



Adam Causse and Mathieu Quintana

Comparing finite-differential approach to discretise one  
dimensional heat equation

Faculty of Engineering and Applied Sciences  
Aerospace computational Engineering

MSc  
Academic Year: 2025–2026

Supervisors: Dr Teschner T. and Dr Sherar P.  
August 2025



Faculty of Engineering and Applied Sciences  
Aerospace computational Engineering

MSc

Academic Year: 2025–2026

Adam Causse and Mathieu Quintana

Comparing finite-differential approach to discretise one  
dimensional heat equation

Supervisors: Dr Teschner T. and Dr Sherar P.  
August 2025

This thesis is submitted in partial fulfilment of the  
requirements for the degree of MSc.

© Cranfield University 2025. All rights reserved. No part of  
this publication may be reproduced without the written  
permission of the copyright owner.

## **Academic Integrity Declaration**

I declare that:

- the thesis submitted has been written by me alone.
- the thesis submitted has not been previously submitted to this university or any other.
- that all content, including primary and/or secondary data, is true to the best of my knowledge.
- that all quotations and references have been duly acknowledged according to the requirements of academic research.

I understand that to knowingly submit work in violation of the above statement will be considered by examiners as academic misconduct.

# Abstract

This study presents a comparative of numerical schemes (stability and accuracy) applied to a one-dimensional parabolic partial differential equations (PDE) as well as the effect of the number of iterations upon the error. After deliberation, the Dufort-Frankel scheme was deemed more suitable for solving this equation. Subsequently, we questioned the usefulness of the time step size and the total number of iterations.

Keywords:

Partial Differential Equation, DuFort-Frankel scheme, Richardson scheme, Laasonen scheme, Crank-Nicolson scheme, C<sup>++</sup> Oriented Object

# Acknowledgements

We express our sincere gratitude to our professors for the knowledge, guidance, and support they have provided throughout this project and during our time in the program.

Dr. Tom-Robin Teschner

Dr. Laszlo Konorzy

Dr. Peter Sherar

Dr. Tamas Jozsa

Mistral AI for its grammatical assistance during the preparation of the assignment report.

# Contents

<b>Academic Integrity Declaration</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>Contents</b>	<b>iv</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>vii</b>
<b>List of Abbreviations</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Literature review</b>	<b>2</b>
<b>3 Methodology</b>	<b>5</b>
3.1 Situation and hypotheses . . . . .	5
3.2 Computing and coding method . . . . .	7
3.2.1 Code management . . . . .	7
3.2.2 Dufort-Frankel scheme . . . . .	7
3.2.3 Richarson . . . . .	7
3.2.4 Laasonen . . . . .	8
3.2.5 Crank-Nicolson . . . . .	9
<b>4 Result analys</b>	<b>10</b>
4.1 Stability . . . . .	10
4.2 Comparing on the case with $dt = 0.01$ . . . . .	10
4.2.1 Dufort-Frankel explicit scheme . . . . .	11
4.2.2 Laasonen . . . . .	12
4.2.3 Crank-Nicolson implicit scheme . . . . .	13
4.2.4 Most useful scheme in our case . . . . .	13
4.3 $T_{max}$ unit impact on scheme accuracy . . . . .	14
4.4 $dt$ impact on scheme accuracy . . . . .	16
<b>5 Conclusion</b>	<b>17</b>

<b>References</b>	<b>18</b>
<b>6 Documentation Doxygen</b>	<b>19</b>
<b>7 C++ Code</b>	<b>27</b>

# List of Figures

3.1	Noneclature and Problem Domain . . . . .	5
4.1	Error using Dufort-Frankel scheme . . . . .	11
4.2	Error using Laasonen scheme . . . . .	12
4.3	Error using Crank-Nicolson scheme . . . . .	13
4.4	Numerical solution using Dufort-Frankel scheme . . . . .	14
4.5	Numerical solution using Dufort-Frankel scheme . . . . .	14
4.6	Numerical solution using Dufort-Frankel scheme . . . . .	15
4.7	* . . . . .	16
4.8	* . . . . .	16
4.9	Laasonen scheme results after 6 min . . . . .	16

# List of Tables

4.1	Impact of the time definition on the time processing . . . . .	15
-----	--	----

# List of Abbreviations

PDE	Partial Differential Equation
L	Thickness between the two walls = 31 cm
D	diffusivity of nickel steel = $93 \text{ cm}^2/\text{hr}$
$T_{sur}$	Temperature on the two sides
$T_{in}$	Temperature between the two sides
$T_{max}$	conduction time
n	number of iterations
$\Delta t$	time iterations size
$\Delta x$	space iterations size
$T_i^k$	Temperature after $k \times \text{time(s)}$ iteration and at the position $i \times \text{space(s)}$ iterations
$\alpha$	$\alpha = \frac{2D\Delta t}{\Delta x^2}$
$\Delta^{j,i}$	Error difference between the numerical result at $T_{max} = j$ and $T_{max} = i$

# Chapter 1

## Introduction

As presented during the course[1], there are many schemes available for approximating one-dimensional parabolic partial differential equations of this type, such as :

- Simple Explicit
- Simple Implicit : Laasonen
- Crank-Nicholson
- DuFort-Frankel
- Combined Explicit/Implicit
- Richarson

While the use of numerical schemes certainly reduces computation time for solving partial differential equations (PDE), it comes with the cost of introducing error and potential instabilities[1]. In their book, "Computational Fluid Dynamics"[2], Hoffman and Chiang present a comparison of various schemes with an forward in time and centered in space scheme (FTCS).The goal of this study is to compare several of these schemes in order to determine which one should be used for solving a 1D parabolic PDE.

# Chapter 2

## Literature review

It seems interesting, in this section, to review the schemes studied, their specific features, and some of the methods covered in course that may prove to be useful during the study. Let us consider a 1D parabolic PDE with the following notation :

$$\frac{\partial f}{\partial t} = D \frac{\partial^2 f}{\partial x^2} \text{ with } D \text{ constant}$$

It's now possible to rewrite this equation using these following schemes [?] :

- Dufort-Frankel scheme[1]

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

- Richardson scheme[1]

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

- Laasonen scheme[1]

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

- Crank-Nicolson scheme[1]

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

Just as a reminder, some numerical schemes, may require matrix solving methods. These were covered in class and can be expressed as follows :

- LU decomposition[1]

Which has a computational cost of  $\frac{2}{3}n^3 + 2pn^2$  with  $p$  the number of system to solve. This could also be written as  $O(n^3)$

This method can be used when the equation is as follows:

$$A \times X = B$$

$$\text{or} \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n-1} & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix} \times \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ \vdots \\ \vdots \\ b_n \end{bmatrix}$$

Then it consists of decomposing  $A$  into an lower trigonal matrix  $L$  and an upper trigonal matrix  $U$ . Then to solve the equation with  $A = LU$

- Cholesky decomposition[1]

Which has a computational cost and a storage twice as low as the LU decomposition.

The equation has the same form as for the LU decomposition but with  $A$  symmetric and positive define.  $A$  could then be decomposed by calculating a lower triangular matrix and its transpose to solve the equation with  $A = L.L^T$

- Thomas Algorithm[1]

Which has a computational cost of  $O(n)$  that is much smaller than any other decomposition but needed a special case to be used :

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & 0 & 0 \\ a_2 & b_2 & c_2 & \ddots & & \vdots \\ 0 & a_3 & b_3 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & c_{n_x-2} & 0 \\ 0 & & \ddots & a_{n_x-1} & b_{n_x-1} & c_{n_x-1} \\ 0 & 0 & \cdots & 0 & a_{n_x} & b_{n_x} \end{bmatrix} \times \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ d_{n_x} \end{bmatrix}$$

Then 2 phases are needed to solve the equation :

Firstly, the forward elimination phase where for  $2 \leq k \leq n$  :

$$m = \frac{a_k}{b_{k-1}}$$

$$b_k = b_k - m \times c_{k-1}$$

$$d_k = d_k - m \times d_{k-1}$$

Finally, the backward substitution phase where :

$$x_n = \frac{d_n}{b_n}$$

and for  $n - 1 \geq k \geq 1$  :

$$x_k = \frac{d_k - c_k \times x_{k+1}}{b_k}$$

# Chapter 3

## Methodology

### 3.1 Situation and hypotheses

To compare the accuracy of the different schemes, we consider a test in which the analytical solution is known:

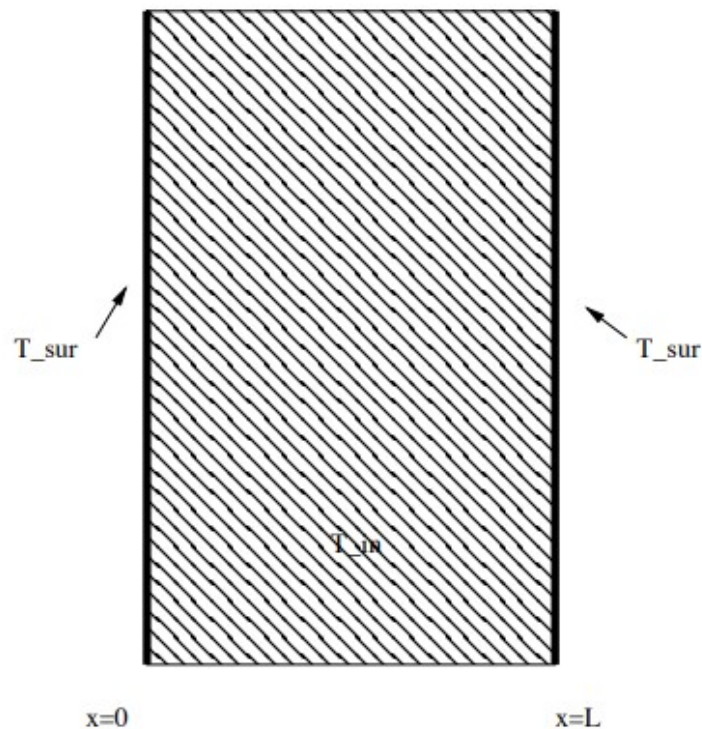


Figure 3.1: Nomenclature and Problem Domain

In our case :

- The thickness between the two walls,  $L$  is 31 cm.
- The height of the walls is considered infinite.
- The conduction time ( $T_{max}$ ) is divided in iterations as follow :

$$T_{max} = \text{iterations size} \times \text{number of iterations}$$

In this case the iterations size is fixed at  $\Delta t = 0.01$  which means that for  $T_{max} = 0.5h$ , there is 50 iterations. If needed,  $T_{max}$  could be change in minute or second to add the number of iterations :  $T_{max} = 0.5h = 30min = 1800s$ ,

For  $T_{max} = 30min$ , number of iterations = 3000

For  $T_{max} = 1800s$ , number of iterations = 180 000

- The initial uniform temperature  $T_{in} = T_{sur} = T(t = 0) = 38^\circ\text{C}$ .

$$|38 \quad 38 \quad \dots \quad 38 \quad 38|$$

- Then, the two sides are suddenly increased and maintained at  $T_{sur} = 149^\circ\text{C}$  but  $T_{in}$  is still equal to  $38^\circ\text{C}$ . Which means that at  $T(t = \Delta t)$  there is this case :

$$|149 \quad 38 \quad \dots \quad 38 \quad 149|$$

This situation can be interpreted as the unsteady one-space dimensional heat conduction in Cartesian coordinates, governed by the equation:

$$\frac{\partial T}{\partial t} = D \frac{\partial^2 T}{\partial x^2}$$

The analytic solution is well known :

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

After computing each scheme for a conduction time :  $T_{max} = 6 \text{ min}, 12 \text{ min}, 18 \text{ min}, 24 \text{ min},$  and  $30 \text{ min}$ , it is possible to see the effect of the conduction time on the numerical solution.

For each value  $T_{max}$ , an absolute error is calculated using the analytic solution and the scheme:

$$\text{Absolute error} = |\text{analytic solution} - \text{numerical solution}|$$

This error will shine the light on the differences in accuracy between all schemes.

## 3.2 Computing and coding method

### 3.2.1 Code management

The code is implemented using object-oriented programming techniques. This aims to limit computation time and numerous repetitions. Indeed, the four schemes share a common foundation. Therefore, four subclasses (one for each scheme) derived from a base class were created. The base class allows us to return the analytical solution and construct the vectors that will enable iteration of the schemes. The four subclasses are constructed in the same way with two constructors and a method that allows for the direct return of the solution vector. For data processing, the error is calculated in the code by a separate function. Finally, a loop in the main code allows extracting four vectors: the position, the analytical solution, the numerical solution, and the absolute error. A .csv file is produced for each scheme and each duration.

### 3.2.2 Dufort-Frankel scheme

As told in our course[1], the Dufort-Frankel scheme discretizes the

$$\frac{\partial T}{\partial t} = D \frac{\partial^2 T}{\partial x^2}$$

in:

$$\frac{T_i^{n+1} - T_i^{n-1}}{2\Delta t} = D \frac{T_{i+1}^n - T_i^{n+1} - T_i^{n-1} + T_{i-1}^n}{\Delta x^2}$$

To compute this explicit scheme, we need to isolate  $T_i^{n+1}$  with  $\alpha = \frac{2D\Delta t}{\Delta x^2}$  :

$$T_i^{n+1} = \frac{2\alpha}{2\alpha + 1} [T_{i+1}^n + T_{i-1}^n] + \frac{1 - 2\alpha}{1 + 2\alpha} T_i^{n-1}$$

### 3.2.3 Richarson

Using the same method as for Dufort-Frankel, the Richardson approximation of the diffusion equation gives[1]:

$$\frac{T_i^{n+1} - T_i^{n-1}}{2\Delta t} = D \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2}$$

And  $T_i^{n+1}$  is isolated to fill the solution vector with  $\alpha = \frac{2D\Delta t}{\Delta x^2}$  :

$$T_i^{n+1} = 2\alpha [T_{i+1}^n - 2T_i^n + T_{i-1}^n] + T_i^{n-1}$$

### 3.2.4 Laasonen

Once again, we start from the discretization of the equation using the Laasonen scheme[1] :

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = D \frac{T_{i+1}^{n+1} - 2T_i^{n+1} + T_{i-1}^{n+1}}{\Delta x^2}$$

However, since the method is implicit, we isolate  $T_i^n$  still with  $\alpha = \frac{2D\Delta t}{\Delta x^2}$  :

$$T_i^n = -\alpha T_{i+1}^{n+1} + (1 + 2\alpha)T_i^{n+1} - \alpha T_{i-1}^{n+1}$$

To solve this equation, several methods are possible. Among those available, the Thomas algorithm has the lowest computational cost and in this case, it is possible to use the Thomas algorithm with :

$$a_k = -\alpha$$

$$b_k = 2\alpha + 1$$

$$c_k = -\alpha$$

$$d_k = T_k^n$$

Thus, the equation in matrix form is :

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & 0 & 0 \\ a_2 & b_2 & c_2 & \ddots & & \vdots \\ 0 & a_3 & b_3 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & c_{n_x-2} & 0 \\ 0 & & \ddots & a_{n_x-1} & b_{n_x-1} & c_{n_x-1} \\ 0 & 0 & \cdots & 0 & a_{n_x} & b_{n_x} \end{bmatrix} \times \begin{bmatrix} T_1^{n+1} \\ \vdots \\ \vdots \\ \vdots \\ T_{n_x}^{n+1} \end{bmatrix} = \begin{bmatrix} d_1 \\ \vdots \\ \vdots \\ \vdots \\ d_{n_x} \end{bmatrix}$$

All that remains is to follow Thomas decomposition

### 3.2.5 Crank-Nicolson

Following the course, the Crank-Nicolson approximation of the heat conduction equation is[1] :

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \frac{D}{2} \left( \frac{T_{i+1}^{n+1} - 2T_i^{n+1} + T_{i-1}^{n+1}}{\Delta x^2} + \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2} \right)$$

We isolate the  $T^{n+1}$  terms from the  $T^n$  terms to proceed with matrix-based solution :

$$-\frac{\alpha}{2}T_{i+1}^{n+1} + (1 + \alpha)T_i^{n+1} - \frac{\alpha}{2}T_{i-1}^{n+1} = \frac{\alpha}{2}T_{i+1}^n + (1 - \alpha)T_i^n + \frac{\alpha}{2}T_{i-1}^n$$

Then, there is the Thomas decomposition with :

$$a_k = -\frac{\alpha}{2}$$

$$b_k = 1 + \alpha$$

$$c_k = -\frac{\alpha}{2}$$

$$d_0 = (1 - \alpha)T_0^n + \frac{\alpha}{2}T_1^n$$

$$d_k = \frac{\alpha}{2}T_{k-1}^n + (1 - \alpha)T_k^n + \frac{\alpha}{2}T_{k+1}^n$$

$$d_{nx-1} = \frac{\alpha}{2}T_{nx-2}^n + (1 - \alpha)T_{nx-1}^n$$

In the same way as for the Laasonen method, we apply the Thomas algorithm decomposition.

# Chapter 4

## Result analys

### 4.1 Stability

Before being precise, numerical schemes must converge and be stable. However, as indicated in Hoffmann and Chiang's book[2], the Richardson discretization is unstable for a 1D parabolic PDE. The results obtained with this method returns very large and unstable values.

The Richardson scheme results were confirmed to be unstable and unusable. Therefore, they will not be considered in further analyses.

### 4.2 Comparing on the case with $dt = 0.01$

Throughout this section, the temperature is set to  $149^{\circ}\text{C}$ , which may result in an apparent discontinuity at the edges of some graphs. Therefore, the analysis of results will not focus on the error near the walls.

### 4.2.1 Dufort-Frankel explicit scheme

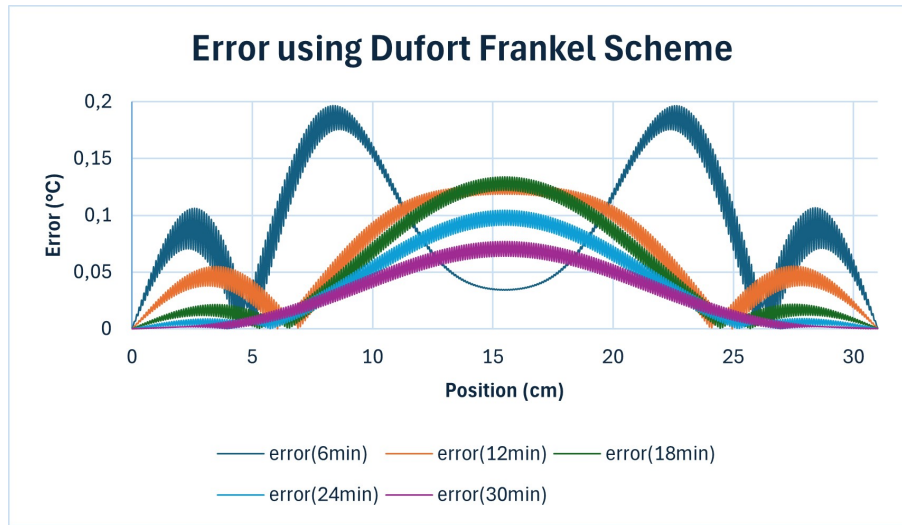


Figure 4.1: Error using Dufort-Frankel scheme

The maximum error is already below  $0.2^{\circ}\text{C}$  and is reached at 6 minutes of simulation, but it does not exceed  $0.1^{\circ}\text{C}$  after 24 minutes. This scheme tends to be particularly accurate at 6 cm from the walls and less so at the center. However, while the regions closer than 7 cm to the walls become increasingly precise over time, at the center, this precision is only valid after 18 minutes (slightly less precise than at 12 minutes of conduction).

Thus, precision continuously improves after 18 minutes of conduction and remains approximately  $0.1^{\circ}\text{C}$  (an error on the order of  $10^{-1}^{\circ}\text{C}$ ) across all positions between the two walls, although the error fluctuates depending on the position.

### 4.2.2 Laasonen

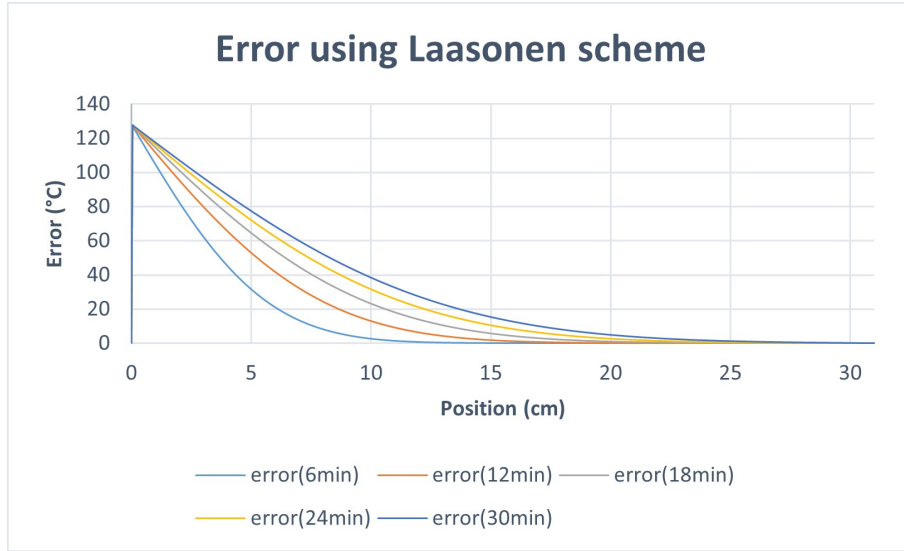


Figure 4.2: Error using Laasonen scheme

For this scheme, the maximum error (over  $120^{\circ}\text{C}$ ) occurs at  $x = 0.31$  cm. This error is significant given that the temperature ranges between  $38^{\circ}\text{C}$  and  $149^{\circ}\text{C}$ . Precision continuously improves as the distance from the position with the maximum error increases. However, the error decreases less rapidly the longer the simulation time. Indeed, the improvement in precision  $\Delta_{max}^{30,24}$  between 30 and 24 min ( $\Delta_{max}^{30,24} = 6.92^{\circ}\text{C}$ ) is less pronounced than that between 6 and 12 min :  $\Delta_{max}^{12,6}$  of simulation ( $\Delta_{max}^{12,6} = 24.30^{\circ}\text{C}$ ). Setting an error threshold below  $0.1^{\circ}\text{C}$  as the target, this level of accuracy is only achieved in the final 8 mm of the simulation and after 30 minutes of simulation time. Overall, the precision of this scheme remains considerably lower than that obtained with the Dufort-Frankel method.

### 4.2.3 Crank-Nicolson implicit scheme

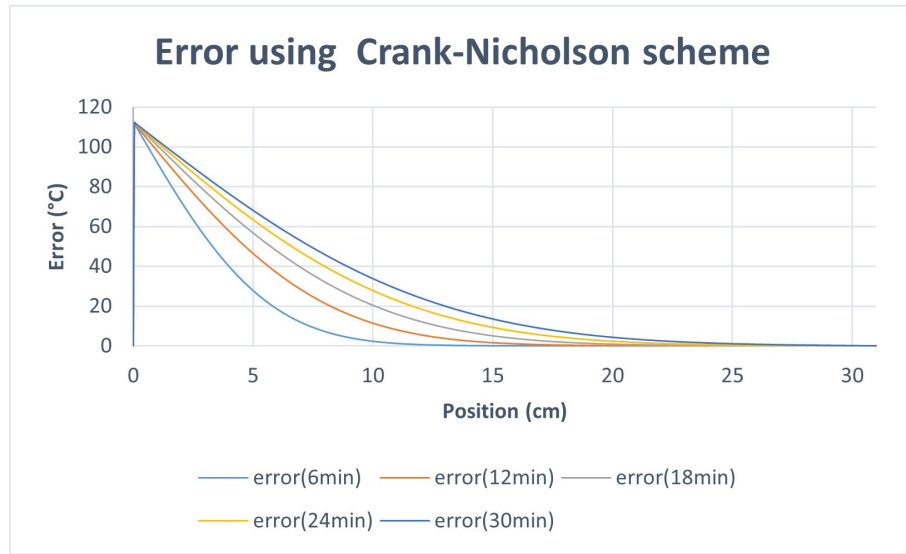


Figure 4.3: Error using Crank-Nicolson scheme

Similar to the Laasonen scheme, the error is maximal at one extremity and continuously decreases until reaching the other extremity, with this decrease becoming less pronounced as simulation time increases. The precision remains lower compared to that achieved with the Dufort-Frankel scheme after 18 minutes. Indeed, only the 6.15 cm farthest from the position of maximum error exhibit an error below  $0.1^{\circ}\text{C}$ , and this error increases as the simulation progresses. Overall, this scheme can be used if the goal is to determine the temperature within 10 cm of a wall for less than 12 minutes of simulation, but the Dufort-Frankel scheme remains superior for analyzing all positions between the two walls.

### 4.2.4 Most useful scheme in our case

In this study, the Dufort-Frankel scheme appears to be the most appropriate. Indeed, it maintains an error below  $0.2^{\circ}\text{C}$  across all positions and is stable. Furthermore, as the simulation progresses beyond 18 minutes, precision continues to improve. It is now possible to visualize the conduction after 30 minutes:

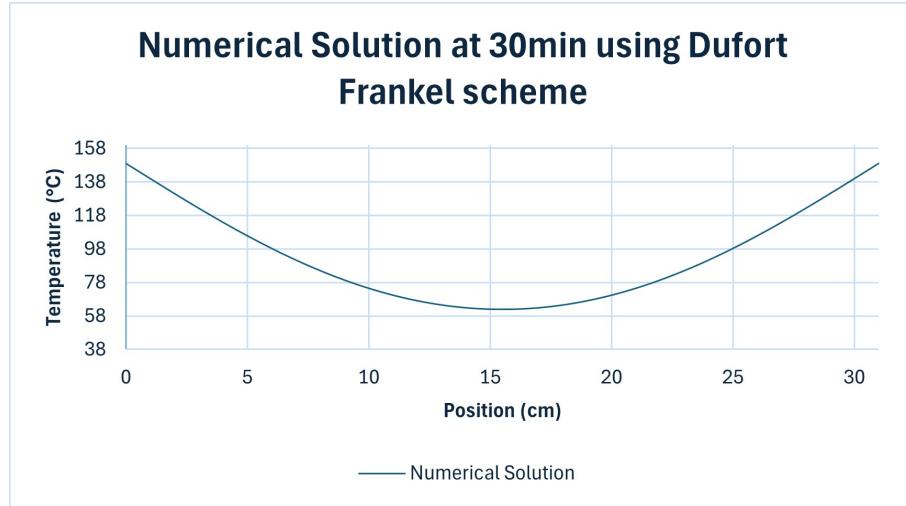


Figure 4.4: Numerical solution using Dufort-Frankel scheme

With a precision of approximately  $\pm 0.1^\circ\text{C}$  and temperatures between  $58^\circ\text{C}$  to  $149^\circ\text{C}$ , the curve with analytical results would have been indistinguishable from the one obtained using the Dufort-Frankel scheme. However, this precision could be adjusted by changing the values of  $\Delta t$  and  $T_{max}$  units.

### 4.3 $T_{max}$ unit impact on scheme accuracy

Since the Dufort–Frankel scheme provided the highest level of accuracy among the tested methods, further investigations were conducted to determine whether its precision could be improved even more. To this end, a study was carried out on the way time is implemented in the code. Initially defined in minutes, the time step was then converted to seconds and finally to tenths of a second. To preserve the stability of the scheme, this refinement was also applied to  $D$ , the diffusivity of the nickel bar.

Switching from minutes to seconds can increase numerical accuracy by up to two orders of magnitude, which may be necessary for experiments requiring higher precision. However, refining the time step from seconds to tenths of a second does not lead to any significant improvement in the results. The scheme appears to reach its accuracy limit when the time variable is defined in seconds.

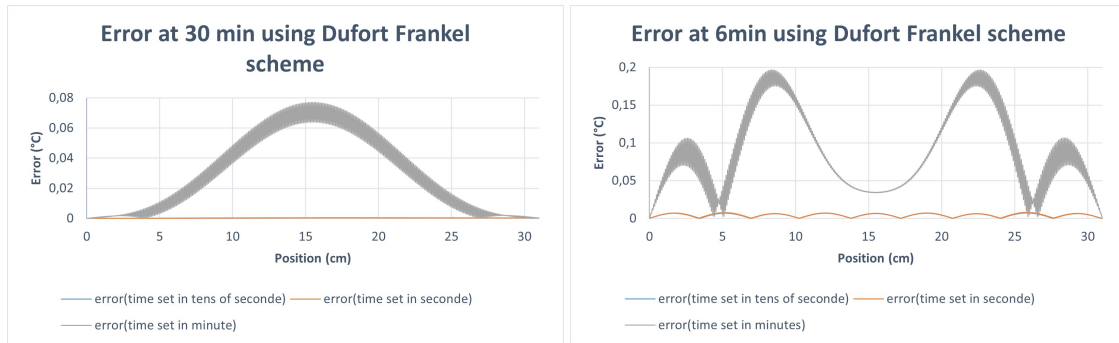


Figure 4.5: Numerical solution using Dufort-Frankel scheme

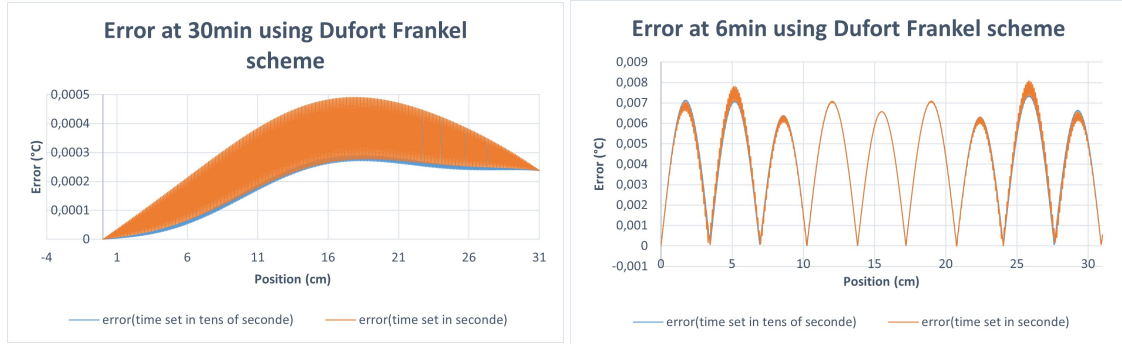


Figure 4.6: Numerical solution using Dufort-Frankel scheme

Simulation time is a key factor when attempting to improve the accuracy of a numerical model. An analysis of the required computation time supports the hypothesis that the Dufort–Frankel scheme may reach an intrinsic accuracy limit. Indeed, the computation time increases by a factor of 2.5 when the time step is refined from minutes to seconds, and by a factor of 4.6 when it is further refined from seconds to tenths of a second.

The computation time therefore appears to increase almost exponentially, while the reduction in numerical error reaches a limit.

Time definition	average error at 6min	max error at 6min	average error at 30min	max error at 30min	time processing
minute	$8,89 \times 10^{-2} \text{ }^{\circ}\text{C}$	$1,96 \times 10^{-1} \text{ }^{\circ}\text{C}$	$2,95 \times 10^{-2} \text{ }^{\circ}\text{C}$	$7,67 \times 10^{-2} \text{ }^{\circ}\text{C}$	3,789 s
seconds	$4,34 \times 10^{-3} \text{ }^{\circ}\text{C}$	$8,07 \times 10^{-3} \text{ }^{\circ}\text{C}$	$2,63 \times 10^{-4} \text{ }^{\circ}\text{C}$	$4,91 \times 10^{-4} \text{ }^{\circ}\text{C}$	9,671 s
tenths of a second	$4,34 \times 10^{-3} \text{ }^{\circ}\text{C}$	$7,38 \times 10^{-3} \text{ }^{\circ}\text{C}$	$1,87 \times 10^{-4} \text{ }^{\circ}\text{C}$	$2,92 \times 10^{-4} \text{ }^{\circ}\text{C}$	45,935 s

Table 4.1: Impact of the time definition on the time processing

## 4.4 dt impact on scheme accuracy

The impact of the time step on the accuracy of the results obtained with the Laasonen scheme is very limited. This is due to its implicit nature, which prevents it from being strongly affected by larger time steps[1, 2]. A degradation in accuracy caused by using a time step ten times larger than the baseline can therefore be acceptable if the goal is to reduce computation time. For this experiment, the overall error of the scheme remains significantly greater than the differences observed between the individual curves.

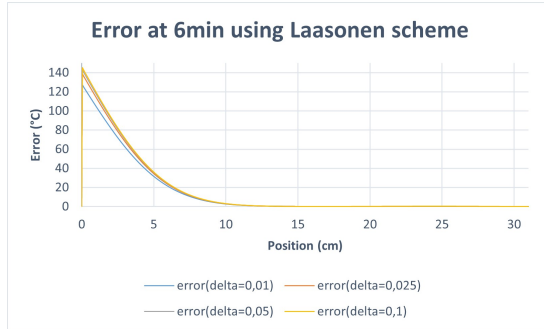


Figure 4.7: \*  
(a) Error after 6 min

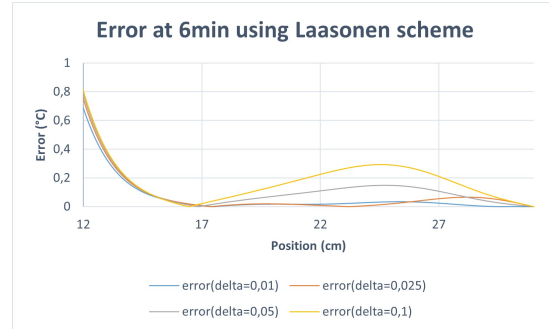


Figure 4.8: \*  
(b) Position with error < 1°C

Figure 4.9: Laasonen scheme results after 6 min

# Chapter 5

## Conclusion

Thus, for a 1D parabolic PDE, the Richardson scheme is unstable, while the Dufort-Frankel scheme provides the highest precision among the stable schemes studied. Switching  $T_{max}$  to seconds and setting  $\Delta t = 0.01$  would be a good alternative to achieve a solution with an average error on the order of  $10^{-3}^{\circ}\text{C}$ , while requiring only 9.671 s to compile the code. It would be possible to draw an analogy with the Laasonen scheme to verify whether the gain in precision would increase significantly in proportion to the computation time. However, given the results obtained by setting  $T_{max}$  in tenths of a second (which adds iterations, similar to decreasing  $\Delta t$ ), it appears that the precision reaches a limit.

s

# References

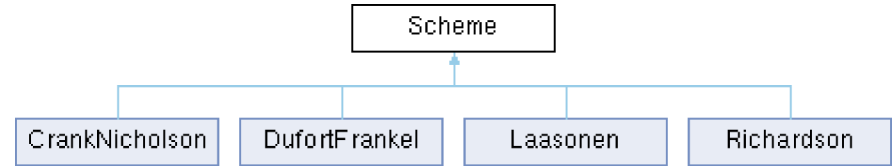
- [1] Moulitsas I. Computational Methods [Lecture]; 2023. Aerospace Computational Engineering. Available from: [https://cranfield.instructure.com/courses/37973/pages/lecture-3?module\\_item\\_id=736836](https://cranfield.instructure.com/courses/37973/pages/lecture-3?module_item_id=736836).
- [2] Hoffman KA, Chiang ST. Computational Fluid Dynamics. vol. 1. Wichita, Kansas 67208-1078, USA: Engineering Education System; 1993.

## **Chapter 6**

### **Documentation Doxygen**

# Scheme Class Reference

Inheritance diagram for Scheme:



## Public Member Functions

<b>Scheme</b> (double boundary_left, double boundary_right, double boundary_bottom, double D, double Xmax, double Tmax, double dt, double dx)
vector< double > <b>get_fnp1</b> ()
vector< double > <b>get_fn</b> ()
vector< double > <b>get_fnm1</b> ()
vector< double > <b>get_fex</b> ()
double <b>get_alfa</b> ()

## Protected Attributes

double <b>boundary_left</b>
double <b>boundary_right</b>
double <b>boundary_bottom</b>
double <b>D</b>
double <b>Xmax</b>
double <b>Tmax</b>
double <b>dt</b>
double <b>dx</b>
vector< double > <b>fn</b>
vector< double > <b>fnm1</b>
vector< double > <b>fnp1</b>
vector< double > <b>fex</b>

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

- [Scheme.h](#)
- [Scheme.cpp](#)

### Public Member Functions

Scheme  
get\_fnp1  
get\_fn  
get\_fnm1  
get\_fex  
get\_alfa

### Protected Attributes

boundary\_left  
boundary\_right  
boundary\_bottom  
D  
Xmax  
Tmax  
dt  
dx  
fn  
fnm1  
fnp1  
fex

[List of all members](#)

## Scheme.h

```

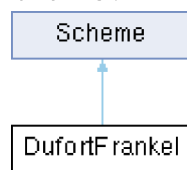
1  ☐ #pragma once
2  #include <iostream>
3  #include <cmath>
4  #include <vector>
5  #include <iomanip>
6  #include<string>
7  #include<fstream>
8
9
10 using namespace std;
11
12 ☐ class Scheme {
13     protected:
14         double boundary_left;
15         double boundary_right;
16         double boundary_bottom;
17         double D;
18         double Xmax;
19         double Tmax;
20         double dt;
21         double dx;
22         vector<double> fn, fnm1, fnp1,fex;
23     public:
24         Scheme();
25         Scheme(double boundary_left, double boundary_right, double boundary_bottom, double D, double Xmax, double Tmax, double dt, double dx);
26         vector<double> get_fnp1();
27         vector<double> get_fn();
28         vector<double> get_fnm1();
29         vector<double> get_fex();
30         double get_alfa();
31
32     };
33
34 ☐ class DufortFrankel : public Scheme {
35     public:
36         DufortFrankel();
37         DufortFrankel(double boundary_left, double boundary_right, double boundary_bottom, double D, double Xmax, double Tmax, double dt, double dx);
38         vector<double> DufortFrankelScheme(vector<double>fnp1, vector<double>fn, vector<double>fnm1,double alfa,double nx,double Tmax, double dt );
39
40     };
41
42 ☐ class Richardson : public Scheme {
43     public:
44         Richardson();
45         Richardson(double boundary_left, double boundary_right, double boundary_bottom, double D, double Xmax, double Tmax, double dt, double dx);
46         vector<double> RichardsonScheme(vector<double>fnp1, vector<double>fn, vector<double>fnm1, double alfa, double nx, double Tmax, double dt);
47
48     };
49 ☐ class CrankNicholson : public Scheme {
50     public:
51         CrankNicholson();
52         CrankNicholson(double boundary_left, double boundary_right, double boundary_bottom, double D, double Xmax, double Tmax, double dt, double dx);
53         vector<double> CrankNicholsonScheme(vector<double>fnp1, vector<double>fn, vector<double>fnm1, double alfa, double nx, double Tmax, double dt);
54
55     };

```

```
56 ☐ class Laasonen : public Scheme {  
57     public:  
58         Laasonen();  
59         Laasonen(double boundary_left, double boundary_right, double boundary_bottom, double D, double Xmax, double Tmax, double dt, double dx);  
60         vector<double> LaasonenScheme(vector<double>fnp1, vector<double>fn, vector<double>fnm1, double alfa, double nx, double Tmax, double dt);  
61     };  
62 }
```

# DufortFrankel Class Reference

Inheritance diagram for DufortFrankel:



## Public Member Functions

**DufortFrankel** (double boundary\_left, double boundary\_right, double boundary\_bottom, double D, double Xmax, double Tmax, double dt, double dx)

vector< double > **DufortFrankelScheme** (vector< double >fnp1, vector< double >fn, vector< double >fnm1, double alfa, double nx, double Tmax, double dt)

➤ Public Member Functions inherited from **Scheme**

## Additional Inherited Members

➤ Protected Attributes inherited from **Scheme**

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

- **Scheme.h**
- **Scheme.cpp**

### Public Member Functions

[DufortFrankel](#)

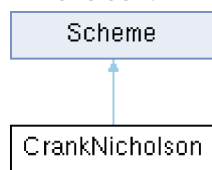
[DufortFrankelScheme](#)

[Additional Inherited Members](#)

[List of all members](#)

# CrankNicholson Class Reference

Inheritance diagram for CrankNicholson:



## Public Member Functions

**CrankNicholson** (double boundary\_left, double boundary\_right, double boundary\_bottom, double D, double Xmax, double Tmax, double dt, double dx)

vector< double > **CrankNicholsonScheme** (vector< double >fnp1, vector< double >fn, vector< double >fnm1, double alfa, double nx, double Tmax, double dt)

➤ Public Member Functions inherited from **Scheme**

## Additional Inherited Members

➤ Protected Attributes inherited from **Scheme**

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

- **Scheme.h**
- **Scheme.cpp**

### ▼ Public Member Functions

[CrankNicholson](#)

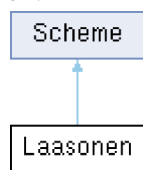
[CrankNicholsonScheme](#)

[Additional Inherited Members](#)

[List of all members](#)

# Laasonen Class Reference

Inheritance diagram for Laasonen:



## Public Member Functions

```
Laasonen (double boundary_left, double  
boundary_right, double boundary_bottom,  
double D, double Xmax, double Tmax,  
double dt, double dx)
```

```
vector< double > LaasonenScheme (vector< double >fnp1,  
vector< double >fn, vector< double  
>fnn1, double alfa, double nx, double  
Tmax, double dt)
```

› Public Member Functions inherited from **Scheme**

## Additional Inherited Members

› Protected Attributes inherited from **Scheme**

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

- **Scheme.h**
- **Scheme.cpp**

### Public Member Functions

[Laasonen](#)

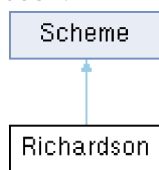
[LaasonenScheme](#)

[Additional Inherited Members](#)

[List of all members](#)

# Richardson Class Reference

Inheritance diagram for Richardson:



## Public Member Functions

**Richardson** (double boundary\_left, double boundary\_right, double boundary\_bottom, double D, double Xmax, double Tmax, double dt, double dx)

vector< double > **RichardsonScheme** (vector< double >fnp1, vector< double >fn, vector< double >fnn1, double alfa, double nx, double Tmax, double dt)

› Public Member Functions inherited from **Scheme**

## Additional Inherited Members

› Protected Attributes inherited from **Scheme**

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

- **Scheme.h**
- **Scheme.cpp**

▼ Public Member Functions

[Richardson](#)

[RichardsonScheme](#)

[Additional Inherited Members](#)

[List of all members](#)

# Chapter 7

## C++ Code

```
1 #include "Scheme.h"
2 #include <iostream>
3 #include <cmath>
4 #include <vector>
5 #include <iomanip>
6 #include <string>
7 #include <fstream>
8 #include <ctime>
9
10
11 using namespace std;
12
13 // Absolute error function
14 vector<double> errorabs(vector<double> solution, vector<double>
    fex)
15 {
16     vector<double> errorabs(solution.size());
17     for (int i = 0; i < solution.size(); i++)
18     {
19         errorabs[i] = abs(solution[i] - fex[i]);
20     }
21     return errorabs;
22 }
23
24 int main()
25 {
26     // Clock to see the compute time
27     clock_t debut = clock();
28
29     // Conditions setting
30     double boundary_left = 149.0;
31     double boundary_right = 149.0;
32     double boundary_bottom = 38.0;
33     double D = 93.0/60.0;
34     double Xmax = 31;
```

```

35     double dx = 0.05;
36
37     vector<double> Tmax = { 0.1*60.0, 0.2 * 60.0, 0.3 * 60.0, 0.4
38         * 60.0, 0.5 * 60.0 };
39
40     vector<double> dt = { 0.01, 0.025, 0.05, 0.1 };
41
42     // Loop to test schemes at 6, 12, 18, 24 and 30 min (cf
43     // Tmax vector)
44     for (int p = 0; p < Tmax.size(); p++)
45     {
46         // /\ Loop to test schemes with diferent delta t (cf
47         // dt vector)
48         // This loop is only useful for the Laasonen scheme
49         // -> You can uncomment this loop when testing
50         // Laasonen scheme (don't forget : "//}" at line 88)
51
52         //for (int q = 0; q < dt.size(); q++)
53         //{
54             // Schemes creation
55             // /\ The code has been created for only one
56             // schemes at time, so please use only one
57             // schemes uncommented
58             DufortFrankel scheme(boundary_left,
59                 boundary_right, boundary_bottom, D, Xmax, Tmax
60                 [p], dt[0], dx);
61             //Richardson scheme(boundary_left, boundary_right
62             // , boundary_bottom,D, Xmax, Tmax[p], dt[0], dx)
63             ;
64             //CrankNicholson scheme(boundary_left,
65             // boundary_right, boundary_bottom, D, Xmax, Tmax
66             // [p], dt[0], dx);
67             //Laasonen scheme(boundary_left, boundary_right,
68             // boundary_bottom, D, Xmax, Tmax[p], dt[q], dx);
69
70             // Alfa, initial vector and number of iterations
71             // recovery
72             vector<double> fnp1 = scheme.get_fnp1();
73             vector<double> fn = scheme.get_fn();
74             vector<double> fnm1 = scheme.get_fnm1();
75             vector<double> fex = scheme.get_fex();
76
77             double alfa = scheme.get_alfa();
78             int n = (Xmax / dx);
79             int nx = n + 1;
80
81             // Schemes call

```

```

69      // /\ The code has been created for only one
      schemes at time, so please use only one
      schemes uncommented

70
71      vector<double> solution = scheme.
          DufortFrankelScheme(fnp1, fn, fnm1, alfa, nx,
          Tmax[p], dt[0]);
72      //vector<double> solution = scheme.
          CrankNicholsonScheme(fnp1, fn, fnm1, alfa, nx,
          Tmax[p], dt[0]);
73      //vector<double> solution = scheme.
          RichardsonScheme(fnp1, fn, fnm1, alfa, nx,
          Tmax[p], dt[0]);
74      //vector<double> solution = scheme.LaasonenScheme
          (fnp1, fn, fnm1, alfa, nx, Tmax[p], dt[q]);

75
76      // Absolute error function call
77      vector<double> abserror = errorabs(solution, fex)
          ;

78
79      // Step, scheme solution and analytic solution
          display and export in a csv file
80      // /\ For Laasonen scheme, replace dt[0] by dt[q
          ]
81      string filename = "resultats_" + to_string(Tmax[p
          ]) + "_" + to_string(dt[0]) + ".csv";
82      ofstream file(filename);
83      file << "x;solution;fex;error" << endl;
84      cout << "Solution finale au temps t = " << Tmax[p
          ] << ":" << endl;

85
86      for (int i = 0; i < solution.size(); i++)
87      {
88          double x = i * dx;
89          cout << "i = " << x << " : u(x,T) = " <<
              solution[i] << " : ex(x,T) = " << fex[i]
              << " : error = " << abserror[i] << endl;
90          file << x << ";" << solution[i] << ";" << fex
              [i] << ";" << abserror[i] << endl;
91      }

92
93      file.close();
94  }
95  //}

96
97      clock_t fin = clock();

98
99      double duree = double(fin - debut) / CLOCKS_PER_SEC;
100

```

```
101     std::cout << "Times : " << duree << " seconds" << std::  
102         endl;  
103 return 0;  
104 }
```

Listing 7.1: Main.cpp

```

1 #include "Scheme.h"
2
3 // Empty scheme creation
4 // Nx and dx are not equal to 0 because we do not want to
   divide by 0 and we want vectors with a size > 0
5 Scheme::Scheme():boundary_left(0.0),boundary_right(0.0),
   boundary_bottom(0.0),Xmax(1.0),Tmax(1.0),D(0.0),dt(0.0),dx
   (1.0)
6 {
7     double alfa = (D * dt) / (dx * dx);
8     int n = Xmax / dx;
9     int nx = n + 1;
10
11     fn.resize(nx);
12     fnp1.resize(nx);
13     fnm1.resize(nx);
14     fex.resize(nx);
15
16     for (int i = 0; i < nx; i++)
17     {
18         fnm1[i] = boundary_bottom;
19         double sum = 0.0;
20
21         for (int m = 1; m < 10; m++)
22         {
23             sum+= exp(-D * (m * 3.14159 / Xmax) * (m * 3.14159 /
               Xmax) * Tmax)
24                 * (1 - pow(-1, m)) / (m * 3.14159)
25                 * sin(m * 3.14159 * i*dx / Xmax);
26         }
27         fex[i] = boundary_left + 2 * (boundary_bottom -
               boundary_left) * sum;
28     }
29     fn = fnm1;
30     fn[0] = boundary_left;
31     fn[nx-1] = boundary_right;
32 }
33
34 // Scheme creation
35 Scheme::Scheme(double boundary_left, double boundary_right,
   double boundary_bottom, double D, double Xmax, double Tmax
   , double dt, double dx):boundary_left(boundary_left),
   boundary_right(boundary_right), boundary_bottom(
   boundary_bottom), Xmax(Xmax), Tmax(Tmax), D(D), dt(dt), dx
   (dx)
36 {
37     double alfa = (D * dt) / (dx * dx);
38     int n = Xmax / dx;
39     int nx = n + 1;

```

```

40
41     fn.resize(nx);
42     fnp1.resize(nx);
43     fnm1.resize(nx);
44     fex.resize(nx);
45
46     // Analytic solution calculation
47     for (int i = 0; i < nx; i++)
48     {
49         fnm1[i] = boundary_bottom;
50         double sum = 0.0;
51
52         for (int m = 1; m < 9; m++)
53         {
54             sum += exp(-D * (m * 3.14159 / Xmax) * (m * 3.14159 /
55                 Xmax) * Tmax)
56                 * (1 - pow(-1, m)) / (m * 3.14159)
57                 * sin(m * 3.14159 * i * dx / Xmax);
58         }
59         fex[i] = boundary_left + 2 * (boundary_bottom -
60             boundary_left) * sum;
61     }
62     fn = fnm1;
63     fn[0] = boundary_left;
64     fn[nx - 1] = boundary_right;
65 }
66
67 // function to get the vector : Tn+1
68 vector<double> Scheme::get_fnp1() {
69     return fnp1;
70 }
71
72 // function to get the vector : Tn
73 vector<double> Scheme::get_fn() {
74     return fn;
75 }
76
77 // function to get the vector : Tn-1
78 vector<double> Scheme::get_fnm1() {
79     return fnm1;
80 }
81
82 // function to get the vector with analytic solutions
83 vector<double> Scheme::get_fex() {
84     return fex;
85 }
86
87 // function to get alpha
88 double Scheme::get_alfa() {

```

```

87     return (D * dt) / (dx * dx);
88 }
89
90 // Empty Dufort-Frankel scheme creation
91 DufortFrankel::DufortFrankel():Scheme(){}
92
93 // Dufort-Frankel scheme creation
94 DufortFrankel::DufortFrankel(double boundary_left, double
    boundary_right, double boundary_bottom, double D, double
    Xmax, double Tmax, double dt, double dx):Scheme(
    boundary_left, boundary_right, boundary_bottom, D, Xmax,
    Tmax, dt, dx)
95 {
96
97 }
98
99 // Dufort-Frankel resolution method
100 vector<double> DufortFrankel::DufortFrankelScheme(vector<
    double>fnp1, vector<double>fn, vector<double>fnm1, double
    alfa, double nx, double Tmax, double dt)
101 {
102
103     for (double t = 2*dt; t < Tmax + dt; t+=dt)
104     {
105         for (int i = 1; i < nx - 1; i++)
106         {
107             fnp1[i] = ((2 * alfa) / (2 * alfa + 1)) * (fn[i + 1] +
                fn[i - 1]) + ((1-2*alfa)/(1+2*alfa))*fnm1[i];
108         }
109         fnp1[0] = boundary_left;
110         fnp1[nx - 1] = boundary_right;
111         fnm1 = fn;
112         fn = fnp1;
113     }
114     return fnp1;
115 }
116
117 // Empty Richardson scheme creation
118 Richardson::Richardson() :Scheme() {}
119
120 // Richardson scheme creation
121 Richardson::Richardson(double boundary_left, double
    boundary_right, double boundary_bottom, double D, double
    Xmax, double Tmax, double dt, double dx) :Scheme(
    boundary_left, boundary_right, boundary_bottom, D, Xmax,
    Tmax, dt, dx)
122 {
123
124 }

```

```

125
126 // Richardson resolution method
127 vector<double> Richardson::RichardsonScheme(vector<double>
    fnp1, vector<double>fn, vector<double>fnm1, double alfa,
    double nx, double Tmax, double dt)
128 {
129
130     for (double t = 2*dt; t < Tmax + dt; t += dt)
131     {
132         for (int i = 1; i < nx - 1; i++)
133         {
134             fnp1[i] = 2*alfa*(fn[i+1]-2*fn[i]-fn[i-1])+fnm1[i];
135         }
136         fnp1[0] = boundary_left;
137         fnp1[nx - 1] = boundary_right;
138         fnm1 = fn;
139         fn = fnp1;
140     }
141     return fnp1;
142 }
143
144 // Empty Crank-Nicholson scheme creation
145 CrankNicholson::CrankNicholson() :Scheme() {}
146
147 // Crank-Nicholson scheme creation
148 CrankNicholson::CrankNicholson(double boundary_left, double
    boundary_right, double boundary_bottom, double D, double
    Xmax, double Tmax, double dt, double dx) :Scheme(
    boundary_left, boundary_right, boundary_bottom, D, Xmax,
    Tmax, dt, dx)
149 {
150
151 }
152
153 // Crank-Nicholson resolution method
154 vector<double> CrankNicholson::CrankNicholsonScheme(vector<
    double>fnp1, vector<double>fn, vector<double>fnm1, double
    alfa, double nx, double Tmax, double dt)
155 {
156
157
158     for (double t = dt; t < Tmax + dt; t += dt)
159     {
160         vector<double>d(nx), b(nx);
161         for (int i = 1; i < nx-1; i++)
162         {
163             b[i] = 1 + alfa;
164             d[i] = (alfa / 2) * fn[i-1] + (1 - alfa) * fn[i] + (
                alfa / 2) * fn[i+1];

```

```

165     }
166     b[0] = 1 + alfa;
167     d[0] = (1 - alfa) * fn[0] + (alfa / 2) * fn[1];
168     b[nx - 1] = 1 + alfa;
169     d[nx - 1] = (alfa / 2) * fn[nx - 2] + (1 - alfa) * fn[nx
        - 1];
170     double m;
171     double c = -alfa / 2;
172     for (int k = 1; k < nx; k++)
173     {
174         m = c / b[k - 1];
175         b[k] = b[k] - m*c;
176         d[k] = d[k] - m * d[k - 1];
177     }
178     fnp1[nx-1] = boundary_right;
179     fnp1[0] = boundary_left;
180     for (int k = nx - 2; k > 0; k--)
181     {
182         fnp1[k] = (d[k] - c * fnp1[k + 1]) / b[k];
183     }
184     fn = fnp1;
185 }
186 return fn;
187
188 }
189
190 // Empty Laasonen scheme creation
191 Laasonen::Laasonen() :Scheme() {}
192
193 // Laasonen scheme creation
194 Laasonen::Laasonen(double boundary_left, double
    boundary_right, double boundary_bottom, double D, double
    Xmax, double Tmax, double dt, double dx) :Scheme(
    boundary_left, boundary_right, boundary_bottom, D, Xmax,
    Tmax, dt, dx)
195 {
196
197 }
198
199 // Laasonen resolution method
200 vector<double> Laasonen::LaasonenScheme(vector<double>fnp1,
    vector<double>fn, vector<double>fnn1, double alfa, double
    nx, double Tmax, double dt)
201 {
202
203
204     for (double t = dt; t < Tmax + dt; t += dt)
205     {
206         vector<double>d(nx), b(nx);

```

```
207     for (int i = 0; i < nx; i++)
208     {
209         b[i] = 1 + 2*alfa;
210         d[i] = fn[i];
211     }
212     double m;
213     double c = -alfa;
214     for (int k = 1; k < nx; k++)
215     {
216         m = c / b[k - 1];
217         b[k] = b[k] - m * c;
218         d[k] = d[k] - m * d[k - 1];
219     }
220     fnp1[nx - 1] = boundary_right;
221     fnp1[0] = boundary_left;
222     for (int k = nx - 2; k > 0; k--)
223     {
224         fnp1[k] = (d[k] - c * fnp1[k + 1]) / b[k];
225     }
226     fn = fnp1;
227 }
228 return fn;
229 }
```

Listing 7.2: Scheme.cpp

```
1 #pragma once
2 #include <iostream>
3 #include <cmath>
4 #include <vector>
5 #include <iomanip>
6 #include <string>
7 #include <fstream>
8
9 // Mother class Scheme with all the parameter used for all
   schemes
10 using namespace std;
11
12 class Scheme {
13 protected:
14     double boundary_left;
15     double boundary_right;
16     double boundary_bottom;
17     double D;
18     double Xmax;
19     double Tmax;
20     double dt;
21     double dx;
22     vector<double> fn, fnm1, fnp1, fex;
23 public:
24     Scheme();
25     Scheme(double boundary_left, double boundary_right, double
        boundary_bottom, double D, double Xmax, double Tmax,
        double dt, double dx);
26     vector<double> get_fnp1();
27     vector<double> get_fn();
28     vector<double> get_fnm1();
29     vector<double> get_fex();
30     double get_alfa();
31
32 };
33
34 // Dufort-Frankel scheme inheritance
35 class DufortFrankel : public Scheme {
36 public:
37     DufortFrankel();
38     DufortFrankel(double boundary_left, double boundary_right,
        double boundary_bottom, double D, double Xmax, double
        Tmax, double dt, double dx);
39     vector<double> DufortFrankelScheme(vector<double>fnp1,
        vector<double>fn, vector<double>fnm1, double alfa, double
        nx, double Tmax, double dt );
40
41 };
42
```

```
43 // Richardson scheme inheritance
44 class Richardson : public Scheme {
45 public:
46     Richardson();
47     Richardson(double boundary_left, double boundary_right,
48               double boundary_bottom, double D, double Xmax, double
49               Tmax, double dt, double dx);
48     vector<double> RichardsonScheme(vector<double>fnp1, vector<
49               double>fn, vector<double>fnm1, double alfa, double nx,
50               double Tmax, double dt);
51 };
52 // Laasonen scheme inheritance
53 class Laasonen : public Scheme {
54 public:
55     Laasonen();
56     Laasonen(double boundary_left, double boundary_right,
57               double boundary_bottom, double D, double Xmax, double
58               Tmax, double dt, double dx);
57     vector<double> LaasonenScheme(vector<double>fnp1, vector<
59               double>fn, vector<double>fnm1, double alfa, double nx,
60               double Tmax, double dt);
61 };
62 // Crank-Nicholson scheme inheritance
63 class CrankNicholson : public Scheme {
64 public:
65     CrankNicholson();
66     CrankNicholson(double boundary_left, double boundary_right,
67               double boundary_bottom, double D, double Xmax, double
68               Tmax, double dt, double dx);
66     vector<double> CrankNicholsonScheme(vector<double>fnp1,
67               vector<double>fn, vector<double>fnm1, double alfa,
68               double nx, double Tmax, double dt);
67 };
68 }
```

Listing 7.3: Scheme.h