Computational Methods and C++

Numerical solving of the heat conduction equation

Victor Ostertag
Student number: 275780

Number of words: 4,785

December 11, 2017

Contents

List of Figures

Nomenclature

Symbol	Meaning	Units
T_{in}	Initial temperature of the wall	°F
T_{sur}	Surface temperature of the wall	°F
D	Diffusivity	ft^2/hr
L	Length of the wall	ft
f	Function solution to the heat equation	No unit

Abstract

In this report, we will solve the heat equation using four different schemes: DuFort-Frankel, Richardson, Laasonen and Crank-Nicholson. We will analyze the properties of each scheme, implement them in a C++ program and see if the results match what we could have predicted in theory. This gives us the opportunity to witness first hand just how complex choosing the right scheme for a problem can be. The results will show how the Richardson scheme is deemed useless because of its instability, the influence of the consistency of a scheme with DuFort-Frankel which results were less accurate than the Laasonen method despite its better truncation error and finally how increasing the mesh size can reduce the computational cost but increase the truncation error as well. In the end, Crank-Nicholson seems to be the best scheme to solve this equation overall, but can be matched by DuFort-Frankel when the step size in space is much bigger than the step size in time. In this case, it would be better to use DuFort-Frankel, as it is an explicit scheme and thus faster to compute than Crank-Nicholson.

1 Introduction

1.1 Numerical solving of partial differential equations

This report focuses on the numerical solving of a Partial Differential Equation (PDE), that is to say an equation in which the unknown's partial derivatives are involved.[1] Finding its analytical solution is often really hard and, sometimes, even impossible. And if numerical solutions were once too demanding in terms of resources to even be effective, it is not the case anymore thanks to computers that quickly became powerful enough for such techniques. Now, computational methods are broadly used and one of the main topics of fluid mechanics engineering.[2]

The most important thing when it comes to creating a numerical solution is choosing the right scheme. So many are available but they are not always suited to the problem at hand: Do we need to be very accurate and therefore use a method that takes a lot of time to implement and more resources or not? But most importantly, some methods, when applied without a proper analysis, can lead to getting the wrong solution, as we will see in this report. In our case, we will witness first hand the importance of choosing the right method by focusing on the solving of the heat equation.

1.2 Heat equation

Consider the following problem, on which we will work in this report:
A one feet thick wall whose initial temperature is of 100°F is being heated, on both of his sides, at 300°F. The wall is composed of nickel steel and has a diffusivity of 0.1 ft^2/hr. How will the temperature inside of the wall evolve in function of the time?

The solution to this problem can be found by solving the unsteady one-space dimensional heat conduction equation:

$$\frac{\delta T}{\delta t} = D\frac{\delta^2 T}{\delta x^2} \tag{1.1}$$

We can already get the boundary conditions of the equation from the introductory part. Let $f(t,x)$ be a function that solves (1.1), with t representing the time and x the space. We know that:

$$f(0,x) = \begin{cases} 300 & \text{for } x = 0 \text{ and } x = \text{L} \\ 100 & \forall x \in]0,\text{L}[\end{cases} \tag{1.2}$$

$$f(t,x) = 300 \quad \text{for } x = 0 \text{ and } x = \text{L} \quad \text{and} \quad \forall t \tag{1.3}$$

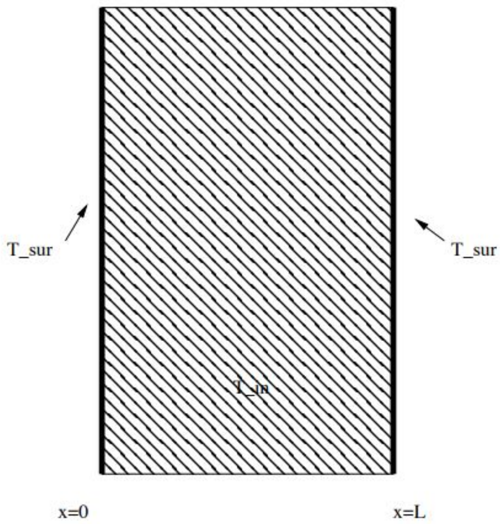


Figure 1.1: Representation of the problem

We will solve the equation using four different numerical schemes:

- DuFort-Frankel
- Richardson
- Laasonen
- Crank-Nicholson

For each scheme, we will analyze their properties, implement them in a C++ program designed to solve this particular equation and then compare our results with the analytic solution, given to us in the assignment's second question:

$$f(t,x) = T_{sur} + 2(T_{in} - T_{sur}) \cdot \sum_{m=1}^{\infty} e^{-D(m\pi/L)^2 t} \cdot \frac{1-(-1)^m}{m\pi} \cdot \sin\left(\frac{m\pi x}{L}\right) \tag{1.4}$$

2 Method

2.1 From PDE to FDE

The heat conduction equation (1.1), in this form, cannot be implemented using a computer, which would not be able to understand what a derivative is. That is why an approximation of these derivatives, using only the unknown function that we will call f, should be found. We will focus on the use of the Taylor series expansion to do so, as described in *Computational Fluid Dynamics*, written by K.A. Hoffmann and S.T. Chiang. [3]

Using the Taylor series expansion of f, we have that:

$$f(x+\Delta x) = f(x) + \Delta x \sum_{n=0}^{\infty} \frac{\Delta x^n}{n!}\frac{\delta^n f}{\delta x^n} = f(x) + \frac{\delta f}{\delta x}\Delta x + \frac{\delta^2 f}{\delta x^2}\frac{\Delta x^2}{2} + \frac{\delta^3 f}{\delta x^3}\frac{\Delta x^3}{3} + \dots \tag{2.1}$$

If we isolate $\frac{\delta f}{\delta x}$ in (2.1):

$$\frac{\delta f}{\delta x} = \frac{f(x+\Delta x) - f(x)}{\Delta x} - \frac{\delta^2 f}{\delta x^2}\frac{\Delta x^2}{2} - \frac{\delta^3 f}{\delta x^3}\frac{\Delta x^3}{3} - \dots \tag{2.2}$$

The red part of the equation (2.2) must be truncated, a computer not being able to deal with infinities:

$$\frac{\delta f}{\delta x} = \frac{f(x+\Delta x) - f(x)}{\Delta x} + O(\Delta x) \tag{2.3}$$

And now we have an approximation of the first derivative in space of the unknown f that can be computed. We just have to keep in mind that truncation error. In this example, I went forward in space, but we could have done the same going backward ($f(x-\Delta x)$). I also chose the derivate in space, but the one in time could have been used as well.

From there, it is just a question of what one wishes to accomplish: we could go further before truncating (2.2) to be more accurate, we could isolate an other term if we wish to approximate the second derivative like it will be the case for our heat equation, or we could get new equations by adding the equation going forward and the one we get going backward for example, and so on.

We can then use these approximations in the heat equation (1.1) and get a Finite Differential Equation (FDE) instead of a Partial Differential Equation (PDE), which can be solved numerically. This is how various schemes are created. I went into more details on how to build the Richardson scheme in the appendix A.

2.2 Properties of a scheme

There are three main properties that a scheme can have and that can be analyzed to insure its usability:

- **Stability:** A scheme is stable if errors made do not grow without limit [4].
- **Consistency:** A scheme is consistent if the FDE approaches the original PDE as the mesh sizes tends to zero [4].
- **Convergence:** A scheme is convergent if its solution approaches that of the PDE as the mesh sizes tends to zero [4].

Often, the Lax theorem is used to prove the convergence of a scheme, as this property is quite hard to demonstrate. It states that a scheme is convergent if and only if it is consistent and stable [4]. Once we proved that a scheme has these three properties for the problem at hand, we can be sure it will give us a correct solution. In this report, we will mainly focus on stability.

2.3 Analyzing each schemes

The first step of applying a scheme is to analyze its stability. This is especially true with explicit schemes, as they, most of the time, are conditionally stable on fairly small intervals while implicit schemes are usually unconditionally stable. In this part of the report, I will go over each scheme that will be used, check their stability and present their FDE. I will go into a lot of details for the Richardson scheme, providing calculations in the appendix B. For the rest of the schemes, the process would have been more or less the same, and the results have simply been taken from *Computational Fluid Dynamics*, written by K.A. Hoffmann and S.T. Chiang. [5].

2.3.1 Richardson

For the Richardson scheme, central differencing is used twice, which gives the following FDE:

$$\frac{f_i^{n+1} - f_i^{n-1}}{2\Delta t} = D \cdot \frac{f_{i+1}^n - 2f_i^n + f_{i-1}^n}{\Delta x^2} \tag{2.4}$$

$$f_i^{n+1} = f_i^{n-1} + \frac{2D\Delta t}{\Delta x^2} \cdot \frac{f_{i+1}^n - 2f_i^n + f_{i-1}^n}{\Delta x^2} \tag{2.5}$$

The accuracy of this method is of order $O(\Delta t^2)$ and $O(\Delta x^2)$ and is unconditionally unstable and therefore useless. This scheme really only helps us to get the next method we will analyze: DuFort-Frankel. All the calculations that led to these results can be found in the appendix B.

2.3.2 DuFort-Frankel

In order to make the Richardson scheme conditionally stable, we can replace f_i^n by the average value of f_i^{n+1} and f_i^{n-1}, which gives us the DuFort-Frankel scheme's FDE:

$$\frac{f_i^{n+1} - f_i^{n-1}}{2\Delta t} = D \cdot \frac{f_{i+1}^n - 2\dfrac{f_i^{n+1} + f_i^{n-1}}{2} + f_{i-1}^n}{\Delta x^2} \tag{2.6}$$

$$f_i^{n+1} = f_i^{n-1} + \frac{2D\Delta t}{\Delta x^2} \cdot \frac{f_{i+1}^n - 2\dfrac{f_i^{n+1} + f_i^{n-1}}{2} + f_{i-1}^n}{\Delta x^2} \tag{2.7}$$

This scheme has an accuracy of $O(\Delta t^2)$, $O(\Delta x^2)$ and is unconditionally stable.

2.3.3 Forward Time / Central Space (FTCS)

This scheme is not one of which we were asked to implement, but the two explicit methods of the assignment (Richardson and DuFort-Frankel) are using the two previous time steps in their FDE (c.f. (2.7) and (2.5)), which will be problematic for the first time step. Therefore, we have to use a starter solution: a scheme that requires only one previous time step to compute to get that missing step. To do so, I used the FTCS method, which FDE is the following:

$$f_i^{n+1} = f_i^n + \frac{D\Delta t}{\Delta x^2} \cdot (f_{i+1}^n - 2f_i^n + f_{i-1}^n) \tag{2.8}$$

The accuracy of this method is of $O(\Delta t)$ and $O(\Delta x^2)$, which impacts the accuracy of the method used to get the other time steps as we will see more closely in the ”Results” section. The FTCS is conditionally stable for $D\frac{\Delta t}{\Delta x^2} \leqslant 0.5$. In our case, this will always be the case.

2.3.4 Laasonen

The Laasonen method is an implicit scheme with the following FDE:

$$\frac{f_i^{n+1} - f_i^n}{\Delta t} = D \cdot \frac{f_{i+1}^{n+1} - 2f_i^{n+1} + f_{i-1}^{n+1}}{\Delta x^2} \tag{2.9}$$

This method is unconditionally stable and has an accuracy of $O(\Delta t), O(\Delta x^2)$ [6].

2.3.5 Crank-Nicholson

The last method we will use is the Crank-Nicholson one, which has the following FDE:

$$\frac{f_i^{n+1} - f_i^n}{\Delta t} = \frac{D}{2} \cdot \left(\frac{f_{i+1}^{n+1} - 2f_i^{n+1} + f_{i-1}^{n+1}}{\Delta x^2} + \frac{f_{i+1}^n - 2f_i^n + f_{i-1}^n}{\Delta x^2} \right) \tag{2.10}$$

This implicit scheme is unconditionally stable and has an accuracy of $O(\Delta t^2), O(\Delta x^2)$.

2.4 Checking the errors

We only checked the stability of each scheme, which, as we have seen earlier, is not the only property a scheme should have for it to be used in practice. This is why we will have to compare the results to experimental data or the analytical solution. In our case, we only have access to the analytical solution, which is why we will calculate the two-norm (also known as Euclidean norm) of each scheme's error matrix. The two-norm is chosen because, as we have seen in class, it penalizes large mistakes and rewards small errors, which makes it more useful in our case.

2.5 Implementing the schemes

In this part of the report, we will discuss how to implement the schemes. The design of the actual solution I came up with will be discussed in the next section. This implementation really only depends on the type of the scheme:

2.5.1 Explicit Schemes

The FDE of our explicit schemes ((2.8), (2.7), (2.5)) only have one unknown. Therefore, we just have to solve that equation for each space point to get the next time step. Let's take the FTCS scheme as an example. Using (2.8), if $n = 0$ and $i = 1$, we have that:

$$f_1^1 = f_1^0 + \frac{D\Delta t}{\Delta x^2} \cdot (f_2^0 - 2f_1^0 + f_0^0)$$

And we can repeat the process for all the space points:

$$\begin{aligned} f_2^1 &= f_2^0 + \frac{D\Delta t}{\Delta x^2} \cdot (f_3^0 - 2f_2^0 + f_1^0) \\ &\vdots \\ f_{i-1}^1 &= f_{i-1}^0 + \frac{D\Delta t}{\Delta x^2} \cdot (f_i^0 - 2f_{i-1}^0 + f_{i-2}^0) \end{aligned}$$

And now, we have $f_i^1 \forall i$ of our grid. We can than repeat the process for n=1, and so on until we reach the desired time step.

2.5.2 Implicit schemes

It is more complex to implement implicit schemes: they have better stability (most of the time unconditionally stable) and are often more accurate, but this comes with a higher implementation cost. In their FDE ((2.9), (2.10)), there are many unknowns, which means that we will have to solve a linear system of equations for each time step that are the following ones (the calculations that led to these systems can be found in the appendix C). For Laasonen:

$$\begin{bmatrix} -c & 2c+1 & 0 & 0 & 0 & \dots & 0 \\ 0 & -c & 2c+1 & -c & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -c & 2c+1 & -c & 0 \\ 0 & & \dots & & 0 & -c & 2c+1 \end{bmatrix} \cdot \begin{bmatrix} f_1^{n+1} \\ f_2^{n+1} \\ \vdots \\ f_{i-2}^{n+1} \\ f_{i-1}^{n+1} \end{bmatrix} = \begin{bmatrix} f_1^n + c \cdot t_{Sur} \\ f_2^n \\ \vdots \\ f_{i-2}^n \\ f_{i-1}^n + c \cdot t_{Sur} \end{bmatrix}$$

with $c = \dfrac{D\Delta t}{\Delta x^2}$. And for Crank-Nicholson:

$$\begin{bmatrix} -c & 2c+1 & 0 & 0 & 0 & \dots & 0 \\ 0 & -c & 2c+1 & -c & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -c & 2c+1 & -c & 0 \\ 0 & & \dots & & 0 & -c & 2c+1 \end{bmatrix} \cdot \begin{bmatrix} f_1^{n+1} \\ f_2^{n+1} \\ \vdots \\ f_{i-2}^{n+1} \\ f_{i-1}^{n+1} \end{bmatrix} = \begin{bmatrix} (1-2c)\cdot f_1^n + c\cdot(f_2^n + f_0^n) + c\cdot t_{Sur} \\ (1-2c)\cdot f_2^n + c\cdot(f_3^n + f_1^n) \\ \vdots \\ (1-2c)\cdot f_{i-2}^n + c\cdot(f_{i-1}^n + f_{i-3}^n) \\ (1-2c)\cdot f_{i-1}^n + c\cdot(f_i^n + f_{i-2}^n) + c\cdot t_{Sur} \end{bmatrix}$$

with $c = \dfrac{D\Delta t}{2\Delta x^2}$.

The solving of a system of equations is generally done with the LU decomposition. But, in this particular case, the left matrix is tridiagonal which allows us to use the Thomas algorithm[7]. This is much better as it will save us a lot of memory, and, more importantly, a lot of time since Thomas algorithm's complexity is of $O(n)$ [8] while LU factorization has a complexity of $O(n^{2.376})$.[9]

Once we solved the equations for the time step $n+1$, we can then use the results to create the next set of equations for $n+2$ and so on until we get the time step we would like to solve. It is important to note that only the right matrix needs to be recalculated with the new values. The left matrix will always be the same, which will save us time.

3 Numerical Algorithms

3.1 In a nutshell

I have designed a program that can solve the heat equation described in the introduction in a way that allows the user to easily change any of the parameters.

Here's an example of how everything works and what can be achieved:
Let's solve the heat equation using the DuFort-Frankel scheme with the parameters given in the assignment.

```
// values given in the subject
Parameters parameters;
parameters.D = 0.1;
parameters.L = 1;
parameters.tIn = 100;
parameters.tSur = 300;
parameters.deltaX = 0.05;
parameters.deltaT = 0.01;
parameters.numberOfTimePoints = 50;
parameters.numberOfSpacePoints = (int)(parameters.L / parameters.deltaX);
// creation of the Laasonen scheme
DuFortFrankel dufort = DuFortFrankel(parameters);
```

The DuFort-Frankel scheme is now initialized and ready to be used. We can solve the equation for all the time points we have asked for simply by doing this:

```
laasonen.solve();
```

If we wish to change one parameter, it can easily be done:

```
parameters.D = 0.2 // parameter I wish to change
dufort.setParameters(parameters);
dufort.solve(); // solve with new parameters
```

We can then print the results in the terminal with the Printer class and its `printInConsole(int t)` method for a precise time step t, or create a .dat file, which will be in the "datFiles" repertory, and get the commands to plot that graph using gnuplot.

```
Printer printer = Printer(&dufort);
// printing the temperature at each space point for the 10th timestep
printer.printInConsole(10);
printer.createDatFileForT(10);
printer.gnuplotForT(10);
```

Using the gnuplot commands, which are stored in the "commands" .txt file, we would get the following graph as a .png stored in the "datFiles" repertory:

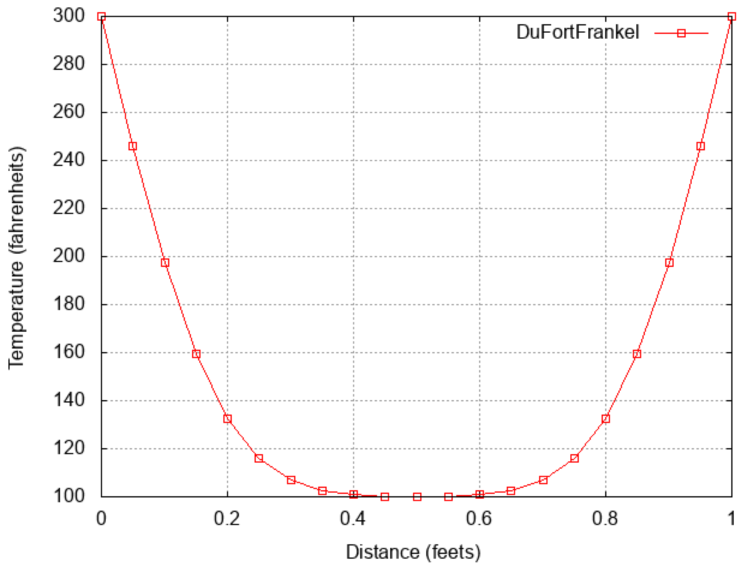


Figure 3.1: Example of graph we can get

We can also use the `Printer` to compare our scheme to an other one:

```
// First we build the scheme we wish to compare and his .dat files
AnalyticalSolution analytical = AnalyticalSolution(parameters);
analytical.solve();
printer.setPdeSolver(&analytical);
printer.createDatFileForT(10);
// Back to our main PDESolver, comparing both schemes using gnuplot at t=10
printer.setPdeSolver(&dufort);
printer.gnuplotForTCompareTo(10, &analytical);
```

Using these commands, we will get the following graph:

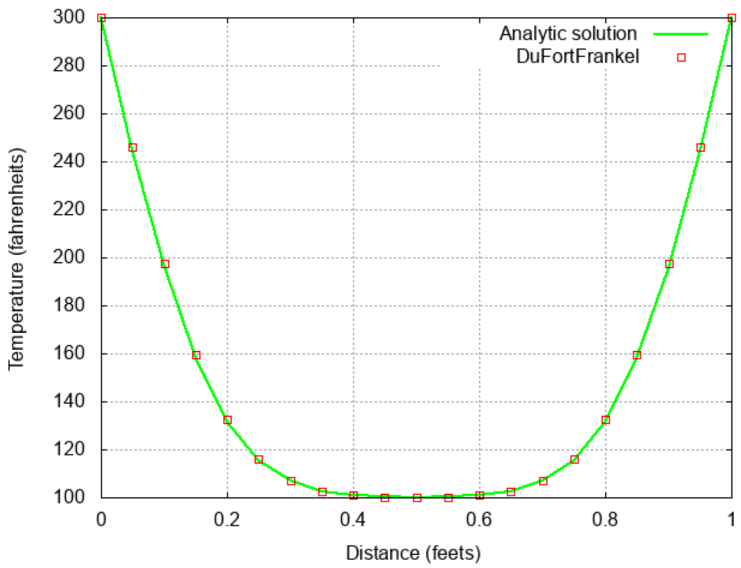


Figure 3.2: Example of comparison graph we can get

Finally, if you wish to get a closer look at the difference between two schemes, you can get the .dat file for the L2 norm calculated with the two schemes:

```
// creating .dat files with the time points at which we want to see the norm
printer.datFileErrorsComparedTo({ 10,20,30,40 }, &analytical);
// getting the gnuplot commands
printer.gnuplotErrorsCompareTo(&as);
```

We would get the following graph:

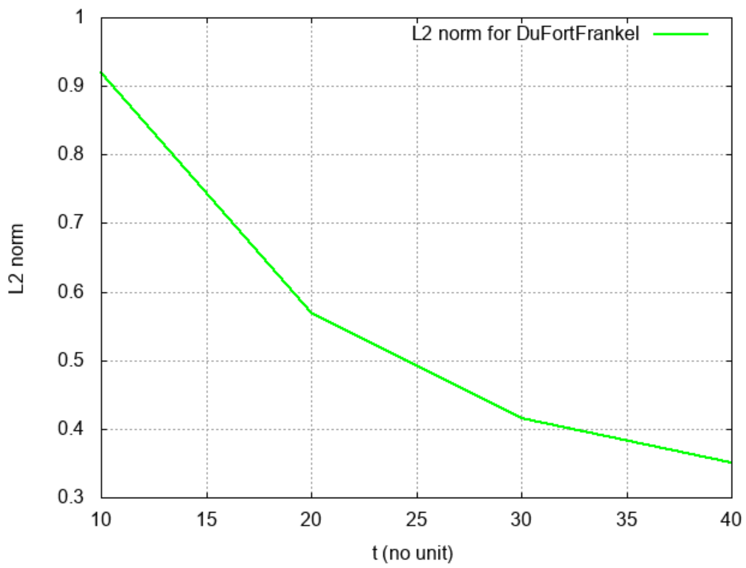


Figure 3.3: Example of L2-Norm difference graph

3.2 UML Diagram

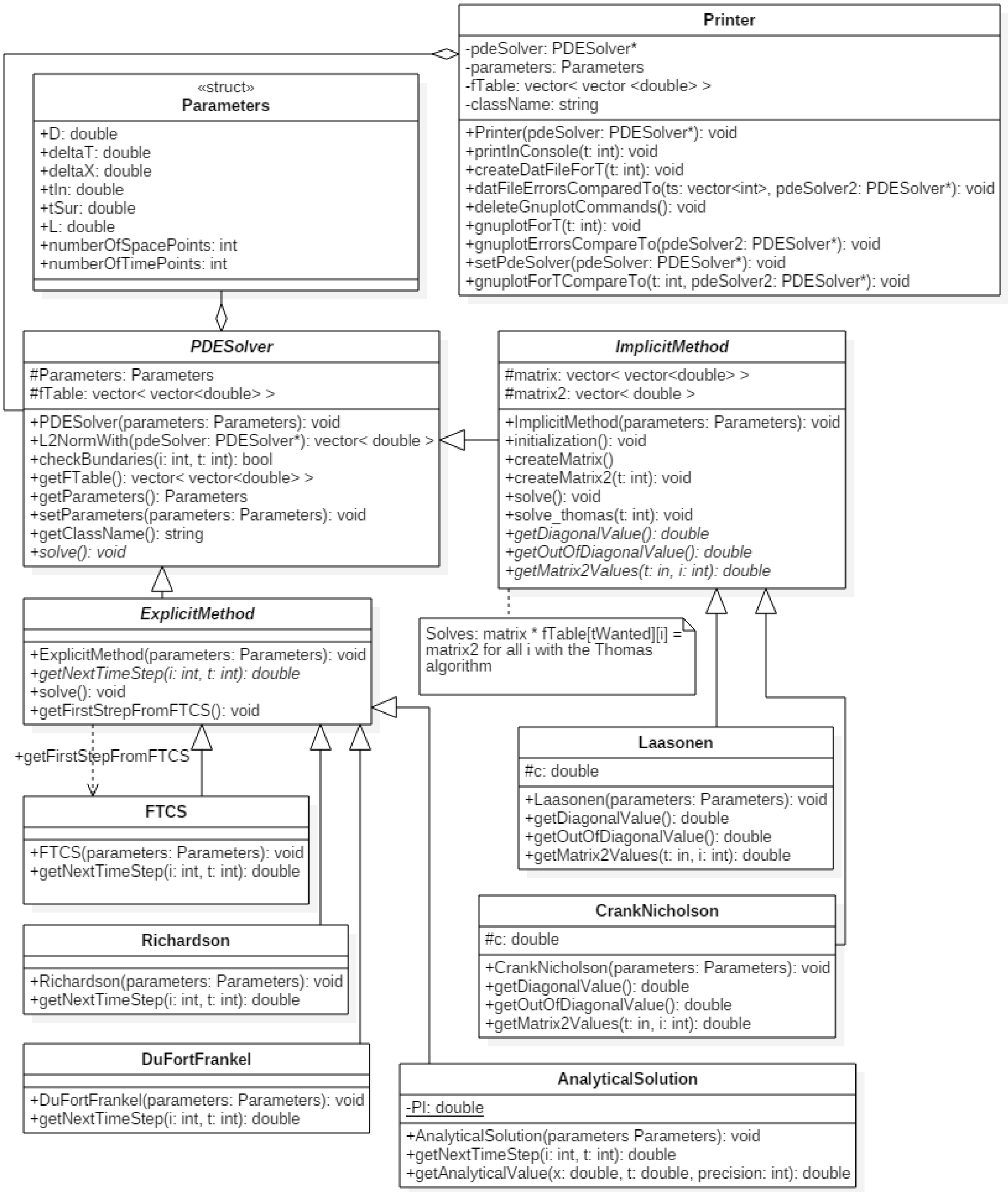


Figure 3.4: UML Diagram of my program

3.3 Design of the solution

3.3.1 PDESolver

We are willing to solve a Partial Differential Equation (PDE) using four different schemes. But, no matter the method we wish to use, quite a lot of things are similar, which is why I implemented an abstract class from which all schemes will inherit: `PDESolver`. In this class, we will find members and methods useful for any scheme:

- the class's members: all the parameters of the equation (gathered in a structure to make it easier to write for the user) as well as a vector of vector (called `fTable`) to store all the values of the solution function f for all the different time steps (from 0 to m) and for each space point of our grid (from 0 to n):

$$\text{fTable} = \begin{bmatrix} \begin{bmatrix} f(t=0,i=0) & f(t=0,i=1) & \dots & f(t=0,i=n-1) & f(t=0,i=n) \end{bmatrix} \\ \begin{bmatrix} f(t=1,i=0) & f(t=1,i=1) & \dots & f(t=1,i=n-1) & f(t=1,i=n) \end{bmatrix} \\ \vdots \\ \begin{bmatrix} f(t=m,i=0) & f(t=m,i=1) & \dots & f(t=m,i=n-1) & f(t=m,i=n) \end{bmatrix} \end{bmatrix}$$

- some methods:
  - The constructor, to get the members from the user and initialize `fTable` with the right size
  - A few getters that will be useful for the `Printer` class
  - A setter for the parameters to use
  - A function that checks if a coordinate is at a boundary condition
  - A function that calculates the two-norm of the error matrix of our scheme with an other one

What makes this an abstract class, is the pure virtual method `solve`. In deed, the way to solve the equation depends on the type of the scheme: explicit or implicit. This is why I created two more abstract classes for both of these types.

3.3.2 ExplicitMethod

`ExplicitMethod` is the class from which all the explicit schemes will inherit. The only difference between two explicit schemes is the way to get the next time step. For example, for the forward time/central space (FTCS) scheme, the relation will be:

$$\texttt{fTable}[n+1][i] = \texttt{fTable}[n][i] + \frac{\alpha\Delta t}{\Delta x^2} \cdot \Big(\texttt{fTable}[n][i+1] - 2\texttt{fTable}[n][i] + \texttt{fTable}[n][i-1]\Big) \tag{3.1}$$

And this is the only thing that changes between each explicit scheme. Which is why we can implement the `solve` method in this abstract class in which we use the pure virtual method `getNextTimeStep` that each explicit scheme will implement in their own class with an equation similar to (3.1). You can find all the relations in the "Method" section. For some schemes, we need two time steps to get to the next, which causes some problem for the first time step. That's why I created the function `getFirstStepFromFTCS` in the `ExplicitMethod` class, to get the first time step using the FTCS scheme.

You probably noticed in the UML diagram that the analytical solution is an explicit scheme. This is because the way to solve the equation analytically is very similar to the way we solve it for an explicit scheme. That way, we avoid duplicate code in our program.

3.3.3 ImplicitScheme

`ImplicitScheme` is the class from which all the implicit schemes will inherit. The way such schemes need to be solved is very different to the explicit ones and a bit more complicated (but, as we will see, they are much more accurate). As we have seen in the "Method" section, we will have to solve the following equation to get all the values of the next time step (`fTable[t]`):

$$\texttt{matrix} \cdot \texttt{fTable[t]} = \texttt{matrix2} \tag{3.2}$$

The name of these variables is not the best, I am aware of it, but I did not manage to find more obvious ones, so I just went with it... Both `matrix` and `matrix2` were calculated in the "Method" section and their value is the only difference between two implicit schemes, which is why I created virtual functions that will give the values of these matrices for each implicit scheme.

As discussed in the "Method" part, the solving of (3.2) will be done thanks to the Thomas algorithm. To gain time, we apply it directly as we are building the two matrices. Once `matrix` is built, we won't need to rebuild it ever again. Only `matrix2` needs to be recreated since its values depends on the time step we are at.

3.3.4 Printer

This class needs to be given a solved scheme and will take care of anything that is related to printing the results. I have not put that in the `PDESolver` class to respect the Single Responsibility Principle. The showing of the results can be done through the terminal or through Gnuplot with commands printed in a .txt file. I could have created a batch file to use Gnuplot directly, but the assignment requires that the code works with any computer. And since gnuplot might not be installed, this is a problem. If that is the case, the class creates .dat files. They can be used to plot the graphs using whatever the user wish to use, like Excel for example.

3.4 Exceptions

The program has some exceptions to help the users understand what went wrong. It will:

- Make sure the parameters make sense (the wall must have a length superior to zero, Δx and Δt must be bigger than zero, and so on)
- Warn the user if a scheme is unstable with the parameters given (but not for the Richardson scheme, as it is always unstable...)
- Make sure a scheme has been solved before printing it or using it in some other way (like for creating an error matrix for example)

4 Results and discussion

In this section, we will have a look at the results we got for each scheme and compare them to what could have been expected in theory. If not mentioned otherwise, all these graphs were made using $\Delta t = 0.01$, $\Delta x = 0.05$ and $D = 0.1$.

4.1 The analytical solution

Here is the analytical solution to the heat equation:

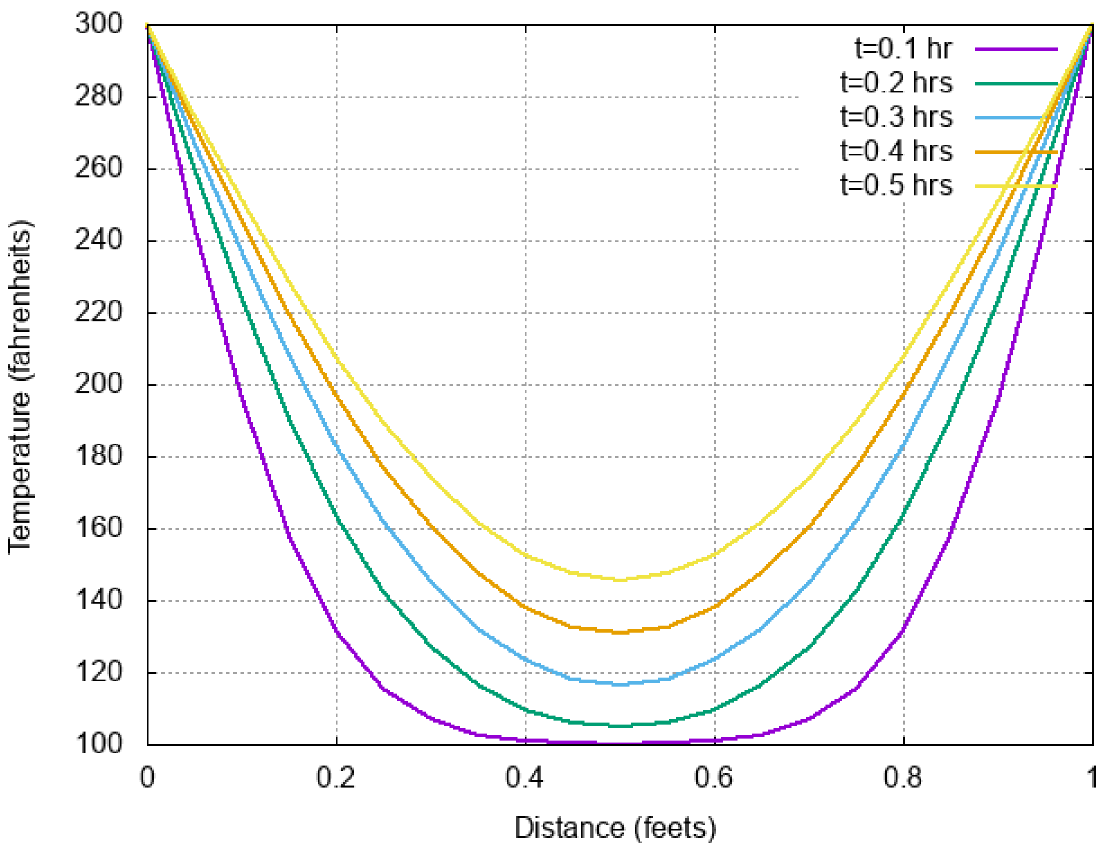


Figure 4.1: Analytical Solution for t=0.1, 0.2, 0.3, 0.4 and 0.5 hours

Nothing surprising here, the heat slowly makes its way towards the middle of the wall and the temperatures slowly rises to T_{Sur}. This graph will be useful to understand the behavior of the first scheme we will analyze: Richardson.

4.2 Richardson

This is the graph we get when comparing the Richardson scheme to the analytical solution:

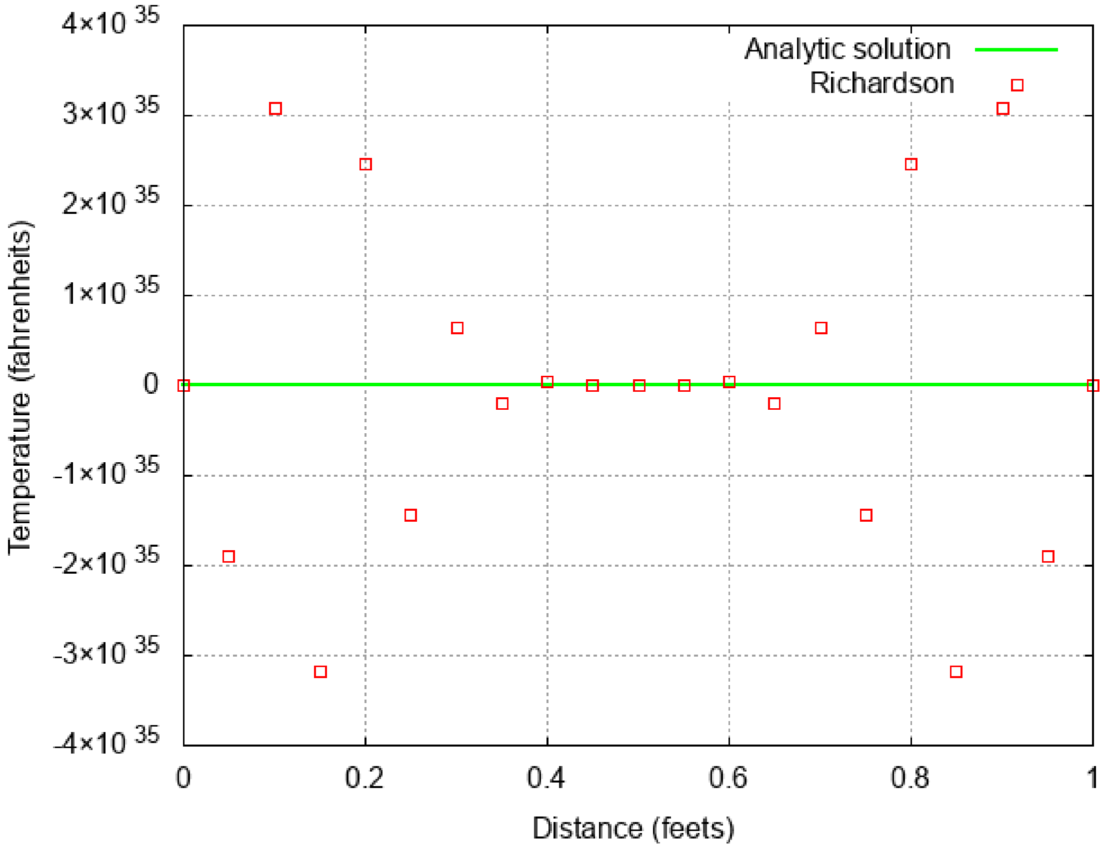


Figure 4.2: Richardson scheme after 6 minutes

The Richardson scheme is an unstable one, meaning that the errors are not bounded and will grow through time, which is clearly the case in the figure (4.2): the analytical solution, displayed in green, cannot even be seen due to how far off the Richardson scheme is, even though we are only at the tenth time step.

We can also observe an oscillation, typical of any unstable scheme. One might wonder what happens in the middle of the graph (from 0.4 fts to 0.6 fts), Richardson seems to be more accurate, all of sudden. If we take a look at (4.1), we can see that the analytical solution is almost constant in this section and this explains the increased accuracy: any scheme is way more accurate if the growth of the function to approximate is slow, which was why in the informative assessment [10], the unstable scheme was so close to the analytical solution that, in this case, was mostly constant. Here is what happens 40 time steps later.

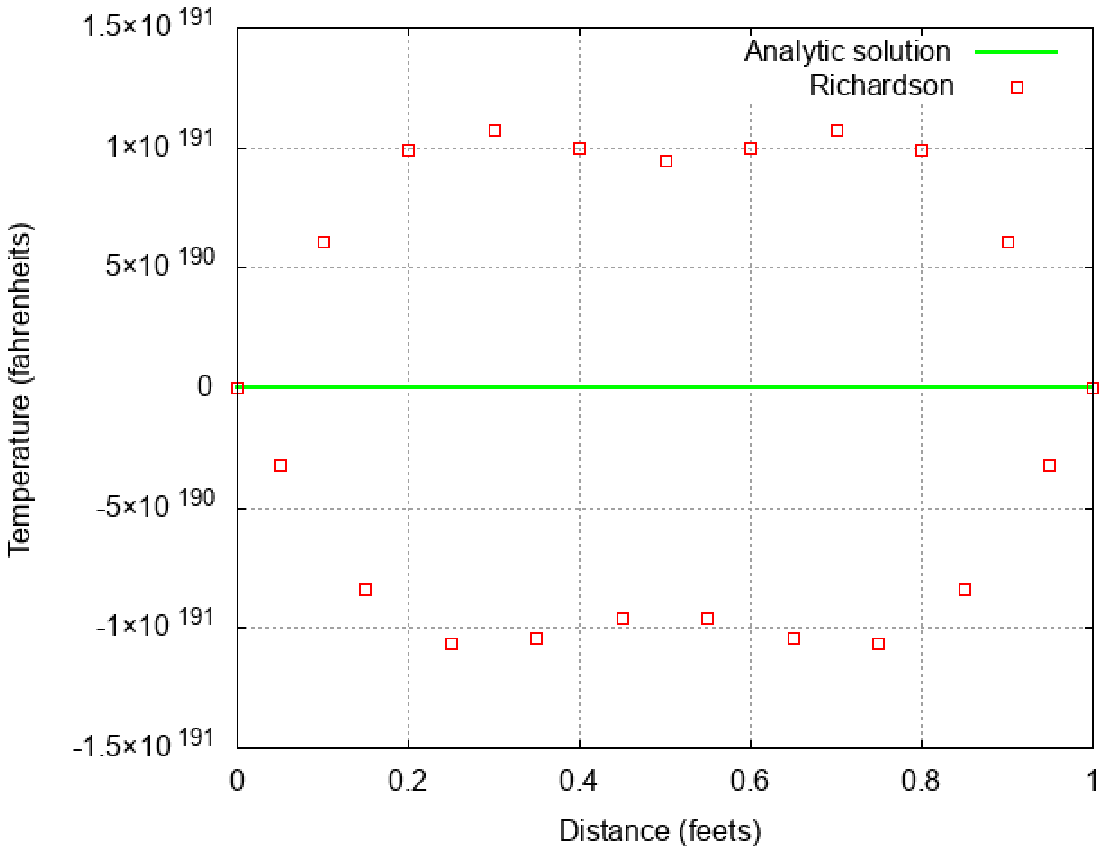


Figure 4.3: Richardson scheme after 30 minutes

As time goes by, the heat reaches the middle of the wall as we can see on (4.1), making the solution grow faster in this section. As a result, the increased accuracy we observed previously is now almost completely gone. The errors have significantly increased (by more than 5 times). If it was not obvious enough before, we can be sure that the errors are unbounded and that the Richardson scheme is unstable and should never be used.

4.3 Comparing the other schemes

The other schemes are all really close to the solution: displaying their graph would be useless, as no difference can be seen that way. Instead, we will analyze the graph of the two-norm applied to the error matrix of the remaining schemes.

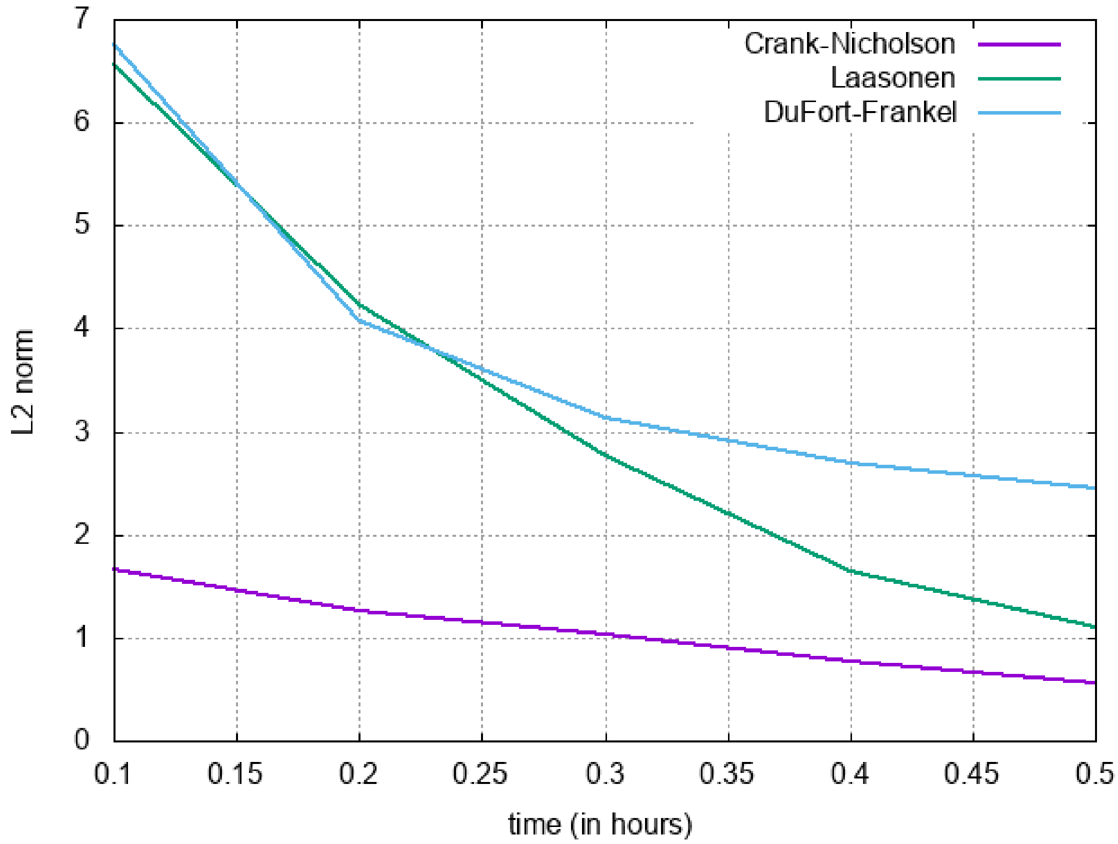


Figure 4.4: Two-norm of the errors matrix for Crank-Nicholson, Laasonen and DuFort-Frankel

4.3.1 General remarks

All the numerical schemes are more and more accurate as time goes by. This was expected since we are dealing with stable methods. Therefore "early errors (due to the imprecision of the method or to an initial value that is slightly incorrect) are damped in as the computations proceed" [11]. The Crank-Nicholson scheme is the best in terms of accuracy for our problem, followed by Laasonen and finally DuFort-Frankel. Let's go over each of these schemes and try to understand why this is so.

4.3.2 DuFort-Frankel

The Dufort-Frankel method is really impressive for an explicit scheme. Not only does it achieve to be unconditionally stable, but it is second-order accurate as well. This would mean that this scheme is just as good as the Crank-Nicholson one. But... as we look at this graph, it is not the case at all. So what happened?

My first idea was that it was due to the starter solution: FTCS, which has the same truncation errors and therefore accuracy as Laasonen and, thus, would explain why DuFort-Frankel and Laasonen go neck and neck with each other here. I tried using Crank-Nicholson as a starter solution instead, to see if it would improve the accuracy:

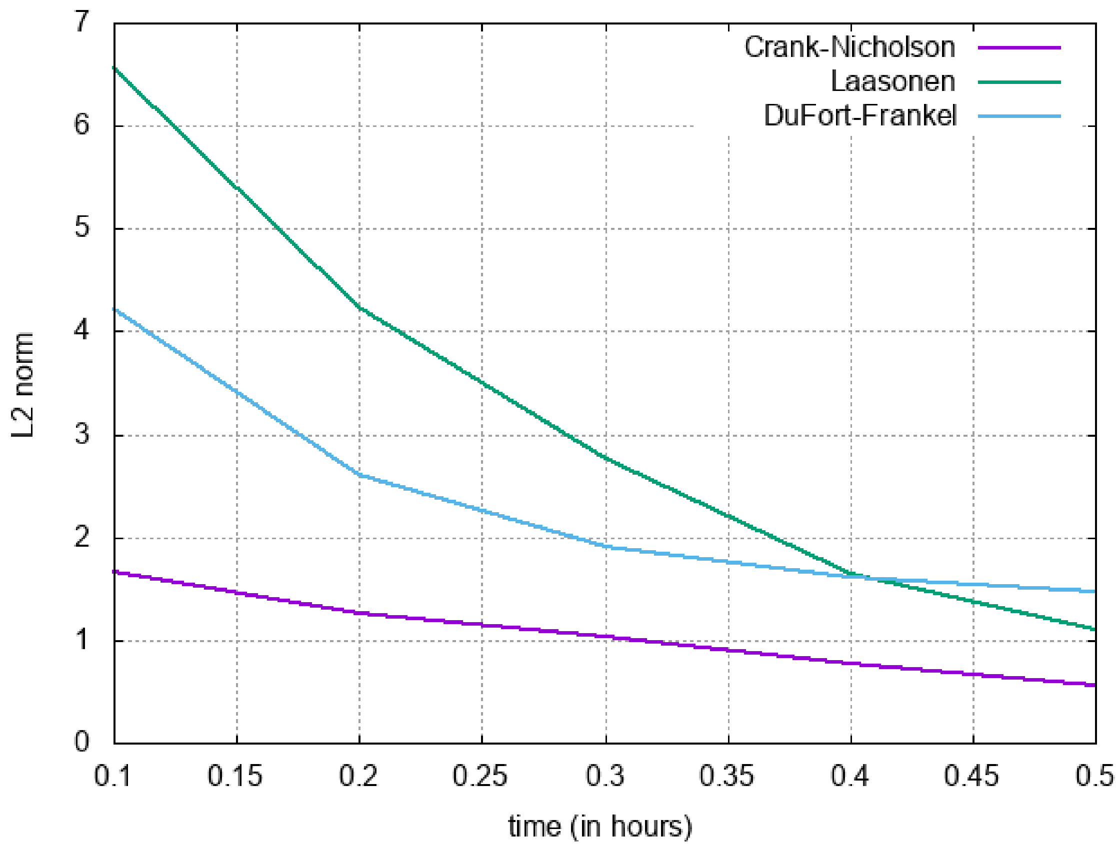


Figure 4.5: Two-norm of the errors matrix for Crank-Nicholson, Laasonen and DuFort-Frankel using Crank-Nicholson as a starter solution

We can see a clear improvement for the early time steps when we compare (4.5) to (4.4), showing just how important the influence of the starter solution is. But still, DuFort-Frankel gets beaten by Laasonen at the very end, despite the lead given to it by the accuracy of Crank-Nicholson. So, this has nothing to do with the starter solution, this is a problem with the scheme itself.

It turns out that the amazing accuracy of DuFort-Frankel can only be achieved under some conditions. In deed, this scheme is not unconditionally consistent. We often assume that the scheme being stable is enough, but here is a perfect of example of why it is not. As shown by C.A.J. Fletcher [12], in his consistency analysis for the scheme, $\Delta t/\Delta x$ must tend to zero, which is to say that Δx must be much bigger than Δt, which is not really the case with our parameters and explains the accuracy not being as good as what we could have expected, since there is an added error of $O((\Delta t/\Delta x)^2)$ to our truncation errors [5]. But here, if we decrease Δt, it should be much better as we can see on this graph:

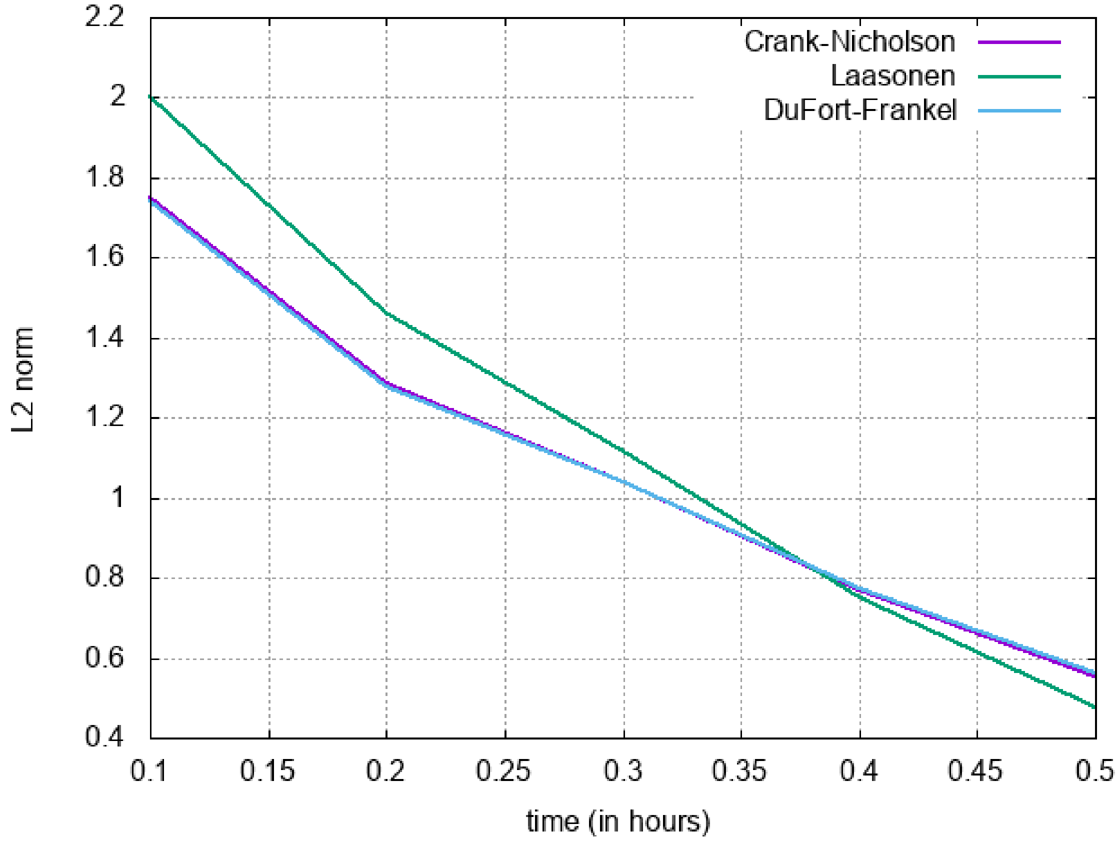


Figure 4.6: Two-norm of the errors matrix for Crank-Nicholson, Laasonen and DuFort-Frankel with $\Delta t = 0.001$

We finally get what we expected out of DuFort-Frankel. Its truncation errors are the same as Crank-Nicholson, which is why both schemes have pretty much the same accuracy here, expect DuFort-Frankel is much faster, making it a clear winner.
But something really surprising happened: Laasonen beat everyone, while it has the worst truncation errors of the three.

4.3.3 Crank-Nicholson

There would not have been much to say about Crank-Nicholson before the graph (4.6). On (4.4), Crank-Nicholson is the most accurate. With DuFort-Frankel being sabotaged by its consistency condition, it is the best scheme in terms of truncation error with a second order accuracy and thus, gives the best results for our problem. So what happened on the graph (4.6)?

After some research, I found out that Crank-Nicholson is not as perfect as we first might think it is, proving once again that there is no perfect scheme to always use. C.A.J. Fletcher talks about some of its drawbacks in the first volume of *Computational Techniques for Fluid Dynamics* [13], noting that the scheme is ”on the boundary of the unconditionally stable regime”, which, in particular, makes it not very efficient for solutions where different parts reach their steady-state at different rates. This leads me to think that, while Crank-Nicholson's initial error is way smaller than for the Laasonen scheme since it has a better truncation error, Laasonen manages to catch up because it is more stable, managing to reduce any errors much faster than Crank-Nicholson, allowing it to be more accurate if enough time steps have passed.

I really was not sure of this explanation and I wanted to check this idea. If I am right, then this would mean that in our initial problem with $\Delta t = 0.01$, Crank-Nicholson would get beat by Laasonen at some point. Here is what happens if we let more time go by:

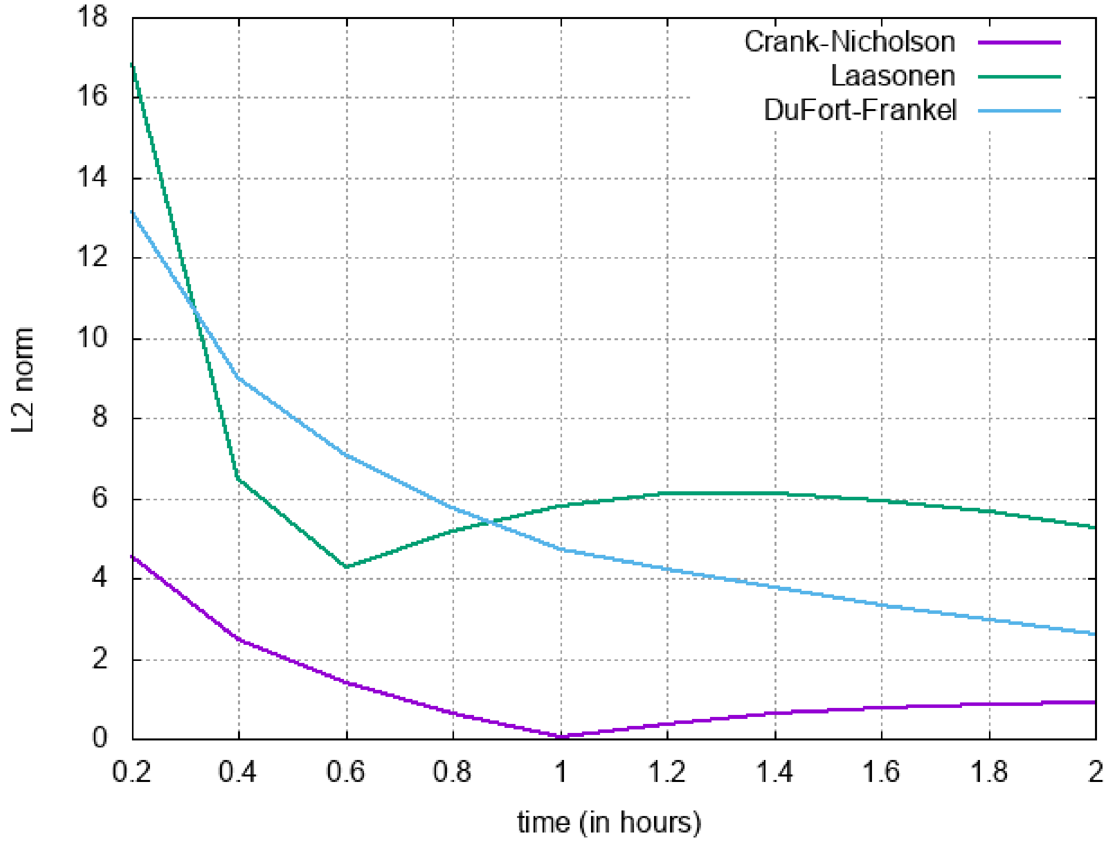


Figure 4.7: Two-norm of the errors matrix for Crank-Nicholson, Laasonen and DuFort-Frankel for $\Delta t = 0.01$ and with more time steps

This is everything but what I would have expected: Laasonen seems to tend towards zero much faster than Crank-Nicholson, but then the errors start growing again for some time and the same happens to Crank-Nicholson a bit later. I was quite shocked by this result, and decided to see if the same event would occur with $\Delta t = 0.001$:

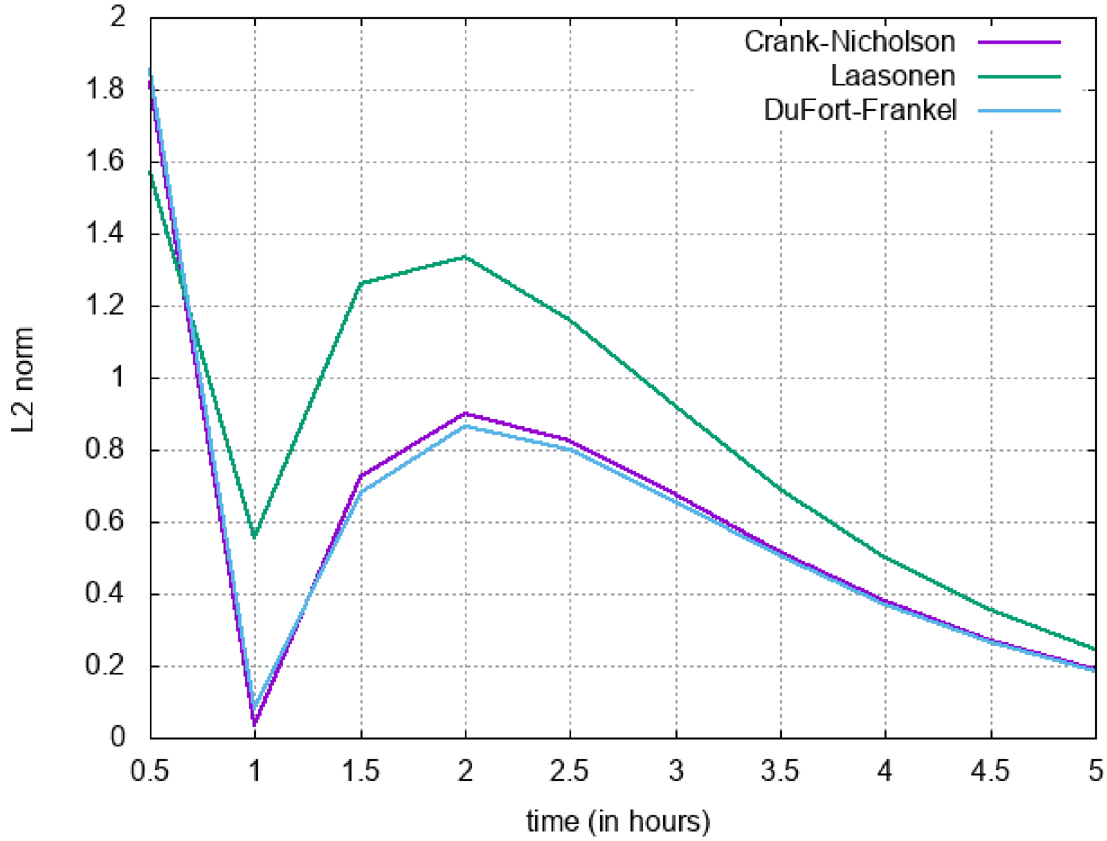


Figure 4.8: Two-norm of the errors matrix for Crank-Nicholson, Laasonen and DuFort-Frankel for $\Delta t = 0.001$ and with more time steps

The same effect can be seen, this time affecting every scheme. It happens around the same time for Crank-Nicholson, but while the errors grow back after 0.6 hours for Laasonen in (4.7), here it grows back after 1 hour, which I can not explain why. And it saddens me to say, but I am not sure what is going on here. I am guessing that this is due to a stability issue, since the error gets amplified, as if the schemes became unstable for a while. But this should not happen as in the stability analysis, we make sure that the ratio between the error at one time step and the next one is smaller than one, meaning that the error should always get smaller. I have not found any book talking about such a sudden increase, so I am pretty sure the error must come from my C++ program, but when I check my results for the initial problem with other students, I get the same values. I do not see why it would get wrong after a while... I thought maybe because of some sort of memory issue? I do store every solution for every time step, which can be quite a lot. But if it was the case, then this increase should happen later with $\Delta t = 0.01$ than $\Delta t = 0.001$, but it actually happens earlier here, so this is not it.

Anyway, as time progresses, we actually get what we could have expected looking at the property of the schemes: the Laasonen scheme has the worst truncation error and is last, then comes DuFort-Frankel and Crank-Nicholson with approximately the same results which is not surprising considering that they have the same accuracy.

4.3.4 Laasonen

For the Laasonen method, we are asked to check what happens as Δt increases:

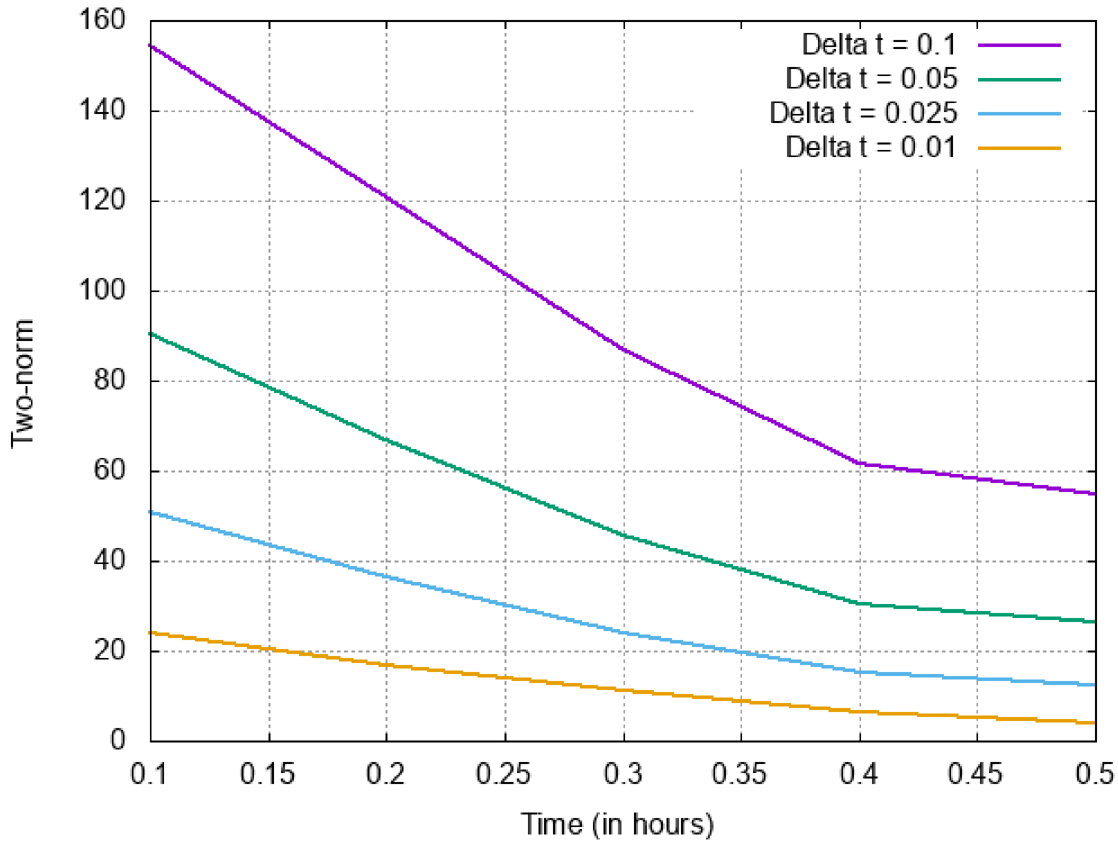


Figure 4.9: Two-norm of the errors matrix for the Laasonen scheme, using various Δts

As Δt increase, so does the truncation error, therefore the scheme is less and less accurate, as we can see. For Laasonen, the truncation in time is of $O(\Delta t)$. So, if we increase Δt by two, so should the truncation error. To check that, we can take a look at the errors for an advanced time step (so that round-off errors and the influence of the initial data are washed out [14]). For t = 0.4, the two-norm of the error matrix is of 15.25 for $\Delta t = 0.025$ and of 30.45 for $\Delta t = 0.05$, confirming what I just said. This is a great way to confirm the theoretical accuracy of a scheme.

Choosing large time step is useful to get further in time without using too much computational power, which is why unconditionally stable methods are so appreciated as we can choose as large of a time step as we want. But we have to keep in mind that the bigger the time step, the worse the truncation error and therefore overall accuracy. It should also be noted that the same can be said for very small time steps: here, the round-off errors would significantly increase [14].

5 Conclusion

In this report, we have seen just how complex it can be to choose the right numerical scheme and just how important a proper analysis is. We have seen why an unstable scheme such as the Richardson one should never be used, that only proving a method's stability is not enough and that consistency should never just be assumed, with the surprising results we got using DuFort-Frankel. We also witnessed how increasing the mesh size changes the truncation error of a scheme and thus its accuracy using the Laasonen method. That proved how useful having an unconditionally stable scheme is, since we can increase the time step and go further in time that way with a decreased cost in computation, as long as that bigger truncation error does not bother us.

Assuming my results are correct, the Crank-Nicholson scheme seems to be the best to use to solve the heat equation overall. But if DuFort-Frankel's consistency conditions are well respected, then it should be used instead as we get the same accuracy, more or less, but with a computational cost that is significantly smaller since it is an explicit scheme. Finally, in some cases, it seems that Laasonen can be better than both of these schemes but only on really short windows.

References

[1] MathWorld, *Partial Differential Equation*. Available at:
`http://mathworld.wolfram.com/PartialDifferentialEquation.html`
(accessed 10/31/2017)

[2] K.A. Hoffmann and S.T. Chiang, *Computational Fluid Dynamics*, Fourth Edition, Vol. I, Engineering Education System Books, pp. 1-2

[3] K.A. Hoffmann and S.T. Chiang, *Computational Fluid Dynamics*, Fourth Edition, Vol. I, Engineering Education System Books, pp. 29 to 34

[4] G. de Vahl Davis, (1986). *Numerical Methods in Engineering and Science,* pp. 251, 254, 256, 257, 259

[5] K.A. Hoffmann and S.T. Chiang, *Computational Fluid Dynamics*, Fourth Edition, Vol. I, Engineering Education System Books, pp. 64 to 67

[6] C.A.J. Fletcher, (1991), *Computational Techniques for Fluid Dynamics*, Volume 1, p. 227

[7] S. Scott Collis, (2005). *An Introduction to Numerical Analysis for Computational Fluid Mechanics*, pp. 58-59

[8] Wikipedia, *Tridiagonal matrix algorithm*. Available at:
`https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm`
(accessed 10/31/2017)

[9] Wikipedia, *LU decomposition*. Available at:
`https://en.wikipedia.org/wiki/LU_decomposition`
(accessed 10/31/2017)

[10] V. Ostertag, (2017). *Computational Methods: Wave equation*, p. 20

[11] G. Wheatley, (2004). *Applied Numerical Analysis*, Seventh edition, p. 335

[12] C.A.J. Fletcher, (1991), *Computational Techniques for Fluid Dynamics*, Volume 1, pp. 220-221

[13] C.A.J. Fletcher, (1991), *Computational Techniques for Fluid Dynamics*, Volume 1, p. 229

[14] K.A. Hoffmann and S.T. Chiang, *Computational Fluid Dynamics*, Fourth Edition, Vol. I, Engineering Education System Books, p. 74

[15] P. Johnson, *Von Neumann Stability Analysis*. Available at:
`http://www.maths.manchester.ac.uk/~pjohnson/pages/math65241.html`
(accessed 12/01/2017)

6 Appendices

6.1 Appendix A: Creating the Richardson scheme

We will start by approximating $\delta f/\delta t$. Using the Taylor series, we know that:

$$f(t+\Delta t) = f(t) + \frac{\delta f}{\delta t}\Delta t + \frac{\delta^2 f}{\delta t^2}\frac{\Delta t^2}{2} + O(\Delta t^3) \tag{6.1}$$

$$f(t-\Delta t) = f(t) - \frac{\delta f}{\delta t}\Delta t + \frac{\delta^2 f}{\delta t^2}\frac{\Delta t^2}{2} + O(\Delta t^3) \tag{6.2}$$

By subtracting (6.2) to (6.1) and isolating $\delta f/\delta t$, we get:

$$\frac{\delta f}{\delta t} = \frac{f(t+\Delta t) - f(t-\Delta t)}{2\Delta t} + O(\Delta t^2) \tag{6.3}$$

We now repeat the process to approximate $\delta^2 f/\delta x^2$:

$$f(t+\Delta x) = f(x) + \frac{\delta f}{\delta x}\Delta x + \frac{\delta^2 f}{\delta x^2}\frac{\Delta x^2}{2} + \frac{\delta^3 f}{\delta x^2}\frac{\Delta x^3}{6} + O(\Delta x^4) \tag{6.4}$$

$$f(t-\Delta x) = f(x) - \frac{\delta f}{\delta x}\Delta x + \frac{\delta^2 f}{\delta x^2}\frac{\Delta x^2}{2} - \frac{\delta^3 f}{\delta x^2}\frac{\Delta x^3}{6} + O(\Delta x^4) \tag{6.5}$$

By adding (6.5) to (6.4) and isolation $\delta^2 f/\delta x^2$, we get:

$$\frac{\delta^2 f}{\delta x^2} = \frac{f(t+\Delta x) - 2f(x) + f(x-\Delta x)}{\Delta x^2} + O(\Delta x^2) \tag{6.6}$$

By replacing $\delta^2 f/\delta x^2$ and $\delta f/\delta t$ in the heat equation, we get the Richardson scheme:

$$\frac{f(t+\Delta t) - f(t-\Delta t)}{2\Delta t} + O(\Delta t^2) = D \cdot \frac{f(t+\Delta x) - 2f(x) + f(x-\Delta x)}{\Delta x^2} + O(\Delta x^2) \tag{6.7}$$

We also get the truncation error of the scheme, that is to say its accuracy, which is of $O(\Delta t^2)+O(\Delta x^2)$

6.2 Appendix B: Richardson's stability analysis

The accuracy of the Richardson scheme has already been discussed in the previous appendix, we will now focus on its stability. To analyze it, we will use the method seen in class and, towards the end, I supported my reflexion with Dr. Johnson's work [15] as I was not able to see how to get the ratio we usually calculated in class.
Assume F is the solution to the Richardson scheme with some errors r, that is to say that:

$$f_i^n = F_i^n + r_i^n \tag{6.8}$$

If we introduce (6.8) in the Richardson's scheme FDE, only the errors will remain:

$$\frac{r_i^{n+1} - r_i^{n-1}}{2\Delta t} = \frac{r_{i+1}^n - 2r_i^n + r_{i-1}^n}{\Delta x^2} \tag{6.9}$$

Using the Fourier series, we can write that:

$$r_i^n = \sum_{-\infty}^{+\infty} g^n(k)e^{ikx_i} \tag{6.10}$$

By introducing (6.10) in (6.9), we get:

$$\frac{\sum_{-\infty}^{+\infty} g^{n+1}(k)e^{ikx_i} - \sum_{-\infty}^{+\infty} g^{n-1}(k)e^{ikx_i}}{2\Delta t} = D \cdot \frac{\sum_{-\infty}^{+\infty} g^n(k)e^{ikx_{i+1}} - 2\sum_{-\infty}^{+\infty} g^n(k)e^{ikx_i} + \sum_{-\infty}^{+\infty} g^n(k)e^{ikx_{i-1}}}{\Delta x^2} \tag{6.11}$$

By replacing x_{i+1} by $x_i + \Delta x$ and x_{i-1} by $x_i - \Delta x$, we can get:

$$\frac{\sum_{-\infty}^{+\infty} g^{n+1} - g^{n-1}}{2\Delta t} = D \cdot \frac{\sum_{-\infty}^{+\infty} g^n(e^{ik\Delta x} - 2 + e^{-ik\Delta x})}{\Delta x^2} \tag{6.12}$$

Since the condition we will get must be true for all the terms of this sum, we can focus our analysis on only one of them:

$$g^{n+1} - g^{n-1} = D\frac{2\Delta t}{\Delta x^2} g^n \cdot (e^{ik\Delta x} - 2 + e^{-ik\Delta x}) \tag{6.13}$$

Now, we will divide (6.13) by g^{n-1}.

$$g^2 - 1 = D\frac{2\Delta t}{\Delta x^2} g^1 \cdot (e^{ik\Delta x} - 2 + e^{-ik\Delta x}) \tag{6.14}$$

Euler's formula states that:

$$cos(x) = \frac{e^{ix} + e^{ix}}{2} \tag{6.15}$$

Using (6.15) in (6.14), we get:

$$g^2 - 1 = D\frac{4\Delta t}{\Delta x^2}g^1 \cdot (cos(k\Delta x) - 1) \tag{6.16}$$

Using the following trigonometry formula:

$$cos(2x) = 1 - 2sin^2(x) \tag{6.17}$$

We get:

$$g^2 + 4g^1\frac{\Delta t}{\Delta x^2}sin(\frac{k\Delta x}{2}) + 1 = 0 \tag{6.18}$$

From which we get that:

$$g^1 + g^2 = -4\frac{\Delta t}{\Delta x^2}sin(\frac{k\Delta x}{2}) \tag{6.19}$$

$$g^1 \cdot g^2 = -1 \tag{6.20}$$

For the scheme to be stable, $| g^2 | \leq 1$ and $| g^1 | \leq 1$. But if $| g^2 | < 1$, then $| g^1 | > 1$. Therefore g^1 must be equal to 1 and, because of (6.20), g^2 must be equal to -1. But if this is the case, then $\frac{\Delta t}{\Delta x^2}$ must be equal to 0, which is impossible. Thus, the Richardson scheme is unconditionally unstable.

6.3 Appendix C: Laasonen and Crank-Nicholson's system of equations

We will start with the Laasonen scheme whose FDE is:

$$\frac{f_i^{n+1} - f_i^n}{\Delta t} = D \cdot \frac{f_{i+1}^{n+1} - 2f_i^{n+1} + f_{i-1}^{n+1}}{\Delta x^2} \tag{6.21}$$

$$f_i^{n+1} - f_i^n = c \cdot (f_{i+1}^{n+1} - 2f_i^{n+1} + f_{i-1}^{n+1}) \tag{6.22}$$

$$-cf_{i+1}^{n+1} + (1 + 2c)f_i^{n+1} - cf_{i-1}^{n+1} = f_i^n \tag{6.23}$$

with $c = \dfrac{D\Delta t}{\Delta x^2}$. For $i = 1$, we get:

$$-cf_2^{n+1} + (1 + 2c)f_1^{n+1} - cf_{i-1}^0 = f_1^n \tag{6.24}$$

Using the boundary condition, this is equivalent to:

$$-cf_2^{n+1} + (1 + 2c)f_1^{n+1} = cT_{sur} + f_1^n \tag{6.25}$$

The same can be done for $i = m - 1$ with m the size of our grid:

$$(1 + 2c)f_m^{n+1} - cf_{m-2}^{n+1} = cT_{sur} + f_m^n \tag{6.26}$$

From this, we know that the Laasonen's system of equations is the following one:

$$\begin{bmatrix} -c & 2c+1 & 0 & 0 & 0 & \dots & 0 \\ 0 & -c & 2c+1 & -c & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -c & 2c+1 & -c & 0 \\ 0 & & \dots & & 0 & -c & 2c+1 \end{bmatrix} \cdot \begin{bmatrix} f_1^{n+1} \\ f_2^{n+1} \\ \vdots \\ f_{i-2}^{n+1} \\ f_{i-1}^{n+1} \end{bmatrix} = \begin{bmatrix} f_1^n + c \cdot t_{Sur} \\ f_2^n \\ \vdots \\ f_{i-2}^n \\ f_{i-1}^n + c \cdot t_{Sur} \end{bmatrix}$$

For the Crank-Nicholson scheme, the process is more or less the same. From its FDE, we get:

$$-cf_{i+1}^{n+1} + (1+2c)f_i^{n+1} - cf_{i-1}^{n+1} = (1-2c)f_i^n + c(f_{i+1}^n + f_{i-1}^n) \tag{6.27}$$

with $c = \dfrac{D\Delta t}{2\Delta x^2}$. Using the boundary conditions just like we have done earlier, we get the following system of equations:

$$\begin{bmatrix} -c & 2c+1 & 0 & 0 & 0 & \dots & 0 \\ 0 & -c & 2c+1 & -c & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -c & 2c+1 & -c & 0 \\ 0 & & \dots & & 0 & -c & 2c+1 \end{bmatrix} \cdot \begin{bmatrix} f_1^{n+1} \\ f_2^{n+1} \\ \vdots \\ f_{i-2}^{n+1} \\ f_{i-1}^{n+1} \end{bmatrix} = \begin{bmatrix} (1-2c) \cdot f_1^n + c \cdot (f_2^n + f_0^n) + c \cdot t_{Sur} \\ (1-2c) \cdot f_2^n + c \cdot (f_3^n + f_1^n) \\ \vdots \\ (1-2c) \cdot f_{i-2}^n + c \cdot (f_{i-1}^n + f_{i-3}^n) \\ (1-2c) \cdot f_{i-1}^n + c \cdot (f_i^n + f_{i-2}^n) + c \cdot t_{Sur} \end{bmatrix}$$

6.4 Appendix D: Source code and Doxygen documentation

```
#include"ExplicitMethod.h"

#ifndef ANALYTICALSOLUTION_H  // include guard
#define ANALYTICALSOLUTION_H

/**
* Analytical Solution to the problem
*
* It is not an explicit scheme but its' behaviour is pretty much the same as
   one, which is
* why it inherits from ExplicitMethod
*/
class AnalyticalSolution : public ExplicitMethod {
private:
     // CLASS MEMBER

     const double PI = 3.141592653589793;    ///< Approximate value of pi
public:
     /**
     * Analytical constructor
     *
     * @see PDESolver (same constructor)
     */
     AnalyticalSolution(Parameters parameters);
     //~AnalyticalSolution();

     /**
     * Implementation of ExplicitMethod's getNextTimeStep
     *
     * @see getNextTimeStep() from ExplicitMethod
     */
     double getNextTimeStep(int i, int t);

     /**
     * Function to get the analytical value of the solution f(t, x) with a
        given precision
     *
     * @param x, value of x
     * @param t, value of t
     * @param precision that we wish to have for the infinite for loop (that
        can't be infinite with a computer...)
     */
     double getAnalyticalValue(double x, double t, int precision);
};

#endif
```

```
#include"ImplicitMethod.h"

#ifndef CRANKNICHOLSON_H // include guard
#define CRANKNICHOLSON_H

/**
* CrankNicholson implicit scheme that we are required to use in this
   assignement
*/
```

```
class CrankNicholson : public ImplicitMethod {
private:
    double c;                                       ///< Value of c in the
        matrix (found during the analysis of the method in the report)
public:
    /**
    * CrankNicholson's constructor
    *
    * @see PDESolver
    */
    CrankNicholson(Parameters parameters);
    //~CrankNicholson();

    /**
    * Get the diagonal value for the CrankNicholson scheme
    *
    * @return value of the diagonal of matrix
    * @see getDiagonalValue() from ImplicitScheme
    */
    double getDiagonalValue();

    /**
    * Get the values out of the diagonal for the CrankNicholson scheme
    *
    * @return value of the points out of the diagonal of matrix
    * @see getOutOfDiagonalValue() from ImplicitScheme
    */
    double getOutOfDiagonalValue();

    /**
    * Get the values of matrix2
    *
    * @return value of matrix2 for t and i
    * @see getBoundaryValue() from ImplicitScheme
    */
    double getMatrix2Values(int t, int i);
};

#endif
```

```
#include"ExplicitMethod.h"

#ifndef DUFORFRANKEL_H  //include guard
#define DUFORTFRANKEL_H

/**
* DuFortFrankel is an explicit scheme that we are required to use in this
   assignement
*/
class DuFortFrankel : public ExplicitMethod {
    public:
        /**
        * DuFortFrankel's constructor
        *
        * @see PDESolver (same constructor)
        */
        DuFortFrankel(Parameters parameters);
        //~DuFortFrankel();
```

```cpp
        /**
        * Implementation of ExplicitMethod's getNextTimeStep
        *
        * @see getNextTimeStep() from ExplicitMethod
        */
        double getNextTimeStep(int i, int t);
};

#endif
```

```cpp
#include"PDESolver.h"

#ifndef EXPLICITMETHOD_H //include guard
#define EXPLICITMETHOD_H

/**
* ExplicitMethod is an abstract class for all of the explicit schemes.
*
* It inherits from PDESolver and contains all of the useful class for an
  explicit scheme.
*/
class ExplicitMethod : public PDESolver {
    public:
        /**
        * ExplicitMethod's constructor
        *
        * @see PDESolver (same constructor)
        */
        ExplicitMethod(Parameters parameters);

        /**
        * Pure virtual function to get the next time step using the
          previous ones we already solved
       *
        * The only difference in each explicit scheme is the way to get the
           next time step, so this will
        * have to be implemented by all the explicit schemes.
        *
        * @param i which space point do we wish to solve
        * @param t at which time step are we
        * @return the value of the temperature for fTable[t][i]
        */
        double virtual getNextTimeStep(int i, int t)=0;
        //~ExplicitMethod();

        /**
        * Solve method to fill up fTable
        *
        * @see solve() function from PDESolver
        */
        void solve();

        /**
        * Get the first time step using the FTCS method
        *
        * Some explicit schemes are using two time steps to solve the next
          one. This is a problem for
```

```
         * t = 1 and we therefore need to use an other explicit method to get all the points for the first
         * time step. In our case, we will be using the FTCS scheme.
         *
         * @see FTCS
         */
         void getFirstStrepFromFTCS();
};

#endif
```

```
#include"ExplicitMethod.h"

#ifndef FTCS_H // include guard
#define FTCS_H

/**
* FTCS (Forward Time, Central Space) is an explicit scheme that was not required to use in the assignement.
*
* I implemented it because we need an explicit scheme that doesn't two time steps to get the
* next one to initialize fTable[1][i] (for all i) for the other explicit schemes.
*/
class FTCS : public ExplicitMethod {
     public:
         /**
         * FTCS constructor
         *
         * @see PDESolver (same constructor)
         */
         FTCS(Parameters parameters);
         //~FTCS();

         /**
         * Implementation of ExplicitMethod's getNextTimeStep
         *
         * @see getNextTimeStep() from ExplicitMethod
         */
         double getNextTimeStep(int i, int t);
};

#endif
```

```
#include"PDESolver.h"

#ifndef IMPLICITMETHOD_H //include guard
#define IMPLICITMETHOD_H

/**
* ImplicitMethod is an abstract class for all of the implicit schemes.
*
* It inherits from PDESolver and contains all of the useful class for an implicit scheme.
```

```
* It has a few new class members that are related to the two matrixes needed in the solving
* of an implicit scheme.
*
* The explicit scheme will solve the equation : matrix * fTable[tWeWantToSolve] = matrix2 using
* the LU decomposition. The value of matrix and matrix2 changes with the scheme we wish to use.
*/
class ImplicitMethod : public PDESolver {
protected:
     // CLASS MEMBERS
     vector< vector<double> > matrix;   ///< Matrix in the left of the equation
     vector< double > matrix2;              ///< Matrix in the right of the equation
public:
     /**
     * ImplicitMethod's constructor
     *
     * @see PDESolver (same constructor)
     */
     ImplicitMethod(Parameters parameters);
     //~ImplicitMethod();

     /**
     * Solve method to fill up fTable
     *
     * @see solve() function from PDESolver
     */
     void solve();

     /**
     * Function that solves the equation matrix * ftable[t+1] = matrix2 once the thomas algorithm has been applied
     */
     void solve_thomas(int t);

     /**
     * Function that creates matrix (3 diagonal matrix)
     */
     void createMatrix();

     /**
     * Function that creates matrix2
     *
     * @param t for which time step we wish to create that matrix
     */
     void createMatrix2(int t);

     /**
     * Pure virtual function that will give the value in the diagonal of matrix
     *
     * The way of creating matrix is always the same (thus the createMatrix() method here),
     * only the value of the matrix changes with the scheme
     *
     * @return value of the diagonal of the matrix
     */
```

```
    double virtual getDiagonalValue() = 0;

    /**
    * Pure virtual that gives the value outside of the diagonal of matrix
    *
    * @return value outside of the diagonal of matrix
    * @see getDiagonalValue() for some additional infos
    */
    double virtual getOutOfDiagonalValue() = 0;

    /**
    * Pure virtual function that gets the values of matrix2
    *
    * @param time step
    * @param space point
    * @return value of matrix2 for i and t
    */
    double virtual getMatrix2Values(int t, int i) = 0;

    /**
    * Method used in the constructor of implicit schemes
    *
    * It gets matrix2 for the first time step ready as well as matrix
    * and its' LU decomposition that will always be the same no matter
    * the time step and time point
    */
    void initialization();
};

#endif
```

```
#include"ImplicitMethod.h"

#ifndef LAASONEN_H  // include guard
#define LAASONEN_H

/**
* Laasonen implicit scheme that we are required to use in this assignement
*/
class Laasonen : public ImplicitMethod {
private:
    double c;                                   ///< Value of c in the
        matrix (found during the analysis of the method in the report)
public:
    /**
    * Laasonen's constructor
    *
    * @see PDESolver (same constructor)
    */
    Laasonen(Parameters parameters);
    //~Laasonen();

    /**
    * Get the diagonal value for the Laasonen scheme
    *
    * @return value of the diagonal of matrix
    * @see getDiagonalValue() from ImplicitScheme
    */
```

```
    double getDiagonalValue();

    /**
    * Get the values out of the diagonal for the Laasonen scheme
    *
    * @return value of the points out of the diagonal of matrix
    * @see getOutOfDiagonalValue() from ImplicitScheme
    */
    double getOutOfDiagonalValue();

    /**
    * Get the values of matrix2
    *
    * @return value of matrix2 for t and i
    * @see getBoundaryValue() from ImplicitScheme
    */
    double getMatrix2Values(int t, int i);
};

#endif
```

```
#ifndef PARAMETERS_H     //include guard
#define PARAMETERS_H

/**
* Parameters is a structure that has all of the useful parameters of the
   problem to solve.
*
* It was created so that if a user uses multiple schemes, he wouldn't have to
    write each parameters
* again and again which is quite frustrating. Here, he'll just define them
   once.
*/
struct Parameters
{
    double D;                               ///< Diffusivity (in ft^2/hr)
    double deltaT;                          ///< deltaT (in hrs)
    double deltaX;                          ///< deltaX (in ft)
    double tIn;                                  ///< Temperature of the
       inside of the wall
    double tSur;                            ///< Temperature of the outside
       of the wall
    double L;                               ///< Size of the wall
    int numberOfTimePoints;                 ///< Number of points in time we
        will need to solve
    int numberOfSpacePoints;           ///< Number of points in our grid
       that we will try to solve
};

#endif
```

```
#include<vector>
#include"Parameters.h"


using namespace std;
```

```cpp

#ifndef PDESOLVER_H      // include guard
#define PDESOLVER_H

/**
* PDESolver is an abstract class that has all the required functions and members to solve the
* partial differential equations (PDE) from the subject. It will be the base for all the schemes.
*
* It provides a constructor that will be used by all of its' derived class to get all the
* variables needed for the solving of the equation.
* It also provides some useful functions that will often be used by its' derived class.
*/
class PDESolver {
     protected:
          // CLASS MEMBERS
          Parameters parameters;                       ///< Structure Parameters, with all of the problem's parameters, to make it easier on the user to write when he uses the  same parameters again and again
          vector< vector<double> > fTable;        ///< Vector of vectors that contains all the solutions f : fTable[1][10] is the temperature at the 1st time point of the 10th space point
     public:
          /**
          * PDESolver's constructor
          *
          * By providing all the wanted parameters, we can set up the solving of the equation.
          * This function also initialize fTable with the solutions of the time step 0 that we
          * already know and will always be the same, no matter the scheme used.
          * You probably noticed that we don't ask for numberOfSpacePoints, that is because we
          * can get it using the other parameters.
          *
          * This function also throws exceptions if the user's value can not be used
          *
          * @param D the diffusivity (in ft^2/hr)
          * @param deltaT (in hrs)
          * @param deltaX (in ft)
          * @param tIn the temperature inside of the wall
          * @param tSur the temperature outside of the wall
          * @param L the size of the wall
          * @param numberOfTimePoints number of points in time we wish to solve
          */
          PDESolver(Parameters parameters);

          //~PDESolver();

          /**
          * Function that checks if we are at an edge of the wall.
          *
          * If it's the case, it will also put the right value for that space
```

```
        point at that time step in fTable
    * (which is the boundary condition, that is to say tSur)
    *
    * @param i space point that needs to be checked
    * @param t timepoint that we are at (useful to fill up fTable if needed)
    * @return true if we are at a bundary and false otherwise
    */
    bool checkBundaries(int i, int t);

    /**
    * Pure virtual function that solves all the points that we desire and put their value in fTable
    *
    * Every type of scheme (implicit and explicit) will solve our equation, but they have different
    * ways to do it, as we will see. Therefore, this is a pure virtual class.
    */
    void virtual solve()=0;

    /**
    * Function that creates a vector with the L2 norm for each time step in comparison of an other PDESolver
    *
    * Mostly used with the analytic solution, to get the errors of the scheme
    *
    * @param pdeSolver to do the L2 norm with
    * @return vector with the values of the norm for each time step of our scheme
    */
    vector< double > L2NormWith(PDESolver* pdeSolver);

    /**
    * Getter for fTable (useful in the Printer class)
    *
    * @return fTable
    */
    vector< vector<double> > getFTable();

    /**
    * Getter for Parameters
    *
    * @return Parameters
    */
    Parameters getParameters();

    /**
    * Setter for Parameters
    */
    void setParameters(Parameters parameters);

    /**
    * Function that gives the name of the class (scheme) being used
    *
    * @return the name of the class
    */
    string getClassName();
};
```

```

#endif
```

```
#include<vector>
#include"Parameters.h"
#include"PDESolver.h"

#ifndef PRINTER_H   // include guard
#define PRINTER_H

using namespace std;

/**
* Printer class used to create all the datFiles as well as the gnuplot
    commands to do to get the graphs from the report
*/
class Printer{
private:
     PDESolver* pdeSolver;                         ///< PDESolver we wish to print
     Parameters parameters;                        ///< Parameters of the PDESolver
     vector< vector<double> > fTable;    ///< fTable of the PDESolver
     string className;                             ///< class name of the PDESolver
          (useful for naming the files)
public:
     /**
     * Printer's constructor
     *
     * @param pdeSolver: scheme for which we would like to print something
     * @param fileName: fileName to use
     */
     Printer(PDESolver* pdeSolver);

     /**
     * Setter for PDESolver (also changes the parameters using the one from
         the new pdeSolver
     *
     * @param pdeSolver: PDESolver* to print
     */
     void setPdeSolver(PDESolver* pdeSolver);

     /**
     * Function that prints the value in console
     *
     * @param t: time step for which we want to see the values
     */
     void printInConsole(int t);

     /**
     * Function that creates a datFile for a given time step (found in the
         datFiles repo)
     *
     * @param t: time step for which we want to get a datFile
     */
     void createDatFileForT(int t);

     /**
     * Function to delete the .txt file with the commands (inside the repo of
          the code)
```

```
    */
    void deleteGnuplotCommands();

    /**
    * Function that add to the command.txt (or create it) the commands to
       plot the scheme at the time step t using gnuplot
    *
    * @param t time step for which we wish to plot a graph
    */
    void gnuplotForT(int t);

    /**
    * Function that add to the command.txt (or create it) the commands to
       plot the L2-norm of the difference between two schemes
    *
    * @param PDESolver* that we want to do the L2-norm with
    */
    void gnuplotErrorsCompareTo(PDESolver* pdeSolver2);

    /**
    * Function that add to the command.txt (or create it) the commands to
       plot 2 schemes on the same graph in order to compare them
    *
    * @param t time step for which we wish to plot the graph
    * @param PDESolver* that we want to plot with our scheme
    */
    void gnuplotForTCompareTo(int t, PDESolver* pdeSolver2);

    /**
    * Print the norm (L2) of the errors in a datFile for given time steps
    *
    * @param ts: time steps for which we will have a look at the errors
    * @param analyticalSolution: to compare with our scheme
    */
    void datFileErrorsComparedTo(vector<int> ts, PDESolver* pdeSolver2);
};

#endif
```

```
#include"ExplicitMethod.h"

#ifndef RICHARDSON_H     //include guard
#define RICHARDSON_H

/**
* Richardson is an explicit scheme that we are required to use in this
   assignement
*/
class Richardson : public ExplicitMethod {
public:
    /**
    * Richardson's constructor
    *
    * @see PDESolver (same constructor)
    */
    Richardson(Parameters parameters);
    //~Richardson();

```

```
	/**
	 * Implementation of ExplicitMethod's getNextTimeStep
	 *
	 * @see getNextTimeStep() from ExplicitMethod
	 */
	double getNextTimeStep(int i, int t);
};

#endif
```

```
#include"AnalyticalSolution.h"
#include"math.h"
#include<iostream>

// CONSTRUCTOR
AnalyticalSolution::AnalyticalSolution(Parameters parameters): ExplicitMethod(parameters) {

}

// Return the analytical value for x and t
double AnalyticalSolution::getAnalyticalValue(double x, double t, int precision) {
	double sum = 0;
	// The sum cannot be infinite like the analytical solution would want it to be. We have to stop at a number (precision)
	for (int m = 1; m < precision; m++) {
		sum += exp(-parameters.D*pow(m*PI / parameters.L, 2)*t) * ((1 - pow(-1, m)) / (m*PI)) * sin(m*PI*x / parameters.L);
	}
	return parameters.tSur + 2 * (parameters.tIn - parameters.tSur) * sum;		// Simply applying what's given in the subject of the assignement
}

// Get the next time step using the analytical solution given in the subject
double AnalyticalSolution::getNextTimeStep(int i, int t) {
	return getAnalyticalValue(i*parameters.deltaX, t*parameters.deltaT, 1000);
}
```

```
#include"CrankNicholson.h"
#include<iostream>

// CONSTRUCTOR
CrankNicholson::CrankNicholson(Parameters parameters) :
	ImplicitMethod(parameters) {
	c = parameters.D*parameters.deltaT / (2 * parameters.deltaX*parameters.deltaX);
	initialization();
}

// Gets the values of matrix2 for CrankNicholson (cf analysis in report)
double CrankNicholson::getMatrix2Values(int t, int i) {
	// Boundary value
	if (i == parameters.numberOfSpacePoints - 2 || i == 0) {
```

```
            return (1 - 2 * c)*fTable[t][i + 1] + c*fTable[t][i + 2] + c*fTable
                [t][i] + c*parameters.tSur;
        }
        // Other values
        else {
            return (1 - 2 * c)*fTable[t][i + 1] + c*fTable[t][i + 2] + c*fTable
                [t][i];
        }
}

// Gets the diagonal values of matrix for CrankNicholson (cf analysis in
    report)
double CrankNicholson::getDiagonalValue() {
        return (2 * c + 1);
}

// Gets the out of diagonal values of matrix for CrankNicholson (cf analysis
    in report)
double CrankNicholson::getOutOfDiagonalValue() {
        return -c;
}
```

```
#include"DuFortFrankel.h"
#include<iostream>
#include<math.h>

// CONSTRUCTOR
DuFortFrankel::DuFortFrankel(Parameters parameters): ExplicitMethod(
    parameters) {
        getFirstStrepFromFTCS(); // This scheme needs to have 2 time step to get
             the next one. Therefore we must get the first time step from an
            other scheme (FTCS here)
}

// Get the next time step using DuFortFrankel's equation
double DuFortFrankel::getNextTimeStep(int i, int t) {
        // If t > 1, we use the equation. Otherwise, we must use the result from
             FTCS as discussed in the comment of the constructor
        if (t > 1) {
            return ((1 - (2 * parameters.D*parameters.deltaT / pow(parameters.
                deltaX, 2))) * fTable[t - 2][i] +
                  (2 * parameters.D*parameters.deltaT / pow(parameters.deltaX,
                      2)) * (fTable[t - 1][i + 1] + fTable[t - 1][i - 1])) *
                  (1 / (1 + (2 * parameters.D*parameters.deltaT / pow(parameters
                      .deltaX, 2))));
        }
        else {
            return fTable[t][i];
        }
}
```

```
#include"ExplicitMethod.h"
#include"AnalyticalSolution.h"
#include"CrankNicholson.h"
#include"FTCS.h"
#include<iostream>
```

```

// Constructor (using PDESolver's constructor)
ExplicitMethod::ExplicitMethod(Parameters parameters): PDESolver(parameters) {
}

// Function to fill up fTable for all the time steps we want
void ExplicitMethod::solve() {
    // For each space point of each time step, we check if we are at a bundary (and use the bundary conditions if so)
    // If we are not, we get the value of the temperature for each point using the equation of whatever scheme we are using
    for (int t = 0; t <= parameters.numberOfTimePoints; t++) {
        for (int i = 0; i <= parameters.numberOfSpacePoints; i++) {
            if (!checkBundaries(i, t)) {
                fTable[t][i] = getNextTimeStep(i, t);
            }
        }
    }
}

// Fill up the first time step with the FTCS scheme
void ExplicitMethod::getFirstStrepFromFTCS() {
    // Create FTCS, solve for only 1 point, copy fTable[1]
    int temp = parameters.numberOfTimePoints;
    parameters.numberOfTimePoints = 1;
    FTCS ftcs = FTCS(parameters);
    parameters.numberOfTimePoints = temp;
    ftcs.solve();
    fTable[1] = ftcs.getFTable()[1];
}
```

```
#include"FTCS.h"
#include<iostream>

// CONSTRUCTOR
FTCS::FTCS(Parameters parameters): ExplicitMethod(parameters) {
    try {
        if (parameters.D*parameters.deltaT / (parameters.deltaX*parameters.deltaX) > 0.5) {
            throw domain_error("FTCS will be unstable with these parameters");
        }
    }
    catch (domain_error &error) {
        cout << "WARNING:" << endl;
        cout << error.what() << endl;
    }
}

// Get the next time step using FTCS's equation
double FTCS::getNextTimeStep(int i, int t) {
    return fTable[t - 1][i] + (parameters.D*parameters.deltaT / (parameters.deltaX*parameters.deltaX)) * (fTable[t - 1][i + 1] - 2 * fTable[t - 1][i] + fTable[t - 1][i - 1]);
}
```

```
#include"ImplicitMethod.h"
#include<iostream>

// CONSTRUCTOR
ImplicitMethod::ImplicitMethod(Parameters parameters): PDESolver(parameters),
     matrix(parameters.numberOfSpacePoints - 1, vector<double>(parameters.
    numberOfSpacePoints - 1)), matrix2(parameters.numberOfSpacePoints - 1) {
}

// Initialization method used in all implicit schemes' constructors
void ImplicitMethod::initialization() {
     // Get time step 0, needed to create matrix2
     for (int t = 0; t <= parameters.numberOfTimePoints; t++) {
          for (int i = 0; i <= parameters.numberOfSpacePoints; i++) {
               checkBundaries(i, t);
          }
     }
     // Creation of both matrix and applying the thomas algorithm right away
     createMatrix();
     createMatrix2(0);
}

// Function to fill up fTable for all the time steps we want
void ImplicitMethod::solve() {
     // For each time step, we solve the new equation matrix * fTable[t] =
         matrix2 to fill up fTable and then get the new matrix2 ready. (
         matrix1 doesn't change)
     for (int t = 1; t <= parameters.numberOfTimePoints; t++) {
          solve_thomas(t);
          createMatrix2(t);
     }
}

// Creates matrix as seen in the analysis (cf report) and applying the thomas
     algorithm directly on each value
void ImplicitMethod::createMatrix() {
     // Only zeros except at i-1, i and i+1 (3-diagonal matrix). For these
         points, the value change with the scheme we are using
     double diag = getDiagonalValue();
     double nonDiag = getOutOfDiagonalValue();
     for (int i = 0; i < parameters.numberOfSpacePoints - 1; i++) {
          if (i == 0) {
               matrix[i][i+1] = nonDiag / diag;
          }
          else if (i != parameters.numberOfSpacePoints - 2) {
               matrix[i][i+1] = nonDiag / (diag - nonDiag * matrix[i - 1][i])
                   ;
          }
          matrix[i][i] = 1;
     }
}

// Creates matrix2 as seen in the analysis (cf report) and applying the
    thomas algorithm directly on each value
void ImplicitMethod::createMatrix2(int t) {
     double diag = getDiagonalValue();
     double nonDiag = getOutOfDiagonalValue();
     for (int i = 0; i < parameters.numberOfSpacePoints - 1; i++) {
          if (i == 0) {
```

```cpp
                matrix2[i] = getMatrix2Values(t, i) / diag;
            }else {
                matrix2[i] = (getMatrix2Values(t, i) - nonDiag*matrix2[i - 1]) / (diag - nonDiag*matrix[i - 1][i]);
            }
        }
}

// Getting the value for fTable[t+1] by solving the different equations
void ImplicitMethod::solve_thomas(int t) {
    for (int i = parameters.numberOfSpacePoints; i >= 0; i--) {
        if(!checkBundaries(i, t)) {
            if (i-1 == parameters.numberOfSpacePoints - 2) {
                fTable[t][i] = matrix2[i - 1];
            }
            else {
                fTable[t][i] = matrix2[i - 1] - matrix[i - 1][i] * fTable[t][i + 1];
            }
        }
    }
}
```

```cpp
#include"Laasonen.h"
#include<iostream>

// CONSTRUCTOR
Laasonen::Laasonen(Parameters parameters) :
    ImplicitMethod(parameters) {
    c = parameters.D*parameters.deltaT / (parameters.deltaX*parameters.deltaX);
    initialization();
}

// Gets the values of matrix2 for Laasonen (cf analysis in report)
double Laasonen::getMatrix2Values(int t, int i) {
    // Boundary value
    if (i == parameters.numberOfSpacePoints - 2 || i == 0) {
        return fTable[t][i + 1] + c*parameters.tSur;
    }
    // Other values
    else {
        return fTable[t][i + 1];
    }
}

// Gets the diagonal values of matrix for Laasonen (cf analysis in report)
double Laasonen::getDiagonalValue() {
    return (2 * c + 1);
}

// Gets the out of diagonal values of matrix for Laasonen (cf analysis in report)
double Laasonen::getOutOfDiagonalValue() {
    return -c;
}
```

```cpp
#include"PDESolver.h"
#include"Parameters.h"
#include"DuFortFrankel.h"
#include"Richardson.h"
#include"Laasonen.h"
#include"FTCS.h"
#include"AnalyticalSolution.h"
#include"CrankNicholson.h"
#include"printer.h"
#include<vector>
#include<iostream>

int main() {
    // values given in the subject
    Parameters parameters;
    parameters.D = 0.1;
    parameters.L = 1;
    parameters.tIn = 100;
    parameters.tSur = 300;
    parameters.deltaX = 0.05;
    parameters.deltaT = 0.01;
    parameters.numberOfTimePoints = (int)(0.5/parameters.deltaT);
    parameters.numberOfSpacePoints = (int)(parameters.L / parameters.deltaX)
        ;

    // creating all the schemes
    AnalyticalSolution analytical = AnalyticalSolution(parameters);
    CrankNicholson crank = CrankNicholson(parameters);
    FTCS ftcs = FTCS(parameters);
    DuFortFrankel dufort = DuFortFrankel(parameters);
    Laasonen laasonen = Laasonen(parameters);
    Richardson richardson = Richardson(parameters);
    vector< PDESolver* > methods{ &analytical, &crank, &laasonen, &dufort, &
        richardson};
    vector< int > timesteps{10,20,30,40,50};
    Printer printer = Printer(&analytical);

    // going over each scheme and creating its .dat files and gnuplot
        commands
    for (int i = 0; i<methods.size(); i++){
        methods[i]->solve();
        printer.setPdeSolver(methods[i]);
        for (int j = 0; j < timesteps.size(); j++) {
            printer.createDatFileForT(timesteps[j]);
            if (i > 0) {
                printer.gnuplotForTCompareTo(timesteps[j], &analytical);
            }
        }
        if (i > 0) {
            printer.datFileErrorsComparedTo(timesteps, &analytical);
            printer.gnuplotErrorsCompareTo(&analytical);
        }
    }
}
```

```cpp
#include"PDESolver.h"
#include<iostream>
```

```cpp
#include <math.h>

// CONSTRUCTOR
// initialisation de fTable et des param tres
PDESolver::PDESolver(Parameters parameters) : parameters(parameters) {
    // catching exceptions
    try {
        if (parameters.L <= 0) {
            throw invalid_argument("The wall must have a length bigger than 0");
        }if (parameters.deltaT <= 0 || parameters.deltaX <= 0) {
            throw invalid_argument("The mesh sizes must be bigger than 0");
        }if (parameters.numberOfSpacePoints * parameters.deltaX != parameters.L) {
            throw invalid_argument("The mesh size times the number of space points should be equal to the length of the wall");
        }
    }
    catch (invalid_argument &error) {
        cout << "ERROR:" << endl;
        cout << error.what() << endl;
        abort();
    }
    this->fTable = vector< vector<double> >(parameters.numberOfTimePoints + 1, vector< double >(parameters.numberOfSpacePoints + 1));
}

// Function checking if we are the edge of the wall
bool PDESolver::checkBundaries(int i, int t) {
    if (i == 0 || i == parameters.numberOfSpacePoints) {
        fTable[t][i] = parameters.tSur;        // If we are at the edge (i=0 or i=last space point), we know the temperature is tSur
        return true;
    }
    else if (t == 0) {
        fTable[t][i] = parameters.tIn;
        return true;
    }
    return false;
}

// Function that returns the name of the class
string PDESolver::getClassName() {
    string className = typeid(*this).name();
    className.erase(0, 6);        // To remove the "class " part of typeid().name()
    return className;
}

// Calculating the L2 norm for each time step compared to PDESolver
vector< double > PDESolver::L2NormWith(PDESolver* pdeSolver) {
    vector< double > L2Norms(parameters.numberOfTimePoints+1);
    vector< vector<double> > fTable2 = pdeSolver->getFTable();
    try {
        // Making sure both schemes are solved, otherwise it is useless to compare them...
        if (fTable[0][0] != parameters.tSur || fTable2[0][0] != parameters.tSur) {
            throw invalid_argument("The two scheme should be solved in
```

```
                    order to compare them");
          }
     }catch (invalid_argument &error) {
          cout << "ERROR:" << endl;
          cout << error.what() << endl;
          abort();
     }
     double sum, norm;
     for (int t = 0; t <= parameters.numberOfTimePoints; t++) {
          sum = 0;
          for (int i = 0; i <= parameters.numberOfSpacePoints; i++) {
               sum += sqrt(pow(fTable[t][i] - fTable2[t][i], 2));
          }
          // Using the norm L2 definition, best one for analysis (cf report)
          norm = sum;
          L2Norms[t] = norm;
     }
     return L2Norms;
}

// Getter for fTable
vector< vector<double> > PDESolver::getFTable() {
     return fTable;
}

// Getter for deltaT
Parameters PDESolver::getParameters() {
     return parameters;
}

void PDESolver::setParameters(Parameters parameters) {
     this->parameters = parameters;
}
```

```
#include"Printer.h"
#include<iostream>
#include<fstream>
#include<sstream>
#include<string>

// CONSTRUCTOR
Printer::Printer(PDESolver* pdeSolver): pdeSolver(pdeSolver), parameters(
    pdeSolver->getParameters()), fTable(pdeSolver->getFTable()), className(
    pdeSolver->getClassName()) {
     deleteGnuplotCommands();
}

// Auxiliary method to throw exceptions if the scheme is not solved
void exceptions(vector< vector<double> > fTable, Parameters parameters) {
     try {
          if (fTable[0][0] != parameters.tSur) {
               throw invalid_argument("The scheme must be solved in order to
                    be printed");
          }
     }
     catch (invalid_argument &error) {
          cout << "ERROR:" << endl;
          cout << error.what() << endl;
```

```
            abort();
        }
}

/**
* Auxiliary function: Convert a float into a string
*
* Useful to get delta t and delta x in the file name. Ex: 0.005 -> 0_005
*
* @param double float, float to be converted
* @return String with float converted in a string
*/
string convertFloatToString(double flt) {
        string str = to_string(flt);
        str.erase(str.find_last_not_of('0') + 1, std::string::npos);
        str.replace(str.find("."), 1, "_");
        return str;
}

// Set Printer's member using the new PDESolver
void Printer::setPdeSolver(PDESolver* pdeSolver) {
        this->pdeSolver = pdeSolver;
        this->parameters = pdeSolver->getParameters();
        this->fTable = pdeSolver->getFTable();
        this->className = pdeSolver->getClassName();
}

// Print values in console for a give time step t
void Printer::printInConsole(int t) {
        exceptions(fTable, parameters);
        cout << "f(" << t << ") = ";
        for (int i = 0; i < fTable[0].size() ; i++) {
            cout << fTable[t][i] << "\t" ;
        }
        cout << endl;
}

// Create a datFile with the values of a give time step t
void Printer::createDatFileForT(int t) {
        exceptions(fTable, parameters);
        string deltaT = convertFloatToString(parameters.deltaT);
        string file = className + "_t" + to_string((int)t) + "_deltaT" + deltaT
            + ".dat";
        ofstream datFile;
        datFile.open("datFiles/" + file);
        // Create the file and fill it up with the values
        for (int i = 0; i < fTable[0].size(); i++) {
            datFile << parameters.deltaX*i << " " << fTable[t][i] << endl;
        }
        datFile.close();
}

// Create a datFile with the norm L2 to check the errors from the scheme
void Printer::datFileErrorsComparedTo(vector<int> ts, PDESolver* pdeSolver2){
        exceptions(fTable, parameters);
        exceptions(pdeSolver2->getFTable(), parameters);
        string deltaT = convertFloatToString(parameters.deltaT);
        string file = className + "_compare_to_" + pdeSolver2->getClassName    ()
            + "_for_deltaT_" + deltaT + ".dat";
        ofstream datFile;
```

```
    datFile.open("datFiles/" + file);                                   // Create the .dat file
    vector< double > L2Norms = pdeSolver->L2NormWith(pdeSolver2);
    for (int t = 0; t < ts.size(); t++) {
        datFile << ts[t]*parameters.deltaT << " " << L2Norms[ts[t]] << endl;
    }
    datFile.close();
}

void Printer::gnuplotForT(int t) {
    string deltaT = convertFloatToString(parameters.deltaT);
    ofstream commands;
    commands.open("commands.txt", ios::out | ios::app);
    commands << "set terminal png" << endl;
    commands << "set output '" << className << "_t" << t << "_deltaT" << deltaT << ".png'" << endl;
    commands << "set grid" << endl;
    commands << "set xlabel \"Distance (feets)\"" << endl;
    commands << "set ylabel \"Temperature (fahrenheits)\"" << endl;
    commands << "plot \"" << className << "_t" << t << "_deltaT" << deltaT << ".dat\" w lp pt 4 lc rgb \"red\" lw 1 title '" << className << "'" << endl;
    commands.close();
}

void Printer::gnuplotForTCompareTo(int t, PDESolver* pdeSolver2) {
    string name = pdeSolver2->getClassName();
    string deltaT = convertFloatToString(parameters.deltaT);
    ofstream commands;
    commands.open("commands.txt", ios::out | ios::app);
    commands << "set terminal png" << endl;
    commands << "set output '" << className << "_compareTo_" << name << "_t_" << t << "_deltaT" << deltaT << ".png'" << endl;
    commands << "set grid" << endl;
    commands << "set xlabel \"Distance (feets)\"" << endl;
    commands << "set ylabel \"Temperature (fahrenheits)\"" << endl;
    commands << "plot \"" + name + "_t" << t << "_deltaT" << deltaT << ".dat\"" << " w l lw 2 lc 'green' title 'Analytic solution',\\" << endl;
    commands << "\"" << className << "_t" << t << "_deltaT" << deltaT << ".dat\" w p pt 4 lc rgb \"red\" lw 1 title '" << className << "'" << endl;
    commands.close();
}

void Printer::gnuplotErrorsCompareTo(PDESolver* pdeSolver2) {
    string name = pdeSolver2->getClassName();
    string deltaT = convertFloatToString(parameters.deltaT);
    ofstream commands;
    commands.open("commands.txt", ios::out | ios::app);
    commands << "set terminal png" << endl;
    commands << "set output '" << className + "_errors_for_deltaT_" + deltaT + ".png" << endl;
    commands << "set grid" << endl;
    commands << "set xlabel \"t (no unit)\"" << endl;
    commands << "set ylabel \"L2 norm\"" << endl;
    commands << "plot \"" << className + "_compare_to_" + name + "_for_deltaT_" + deltaT + ".dat\"" << " w l lw 2 lc 'green' title 'L2 norm for " << className << "'" << endl;     // Graph of the errors
    commands.close();
```

```
}

// Delete the previous gnuplotcommands file
void Printer::deleteGnuplotCommands() {
    ofstream commands;
    commands.open("commands.txt");
    commands.close();
}
```

```
#include"Richardson.h"
#include<iostream>

// CONSTRUCTOR
Richardson::Richardson(Parameters parameters): ExplicitMethod(parameters) {
    getFirstStrepFromFTCS(); // This scheme needs to have 2 time step to get
        the next one. Therefore we must get the first time step from an
        other scheme (FTCS here)
}

// Get the next time step using Richardson's equation
double Richardson::getNextTimeStep(int i, int t) {
    // If t > 1, we use the equation. Otherwise, we must use the result from
        FTCS as discussed in the comment of the constructor
    if (t > 1) {
        return fTable[t-2][i] + parameters.D/(2* parameters.deltaT*
            parameters.deltaX*parameters.deltaX) * (fTable[t-1][i+1] - 2*
            fTable[t - 1][i] + fTable[t - 1][i - 1]);
    }
    else {
        return fTable[t][i];
    }
}
```

Computational methods and C++ assignment

Generated by Doxygen 1.8.13

Contents

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

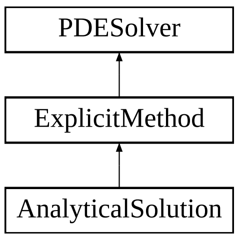
Chapter 3

Class Documentation

3.1 AnalyticalSolution Class Reference

```
#include <AnalyticalSolution.h>
```

Inheritance diagram for AnalyticalSolution:



Public Member Functions

- AnalyticalSolution (Parameters parameters)
- double getNextTimeStep (int i, int t)
- double getAnalyticalValue (double x, double t, int precision)

Additional Inherited Members

3.1.1 Detailed Description

Analytical Solution to the problem

It is not an explicit scheme but its' behaviour is pretty much the same as one, which is why it inherits from ExplicitMethod

3.1.2 Constructor & Destructor Documentation

3.1.2.1 AnalyticalSolution()

```
AnalyticalSolution::AnalyticalSolution (
            Parameters parameters )
```

Analytical constructor

See also

PDESolver (same constructor)

3.1.3 Member Function Documentation

3.1.3.1 getAnalyticalValue()

```
double AnalyticalSolution::getAnalyticalValue (
            double x,
            double t,
            int precision )
```

Function to get the analytical value of the solution f(t, x) with a given precision

Parameters

x,value	of x
t,value	of t
precision	that we wish to have for the infinite for loop (that can't be infinite with a computer...)

3.1.3.2 getNextTimeStep()

```
double AnalyticalSolution::getNextTimeStep (
            int i,
            int t ) [virtual]
```

Implementation of ExplicitMethod's getNextTimeStep

See also

getNextTimeStep() from ExplicitMethod

Implements ExplicitMethod.

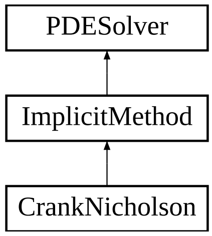
The documentation for this class was generated from the following files:

- AnalyticalSolution.h
- AnalyticalSolution.cpp

3.2 CrankNicholson Class Reference

```
#include <CrankNicholson.h>
```

Inheritance diagram for CrankNicholson:



Public Member Functions

- CrankNicholson (Parameters parameters)
- double getDiagonalValue ()
- double getOutOfDiagonalValue ()
- double getMatrix2Values (int t, int i)

Additional Inherited Members

3.2.1 Detailed Description

CrankNicholson implicit scheme that we are required to use in this assignement

3.2.2 Constructor & Destructor Documentation

3.2.2.1 CrankNicholson()

```
CrankNicholson::CrankNicholson (
            Parameters parameters )
```

CrankNicholson's constructor

See also

PDESolver

3.2.3 Member Function Documentation

3.2.3.1 getDiagonalValue()

```
double CrankNicholson::getDiagonalValue ( )  [virtual]
```

Get the diagonal value for the CrankNicholson scheme

Returns

value of the diagonal of matrix

See also

getDiagonalValue() from ImplicitScheme

Implements ImplicitMethod.

3.2.3.2 getMatrix2Values()

```
double CrankNicholson::getMatrix2Values (
            int t,
            int i )  [virtual]
```

Get the values of matrix2

Returns

value of matrix2 for t and i

See also

getBoundaryValue() from ImplicitScheme

Implements ImplicitMethod.

3.2.3.3 getOutOfDiagonalValue()

```
double CrankNicholson::getOutOfDiagonalValue ( )  [virtual]
```

Get the values out of the diagonal for the CrankNicholson scheme

Returns

value of the points out of the diagonal of matrix

See also

getOutOfDiagonalValue() from ImplicitScheme

Implements ImplicitMethod.

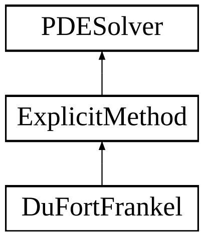
The documentation for this class was generated from the following files:

- CrankNicholson.h
- CrankNicholson.cpp

3.3 DuFortFrankel Class Reference

```
#include <DuFortFrankel.h>
```

Inheritance diagram for DuFortFrankel:



Public Member Functions

- DuFortFrankel (Parameters parameters)
- double getNextTimeStep (int i, int t)

Additional Inherited Members

3.3.1 Detailed Description

DuFortFrankel is an explicit scheme that we are required to use in this assignement

3.3.2 Constructor & Destructor Documentation

3.3.2.1 DuFortFrankel()

```
DuFortFrankel::DuFortFrankel (
            Parameters parameters )
```

DuFortFrankel's constructor

See also

PDESolver (same constructor)

3.3.3 Member Function Documentation

3.3.3.1 getNextTimeStep()

```
double DuFortFrankel::getNextTimeStep (
            int i,
            int t ) [virtual]
```

Implementation of ExplicitMethod's getNextTimeStep

See also

getNextTimeStep() from ExplicitMethod

Implements ExplicitMethod.

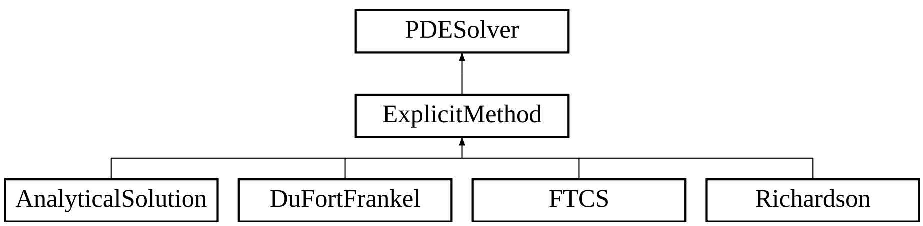
The documentation for this class was generated from the following files:

- DuFortFrankel.h
- DuFortFrankel.cpp

3.4 ExplicitMethod Class Reference

```
#include <ExplicitMethod.h>
```

Inheritance diagram for ExplicitMethod:



Public Member Functions

- ExplicitMethod (Parameters parameters)
- virtual double getNextTimeStep (int i, int t)=0
- void solve ()
- void getFirstStrepFromFTCS ()

Additional Inherited Members

3.4.1 Detailed Description

ExplicitMethod is an abstract class for all of the explicit schemes.

It inherits from PDESolver and contains all of the useful class for an explicit scheme.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 ExplicitMethod()

```
ExplicitMethod::ExplicitMethod (
            Parameters parameters )
```

ExplicitMethod's constructor

See also

PDESolver (same constructor)

3.4.3 Member Function Documentation

3.4.3.1 getFirstStrepFromFTCS()

```
void ExplicitMethod::getFirstStrepFromFTCS ( )
```

Get the first time step using the FTCS method

Some explicit schemes are using two time steps to solve the next one. This is a problem for t = 1 and we therefore need to use an other explicit method to get all the points for the first time step. In our case, we will be using the FTCS scheme.

See also

FTCS

3.4.3.2 getNextTimeStep()

```
virtual double ExplicitMethod::getNextTimeStep (
            int i,
            int t ) [pure virtual]
```

Pure virtual function to get the next time step using the previous ones we already solved

The only difference in each explicit scheme is the way to get the next time step, so this will have to be implemented by all the explicit schemes.

Parameters

i	which space point do we wish to solve
t	at which time step are we

Returns

the value of the temperature for fTable[t][i]

Implemented in AnalyticalSolution, FTCS, DuFortFrankel, and Richardson.

3.4.3.3 solve()

```
void ExplicitMethod::solve ( )  [virtual]
```

Solve method to fill up fTable

See also

solve() function from PDESolver

Implements PDESolver.

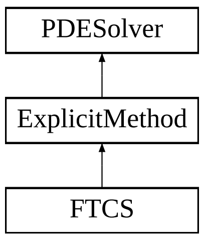
The documentation for this class was generated from the following files:

- ExplicitMethod.h
- ExplicitMethod.cpp

3.5 FTCS Class Reference

```
#include <FTCS.h>
```

Inheritance diagram for FTCS:



Public Member Functions

- FTCS (Parameters parameters)
- double getNextTimeStep (int i, int t)

Additional Inherited Members

3.5.1 Detailed Description

FTCS (Forward Time, Central Space) is an explicit scheme that was not required to use in the assignement.

I implemented it because we need an explicit scheme that doesn't two time steps to get the next one to initialize fTable[1][i] (for all i) for the other explicit schemes.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 FTCS()

```
FTCS::FTCS (
            Parameters parameters )
```

FTCS constructor

See also

PDESolver (same constructor)

3.5.3 Member Function Documentation

3.5.3.1 getNextTimeStep()

```
double FTCS::getNextTimeStep (
            int i,
            int t )  [virtual]
```

Implementation of ExplicitMethod's getNextTimeStep

See also

getNextTimeStep() from ExplicitMethod

Implements ExplicitMethod.

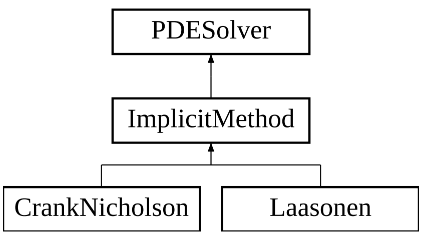
The documentation for this class was generated from the following files:

- FTCS.h
- FTCS.cpp

3.6 ImplicitMethod Class Reference

```
#include <ImplicitMethod.h>
```

Inheritance diagram for ImplicitMethod:



Public Member Functions

- ImplicitMethod (Parameters parameters)
- void solve ()
- void solve_thomas (int t)
- void createMatrix ()
- void createMatrix2 (int t)
- virtual double getDiagonalValue ()=0
- virtual double getOutOfDiagonalValue ()=0
- virtual double getMatrix2Values (int t, int i)=0
- void initialization ()

Protected Attributes

- vector$<$ vector$<$ double $>$ $>$ matrix

  Matrix in the left of the equation.
- vector$<$ double $>$ matrix2

  Matrix in the right of the equation.

3.6.1 Detailed Description

ImplicitMethod is an abstract class for all of the implicit schemes.

It inherits from PDESolver and contains all of the useful class for an implicit scheme. It has a few new class members that are related to the two matrixes needed in the solving of an implicit scheme.

The explicit scheme will solve the equation : matrix $*$ fTable[tWeWantToSolve] = matrix2 using the LU decomposition. The value of matrix and matrix2 changes with the scheme we wish to use.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 ImplicitMethod()

```
ImplicitMethod::ImplicitMethod (
            Parameters parameters )
```

ImplicitMethod's constructor

See also

PDESolver (same constructor)

3.6.3 Member Function Documentation

3.6.3.1 createMatrix()

```
void ImplicitMethod::createMatrix ( )
```

Function that creates matrix (3 diagonal matrix)

3.6.3.2 createMatrix2()

```
void ImplicitMethod::createMatrix2 (
             int t )
```

Function that creates matrix2

Parameters

t	for which time step we wish to create that matrix

3.6.3.3 getDiagonalValue()

```
virtual double ImplicitMethod::getDiagonalValue ( )  [pure virtual]
```

Pure virtual function that will give the value in the diagonal of matrix

The way of creating matrix is always the same (thus the createMatrix() method here), only the value of the matrix changes with the scheme

Returns

value of the diagonal of the matrix

Implemented in CrankNicholson, and Laasonen.

3.6.3.4 getMatrix2Values()

```
virtual double ImplicitMethod::getMatrix2Values (
             int t,
             int i )  [pure virtual]
```

Pure virtual function that gets the values of matrix2

Parameters

time	step
space	point

Returns

value of matrix2 for i and t

Implemented in CrankNicholson, and Laasonen.

3.6.3.5 getOutOfDiagonalValue()

```
virtual double ImplicitMethod::getOutOfDiagonalValue ( )  [pure virtual]
```

Pure virtual that gives the value outside of the diagonal of matrix

Returns

value outside of the diagonal of matrix

See also

getDiagonalValue() for some additional infos

Implemented in CrankNicholson, and Laasonen.

3.6.3.6 initialization()

```
void ImplicitMethod::initialization ( )
```

Method used in the constructor of implicit schemes

It gets matrix2 for the first time step ready as well as matrix and its' LU decomposition that will always be the same no matter the time step and time point

3.6.3.7 solve()

```
void ImplicitMethod::solve ( )  [virtual]
```

Solve method to fill up fTable

See also

solve() function from PDESolver

Implements PDESolver.

3.6.3.8 solve_thomas()

```
void ImplicitMethod::solve_thomas (
            int t )
```

Function that solves the equation matrix ∗ ftable[t+1] = matrix2 once the thomas algorithm has been applied

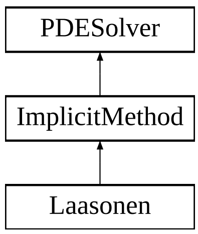
The documentation for this class was generated from the following files:

- ImplicitMethod.h
- ImplicitMethod.cpp

3.7 Laasonen Class Reference

```
#include <Laasonen.h>
```

Inheritance diagram for Laasonen:



Public Member Functions

- Laasonen (Parameters parameters)
- double getDiagonalValue ()
- double getOutOfDiagonalValue ()
- double getMatrix2Values (int t, int i)

Additional Inherited Members

3.7.1 Detailed Description

Laasonen implicit scheme that we are required to use in this assignement

3.7.2 Constructor & Destructor Documentation

3.7.2.1 Laasonen()

```
Laasonen::Laasonen (
            Parameters parameters )
```

Laasonen's constructor

See also

PDESolver (same constructor)

3.7.3 Member Function Documentation

3.7.3.1 getDiagonalValue()

```
double Laasonen::getDiagonalValue ( ) [virtual]
```

Get the diagonal value for the Laasonen scheme

Returns

value of the diagonal of matrix

See also

getDiagonalValue() from ImplicitScheme

Implements ImplicitMethod.

3.7.3.2 getMatrix2Values()

```
double Laasonen::getMatrix2Values (
            int t,
            int i ) [virtual]
```

Get the values of matrix2

Returns

value of matrix2 for t and i

See also

getBoundaryValue() from ImplicitScheme

Implements ImplicitMethod.

3.7.3.3 getOutOfDiagonalValue()

```
double Laasonen::getOutOfDiagonalValue ( ) [virtual]
```

Get the values out of the diagonal for the Laasonen scheme

Returns

value of the points out of the diagonal of matrix

See also

getOutOfDiagonalValue() from ImplicitScheme

Implements ImplicitMethod.

The documentation for this class was generated from the following files:

- Laasonen.h
- Laasonen.cpp

3.8 Parameters Struct Reference

```
#include <Parameters.h>
```

Public Attributes

- double D

  Diffusivity (in ft^2/hr)
- double deltaT

  deltaT (in hrs)
- double deltaX

  deltaX (in ft)
- double tIn

  Temperature of the inside of the wall.
- double tSur

  Temperature of the outside of the wall.
- double L

  Size of the wall.
- int numberOfTimePoints

  Number of points in time we will need to solve.
- int numberOfSpacePoints

  Number of points in our grid that we will try to solve.

3.8.1 Detailed Description

Parameters is a structure that has all of the useful parameters of the problem to solve.

It was created so that if a user uses multiple schemes, he wouldn't have to write each parameters again and again which is quite frustrating. Here, he'll just define them once.

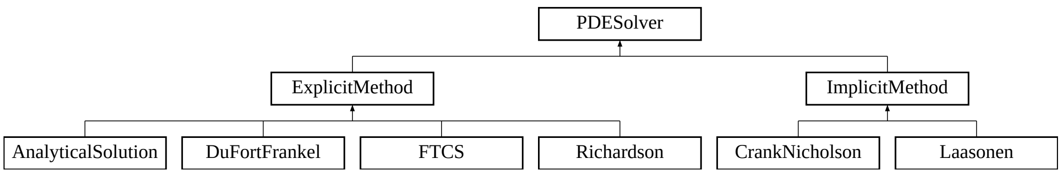
The documentation for this struct was generated from the following file:

- Parameters.h

3.9 PDESolver Class Reference

```
#include <PDESolver.h>
```

Inheritance diagram for PDESolver:



Public Member Functions

- PDESolver (Parameters parameters)
- bool checkBundaries (int i, int t)
- virtual void solve ()=0
- vector< double > L2NormWith (PDESolver *pdeSolver)
- vector< vector< double > > getFTable ()
- Parameters getParameters ()
- void setParameters (Parameters parameters)
- string getClassName ()

Protected Attributes

- Parameters parameters

  Structure Parameters, with all of the problem's parameters, to make it easier on the user to write when he uses the same parameters again and again.
- vector< vector< double > > fTable

  Vector of vectors that contains all the solutions f : fTable[1][10] is the temperature at the 1st time point of the 10th space point.

3.9.1 Detailed Description

PDESolver is an abstract class that has all the required functions and members to solve the partial differential equations (PDE) from the subject. It will be the base for all the schemes.

It provides a constructor that will be used by all of its' derived class to get all the variables needed for the solving of the equation. It also provides some useful functions that will often be used by its' derived class.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 PDESolver()

```
PDESolver::PDESolver (
            Parameters parameters )
```

PDESolver's constructor

By providing all the wanted parameters, we can set up the solving of the equation. This function also initialize fTable with the solutions of the time step 0 that we already know and will always be the same, no matter the scheme used. You probably noticed that we don't ask for numberOfSpacePoints, that is because we can get it using the other parameters.

This function also throws exceptions if the user's value can not be used

Parameters

D	the diffusivity (in ft^2/hr)
deltaT	(in hrs)
deltaX	(in ft)
tIn	the temperature inside of the wall
tSur	the temperature outside of the wall
L	the size of the wall
numberOfTimePoints	number of points in time we wish to solve

3.9.3 Member Function Documentation

3.9.3.1 checkBundaries()

```
bool PDESolver::checkBundaries (
            int i,
            int t )
```

Function that checks if we are at an edge of the wall.

If it's the case, it will also put the right value for that space point at that time step in fTable (which is the boundary condition, that is to say tSur)

Parameters

i	space point that needs to be checked
t	timepoint that we are at (useful to fill up fTable if needed)

Returns

true if we are at a bundary and false otherwise

3.9.3.2 getClassName()

```
string PDESolver::getClassName ( )
```

Function that gives the name of the class (scheme) being used

Returns

the name of the class

3.9.3.3 getFTable()

```
vector< vector< double > > PDESolver::getFTable ( )
```

Getter for fTable (useful in the Printer class)

Returns

fTable

3.9.3.4 getParameters()

```
Parameters PDESolver::getParameters ( )
```

Getter for Parameters

Returns

Parameters

3.9.3.5 L2NormWith()

```
vector< double > PDESolver::L2NormWith (
            PDESolver * pdeSolver )
```

Function that creates a vector with the L2 norm for each time step in comparison of an other PDESolver

Mostly used with the analytic solution, to get the errors of the scheme

Parameters

pdeSolver	to do the L2 norm with

Returns

vector with the values of the norm for each time step of our scheme

3.9.3.6 setParameters()

```
void PDESolver::setParameters (
            Parameters parameters )
```

Setter for Parameters

3.9.3.7 solve()

```
virtual void PDESolver::solve ( ) [pure virtual]
```

Pure virtual function that solves all the points that we desire and put their value in fTable

Every type of scheme (implicit and explicit) will solve our equation, but they have different ways to do it, as we will see. Therefore, this is a pure virtual class.

Implemented in ExplicitMethod, and ImplicitMethod.

The documentation for this class was generated from the following files:

- PDESolver.h
- PDESolver.cpp

3.10 Printer Class Reference

```
#include <Printer.h>
```

Public Member Functions

- Printer (PDESolver ∗pdeSolver)
- void setPdeSolver (PDESolver ∗pdeSolver)
- void printInConsole (int t)
- void createDatFileForT (int t)
- void deleteGnuplotCommands ()
- void gnuplotForT (int t)
- void gnuplotErrorsCompareTo (PDESolver ∗pdeSolver2)
- void gnuplotForTCompareTo (int t, PDESolver ∗pdeSolver2)
- void datFileErrorsComparedTo (vector< int > ts, PDESolver ∗pdeSolver2)

3.10.1 Detailed Description

Printer class used to create all the datFiles as well as the gnuplot commands to do to get the graphs from the report

3.10.2 Constructor & Destructor Documentation

3.10.2.1 Printer()

```
Printer::Printer (
            PDESolver * pdeSolver )
```

Printer's constructor

Parameters

pdeSolver	scheme for which we would like to print something
fileName	fileName to use

3.10.3 Member Function Documentation

3.10.3.1 createDatFileForT()

```
void Printer::createDatFileForT (
            int t )
```

Function that creates a datFile for a given time step (found in the datFiles repo)

Parameters

t	time step for which we want to get a datFile

3.10.3.2 datFileErrorsComparedTo()

```
void Printer::datFileErrorsComparedTo (
            vector< int > ts,
            PDESolver * pdeSolver2 )
```

Print the norm (L2) of the errors in a datFile for given time steps

Parameters

ts	time steps for which we will have a look at the errors
analyticalSolution	to compare with our scheme

3.10.3.3 deleteGnuplotCommands()

```
void Printer::deleteGnuplotCommands ( )
```

Function to delete the .txt file with the commands (inside the repo of the code)

3.10.3.4 gnuplotErrorsCompareTo()

```
void Printer::gnuplotErrorsCompareTo (
            PDESolver * pdeSolver2 )
```

Function that add to the command.txt (or create it) the commands to plot the L2-norm of the difference between two schemes

Parameters

*PDESolver**	that we want to do the L2-norm with

3.10.3.5 gnuplotForT()

```
void Printer::gnuplotForT (
            int t )
```

Function that add to the command.txt (or create it) the commands to plot the scheme at the time step t using gnuplot

Parameters

t	time step for which we wish to plot a graph

3.10.3.6 gnuplotForTCompareTo()

```
void Printer::gnuplotForTCompareTo (
            int t,
            PDESolver * pdeSolver2 )
```

Function that add to the command.txt (or create it) the commands to plot 2 schemes on the same graph in order to compare them

Parameters

t	time step for which we wish to plot the graph
PDESolver∗	that we want to plot with our scheme

3.10.3.7 printInConsole()

```
void Printer::printInConsole (
            int t )
```

Function that prints the value in console

Parameters

t	time step for which we want to see the values

3.10.3.8 setPdeSolver()

```
void Printer::setPdeSolver (
            PDESolver * pdeSolver )
```

Setter for PDESolver (also changes the parameters using the one from the new pdeSolver

Parameters

pdeSolver	PDESolver∗ to print

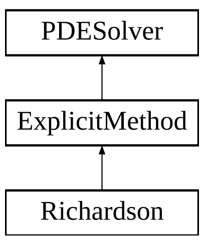
The documentation for this class was generated from the following files:

- Printer.h
- Printer.cpp

3.11 Richardson Class Reference

```
#include <Richardson.h>
```

Inheritance diagram for Richardson:



Public Member Functions

- Richardson (Parameters parameters)
- double getNextTimeStep (int i, int t)

Additional Inherited Members

3.11.1 Detailed Description

Richardson is an explicit scheme that we are required to use in this assignement

3.11.2 Constructor & Destructor Documentation

3.11.2.1 Richardson()

```
Richardson::Richardson (
             Parameters parameters )
```

Richardson's constructor

See also

PDESolver (same constructor)

3.11.3 Member Function Documentation

3.11.3.1 getNextTimeStep()

```
double Richardson::getNextTimeStep (
             int i,
             int t )  [virtual]
```

Implementation of ExplicitMethod's getNextTimeStep

See also

getNextTimeStep() from ExplicitMethod

Implements ExplicitMethod.

The documentation for this class was generated from the following files:

- Richardson.h
- Richardson.cpp

Index