 1 / 100
t = np.arange(0, 4*delta_t, delta_t)
#calculate the numerical solution
solution = explicit_difference_scheme(x, t, delta_x, delta_t, D)
#calculate the exact solution
exact = exact_solution(x,t,D)

plt.figure(figsize=(8, 5))
for target_time in t:
    #plot the numerical solution
    t_index = np.abs(t - target_time).argmin()
    plt.plot(x, solution[t_index, :], label=f'NS: t = {target_time}')
    plt.plot(x, exact[t_index, :], label=f'ES: t = {target_time}')


plt.xlabel('X')
plt.ylabel('U')
plt.title('Solution for Delta_t=1/100')
plt.legend()
plt.grid(True)
plt.show()

###################################3
#for delta_t=1/200
delta_t = 1 / 200
t = np.arange(0, 4*delta_t, delta_t)
#calculate the numerical solution
solution = explicit_difference_scheme(x, t, delta_x, delta_t, D)
#calculate the exact solution
exact = exact_solution(x,t,D)

plt.figure(figsize=(8, 5))
for target_time in t:
    #plot the numerical solution
    t_index = np.abs(t - target_time).argmin()
    plt.plot(x, solution[t_index, :], label=f'NS: t = {target_time}')
    plt.plot(x, exact[t_index, :], label=f'ES: t = {target_time}')


```

```python
plt.xlabel('X')
plt.ylabel('U')
plt.title('Solution for Delta_t=1/200')
plt.legend()
plt.grid(True)
plt.show()
```

We denote the numerical Solution with "NS" and the exact Solution with "ES". For the different Δt values, we get

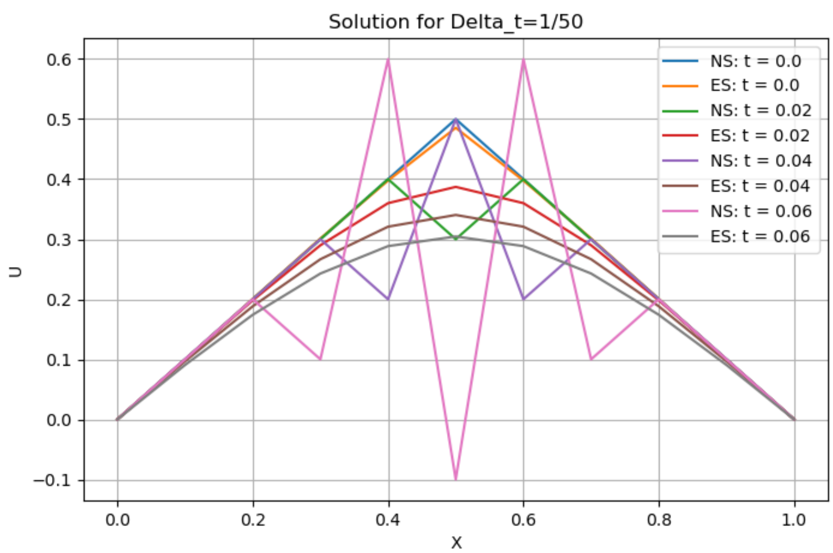

Figure 4: Numerical and exact Solution for $\Delta t = \frac{1}{50}$

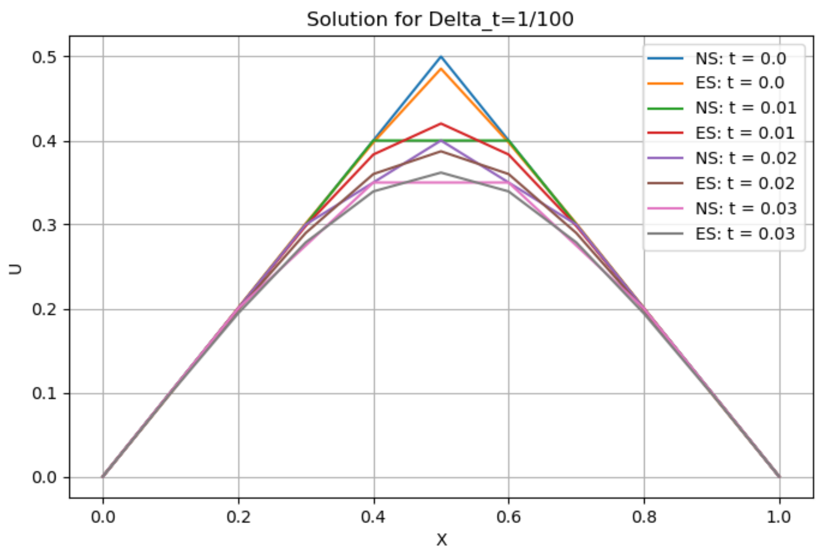

Figure 5: Numerical and exact Solution for $\Delta t = \frac{1}{100}$

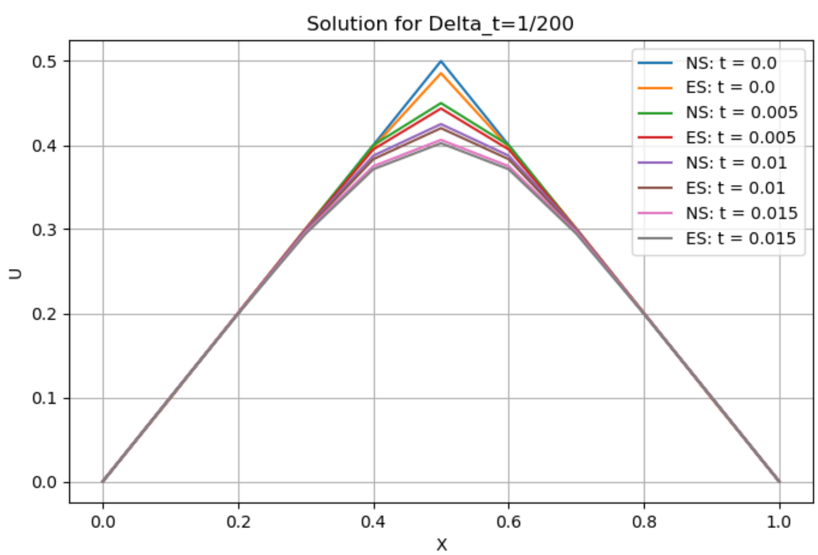

Figure 6: Numerical and exact Solution for $\Delta t = \frac{1}{200}$

3. Observations:
   Lets first take a look at the numerical and exact solutions. We can see that for smaller t values the numerical solution is relatively closer to the exact solution (Figure 4). If we take $t = 0.06$ we can already see great fluctuations. These fluctuations get smaller if we decrease Δt. For $\Delta t = \frac{1}{200}$ the numerical solution is already very close to the exact one (Figure 6).
   If we take a look at part 1), we can see that our observation is correct. The errors for $\Delta t = \frac{1}{50}$ are relatively big (Figure 1) and they decrease if we decrease Δt (Figure 2,3).
   Notice that all graphs are symmetric with maximum at 0.5. This might get explained by the fact that the initial condition is symmetric at $x = 0.5$. But this is just a speculation at this point.

   Stability Analysis:
   I will follow the principle of page 619 from the book "Numerical Analysis" from David Kindcaid and Ward Cheney.
   Recall that $u_j^{n+1} = D\frac{\Delta t}{(\Delta x)^2}(u_{j+1}^n - 2u_j^n + u_{j-1}^n) + u_j^n$.
   We define $s = D\frac{\Delta t}{(\Delta x)^2}$ and

$$V_j = \begin{pmatrix} v_{1,j} \\ ... \\ v_{n,j} \end{pmatrix}.$$

   as the vector of values at time $t = jk$. We get $V_{j+1} = AV_j$ where

$$A = \begin{bmatrix} 1-2s & s & 0 & 0 & . & . & . & . \\ s & 1-2s & s & 0 & . & . & . & . \\ 0 & s & 1-2s & s & . & . & . & . \\ . & . & . & . & . & 1-2s & s & 0 \\ . & . & . & . & . & s & 1-2s & s \\ . & . & . & . & . & 0 & s & 1-2s \end{bmatrix}.$$

   Since $u(0,t) = u(1,t) = 0$ we know that $v_{0,j} = v_{n+1,j} = 0$
   We can write V_j as $V_j = AV_{j-1} = A^2V_{j-2} = ... = A^jV_0$.

   Theorem 5 on page 215 states that

   (a) $lim_{j \to 0} A^jV = 0$ for all vectors V
   (b) $\rho(A) < 1$

   are equivalent, where $\rho(A) := \max(|\lambda_1|, ..., |\lambda_n|)$ for eigenvalues λ_i for $1 \le i \le n$. Therefore we have to first calculate the Eigenvalues.

We can write $A = I_n - sB$ with

$$B = \begin{bmatrix} -2 & 1 & 0 & 0 & . & . & . & . \\ 1 & -2 & 1 & 0 & . & . & . & . \\ 0 & 1 & -2 & 1 & . & . & . & . \\ . & . & . & . & . & -2 & 1 & 0 \\ . & . & . & . & . & 1 & -2 & 1 \\ . & . & . & . & . & 0 & 1 & -2 \end{bmatrix}.$$

Let μ_i be an Eigenvalue of B, then $\lambda_i = 1 - s\mu_i$ is an Eigenvalue of A. (This can be proven with linear algebra, which is of no importance at this point.) From Lemma 1 page 621 we know that $\mu_i = 2(1 - \cos(\frac{i\pi}{n+1}))$. Therefore $\lambda_i = 1 - 2s(1 - \cos(\frac{i\pi}{n+1}))$.

Now we can check if the maximum of all $|\lambda_i| < 1$. So we get $-1 < \lambda_i < 1 \Rightarrow 0 < s < \frac{1}{1-\cos(\frac{i\pi}{n+1})}$. So, we need to find the smallest possible value of $\frac{1}{1=\cos(\frac{i\pi}{n+1})}$ which is when $\cos(\frac{i\pi}{n+1}) = -1$. Therefore $s < \frac{1}{1+1} = \frac{1}{2}$ Since we defined $s = D\frac{\Delta t}{(\Delta x)^2}$ in the beginning, we can now say that $2D\Delta t < \Delta x$ to guarantee stability.

2 Problem 2

Solve Computer problem 1 in Section 8.3. Write you own program from scratch. Then compare with the exact analytic solution that can be obtained by the method of integrating factors studied in Math 315. The exact solution is

$$x(t) = 12\frac{e^t}{(e^t+1)^2}.$$

Computer problem 1 in Section 8.3.
Write a computer program to solve an initial-value problem $x' = f(t, x)$ with $x(t_0) = x_0$ on an interval $t_0 \le t \le t_m$ or $t_m \le t \le t_0$. Use the fourth-order Runge-Kutta method. Test it on this example:

$$\begin{cases} (e^t+1)x' + xe^t - x = 0 \\ x(0) = 3 \end{cases} \tag{6}$$

Determine the analytic solution and compare it to the computed solution on the interval $-2 \le t \le 0$. Use h=-0.01.

Solution: We first should check if the given exact solution is a solution.

$$x'(t) = 12\frac{e^t(e^t+1)^2 - 2e^{2t}(e^t+1)}{(e^t+1)^4} = 12e^t\frac{1-e^t}{(e^t+1)^3}$$

So we have

$$\begin{aligned} &(e^t+1)12e^t\frac{1-e^t}{(e^t+1)^3} + 12\frac{e^t}{(e^t+1)^2}e^t - 12\frac{e^t}{(e^t+1)^2} \\ &= 12e^t\left(\frac{1}{(e^t+1)^2} - \frac{e^t}{(e^t+1)^2} + \frac{e^t}{(e^t+1)^2} - \frac{1}{(e^t+1)^2}\right) \\ &= 0 \end{aligned}$$

and obviously $x(0) = 3$

The fourth-order Runge-Kutta method:

$$x(t+1) = x(t) + \frac{1}{6}(F_1 + 2F_2 + 2F_3 + F_4)$$

where

$$\begin{cases} F_1 = hf(t,x) \\ F_2 = hf(t+0.5h, x+0.5F_1) \\ F_3 = hf(t+0.5h, x+0.5F_2) \\ F_4 = hf(t+h, x+F_3) \end{cases} \tag{7}$$

We have $(e^t+1)x' + xe^t - x = 0$, so we get

$$x' = x\frac{1-e^t}{e^t+1}$$

Now we can implement all necessary functions.

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


def f(x,t):
    return x *(1 - math.exp(t))/(math.exp(t) + 1)
def u(t):
    return 12* math.e**t/(math.e**t + 1)**2

def runge_kutta(M,t,x,h):
    ret=[]
    e=abs(u(t )-x)
    ret.append([0,t,x,e])
    for k in range (1,M):
        F1 = h* f(x,t )
        F2 = h* f(x+F1/2,t+h/2 )
        F3 = h* f(x+F2/2,t+h/2 )
        F4 = h* f(x+F3,t+h )
        x= x+ (F1+2*F2+2*F3+F4)/6
        t=t+h
        e=abs(u(t )-x)
        ret.append([k,t,x,e])

    return ret


#exactness
M = 200
h = -0.01
#initial value
t = 0
x = 3

#calculate the numerical solution
solution = runge_kutta(M,t,x,h )

t_values = [i[1] for i in solution]
x_values = [i[2] for i in solution]
error_values = [i[3] for i in solution]

#calculate the exact solution
exact_values = [u(i) for i in t_values]

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(t_values, x_values, label='Numerical Solution')
```

```
plt.plot(t_values, exact_values, label='Exact Solution')
plt.xlabel('t')
plt.ylabel('x')
plt.title('Runge-Kutta Method for Differential Equation')
plt.legend()
plt.grid(True)
plt.show()
```

Then we get the following output (Figure 7).

To get a more accurate result to how close the numerical and exact solution

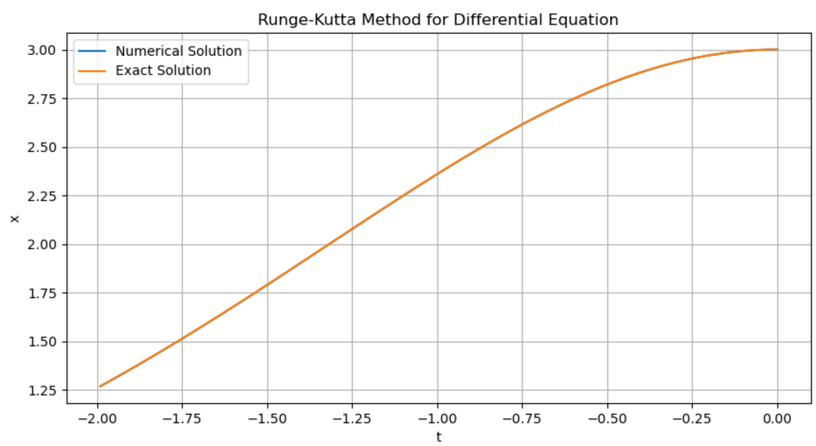

Figure 7: Numerical and exact Solution for the Runge Kutta method

actually are, we want to look at the errors.

```
#Error
plt.figure(figsize=(10, 6))
plt.plot(t_values, error_values, label='Error')
plt.xlabel('t')
plt.ylabel('x')
plt.title('Errors of the Runge-Kutta Method')
plt.legend()
plt.grid(True)
plt.show()
```

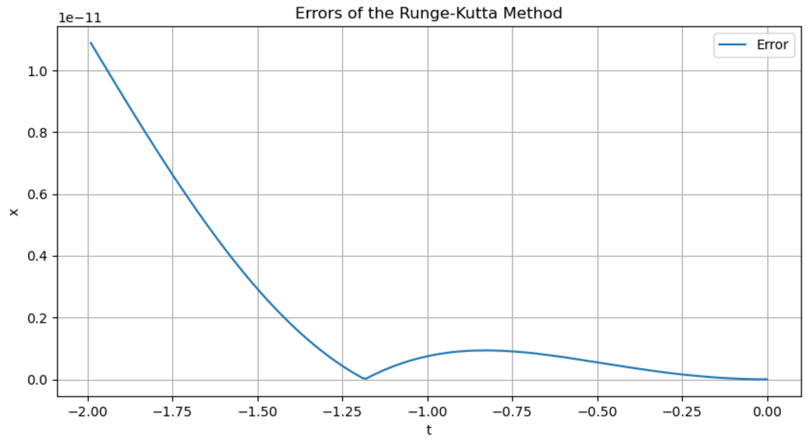

Figure 8: Error for the Runge Kutta method

Conclusion:
In Figure 7 we can assume that the numerical and exact solution are almost identical. Looking at the Error values (Figure 8) we can see that the maximum error is about $8.7 * 1e - 8$. So the Runge Kutta method computed the values up to 7 digits correctly on the given Interval.
The error also decreases as we get closer to 0.

Analysis:
We want to take a closer look at the local and global truncation error. The Runge-Kutta method has a local truncation error of $\mathcal{O}(h^5)$ and a global truncation error of $\mathcal{O}(h^4)$ (see page 543). Therefore we know that if we reduce the step size, we get more accurate results.

3 Problem 3

Solve Computer problem 3 in Section 8.4.
Compute the solution of

$$\begin{cases} y' = -2xy^2 \\ y(0) = 1 \end{cases} \tag{8}$$

at $x = 1.0$ using $h = 0.25$ and the fourth-order Adams-Bashforth-Moulton method (Problem 8.4.4-5 p.555) together with the fourth-order Runge-Kutta method. Give the computed solution to five significant digits at $0.25, 0.5, 0.75$ and 1.0. Compare your results to the exact solution $y = \frac{1}{1+x^2}$

Solution: First we need to check if the given exact solution is in fact a solution to the given differential equation.

$$y' = -\frac{2x}{(1+x^2)^2}$$

Therefore

$$-2xy^2 = -2x\left(\frac{1}{1+x^2}\right)^2 = -\frac{2x}{(1+x^2)^2} = y'$$

The boundary condition is obviously correct as well.

The fourth-Order Adams-Bashforth formula:

$$x_{n+1} = x_n + \frac{h}{24}[55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}]$$

where $f_n = f(x_n, t_n)$.
The fourth-Order Adams-Moulton formula:

$$x_{n+1} = x_n + \frac{h}{24}[9f_{n+1} + 19f_n - 5f_{n-1} + f_{n-2}]$$

where $f_n = f(x_n, t_n)$ and $f_{n+1} = f(x^*_{n+1}, t_{n+1})$. x^*_{n+1} denotes the predicted value that can be obtained from the fourth order Adams-Bashforth formula.

The fourth-order Runge-Kutta method:

$$x(t+1) = x(t) + \frac{1}{6}(F_1 + 2F_2 + 2F_3 + F_4)$$

where

$$\begin{cases} F_1 = hf(t, x) \\ F_2 = hf(t + 0.5h, x + 0.5F_1) \\ F_3 = hf(t + 0.5h, x + 0.5F_2) \\ F_4 = hf(t + h, x + F_3) \end{cases} \tag{9}$$

First we need to implement the Adams-Bashforth formula. We don't need to implement the entire method. We only need the formular to compute the x values in the Adams-Moulton method.

```
import numpy as np
import math
import matplotlib.pyplot as plt
from tabulate import tabulate



def f(x,t):
    return -2*t*x**2
def u(t):
    return 1/(1+t**2)

h = 0.25
t = 0
x=1
e = 5
steps=5
#adam bashfort (fourth order)

print('Adams-Bashforth fourth order formular')

def adams_bashforth_one_step(T, X, h, step):
    if step <=3:
        #we use the runge kutta method for the first steps
        F1 = h * f(X[step - 1],T[step - 1])
        F2 = h * f(X[step - 1] + F1/2, T[step - 1] + h/2)
        F3 = h * f(X[step - 1] + F2/2, T[step - 1] + h/2)
        F4 = h * f(X[step - 1] + F3, T[step - 1] + h)
        return X[step - 1] + (F1 + 2*F2 + 2*F3 + F4) / 6

    else:
        return X[step - 1] + h * (55 * f(X[step - 1],T[step - 1]) - 59 * f(X[step - 2],T[step - 2]) + 37 * f(X[st


```

Now we can implement the Adams-Moulton formula

```
#adam moulton (fourth order)
print('Adams-Moulton fourth order method')


def adams_moulton(t, x, h, steps):
    T = [t]
    X = [x]
    #compute the values we need from the adams-bashforth formular
    x_values = [adams_bashforth_one_step(T,X,h,1)]
```

```

    for i in range (1,min(3, steps)+1):
        T.append( T[i - 1] + h)
        b=adams_bashforth_one_step(T,X,h,i)
        X.append( b)
        x_values.append(b)

    for i in range(4, steps):
        T.append( T[i - 1] + h)
        #here we need to use the adams bashforth formular
        x_values.append(adams_bashforth_one_step(T,X,h,i))
        X.append( X[i - 1] + h / 24 * (9 * f(x_values[i],T[i]) + 19 * f(X[i - 1], T[i - 1]) - 5 * f(X[i - 2],T[i

    return T, X

# Calculate the numerical solution
solution = adams_moulton(t,x,h,steps)

t_values = solution[0]
x_values = [round(i,e) for i in solution[1]]

# Calculate the exact solution
exact_values = [round(u(i),e) for i in t_values]

#we create a table with the values
data = [[t, x, exact_values] for t, x, exact_values in zip(t_values, x_values, exact_values)]
headers = ["t", "x","exact"]
table = tabulate(data, headers, tablefmt="pretty")
print(table)
```

Lastly we can use the code from Problem 2 to compute the values using the Runge-Kutta method.

```
print('Runge Kutta method')

def runge_kutta(M,t,x,h):
    ret=[]
    e=abs(u(t )-x)
    ret.append([0,t,x,e])
    for k in range (1,M):
        F1 = h* f(x,t )
        F2 = h* f(x+F1/2,t+h/2 )
        F3 = h* f(x+F2/2,t+h/2 )
        F4 = h* f(x+F3,t+h )
        x= x+ (F1+2*F2+2*F3+F4)/6
        t=t+h
        e=abs(u(t )-x)
```

```
        ret.append([k,t,x,e])

    return ret


# Calculate the numerical solution
solution = runge_kutta(steps,t,x,h )

t_values = [i[1] for i in solution]
x_values = [round(i[2],5) for i in solution]

# Calculate the exact solution
exact_values = [round(u(i),e) for i in t_values]

#we create a table with the values
data = [[t, x, exact_values] for t, x, exact_values in zip(t_values, x_values, exact_values)]
headers = ["t", "x","exact"]
table = tabulate(data, headers, tablefmt="pretty")
print(table)
```

With the Adams-Bashforth-Moulton method and the Runge-Kutta method we compute the following values:

Adams-Moulton fourth order formular

t	x	exact
0	1	1.0
0.25	0.94115	0.94118
0.5	0.79995	0.8
0.75	0.63997	0.64
1.0	0.49822	0.5

Runge Kutta method

t	x	exact
0	1	1.0
0.25	0.94115	0.94118
0.5	0.79995	0.8
0.75	0.63997	0.64
1.0	0.50001	0.5

Figure 9: Solutions values for both methods

We want to compare the methods. So let's take a look at the error values.

```
#Errors
error_am=[]
error_rk=[]
for i in range(0, len(exact_values)):
    error_am.append(round(abs(x_values_am[i]-exact_values[i]),5))
    error_rk.append(round(abs(x_values_rk[i]-exact_values[i]),5))

print(error_ab)
# Plotting
plt.figure(figsize=(10, 6))
plt.plot(t_values_am, error_am, label='Error Adams-Moulton method')
plt.plot(t_values_rk, error_rk, label='Error Runge Kutta method',linestyle='--')
plt.xlabel('t')
plt.ylabel('x')
plt.title('Runge-Kutta Method for Differential Equation')
```

```
plt.legend()
plt.grid(True)
plt.show()
```

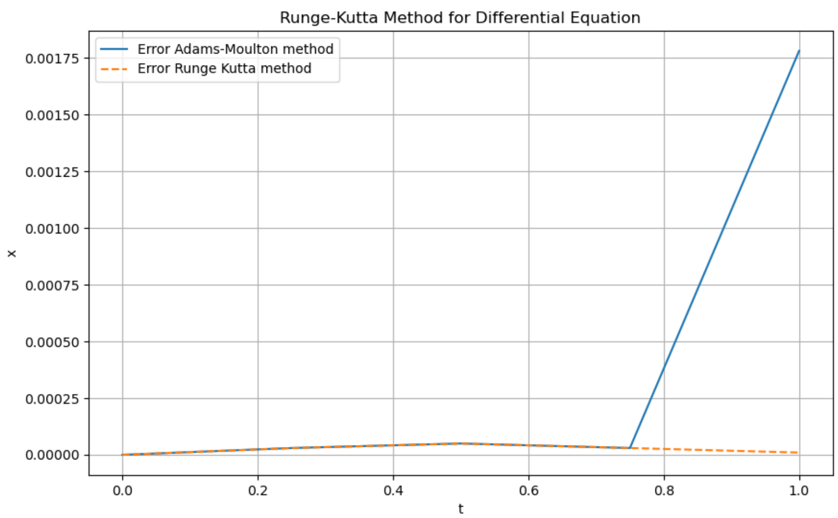

Figure 10: Error values for both methods

We can see that for the first three values, both methods have the same error. That can be explained by how we computed the Adams-Bashforth-Moulton solution. If we want to use this formula, we first need to compute the initial values $X[i-1]$, $X[i-2]$, $X[i-3]$ and $X[i-4]$. We computed them using the Runge Kutta method, so we had to expect, the values to be the same.
After computing the initial values, we can use the formula given by Adams-Moulton and compute the next value. Now we can expect to get different values. That is displayed in the errors. Fot $t = 1$ we get an error of about $e = 0.00178$ for the Adams-Bashforth-Moulton method and an error of about $e = 0.00001$ for the Runge-Kutta method.
So in this case we get a more accurate solution using the Runge Kutta method.