Finite Difference Methods for Partial Differential Equations

1 problem 1 & 2 (considering $\Delta x = h$)

Rate of convergence for forward Euler and Crank-Nicolson numerical scheme:

1.1 Forward Central difference method

This method is second order in space and first order in time. since, Δt is related to Δx by the value of α. So, for the truncation error we have:

$$\tau(x,t)_j^{n+1} \leq C_1 \Delta t + C_2 \Delta x^2 \tag{1}$$

$$\Delta t = \frac{\alpha \Delta x^2}{\sigma} \tag{2}$$

σ is equal to one here, we have:

$$\tau(x,t)_j^{n+1} \leq C_1 \alpha \Delta x^2 + C_2 \Delta x^2 \tag{3}$$

$$\tau(x,t)_j^{n+1} = O(\Delta x^2) \tag{4}$$

So, we expect to see the rate of convergence be equal to 2 for the forward Euler method.

We implement a code for different domain discretization. we define a value M as the number of parts, we want to divide the domain. We will put it equal to 128, 256, 512 for different values of α.

Since we don't know the exact solution to see what the value for local truncation error looks like, we use a numerical trick as bellow. Let's consider W_h a numerical approximation of W with time step h, which converges with a rate of β we have:

$$q = \frac{w_h - w_{\frac{h}{2}}}{w_{\frac{h}{2}} - w_{\frac{h}{4}}} = 2^\beta \tag{5}$$

1.1.1 $alpha = 1/2$

When α is equal to $\frac{1}{2}$, we have the values bellow as the parameters of the method. This is the stability condition for this method. So, we have to choose the values of Δt and Δx in a way that α stays in the stability zone. We expect β be equals around 2 by the algorithm we wrote down for the rate of convergence by `Python`. We consider a fixed-pint in $x-t$ plane, to see how the approximation for temperature function changes as the time and space steps get smaller and smaller. I considered this point in the middle of the rode and at $t = 0.2$ seconds.

M	Δx	Δt	α
128	$\frac{pi}{128}$	$\frac{pi}{128}$	$\frac{1}{2}$
256	$\frac{pi}{256}$	$\frac{pi}{256}$	$\frac{1}{2}$
512	$\frac{pi}{512}$	$\frac{pi}{512}$	$\frac{1}{2}$

Table 1: Values for the parameters in forward Central Difference method, $\alpha = \frac{1}{2}$

1.1.2 $alpha = 1/6$

The local truncation error for this method is exactly equals to equation (6).

$$\tau(x,t) = \frac{1}{2}(\Delta t - \frac{1}{6}\Delta x^2)u_{xxxx} + O(\Delta t^2 + \Delta x^4) \tag{6}$$

If $\sigma = 1$, then by considering $\alpha = \frac{1}{6}$, for the local truncation error, we have:

$$\tau(x,t) = O(\Delta t^2 + \Delta x^4) \tag{7}$$

In this case, Forward Euler method is of order 2 in time and 4 in space.

M	Δx	Δt	α
128	$\frac{pi}{128}$	$\frac{pi}{128}$	$\frac{1}{6}$
256	$\frac{pi}{256}$	$\frac{pi}{256}$	$\frac{1}{6}$
512	$\frac{pi}{512}$	$\frac{pi}{512}$	$\frac{1}{6}$

Table 2: Values for the parameters in forward Central Difference method

Therefore, we expect the value of β becomes equal to 4 at the end of the computation.

1.2 Crank-Nicolson method

We only consider one option for the value of alpha for this method, we consider the values of Δt and Δx the same. So, in this case, the value of α will be:

$$\alpha = \frac{\sigma \Delta t}{2(\Delta x)^2} = \frac{1}{2\Delta x} \tag{8}$$

The Crank–Nicolson method is centered in both space and time, and an analysis of its local truncation error shows that it is second order accurate in both space and time.

$$\tau(x,t) = O(\Delta t^2 + \Delta x^2) \tag{9}$$

Since, this is an implicit method, we don't need any restriction on α. Defining the values of Δt and Δx equals to each other, means that we don't care about α. We used a traditional solver

for this method. It's essential like the one for the backward Euler method, except the known values (u^n) is an array of different values as for j, $j+1$ and $j-1$.

Let's divide the spatial domain to M parts, in this case, the values for Δt, Δx, and α will be as table 3:

M	Δx	Δt	α
128	$\frac{pi}{128}$	$\frac{pi}{128}$	20.37
256	$\frac{pi}{256}$	$\frac{pi}{256}$	40.74
512	$\frac{pi}{512}$	$\frac{pi}{512}$	81.49

Table 3: Values for the parameters in Crank-Nicolson method

2 problem 1 & 2 (considering $\Delta t = h$)

As we discussed in previous section the central difference method is first order in time, and the crank niclson is second order in time. If we change the code α equals to $\frac{1}{6}$ we would have the same the thing as the crank-nicolson one. Therefore, if we consider Δt as h, we would get beta equals to one for the first method when $\alpha = 0.5$ and equals to two when $\alpha = \frac{1}{6}$ and we would get β equals to two using the crank-nicolson method.

2.1 Results and Discussion

As discussed above and for the values set to be as the ones in table 1, 2 and 3, we get the values for β as in table 4 for the fixed-point as $(x,t) = (\frac{\pi}{2}, 0.2)$. This happens when we consider the value of h to be equals to me I don't Δx.

Also, The table 5 is the result of the code considering $\Delta t = h$.

As it can be seen, the values of the β are around the desired ones. We can conclude that:

1. We have stability restriction for the forward euler method, since it's an explicit method. But, there is no such a thing for Cranc-Nicolson method. By setting $\Delta x = \Delta t$, α would depend on the value of Δt and σ, which is 1 here. Therfore, we can have any value of α we want.

Method	Value of β
Forward Euler ($\alpha = \frac{1}{2}$)	1.9997941151677683
Forward Euler ($\alpha = \frac{1}{6}$)	3.999889659243995
Crank-Nicolson (different values of α)	1.9825505233730345

Table 4: Values for β using different methods (considering $\Delta x = h$)

Method	Value of β
Forward Euler ($\alpha = \frac{1}{2}$)	0.9989733635769921
Forward Euler ($\alpha = \frac{1}{6}$)	1.999889659243995
Crank-Nicolson (different values of α)	1.99999179764568

Table 5: Values for β using different methods (considering $\Delta t = h$)

2. By discretizing the spatial and time domain in a way that results in $\alpha = \frac{1}{6}$, the order of the accuracy and convergence will be raised up. But, we have to change the time steps and spatial discretization in a bounded area, so it neither hurts the stability nor the accuracy.

3. The cost we pay for an implicit method, will be the stability we get. But, if we make the mesh finer and finer, it still converges with the same order. As we saw this behavior in Crank-Nicolson method. So, it has the same accuracy and convergence rate with forward Euler, but there is no limit for alpha in case of stability.

4. Since at the first couple of time steps for heat equation, the temperature function changes are bigger than as we progress in time, by the numerical trick we used, the difference between W_h, $W_{\frac{1}{2}}$ and $W_{\frac{h}{4}}$ is going to be so small, that may cause round-off errors and loss of significants, which may make the value of q unstable. But, it works for a fixed-point in time for smaller values.

5. Considering $\Delta t = h$, everything seems more reasonable than the previous one. We checked the results for 10 seconds. As we expected the centeral difference scheme was of order 1 in time when $\alpha = \frac{1}{2}$, and was as order of two when $\alpha = \frac{1}{6}$ this is the same for the Crank-Nicolson method.

3 problem 3

Determining the convergence of a Cauchy sequence:

It's trivial by the code we wrote in `matlab` (shown in figure 1), that as n goes to infinity, the sequence converges to the Heaviside function, which is not a continuous function. We have to show that analytically.

Now, the analytical proof is shown in figure bellow.

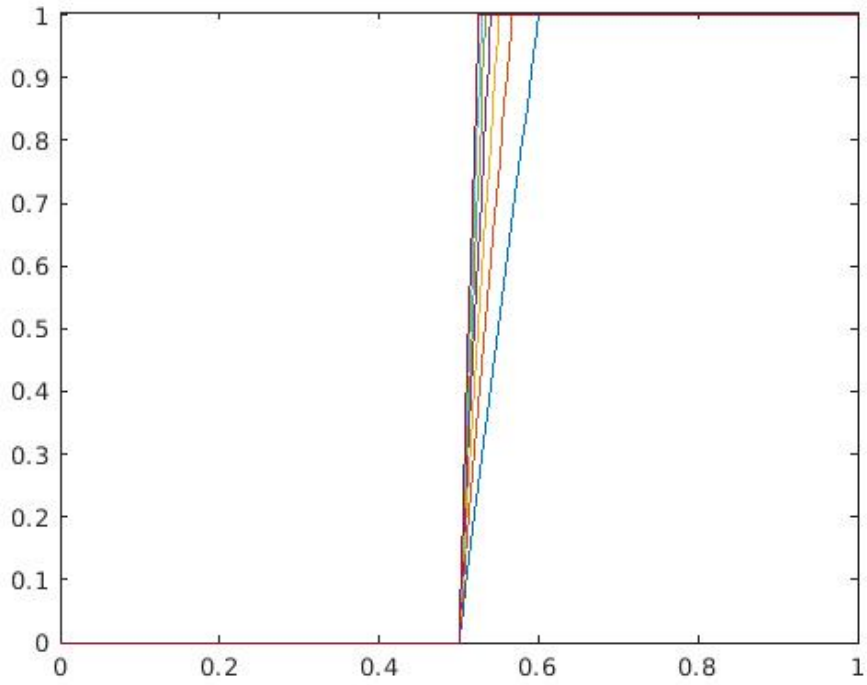

Figure 1: The sequence of continuous functions

(1)

we have to show that $\|u^n - u^m\|_1 \to 0$ as $m,n \to \infty$:

we have the triangle equality for normed spaces:

$$\|u^n - u^m\| \leq \|u^n\| - \|u^m\|$$

1. For $n = m$:

$$\|u^n\| = \|u\|^m \longrightarrow \|u^n - u^m\| \leq 0$$

for any value of n, m

2. For $n \neq m$:

if $n > m$: $\|u^n - u^m\| \leq \|u^n\| - \|u^m\|$

$$\leq \int_0^1 |u^n|\,dx - \int_0^1 |u^m|\,dx$$

$$\leq \int_0^{1/2} 0\,dx + \int_{1/2}^{1/2+1/n} |n(x-1/2)|\,dx + \int_{1/2+1/n}^1 (1)\,dx$$

$$-\left(\int_0^{1/2} 0\,dx + \int_{1/2}^{1/2+1/m} |m(x-1/2)|\,dx + \int_{1/2+1/m}^1 (1)\,dx \right)$$

(*)

for the 2nd and 6th term, we have:

$x \geq 1/2 \longrightarrow$

$x - 1/2 \geq 0 \longrightarrow$ n is positive

$n(x-1/2) \geq 0 \longrightarrow$

$|n(x-1/2)| = n(x-1/2)$ (I)

we have the same for "m"!

(I), (*) $\longrightarrow$

$$\|u^n - u^m\| \leq \int_{1/2}^{1/2+1/n} n(x-1/2)\,dx + \int_{1/2+1/n}^1 (1)\,dx$$

$$- \left(\int_{1/2}^{1/2+1/m} m(x-1/2)\,dx + \int_{1/2+1/m}^1 (1)\,dx \right)$$

$$\|u^n - u^m\| \leq \frac{n}{2}(x^2-x)\Big|_{1/2}^{1/2+1/n} + x\Big|_{1/2+1/n}^{1} - \left(\frac{m}{2}(x^2-x)\Big|_{1/2}^{1/2+1/m} + x\Big|_{1/2+1/m}^{1}\right)$$

$$\leq \frac{n}{2}\left(\left(\tfrac{1}{2}+\tfrac{1}{n}\right)^2 + \left(\tfrac{1}{n}\right)^2 + \tfrac{1}{n} - \tfrac{1}{2} - \tfrac{1}{n}\right) - \frac{n}{2}\left(\left(\tfrac{1}{2}\right)^2 - \tfrac{1}{2}\right) + \left(1 - \tfrac{1}{2} - \tfrac{1}{n}\right)$$

$$- \left(\frac{m}{2}\left(\left(\tfrac{1}{2}\right)^2 + \left(\tfrac{1}{m}\right)^2 + \tfrac{1}{m} - \tfrac{1}{2} - \tfrac{1}{m}\right) - \frac{m}{2}\left(\left(\tfrac{1}{2}\right)^2 - \tfrac{1}{2}\right) + \left(1 - \tfrac{1}{2} - \tfrac{1}{m}\right)\right)$$

$$\leq \frac{n}{2}\left(\frac{1}{n^2}\right) + \cancel{\frac{1}{2}} - \frac{1}{n} - \frac{m}{2}\left(\frac{1}{m^2}\right) - \cancel{\frac{1}{2}} + \frac{1}{m}$$

$$\leq \frac{1}{2m} - \frac{1}{2n}$$

as $m, n \to \infty$, $\frac{1}{n}, \frac{1}{m} \to 0$

then the whole thing goes to 0. So:

$$\|u^m - u^n\| \to 0 \quad \text{as} \quad m, n \to \infty \quad . \square$$

if $m > n$: everything is the same expect the signs are different.

and it Convergences to 0 again.

(2) It's not Convergent to a Continouos function:

Let's defin: $f(x) = \lim_{n\to\infty} u^n(x)$, we have:

$$u^n(x) = \begin{cases} 0 & [0, 1/2] \\ n(x-0.5) & [1/2,\ 1/2+1/n] \\ 1 & [1/n+1/2,\ 1] \end{cases} \longrightarrow \text{as } n \to \infty \text{ this interval dissipate!}$$

then $f(x) = \lim_{n\to\infty} u(x) = \begin{cases} 0 & [0, 1/2] \\ 1 & [1/2, 1] \end{cases} \longrightarrow f(x) = \begin{cases} 0 & [-, 1/2] \\ 1 & [1/2,] \end{cases}$

$$\begin{cases} \lim_{x\to 1/2^+} f(x) = 1 \\ \lim_{x\to 1/2^-} f(x) = 0 \end{cases} \longrightarrow \boxed{\lim_{x\to 1/2^+} f(x) \neq \lim_{x\to 1/2^-} f(x)}$$

it doesn't satisfy the deffiniti-n of Gntinuity!

it's Heaviside function as $n \to \infty$!

going to look like

A The Appendix for the Python Code

A.1 problem 1

Python code for the problem one is as what follows.

```
import sys
import math
import matplotlib.pyplot as plt


def init_1(M):
    U0 = []
    for j in range(0, M):
        if j <= M / 2:
            U0.append(j * math.pi / M)
        else:
            U0.append(math.pi - j * math.pi / M)
    U0.append(0)
    return U0


def init_2(M):
    U0 = []
    for j in range(0, M):
        x = j * math.pi / M
        U0.append(1 - math.exp(- math.sin(x) ** 2))
    U0.append(0)
    return U0


def fwd_solver(alpha, M, T):
    U0 = init_2(M)

    U_list = [U0]
    for n in range(0, T):
        new_U = update_temperature(U_list[-1], alpha)
        U_list.append(new_U)

    return U_list[-1]


def update_temperature(current_temp, alpha):
```

```
    new_temp = []
    M = len(current_temp)
    for j in range(0, M):

        if j == 0 or j == M - 1:
            new_temp_j = 0
        else:
            new_temp_j = current_temp[j] +
            alpha * (current_temp[j - 1] -
            2 * current_temp[j] + current_temp[j + 1])

        new_temp.append(new_temp_j)

    return new_temp


alpha = 1 / 6
sigma = 1

fixed_time = 0.032  # in seconds

mid_temps = []

init_M = 128
for M in [init_M, init_M*2, init_M*4]:
    deltaX = math.pi / M
    deltaT = (deltaX ** 2) * alpha / sigma

    T = int(fixed_time / deltaT)
    print(T)

    curr_temp = fwd_solver(alpha, M, T)

    mid_temp = curr_temp[int(M / 2)]
    mid_temps.append(mid_temp)

q = (mid_temps[0] - mid_temps[1]) / (mid_temps[1] - mid_temps[2])
print(mid_temps, q)

print(math.log(q, 2))
```

A.2 problem 2

Python code for the problem two is as what follows.

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


def TDMSolver(a, b, c, u):
    N = len(a)

    ac = np.array(a)
    bc = np.array(b)
    cc = np.array(c)
    uc = np.array(u)

    for i in range(1, N):
        w = ac[i] / bc[i - 1]
        bc[i] = bc[i] - w * cc[i - 1]
        uc[i] = uc[i] - w * uc[i - 1]

    x = np.zeros((N,))
    x[N - 1] = uc[N - 1] / bc[N - 1]

    for i in range(N - 2, -1, -1):
        x[i] = (uc[i] - cc[i] * x[i + 1]) / bc[i]

    return x


def init_1(N):
    dx = math.pi / N

    # initial function
    U0 = []
    for j in range(0, N + 1):
        u = j * dx
        if j == N:
            u = 0
        elif j >= N / 2:
            u = math.pi - j * dx
```

```
        U0.append(u)

    U0 = np.array(U0)
    return U0


def init_2(N):
    U0 = []
    for j in range(0, N + 1):
        x = j * math.pi / N
        if j == N:
            x = 0
        U0.append(1 - math.exp(- math.sin(x) ** 2))
    return U0


def solver(alpha, N, T):

    # initial function
    U0 = init_1(N)

    # U0: a list showing temprature for N+1 points on the pole.

    # Tridiagonal solver

    a = np.zeros((N + 1, ))
    b = np.zeros((N + 1, ))
    c = np.zeros((N + 1, ))

    for i in range(0, N + 1):
        if i == 0 or i == N:
            b[i] = 1
            a[i] = 0
            c[i] = 0

        else:

            b[i] = 1 + alpha
            c[i] = -alpha/2
            a[i] = -alpha/2

    U = U0
```

```
    U_checkpoints = [U0]
    for t in range(1, T + 1):
        R = []
        for j in range(len(U)):
            if j == 0:
                Rj = 0
            elif j == len(U) - 1:
                Rj = 0
            else:
                Rj = alpha/2 * U[j-1] +
                (1 - alpha) * U[j] + alpha/2 * U[j+1]

            R.append(Rj)

        new_U = TDMSolver(a, b, c, R)
        U_checkpoints.append(new_U)
        U = new_U

    return U_checkpoints[-1]


sigma = 1
fixed_time = 0.05  # in seconds

mid_temps = []

init_N = 128
for N in [init_N, init_N*2, init_N*4]:
    dx = math.pi / N
    dt = dx
    alpha = sigma / dx

    T = int(fixed_time / dt)
    print(alpha, T)

    curr_temp = solver(alpha, N, T)

    mid_temp = curr_temp[int(N / 2)]
    mid_temps.append(mid_temp)

q = (mid_temps[0] - mid_temps[1]) / (mid_temps[1] - mid_temps[2])
print(mid_temps, q)
```

A.3 problem 3

The matlab code for problem 3 is as what follows.

```
clc
close all
clear all

for n = 10:5:40
    syms x
    y = piecewise(0 <= x <= 0.5,0 ...
        ,0.5 <= x <= 0.5 + (1/n),n*(x-0.5), ...
        0.5 + (1/n) <= x <= 1,1);
    fplot(y)
    axis([0 1 0 1.005])
    hold on
end
```