Computational Methods
Lab Session 1

Augustin Reille

2 october 2017

Contents

Abstract

In these lab sessions we had to use a C++ code to find the results of one equation with two functions, using differents discretization schemes and compare them to the analytical solutions to see how the errors change.

Chapter 1

Introduction

In this first lab, we looked how the accuracy of a solution of a linear equation was changing by using differents numerical schemes.
We are considering the following problem :

$$\frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} = 0$$

$$x \in [\,-40, 40]$$

$$u = 1$$

We have two functions to study, with boudaries conditions :

$$f_0(x, 0) = \frac{1}{2}(sign(x) + 1)$$

$$f_0(-40, t) = 0$$

$$f_0(40, t) = 1$$

and

$$f_1(x, 0) = \frac{1}{2}exp(-x^2)$$

$$f_1(-40, t) = 0$$

$$f_1(40, t) = 0$$

Chapter 2

Methods

We want to solve the above problem with the initial conditions f_0 and f_1 on an uniform grid containing 100 points in x, using :

- Upwind scheme
- Central difference scheme
- Lax scheme
- Leapfrog scheme

Chapter 3

Results

Upwind scheme

3.1 Equations

The upwind scheme corresponds to a discretisation with forward time and backward space.
We have :

$$\frac{f_i^{n+1} - f_i^n}{\Delta t} + a\frac{f_i^n - f_{i-1}^n}{\Delta x} = 0$$

$$\Leftrightarrow f_i^{n+1} = f_i^n(1 - a\frac{\Delta t}{\Delta x}) + a\frac{\Delta t}{\Delta x}f_{i-1}^n \tag{3.1}$$

3.2 Steps and Courant number

Our study interval is $[-40, 40]$, so we have 80 points. We need to make our study over 100 points.
Our Δx will be :

$$\Delta x = \frac{80}{100} = 0,8$$

To have a stable discretization scheme, we need to have our courant number C:

$$C = u\frac{\Delta t}{\Delta x} < 1$$

As we have $u = 1$, by taking $\Delta t = \Delta x/2$, we will always have $C = 0,5 < 1$.
(see Appendix A)

3.3 Initialization and results calculation

For each time value, we want the discrete solution for each of the 100 points of the interval. Our matrix containing all results will look like :

$$\begin{matrix} t_0 & t_1 & t_2 & \cdots & t_n \end{matrix}$$
$$\begin{pmatrix} a & b & c & \cdots & j \\ d & e & f & \cdots & k \\ g & h & i & \cdots & l \\ \vdots & \vdots & \vdots & \ddots & z \\ w & x & y & \cdots & z \end{pmatrix} \begin{matrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_i \end{matrix}$$

Here with the initial conditions we have, we are able to fullfill vectors t_0 and x_0. (see Appendix A) Then, with equation (3.1), we will be able to fullfill recursively the rest of the matrix, with a double for-loop (see Appendix B).

3.4 Graphical results and errors calcuation

The red curve is the analytical solution, and the blue one is the one we get with the code.
We can see that the error increase with the time : if we take a value of a later time, the error will be bigger.

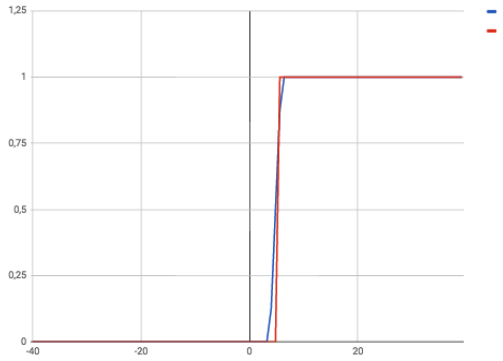

Figure 3.1: f0 with t = 5s

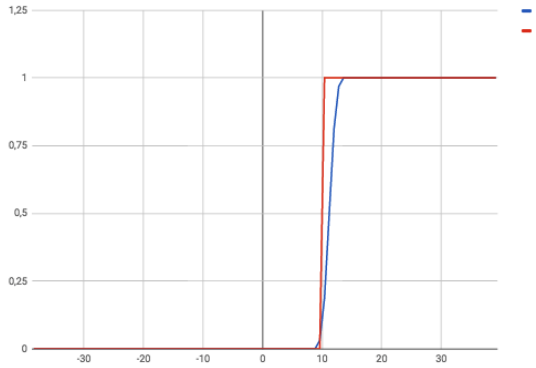

Figure 3.2: f0 with t = 10s

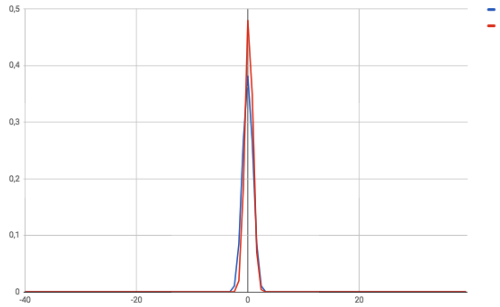

Figure 3.3: f1 with t = 5s

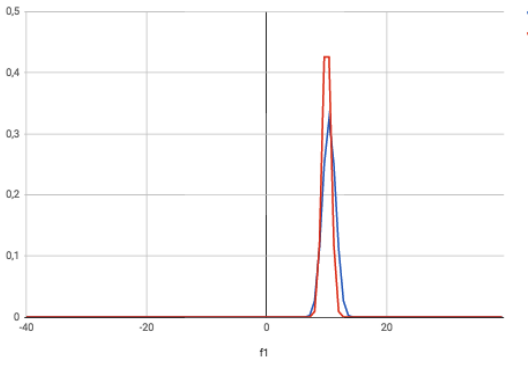

Figure 3.4: f1 with t = 10s

Chapter 4

Discussion

4.1 Conclusion

We see that in most of cases, the error increases with the time. By doing a grid convergence study, and setting a grid with more points, we see that the error decrease; however the error is susceptible to increase locally due to computational errors (like truncation or calculation errors).

4.2 Future work

I only studied this grid convergence for the upwind scheme. I will have to make the study with the others schemes, to see if they are more efficients, more stable with time and with a greater number of mesh points.

Chapter 5

Appendices

5.1 Appendix A : initialization

C++ code to initialize our result table

```
int nspace = 100;
int tottime = 22;
double dx = 0.8;
double dt = dx / 2;
int ntime = tottime / dt;
double courantNumber = dt / dx;

// the vector containing our results :
double results[ntime][nspace];

/* initial conditions
* here is the code for f1 only
*/
for (int i = 0; i < nspace; i++)
{
    double x = -40 + i * dx;
    results[0][i] = 0.5 * exp(-(x * x));
}
for (int n = 0; n < ntime; n++)
{
    results[n][0] = 0;
    results[n][nspace - 1] = 0;
}
```

5.2 Appendix B : the double for-loop

C++ code to fullfill our result table

```
// loop
for (int n = 1; n < ntime; n++)
{
    for (int i = 1; i < nspace - 1; i++)
    {
        results[n][i] = results[n - 1][i] * (1 - courantNumber)
                        + courantNumber * results[n - 1][i - 1];
    }
}
```