MSc in Aerospace Computational Engineering
Computational Methods & C++

Assignment

Irene Moulitsas & Peter Sherar

Cranfield University

Marc Barceló
Julio Maldonado
Group 1

December 9th, 2019

Content

List of Figures

List of Tables

List of Symbols

$\left(\frac{\partial T}{\partial t}\right)$	First derivative in time
$\left(\frac{\partial T}{\partial x}\right)$	First derivative in space
Δt	Time step
Δx	Space step
$\vec{a}$	Acceleration vector
D	Diffusivity
$\vec{F}$	External forces vector
k	Hooke Constant
m	Mass
t	Time variable
x	Space variable

List of Acronyms

1D	One Dimension
3D	Three Dimensions
AGC	Apollo Guidance Computer
CPU	Central Processing Unit
NASA	National Aeronautics and Space Administration
ODE	Ordinary Differential Equation
OOP	Object Oriented Programming
PDE	Partial Differential Equation
STL	Standard Library
TDMA	Tridiagonal Matrix Algorithm

Abstract

The aim of this document is to obtain the numerical results of the One-Dimensional Heat Transfer PDE of a given problem by using four different computational methods or schemes (DuFort-Frankel, Richardson, Laasonen, Crank-Nicolson). Since the analytical solution for the continuous problem can be obtained, it will serve as reference to calculate the deviation (errors) of the computational methods. Then, thorough comparison between them will be made in terms of accuracy, stability, and computational cost and then discussion of the results will be also made and briefly highlighted in the conclusion section.

Introduction

The objectives and procedures to achieve stated in the document's abstract could not be generated without the appropriate **tools**: at a first glance, hardware and software but also mathematical knowledge, English language, and a large series of other sources mainly driven by the authors' analysis and criteria.

According to Cambridge Dictionary, the highlighted word **tool** is defined as *"something that helps you to do a particular activity"*. So, should the reader allow us to apply this word to our field: science and engineering.

Although science's fuel is mainly curiosity, and its methodology is based on analysis, the tools required to understand our environment are -definitely- differential equations. Served as human-understandable expressions, they transcript and state elegantly the behaviour of a particular entity or phenomena through the variation of their involved parameters. Therefore, in the procedure of solving -with the appropriate integration method- those general equations, when the conditions of the particular problem are applied (boundary and initial conditions), a function is obtained as a solution and allow us to calculate any of the parameters involved within its boundary scope.

In physics and engineering, parameters primarily depend on the dimensions that humans are prone to handle (space, time) but also may vary according to other physical parameters. Thus, being subject to the dependence on dimensions or variables, differential equations can be split into:

·**Ordinary Differential Equation** (from now on, **ODE**). Its derivatives (changing variables) only depend on a single variable or dimension. The simplest and well-known example of ODE in the physics field is the movement of the simple harmonic motion (Hooke's Law). Using the 2nd Law of Newton:

$$\vec{F} = m\vec{a} = m \cdot \frac{d^2x}{dt^2}\ ; F = -k \cdot x \quad \rightarrow \ -k \cdot x = m \cdot \frac{d^2x}{dt^2} \tag{1}$$

Further information of this case and other examples of ODE can be checked at the website: https://ccrma.stanford.edu/~jos/DigitizingNewton/Mass_Spring_ODE.html

·**Partial Differential Equation** (from now on, **PDE**). In nature, most of its quantitative variables depend on the variation of other magnitudes so the most common scenario to be examined by physicians and engineers lead to solve PDEs rather than ODEs. According to (Phrasad and Ravindran, 1984), partial differential equations are *"a relationship between the function and its partial derivatives"*. Hence, their derivatives depend on two or more variables or dimensions.

The Heat Equation discussed in this document and solved through a variety of computational methods is -in fact- one of the most studied PDE's. Moreover, inside PDE's other classification could be made depending on the "appearance" of the equation and therefore the convergence and nature of the obtained results. This classification will be briefly discussed in the following lines.

Let's take a look at the general and linear second order equation preceded by some "constants" - being them called as in the general notation *A,B,C,D,E,F,G*- which will have a particular value on each faced problem:

$$A \cdot \frac{\partial^2 f}{\partial x^2} + B \cdot \frac{\partial^2 f}{\partial x\, \partial y} + C \cdot \frac{\partial^2 f}{\partial y^2} + D \cdot \frac{\partial f}{\partial x} + E \cdot \frac{\partial f}{\partial y} + F \cdot f = G \qquad (2)$$

The elegant procedure following has been extracted from (Grigoryan, 2010) and will be briefly shown on this document.

Compacting the terms that do not correspond to second order into a general constant "I" dependant of on those factors:

$$A \cdot \frac{\partial^2 f}{\partial x^2} + B \cdot \frac{\partial^2 f}{\partial x\, \partial y} + C \cdot \frac{\partial^2 f}{\partial y^2} + I(x, y, f, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, f) = 0 \qquad (3)$$

Not getting into deeper math, (which can be consulted on the reference book) the nature of the previous equation depends on the sign of the next factor, which is known as its "discriminant":

$$\Delta(\mathrm{x,y}) = \mathrm{B}^2\,(\mathrm{x,y}) - 4 \cdot \mathrm{A}(\mathrm{x,y}) \cdot \mathrm{C}(\mathrm{x,y}) \qquad (4)$$

Thus, the classification of PDEs is done according to:

·if the discriminant $\Delta(\mathrm{x,y})$ is > 0, the PDE will be known as a **hyperbolic** PDE.
·if the discriminant $\Delta(\mathrm{x,y})$ is equal to zero, we will consider it a **parabolic** PDE.
·at last, if $\Delta(\mathrm{x,y})$ is < 0, it will be defined as an **elliptic** PDE.

In order to understand the problem we are facing, a little literature on the basics of each type of PDEs will be attached. According to (Chapra and Canale, 1984):

· **Elliptic equations** *"are typically used to characterize steady-state systems. As in the Laplace equation, this is indicated by the absence of a time derivative. Thus, these equations are typically employed to determine the steady-state distribution of an unknown in two spatial dimensions."*

·**Parabolic equations** "*determine how an unknown varies in both space and time. This is manifested by the presence of both spatial and temporal derivatives in the heat conduction equation. Such cases are referred to as propagation problems because the solution "propagates,'' or changes, in time."*

· At last, **hyperbolic equations**, *"also deal with propagation problem. However, an important distinction manifested by the wave equation is that the unknown is characterized by a second derivative with respect to time. As a consequence, the solution oscillates."*

If some analysis of our initial problem is made – obtaining the solution in time and space for the heat equation, through numerical algorithms based on particular discretization- it can be seen either by the name of the equation (Heat Conduction Problem, the nature or through the calculus of the discriminant, that the 1-D equation given, is clearly a case of a parabolic PDE.

$$\frac{\partial T}{\partial t} = D \cdot \frac{\partial^2 T}{\partial x^2} \tag{5}$$

The analytic solution -which is actually given in order to be the reference for computational method's deviations or errors- can be obtained by solving the typical heat equation PDE with non-homogeneous boundary conditions. The D'Alembert solution, the Sturm-Liouville conditions and the decomposition into Fourier Series will be must-do parts of the analytic solution procedure. Nevertheless, this process is not of this document's interest and if the reader would like to know further about solving this type of PDEs, it is highly recommended to look at: (Smith, 2011)
Being the analytical solution of this problem:

$$\mathrm{T} = \mathrm{T_{sur}} + 2(\mathrm{T_{in}} - \mathrm{T_{sur}}) \sum_{m=1}^{m=\infty} \mathrm{e}^{-\mathrm{D} \cdot \left(\frac{\mathrm{m}\pi}{\mathrm{L}}\right)^2 \cdot t} \cdot \frac{1 - (-1)^{\mathrm{m}}}{\mathrm{m}\pi} \sin\left(\frac{\mathrm{m}\pi\mathrm{x}}{\mathrm{L}}\right) \tag{6}$$

Will provide us the value of temperature at any chosen time or space point.

However, it would be senseless to perform a solution without the best tools available in our days. Would anybody build a handmade car with sticks and stones? Since the XVII century where outstanding mathematicians such as Euler, Laplace, Fourier, etc. a great improvement has been made with the introduction of computation. Even a parallel part of maths' branch called numerical math has been created and highly used in most of engineering procedures. Quite every single object we possess has been created with a model (mathematical background and procedures) and built with automatic machines. Moreover, this way of proceeding has its clear explanation.

· By the one hand, the vast amount of calculations per second performed by any current computer exceeds by far the average of human's ability to perform them. In the recent past, in NASA's first missions enrolled on the "Cold War's Space Race", calculator was not an object but a job performed by a huge amount of humans who had to calculate most of the operation's dynamics by hand and then, it had to be reviewed since errors with those complex operations were easily made. Even the computer boarded in the Apollo 11's mission (the *AGC*) could calculate and provide any necessary navigation data, and its power was vastly lower than any recent low-cost cell phone. However, nowadays, it can be done by computer within relatively hugely less amount of time and effort. Additionally, risk of errors in "pure calculations" have been reduced to the minimum since they only depend on the precision -bits- that the computer is able to handle.

· By the other hand, and possibly being a more attractive idea than the previous insight, computers can calculate solutions, which are actually analytically unsolvable by humans. A classic example of it is the "3-Body Problem" which studies the dynamics of three mass points taking into account the gravitational and motion influence made between themselves. The solutions of this problem generally lead into chaotic results and only reliable information can be obtained through numerical methods; even more crucial if we consider that in our environment this problem arises to n-body.

Being specific in our career -aerospace and mainly fluid dynamics- the Navier-Stokes' equations (combined with continuity and energy principles) achieve to describe the behaviour of any particular case of fluids once "solved" the PDEs and applied correctly boundary and initial conditions. However, they are hard -in most of cases, impossible- to be integrated analytically and only "acceptable" results can be achieved through approximations and numerical calculation. In fact, due to the importance of its applications and mainly the intractability nature of the equations, Navier-Stokes' equations has been classified as one of the "Millennium Problems" and an award is provided to anyone capable of providing a new -but useful- perspective of them.

Once it is seen the importance of the computational methods in our current society, we would like to describe the procedure and schemes which will be used to numerically calculate our Heat Transfer Equation Problem. However, the accuracy and stability of numerical solutions cannot be exact by nature and may vary depending on several factors: differentiation schemes, implicit or explicit methods used, chosen increments, values of involved variables, etc. Briefly summarizing the information clarified in our "Mock Assignment" (Barcelo and Maldonado, 2019) numeric errors can be mainly classified into:

·Truncation errors: they take place when using approximation methods instead of exact analytic math functions.
·Round-off errors: they appear according to the way each computer displays the obtained results.

In order to evaluate both errors, the deviation from the analytical solution will be calculated and translated into a single representative number through the application of norms. These can also be seen in more detail in the Mock Assignment (Barcelo and Maldonado, 2019) Nonetheless, find attached here their main expression:

- $|x_1| = \frac{1}{n} \cdot \sum_{i=0}^{n} |x_i|$
- $|x_2| = \frac{1}{n} \cdot \sqrt{\sum_{i=0}^{n} x_i}$
- $|x_{inf}| = \max|x_i|$

Therefore, the value of the norms altogether with compilation time and memory expend, will help us to determine which of the four analysed computational methods might be the most appropriate for solving the one-dimensional Heat Transfer PDE.

Procedures

Computational Schemes Applied

The first thing to have in mind in computational methods is the fact that purely continuous problems cannot be solved: computers are only capable of managing grids of points and values so the problem known as “discretization” needs to be made. However, if we manage to perform a greatly-close but stable grid of points that describe correctly the continuous behaviour of the discretized problem, further analysis can be made like if the continuous problem was dealt. Nevertheless, some conditions need to be satisfied to rely on the results and will be discussed below.

Depending on the characteristics of accuracy and stability of the discretization we are looking, the first and second derivatives contained in the Heat Transfer Equation can be discretized (translated into computer-understanding language) in three main difference schemes: forward, backward and central derivative schemes. A deeper explanation on these concepts can be seen in our last assignment: (Barcelo and Maldonado, 2019).

Applying these different schemes (or even more explicit: combining them in the separate derivatives) will result in different accuracy and stability of the computational methods as well. That is why, in this assignment it will be seen how the different combination of the schemes into computational methods (DuFort-Frankel, Richardson, Laasonen and Crank-Nicolson) will vary numerical result for the Heat Transfer Equation.

The two first stated methods - DuFort-Frankel and Richardson - are known to be **“explicit methods”**. This means that the problem’s unknown value is single and isolated in one of the sides of the equation. In our problem, the unknown term will constitute the next time step value, calculated by the operation of different spatial but current time values of the function (which are obviously known). Therefore, once having all the spatial values in a particular (specifically: initial) time, a loop in space will be made in order to calculate all the next-time values for each spatial coordinate.

If a descriptive graph of the explicit method is made, figures known as stencils are obtained and help to recognize how to deal with the problem: applying implicit or explicit procedures.

Explicit methods' stencil usually look like a right side up triangle (shape of it may vary depending on the difference scheme applied) and accordingly demonstrate that the upper point (our isolate unknown) needs from the value of the lower to be calculated. Serve the following image as an example of explicit stencil: used forward difference scheme in time and central in space.

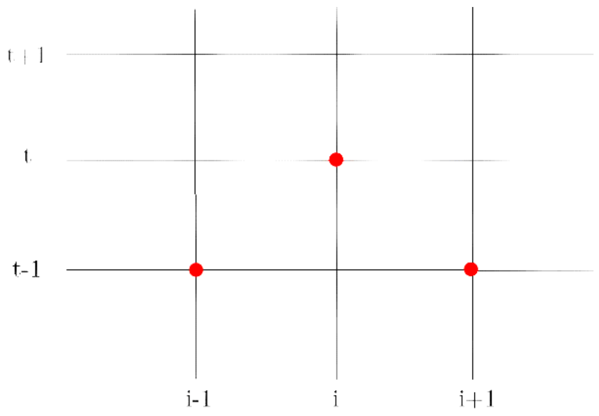


Figure 1.-Explicit methods' stencil

According to the theory learnt in the module, available in the slides from: (Moulitsas, 2019) , explicit methods entail fast and cheap (less operation) methods but may have issues in the both Von Neumann's and Courant's stability criteria.

In the other hand, **implicit methods** -such as Crank-Nicolson and Laasonen- involve more than a single unknown value and therefore need a system of equations to be solved. In order to face them in the most comfortable strategy, the unknowns are grouped in one side of the equation whilst the known values remain in the other side. Specifically, in our application the unknowns will be the next time steps involving consecutive space steps, and the known will be previous or current time steps but different consecutive space steps as well. Thus, a square matrix of unknown next-time steps constant terms (A) is built in the form:

$$\bar{\bar{A}} \cdot \bar{x} = \bar{b} \tag{7}$$

Being *x* the next time-step spatial solutions and *b* the current time spatial known terms. This system of linear equations can be solved through applying a wide variety of methods in which both linear algebra theorems and iteration properties are elegantly merged.

In consonance with this, the stencil of any implicit method will result in an inverted or upside down triangle in which the current space and time point will influence in future time and spatial points. For instance, an implicit stencil applying forward in time and central in space should look like the following figure:

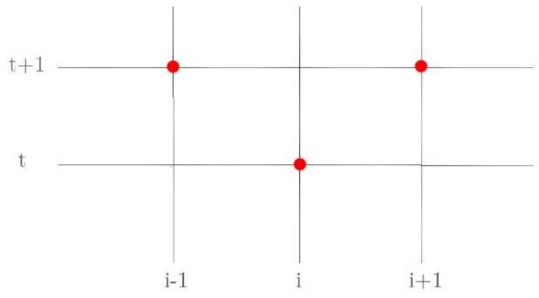


Figure 2.-Implicit methods' stencil

Considering the information shown in (Collis, 2005), the main advantage of using an implicit scheme is that the stability of the solution is quite always guaranteed. Nevertheless, the computational time and cost will be highly superior than an explicit method.

The choice of either using an explicit or implicit method is not always the same or predefined: it essentially depends on the faced problem, the expected accuracy and stability, on the available resources, the cost of time and money that can be taken, among other variables. Hence, a thorough previous analysis of the problem must be done.

The explicit and implicit methods that will be used in this document will be specified in the following lines.

<u>Richardson – Explicit</u>

This is the first of the explicit methods. It uses central difference scheme in both time and space, hence, the order of accuracy will be 2nd in both dimensions which is considerably better than if forward or backward schemes were used. Thus, if we apply this scheme to the Heat Transfer Equation, the Richardson scheme for this problem is obtained:

$$\frac{t_i^{n+1} - t_i^{n-1}}{2 \cdot \Delta\mathrm{t}} = \frac{D}{\Delta x} \cdot \left(t_{i+1}^n - 2 \cdot t_i^n + t_{i-1}^n \right) \tag{8}$$

And Richardson’s stencil can be obtained as well:

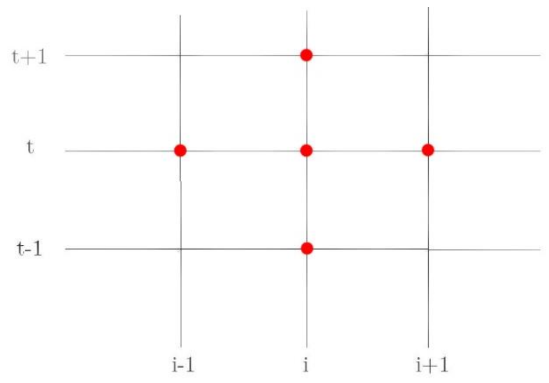


Figure 3.-Richardson method stencil

However, as it has been already stated, the stability in numerical methods must be also proven. Pure central difference schemes, although they provide better accuracy, are prove to drive the solution into instability. If the Von-Neumann Stability Criteria is reviewed, as it can be seen in: (Hellevik R., 2018.), the convergence of error is:

$$CE_{1,2} = -4 \cdot D \cdot \sin\left(\frac{\delta}{2}\right) \pm \sqrt{16 \cdot D^2 \cdot \sin^4\left(\frac{\delta}{2}\right) + 1} \tag{9}$$

As it can be seen, if -4·D·$\sin\left(\frac{\delta}{2}\right)$ is = 0, CE is equal to 0, but for all other values, |CE| is >0 for any value of the diffusion coefficient "D". Therefore, the Richardson method is unconditionally unstable.

DuFort-Frankel – Explicit

This explicit scheme achieves to turn the previous scheme (Richardson) into a stable method by approximating the t_i^n terms into $\frac{t_i^{n+1}+t_i^{n-1}}{2}$ and subsequently erasing a spatial-depending node as it can be seen either in the new expression or the DuFort-Frankel Scheme. Allow, from now on, to substitute the user-fixed and constant term $\frac{D \cdot \Delta t}{\Delta x}$ in the constant "r" and to show the final expression of the calculations:

$$t_i^{n+1} = \frac{t_i^{n-1} + r \cdot (t_{i+1}^n - t_i^{n-1} + t_{i-1}^n)}{1 + 2 \cdot r} \tag{10}$$

Accordingly, the DuFort-Frankel scheme will result in:

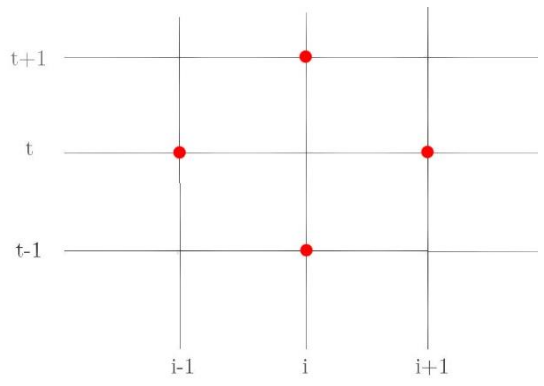


Figure 4.-DuFort-Frankel method stencil

If the Von-Neumann Analysis Criteria is evaluated (Ferziger and Peric, 2002) , the convergence of error is:

$$CE_{1,2}^{2} = \left|\frac{2D-1}{2D+1}\right| < 1 \tag{11}$$

Hence, the DuFort-Frankel is an explicit unconditionally stable method. Its only inconvenient, according to (Ferziger and Peric, 2002) is the fact that *"Surprisingly, the above approximation introduces another truncation error, which has the unusual property of being proportional to (Δt/Δx)^2. This term stems from the substitution and is an undesirable feature of the method because the method is consistent (that is, it yields a solution of the partial differential equation in the limit of small step sizes) only if Δt tends faster to zero than Δx."*

Laasonen Simple Implicit

This implicit method discretizes the Heat Transfer equation with forward difference scheme in time and central in space. Hence, once performed, our parabolic PDE should look like the following expression:

With this information, the Laasonen stencil is graphed in the following figure:

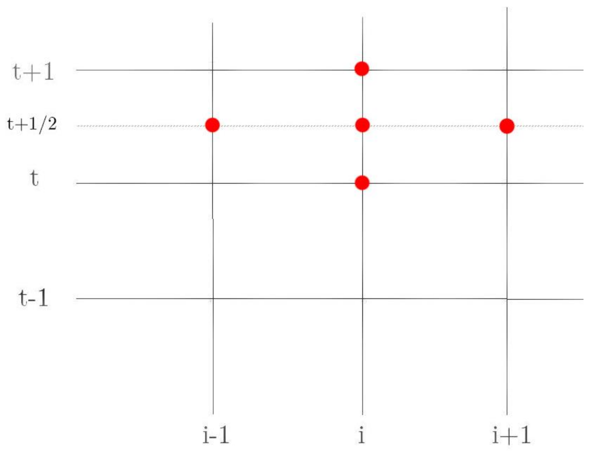


Figure 5.-Laasonen Simple Implicit method stencil

As it has already been pointed, any implicit scheme is unconditionally stable. However, computational cost will be outstandingly larger than any explicit method.

<u>Crank - Nicholson</u>

Also known as the trapezoidal method, the particularity of this implicit method is the fact that the accuracy of the solution is 2nd order for both space and time independent terms. According to (Collis, 2005) the discretization of the Heat Equation PDE casts the expression:

$$T_i^{n+1} - T_i^n = \frac{D \cdot \Delta t}{\Delta x^2} \cdot [T_{i+1}^{n+1} - 2 \cdot T_i^{n+1} + T_{i-1}^{n+1} + T_{i+1}^n - 2 \cdot T_i^n + T_{i-1}^n] \qquad (12)$$

Then, by substituting the term in (12) and displaying the next time step indexes in one side of the equation (the unknowns' side) and the current-time step indexes (theoretically their value is known), the following formula is obtained:

$$\begin{aligned} -r \cdot T_{i+1}^{n+1} + (1 + 2 \cdot r) \cdot T_i^{n+1} - r \cdot T_{i-1}^{n+1} \\ = -r \cdot T_{i+1}^n + (1 - 2 \cdot r) \cdot T_i^n - r \cdot T_{i-1}^n \end{aligned} \qquad (13)$$

As it can be seen, the matrix A obtained within this method is tridiagonal (banded, sparse) and some methods may be more appropriate and faster than general solvers such as pivoting LU, Jacobi, Gauss-Seidel… For instance, basic iterative solvers such as Thomas Algorithm (TDMA) or Cholensky Method may provide the less computational effort and memory consumption. As it will be seen, the Thomas Algorithm will have relatively good performance on our 1-D Heat Transfer PDE.
The stencil for Crank-Nicholson should look like the following graph:

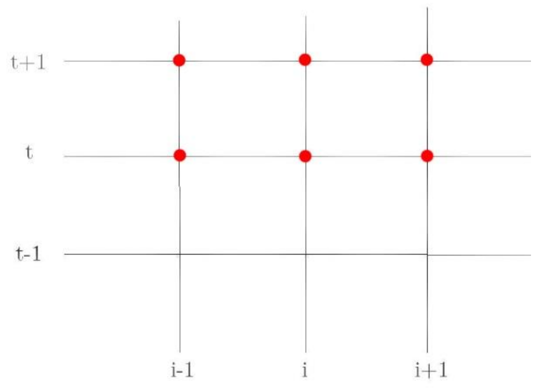


Figure 6.-Crank-Nicholson method stencil

The stability of the Crank-Nicolson Method is considered to be unconditionally unstable since it clearly is an implicit method.

Supposing a particular problem, once it is analysed that the implicit scheme is preferable than explicit in that case, the choice between Laasonen and Crank-Nicolson will depend on the accuracy of each's withdrawn solution. Here, a comparison for the 1-D Heat Transfer PDE will be done for both methods and the singularities and differences will be properly commented.

Methodology

Before introducing the methodology followed, shall the fundamentals of the given problem be introduced beforehand.

Suppose a One-Dimensional wall composed of nickel steel (40% Ni) with a diffusivity of D = 93 cm^2/hr. Its longitude is 31 cm and the initial temperature is 38 ºC. At one instant, the temperature in both sides arises instantly to 149ºC.

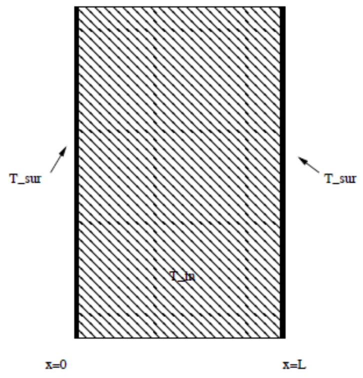


Figure 7.-Problem definition. Source: (Moulitsas, 2019)

Being aware that the PDE of the Heat Transfer Equation is (5) and its analytical solution is (6), a comparison between the 4 numerical methods needs to be done by varying time increments and the final time.

Hence, a program in C++ must be designed beforehand. Our idea is to create a "father" class (Abstract Solver) from which each different numerical solver class will inherit the general attributes. Therefore, the information from this general class Abstract Numerical Solver will be scant and two voids will configure the methods: One calculating the numerical solutions and the other to compare results with the analytical solution and withdraw norms. Since norms and analytical solutions will be common in all the inherited classes, three voids for norms and one double function with the analytical solution will be included in the "public" methods of this father class "Abstract Solver". Consequently, all 4 inherited classes (DuFort-Frankel, Richardson, Crank-Nicolson and Laasonen) will be performing the same two voids but with different instructions (math procedures in it), which is actually known as class polymorphism.

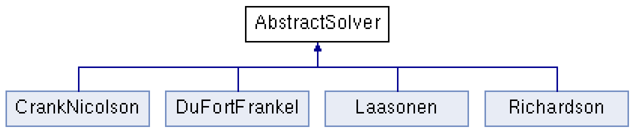


Figure 8. Class Inheritance.

The encapsulation of the attributes and procedures conveniently follows the general case: private or protected for internal variables of the class and public for procedures (voids or functions). It is important to remind that each class should have a class constructor definition and implementation and subsequently a class destructor too.

The use of the STL library (especially classes: vector and array) which is available in our compiler – “work environment” will help us to simplify the calculations but will also bust the readability of our code. This point may be important since in current businesses, tasks are divided into maintenance of code (and using of it) and coding. Anybody could have access to our code and reuse it or even improve it so clarity is an important factor for granted. Then, it is also significant in terms of troubleshooting and debugging – less processes can be needed, therefore less lines to examine and more chances to catch a mismatch.

The main bibliography used to acquire the C++ Programming (besides the module material) are: (Stroustrup, 2013), the website *cpp.org* , and (Bronson and Borse, 2010).

However, it may be necessary to use try catch-except procedures in order to ensure the correct performance of our program. Initially, we thought of involving three undesirable scenarios: having relatively low values, too high values that could make the solution fail; if using iterative solver such as Gauss-Seidel, Jacobi, etc. where values of matrix were 0 and could cause Infinite values in solution; and finally, errors produced by user or by trying to charge files that do not exist (see *compareTo*() void that charges ".txt" files from results). In the end, only the user-choice of computing norms has been needed to be implemented since Thomas Algorithm does not present the issue described with Jacobi-Gauss Seidel Algorithms and file creation-loading is implemented in the correct order.

Once designed this programme scheme for our code, we should discuss how we managed to make it work.

As it can be seen in the voids of the numeric solvers, the procedure to be applied is the same in both explicit methods and implicit methods. Only the discretization formula for each method may vary with the boundary conditions as well, especially in the implicit method.

For the explicit methods, after defining the known variables, vectors are created with the boundary and initial conditions. Then, a forward difference scheme is performed to calculate the first time and space point (with the appropriate loop). Once achieved, with two loops (one outer of time and the inside loop for space) the calculations of a matrix with the solutions is obtained. The values will depend on the discretization of the method:

· DuFort-Frankel: `f[i][n + 1] = (f[i][n - 1] + 2 * r * (f[i + 1][n] - f[i][n - 1] + f[i - 1][n])) / (1 + 2 * r)`

· Richardson: `f[i][n + 1] = f[i][n - 1] + 2 * r * (f[i + 1][n] - 2 * f[i][n] + f[i - 1][n])`

To conclude, the results are exported into a *".txt"* file by using the *ofstream* command (consequently, the *fstream* library must be included within the code).

Actually, two "*txt*" files are created for each computational method/class. It is due to the fact that one of them will be charged to the norms calculation (void compareTo()) and in order to read It properly, no Information about x and t must be shown. Hence, another file more user-friendly is also built up in case data wants to be checked.

Furthermore, in the implicit method, the definitions of variables and boundary conditions are similarly done. The main difference is the fact that now we have to solve a system of linear equations in the algebraic form A·x=b. To achieve this, being both A (tridiagonal constant matrix) and b (being the initial conditions firstly and then the results for the previous time-step) known, three "*for*" loops are done by using the Thomas Algorithm (TDMA). This method will be deeply explained In the Appendix 4. Furthermore, the boundary conditions and the vector of known values (b) of each method will be the only substantial change in the code:

· <u>Laasonen:</u>

For the second to the last time iteration, the vector b will be:

$b_{[1]}$ = sol[1][n - 1] + r * tsup;
$b_{[xstep - 1]}$ = sol[1][n - 1] + r * t_{sup};

· <u>Crank-Nicolson:</u>

Differently to Laasonen, the values of b obtained in the Crank-Nicolson's discretisation from the second to the last time iteration are:

$b_{[1]}$ = r * sol[2][n - 1] + (1 - 2 * r)*sol[1][n - 1] + 2 * r * tsup;
$b_{[xstep - 1]}$ = r * sol[xstep - 2][n - 1] + (1 - 2 * r)*sol[xstep - 1][n - 1] + 2 * r * 149;

Done this, the current space solutions are transferred to the b vector for the next time-step loop until the maximum time is reached.

The numerical methods applied have been just explained in the previous paragraphs. Additionally, the analytical solution can be calculated and actually provided so the deviation from the numerical methods with the real solutions can be easily achieved by subtracting the difference of each values. Here, norms will be applied to sum up in a single value the deviation from numerical methods to analytic solutions and will elegantly define the accuracy of the method. Since each norm somehow explains one particular trait, both 3 (norm 1, 2 and Infinite) will be computed and displayed in order to have a more complete perspective of the performance for each numerical method.

Results & Analysis

The results of the procedure explained in the sections above are presented in this section. The analytical solution of the problem is shown in order to compare the numerical methods and errors produced by them. Accuracy and errors produced by the computational methods below will be discussed.

Analytical Solution

First, before introducing the results of the numerical methods, the analytical solution is presented in order to later on, compare both. The time and space discretization chosen have been $\Delta x = 0.05$, and $\Delta t = 0.01$.

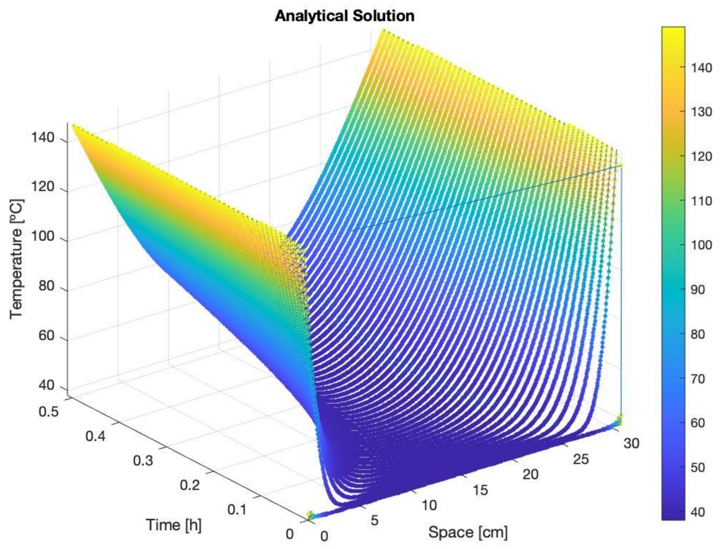


Figure 9.-Analytical solution 3D representation

In Figure 9, a 3-Dimensions representation of the analytical solution is shown. In X axis, space is represented with boundaries that go from 0 to 31 cm, in Y axis we have time in hours from 0 to 0.5 h, and in Z axis the temperatures is shown from 38 ºC to 149 ºC. The analytical solution states that the wall was initially at 38 ºC, and the two sides the temperature were then suddenly increased until 149 ºC. Then, the evolution of the temperature inside the wall increases as time rises up and spreads to the interior of the wall, which increases at every time step. To analyse the solution in more detail, the solution is printed for every X location every 0.1 h step from 0 to 0.5 h.

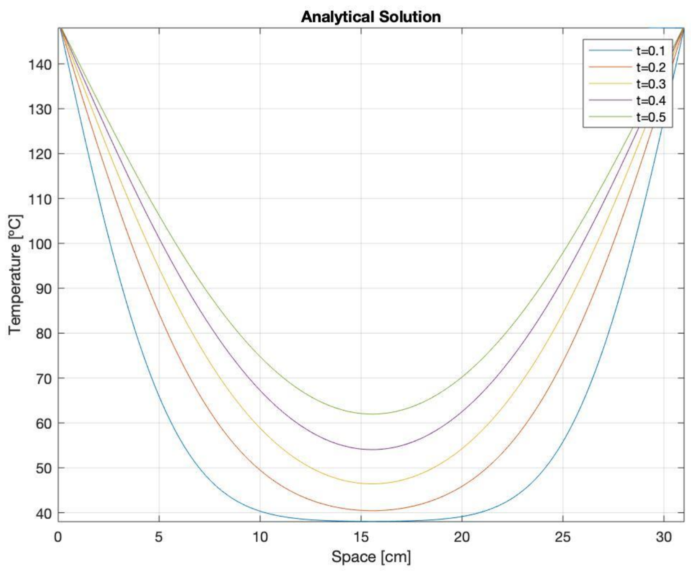


Figure 10.-Analytical Solution 2D representation

Figure 10 shows for every 0.1 h time step the behaviour of the temperature for every x position. As the graphic reveals, as time increases the temperature does so in all the x positions, being the central position of the wall the zone where the temperature increases in the slowest ratio.
As this is the analytical result, this is the exact solution of the problem, consequently the numerical results presented in the subsections below, will be compared to this one.

DuFort-Frankel Method

In this subsection, the results of first numerical method performed are shown. The DuFort-Frankel method is an unconditional stable explicit scheme (Chapra and Canale, 1984) , as explained in section the section of "Procedures". A 3D representation of the results obtained with this scheme is shown in Figure 11.

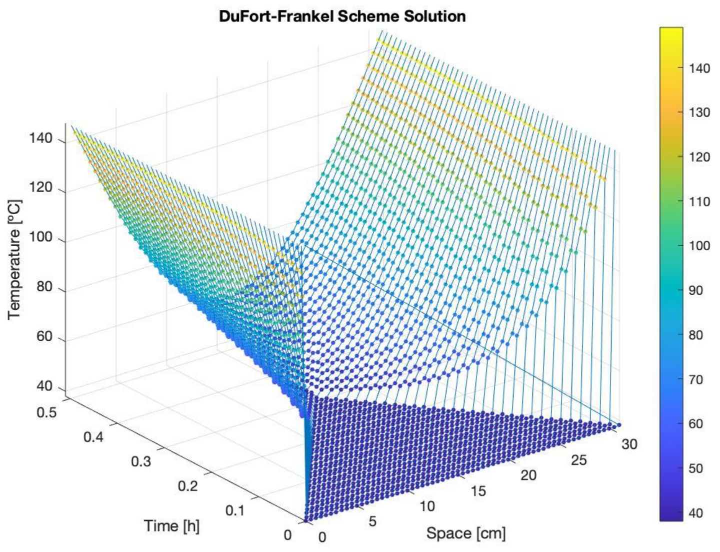


Figure 11.-DuFort-Frankel 3D representation

Figure 11 demonstrates the behaviour of the temperature along time and space. As It can be observed, the graphic behaviour is similar to the analytical, which is actually the exact solution. The main difference of this method is that the temperature propagation in the X axis is slower, and the central zone of the wall increases the temperature in a slower way too. However, a more in depth analysis is going to be done by printing the solution for every X location from 0 to 0.5 h with 0.1 h step.

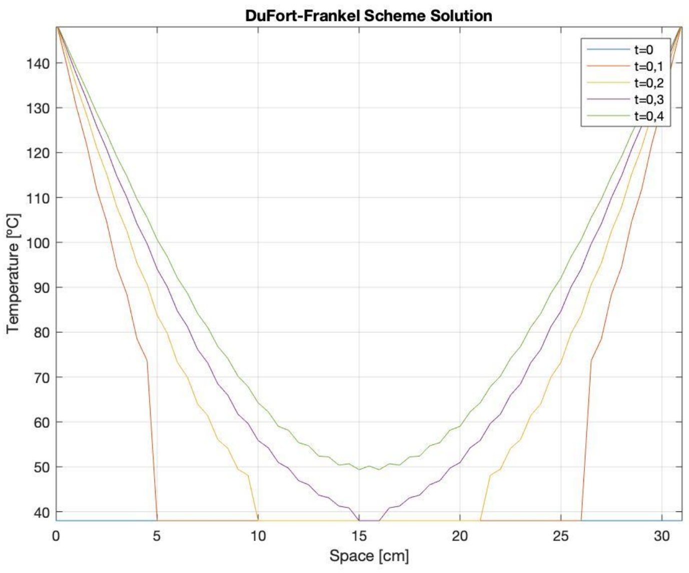


Figure 12.-DuFort-Frankel 2D representation

As it can be seen in Figure 12, the behaviour is similar to the expected one, however as said before, the temperature propagation at the X axis is slower than in the exact solution as can also be observed in the figure, consequently for the first time steps the temperature in the central zone of the X axis remains equals to the initial one. In order to evaluate the solution in a quantitative way, computation of the norms of the errors have been performed, as is shown in Table 1.

Table 1.-Norms results for DuFort-Frankel

Norm 1	2,9936
Norm 2	0,0305
Norm Inf	79,2420

The calculations in Table 1 shows that Norm 1, which is the average error of DuFort-Frankel scheme, tells that there are about 3ºC of error of difference between the results given and the exact solution in average. It can be considered a significant error, however is acceptable in comparison with the maximum error obtained which has a value of 79.242ºC.

Richardson Method

In this subsection, the problem will be solved with Richardson scheme. This is an explicit method, however as explained in Appendix 3: Richardson Method Study, this is an unconditionally unstable method, consequently the results presented are expected not to be similar at the analytical ones as time goes by. The 3 dimensions representation of the solutions is shown on Figure 13.

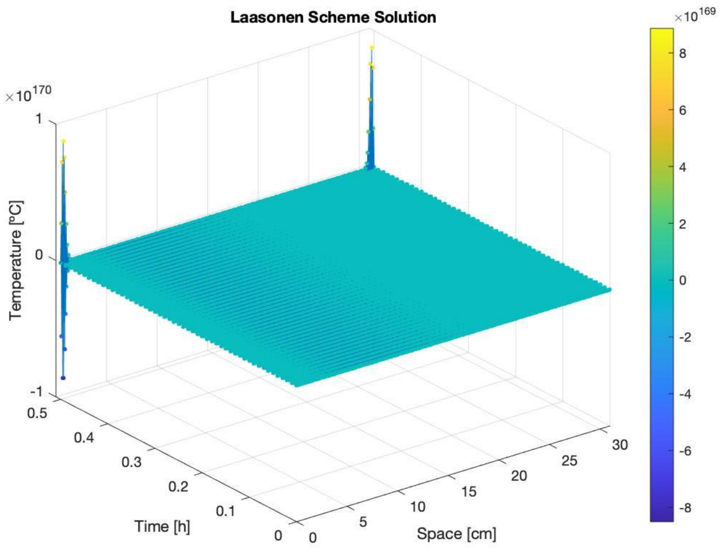


Figure 13.-Richardson results 3D representation

The graph above (Figure 13) details the results obtained with Richardson numerical scheme. As explained, the results are completely different to the analytical ones. In this case, the temperature seems not to evolve neither on the space or in the time axis, only present some temperature variations at the extreme values of space; however, those do not have any sense considering the definition of the problem. To analyse more deeply the behaviour of these results is going to be printed the solution for every X location every 0.1 h step from 0 to 0.5 h.

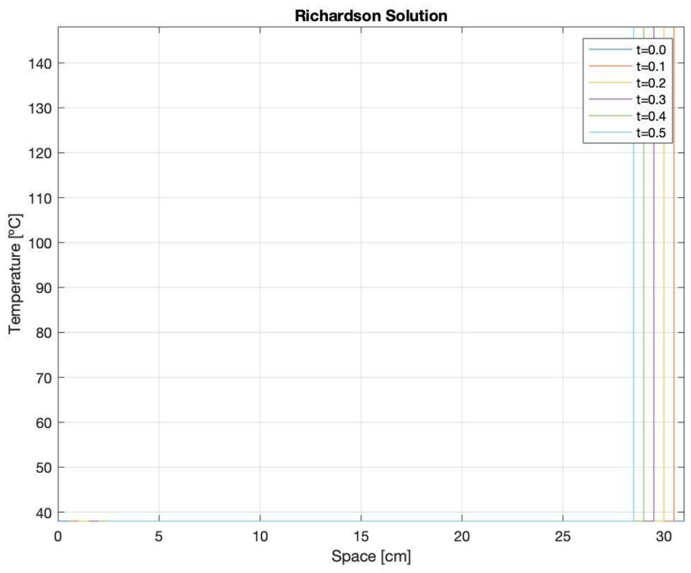


Figure 14.-Richardson 2D representation

Hence, Figure 14 reveals for every 0.1 h time step the behaviour of the temperature for every x position, and as expected by the results shown on Figure 13, there is no temperature evolution on the X axis, and neither on time axis, only some variation at extreme of the X axis. As expected, the results of this method is completely unstable and it is not possible to achieve an acceptable result, with similarity to the analytical one.

A quantitative analysis has also been performed in order to know the magnitude of the errors of the method. The calculations of the norms are presented in Table 2.

Table 2.-Norms results for Richardson

Norm 1	4,55e166
Norm 2	1,20e81
Norm Inf	8,86e169

The calculations performed in Table 2 shows that the error obtained with the Richardson scheme are huge as expected. The average error calculated is almost the same magnitude order than the maximum error obtained, this indicates, as already stated, that the results of this method are not trustworthy.

Laasonen Simple Implicit Method

Let the problem be solved with the Laasonen Simple Implicit numerical scheme. At its name says, it is an implicit method, which means that a linear system of equations have been solved in order to obtain the results, therefore the computational cost will be higher (Collis, 2005) . The results obtained with this scheme are presented in the following image.

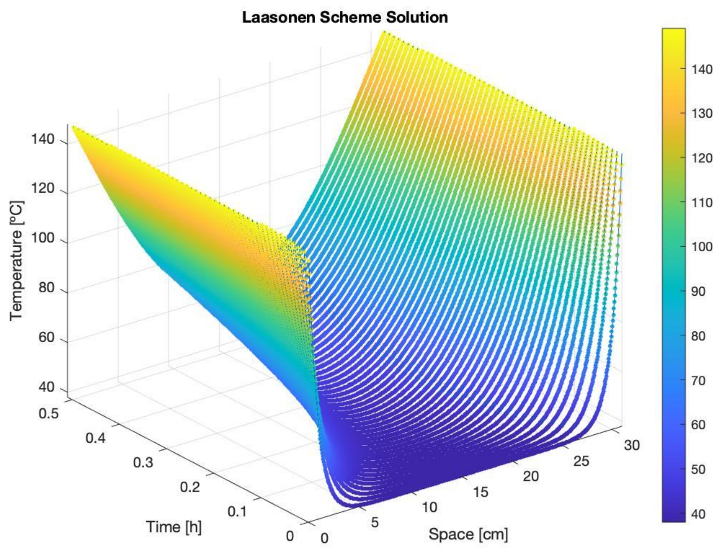


Figure 15.-Laasonen solution 3D representation

Figure 15 presents the Laasonen scheme results. As it can be seen, the solution really assembles the analytical: the temperature increases with the space and the time axis, which could mean this solution is more accurate than the other that have been analysed. To analyse more deeply the behaviour of these results is going to be printed the solution for every X location every 0.1 h step from 0 to 0.5 h.

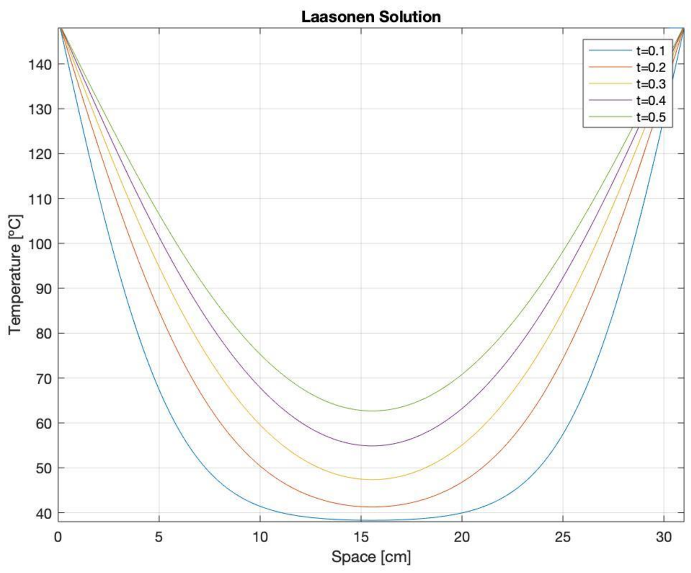


Figure 16.-Laasonen solution 2D representation

Figure 16 presents the results within a 0.5 h time step, and as can be observed the behaviour of the temperature is the expected one. The temperature is increasing along X axis until the centre of the wall from the sides of the wall, and it also increases by every time step. This solution is apparently, is very similar to the exact one, and in order to evaluate that in a more quantitative way, the calculation of errors is going to be performed.

Table 3.-Norms results for Laasonen

Norm 1	0,7130
Norm 2	0,0047
Norm Inf	12,6247

Table 3 shows the results of the computation of the norms. As can be observed, the average error obtained is in this case considerably little, is less than 1ºC of average error. The maximum error obtained computed by the Infinite Norm, is 12,6274 ºC which is high value, however, as the norm 1 is low value, and norm 2 tells that little errors are more numerous than bigger ones, it can be supposed that big errors are just punctual error cases, and the overall result is acceptable.

Crank-Nicholson Method

In this subsection, the problem will be solved by the Crank-Nicholson numerical scheme. As before, this is an implicit scheme which means that a linear system of equations have been solved in order to obtain the results. The results obtained with this method are presented in the following lines.

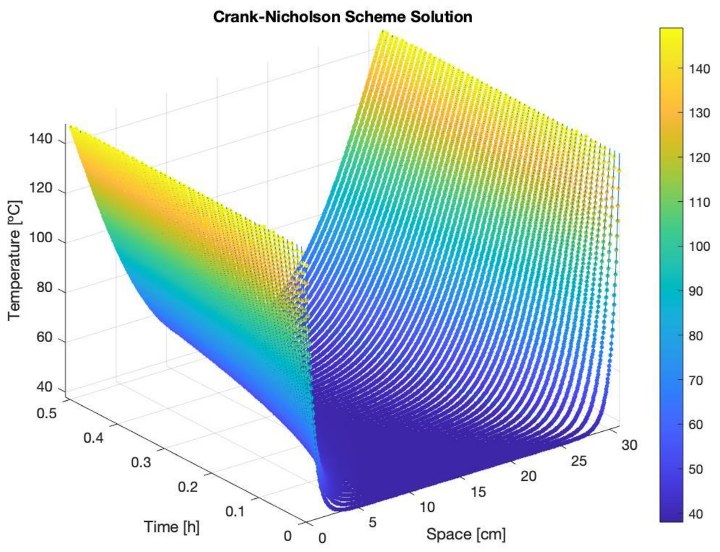


Figure 17.-Crank-Nicholson 3D representation

Figure 17 presents the Crank-Nicholson scheme results. It is shown how the solution is similar to the analytical: the temperature increases from the sides of the wall to the central zone, and it increases by every time step. However, the increase of the temperature along the time axis is slightly slower than in the analytical solution. Let's take a further sight into it from time t=0.1 h to 0.5 h.

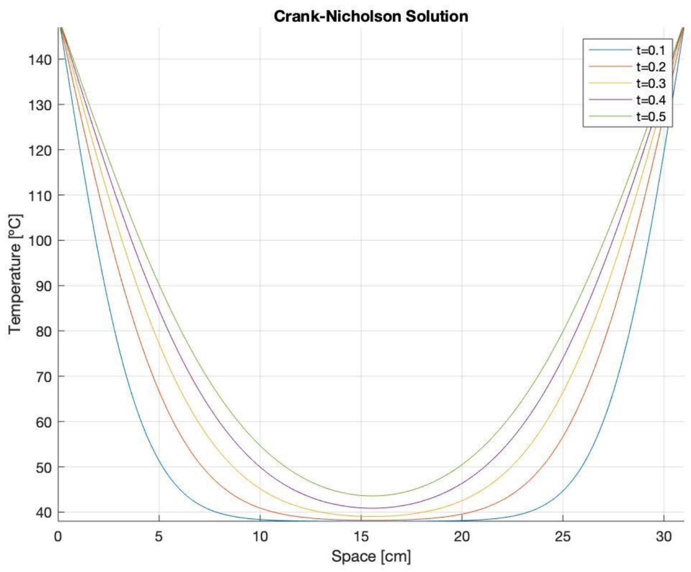


Figure 18.-Crank-Nicholson 2D representation

Figure 18 show more detailed how the behaviour of the temperature is along all values for every 0.5 h time step. It is revealed how temperature increases along the X-axis in a similar way as the analytical solution, however it is easily to see that the temperature increases slightly slower in time than the analytical solution. This solution is very similar to the exact one; nevertheless, in order to evaluate that in a quantitative way shall the norms be computed below:

Table 4.-Norms results for Crank-Nicholson

Norm 1	10,1711
Norm 2	0,0179
Norm Inf	20,1626

Table 4 shows the results of the calculations of the norms for the Crank-Nicholson scheme. Norm 1 is in this case significant, since reveals that there is an average error of about 10ºC with the exact solution. However, according to norm 2, it must be a punctual mismatch which should not be considered in our analysis. This mismatch (or for so few points) could be the highest what Infinite Norm shows of about 20ºC. These results tells that even the schemes seems to have a similar solution to the exact one, they also have significant errors, and its results have to be taken carefully. A study of both 3 norms should be performed to understand the nature and distribution of errors.

Effect of the step size on the accuracy

In order to the study the effect of the step size in the accuracy of the solution, it is going to be selected the Laasonen method- that has been demonstrated to achieve accurate results- and the steps sizes are going to be changed to see what changes are introduced. The study cases will be the presented in Table 5.

Table 5.-Accuracy study cases

Δt	0,01 h
Δt	0,025 h
Δt	0,05 h
Δt	0,10 h

To compare the different solutions of the Laasonen scheme, for different time steps, a calculation of the norms has been done for each study case. This will allow studying how the error changes with the variation of the step size, and consequently the accuracy of the solution according to that. Table 6 presents the results of these calculations.

Table 6.-Laasonen Norms results for different Δt

Δt = 0,01 h Δx = 0,05 cm	Norm 1	12.098
	Norm 2	0.09218
	Norm Inf	101.486
Δt = 0,025 h Δx = 0,05 cm	Norm 1	9.88199
	Norm 2	0.11493
	Norm Inf	106.792
Δt = 0,05 h Δx = 0,05 cm	Norm 1	9.024
	Norm 2	0.1613
	Norm Inf	109.577
Δt = 0,10 h Δx = 0,05 cm	Norm 1	8.79296
	Norm 2	0.27133
	Norm Inf	111.593

Before taking into consideration the data shown on Table 6, we will briefly mention what each norm should indicate. Norm 1 reveals the average error that could result from each point. Then, since Norm 2 is quadratically-averaged, it estimates the distribution of errors: Norm 2 value will be smaller if errors are punctual but will grow as they are normally distributed along our sample space. Finally, Norm Infinite states the max error in our model, the worst-case scenario.
As it can be observed in Table 6, Norm 1 indicates that the averaged error is approximately 10°C, but the small value Norm 2 reveals that this averaged norm comes from few points. If the 3-D graphs of each numerical method are seen, it makes sense: temperature does not propagate as much as it does in the analytic solution, that is why it is normal to have norm infinite relatively high when it is really close to the boundary. Thus, these values are reasonable.
As far as we can perceive, the results withdrawn make sense and seem to be reliable. As we loose precision by increasing Δt in factors of 2.5, 5 and 10; norms 2 and 3 demonstrate that errors become slightly bigger. However, the fact that less points are computed, may incurr in less generalized error, in case of norm 1.
Furthermore, by looking into the results, the errors are not so significant in terms of magnitude. Hence, if computational cost is the primary criteria, we could allow Δt to be 0.1 without loosing high quantity of precision.

Computational Time Comparison

Other interesting analysis that can be performed due the problem that has been solved in this report is the computational cost that each numerical method involve. In Table 7 the computational times of each method are presented for two different computers. It has been decided to compute the time in a couple of computers in order to see the time variation that it could be. Both computers specifications have been Included in Appendix 5.

Table 7.-Computational times of each numerical method

Method	University Computer Time	Personal Computer Time
DuFort-Frankel	0,014 s	0,021 s
Richardson	0,015 s	0,020 s
Laasonen Simple Implicit	1,192 s	0,956 s
Crank-Nicholson	1,314 s	1,169 s

As can be observed, the explicit methods have a very similar and lower duration than the implicit ones in both cases. This is easily understandable due for each implicit method, a linear system of equations has to be solved for each time step, hence this consumes a lot of time in comparison with the explicit schemes. In this particular case, the computational cost is about a hundred times bigger for both implicit methods. This indicates that depending on the problem that is faced, it would be preferably to not have such accurate results, and choose a method that gives results faster and without that huge requirement of memory.

Conclusions

The aim of this assignment was to solve the heat transfer first order partial derivation equation with different numerical methodologies, including explicit and implicit ones, using for that a C++ Object Oriented software.

In terms of creating the code for solving the present code, it can be concluded that an object-oriented code makes the program more efficient but also much understandable, and therefore easier to modify and maintain. The use of objects gets critical if processes must be repeated, imagine, a grid of particles in a fluid, which could not be programmed in another logic way.

As the results that sections above have shown, there is a clear difference between the results if using explicit or implicit method. As proved, implicit methods have produced significant more accurate results than the explicit ones, consequently, these methods are better in terms of accuracy, but they have a higher computational time, as has also been demonstrated.

Regarding the explicit methods, it can be concluded that DuFort-Frankel scheme give acceptable results with a low computational cost; Richardson scheme, as has been demonstrated is unconditionally unstable, and the results obtained from it were nonsense, as expected.

Looking now into the implicit methods, it can be observed that both have produced acceptable results, however, as said, at the sacrifice of a higher computational cost; being the Laasonen Simple Implicit Method the scheme which more accurate results has produced. Analysing in depth this method, it has been observed that step size has a big impact on the accuracy of the scheme. As the step size gets bigger, the accuracy of the scheme decreases, however, also the computational time decreases, consequently, a compromise solution could be found, in which not such accurate results were obtained, but get the results in a faster way.

However, as it has been posed along the document, the choice of the numerical method is not easy: it actually deals a compromise solution between computational time aimed (cost), and stability and accuracy of results desired. Therefore, the scope of the problem and its application determine which method may be the most appropriate.

Future work

The problem given: 1-D Heat Transfer PDE, assures the expansion and application of it through almost any thermodynamic subject. Thus, a lot of research and contributions can be done in the top of this basic example.

Due to this, if "heat transfer PDE" is introduced in the Scopus database, 789 results (documents) are presented related with this topic. If we consider that those documents represent a higher level and research study of the heat transfer applied to a real case than this basic document, it can be seen how far we can develop the problem given.

Nevertheless, it may be more appropriate to comment the next steps that could be done to improve the 1-D Heat Transfer PDE.

Firstly, it would make sense to translate it into the two or three space dimension which are more accurate to any real-life problem in which dimensions do not tend to infinite. Although in fluid dynamics, depending on the orders of magnitude of the Navier-Stokes equations' terms and satisfying the Pi theorem of Vaschy-Buckingham, it can be demonstrated that certain dimension may not introduce substantial changes compared to others and hence may be depreciated. The methodology to apply it would suppose clearly a different analytical solution, but in the computational environment, it would incur in the introduction of another space loop. Additionally, it may be also necessary to introduce a new term in the PDE for convective flows, which normally appear in non-idealised real problems.

Since some bibliographic research has been made to complete and back-up the discussion of results, we have found in (Chapra and Canale, 1984) , page 885, the solution for the two-dimensional problem in which the ADI method is presented as the most appropriate method to face the two-dimensional tridiagonal matrices.

Another improvement that could be applied would be the implementation of other iterative solvers or linear-algebra procedures to determine which is the fastest and less restrictive in terms of used memory. By the reasons exposed in the appendix (mainly that is simple to implement and fast), we have chosen the *Thomas Algorithm* to solve the implicit schemes, but according to (Golub and van Goal, 2013), the *Cholensky Method* would also be a strong candidate to be implemented due to its characteristics and suitability to the faced problem. Consequently, our current work would be considerably improved if the *Cholensky Method* and others with more complexity would be applied and compared to our Thomas Algorithm.

Finally, we are aware of the huge impact in terms of time that loops (for, while …) cause to CPU and it can be seen in the compilation time. That is why, our code could be highly boosted if loops could be reduced to the lowest number of them possible.

Acknowledgments

Marc: Firstly, I would like to thank Julio and to acknowledge him for his work done. He has always been helpful and collaborative, I think his discussion and analysis of results is excellent and I look forward to doing more module group assignments with him.
Since we had a really positive feedback for the mock assignment in which the PhD student Aaron Feria commented that was greatly performed, we decided to maintain the task division in the same way: both working in the plain code (I faced Implicit methods and Julio worked in explicits'); abstract, introduction, procedures and future work have been develop by myself whilst Julio faced the Results and conclusions. In terms of coding, I managed to implement the plain code for each numerical method into a single Object-Oriented Program, Julio translated arrays and vectors in STL format and then I performed the Doxygen comment.
I think that both of us, in our degree of "Aerospace Engineering", we have been trained to deal with any of those tasks and it would be relatively the same if we swapped the tasks. However, we had barely not much background in programming science and our final program could be simplified and more effective if it was coded with further knowledge.
The other sections were equally made collaboratively as both of us needed to reference points from appendixes.
Overall, despite it may be larger than the mock assignment and a bit tougher to read since the required knowledge to understand the problem is higher, I am proud of the final document. I hope the reader enjoys this document as much as we have had in the process of elaborating it (and especially when logical results were obtained).

Julio: First thing I would like to do is acknowledge Marc for doing this assignment with me and for all entire job he has done. The work to complete this assignment, has been in my opinion well balanced, as Marc and I have always share all the responsibilities, and both have discussed and agreed all the decisions and way of doing the present assignment.
The code done for solving the heat transfer PDE have been developed by both of us, and all the time, we were supporting each other until good results came. In my case after getting reliable results, I focused on creating a Matlab code to represent in a proper way the data obtained. In terms of the report, Marc has developed the abstract, introduction, procedures and methodology, performing an excellent job. Myself I have done all the results and analysis sections, including the conclusions and the Richardson scheme study included on the appendices.
To sum up, this assignment has been harder than the mock one, but lot of more effort has been put in, and I feel proud of the work we have performed.

References

Moin, P. (2010). *Fundamentals of Engineering Numerical Analysis.* Cambridge: Cambridge University Press - M.U.A.

Ferziger, J.H. and Peric, M. (2002) *Computational Methods for Fluid Dynamics*.

Grigoryan, V. (2010) "PARTIAL DIFFERENTIAL EQUATIONS," *Department of Mathematics University of California, Santa Barbara*

Phrasad, P. and Ravindran, R. (1984) "Partial Differential Equations," *John Wiley and Sons*

Smith, W. v (2011) *Elementary Partial Differential Equations*.

Barcelo, M. and Maldonado, J. (2019) "*Mock Assignment*"

Bronson, G.J. and Borse, G.J. (Garold J.) (2010) *C++ for engineers and scientists*. Thomson Course Technology.

Chapra, S.C. and Canale, R.P. (1984) *Numerical methods for engineers*.

Collis, S.S. (2005) *SANDIA REPORT An Introduction to Numerical Analysis for Computational Fluid Mechanics*.

Golub, G.H. and van Goal, C.F. (2013) *Matrix Computations*.

Lee, W.T. (n.d.) *Tridiagonal Matrices: Thomas Algorithm*. University of Limerick

Moulitsas, I. (2019) *Computational Methods-I*. Available at: Must specify DOI or URL if using the accessed field (Accessed: October 7, 2019).

Stroustrup, Bjarne. (2013) *The C++ programming language*.

Appendices

Appendix 1: C++ Code

"Numerical Methods.h"

```
#pragma once
#include <iostream>
#include <sstream>
#include <math.h>
#include <cmath>
#include <iomanip>
#include <fstream>
#include <array>
#include <fstream>
#include <ctime>
#include <time.h>
#include <stdio.h>
#include <exception>
using namespace std;

class AbstractSolver
{
        protected:
                double dx, dt, xmax, tmax, D, tini, tsup;
        public: // Methods - the ones that have polymorphism, are =0 and then defined in each
class
                AbstractSolver(double,double,double,double,double,double,double); // Class
constructor
                virtual void solve()=0;
                virtual void compareTo()=0;
                double analytic(double, double, double, double, double, double);
                void norm1(array<array<double, 621>, 621> &, double, double, double, double);
                void norm2(array<array<double, 621>, 621> &, double, double, double, double);
                void norminf(array<array<double, 621>, 621> &, double, double, double, double);
};

class DuFortFrankel : public AbstractSolver {
        public:
                DuFortFrankel(double, double, double, double, double,double, double); // Class
constructor
                ~DuFortFrankel();// Class destructor
                void solve();
                void compareTo();
};

class Richardson : public AbstractSolver{
        public:
                Richardson(double, double, double, double, double, double, double); // Class
constructor
                ~Richardson(); // Class destructor
                void solve();
                void compareTo();
};

class CrankNicolson : public AbstractSolver{
        public:
                CrankNicolson(double, double, double, double, double, double, double); // Class
constructor
                ~CrankNicolson(); // Class destructor
                void solve();
                void compareTo();
};

class Laasonen : public AbstractSolver {
public:
        Laasonen(double, double, double, double, double, double, double);
        ~Laasonen();
        void solve();
        void compareTo(); };
```

"Main.h"

```
#pragma once
#include "Numerical Methods.h"
```

"Methods.cpp"

```
#include "Numerical Methods.h"
#define PI 3.14159
// Constructor
AbstractSolver:: AbstractSolver(double _dx, double _dt, double _xmax, double _tmax, double _D, double _tini,
double _tsup) // Global variables for all comp methods( inherited classes)
{
	dx = _dx;
	dt = _dt;
	xmax = _xmax;
	tmax = _tmax;
	D = _D;
	tini = _tini;
	tsup = _tsup;
}
//Norms Abstract Solver
double AbstractSolver::analytic(double D, double xmax,  double x, double t, double tini, double tsup) { // Calculation
of analytical solution, common for all classes. It will be used in compareTo() void.
	double sum = 0;
	for (int k = 1; k < 1000; k++) {
		sum += exp(-D * pow(k * PI / xmax, 2) * t) * ((1 - pow(-1, k)) / (k * PI)) * sin(k * PI * x / xmax);
	}
	return tsup + 2 * (tini - tsup) * sum;
}
void AbstractSolver::norm1(array<array<double, 621>, 621>& error, double xmax, double tmax, double dt, double
dx) // Norm 1, common procedure for all classes in void compareTo()
{
	int xstep = int(xmax / dx);
	int tstep = int(tmax / dt);
	double sum = 0;
	double max = 0;
	double n;
	n = double(xstep) * double(tstep);

	for (int j = 0; j < tstep; j++)
	{
		for (int i = 0; i < xstep; i++)
		{
			sum = sum + error[i][j];
		}
	}

	cout<< "Norm 1: " << sum / n<<endl;
}
void AbstractSolver::norm2(array<array<double, 621>, 621>& error, double xmax, double tmax, double dt, double
dx) // Norm 2, common procedure for all classes in void compareTo()
{
	int xstep = int(xmax / dx);
	int tstep = int(tmax / dt);
	double sum = 0;
	double max = 0;

	for (int j = 0; j < tstep; j++)
	{
		for (int i = 0; i < xstep; i++)
		{
			sum = sum + abs(error[i][j] * error[i][j]);
```

```
                }

        }
        cout << "Norm 2: " << sqrt(sum) / (double(xstep) * double(tstep))<<endl;
}
void AbstractSolver::norminf(array<array<double, 621>, 621>& error, double xmax, double tmax, double dt, double
dx) // Norm Infinite, common procedure for all classes in void compareTo()
{
        int xstep = int(xmax / dx);
        int tstep = int(tmax / dt);
        double max = 0;
        for (int j = 0; j < tstep; j++)
        {
                for (int i = 0; i < xstep; i++)
                {
                        if (error[i][j] > max) { max = error[i][j]; }
                }

        }
        cout << "Norm Infinite: " << max<<endl;
}

// Inheritance Protected Values -> Constructing Each Inherited Class, Numerical Method
DuFortFrankel::DuFortFrankel(double _dx, double _dt, double _xmax, double _tmax, double _D, double _tini,
double _tsup) : AbstractSolver(_dx, _dt, _xmax, _tmax, _D, _tini, _tsup) { ; };
Richardson::Richardson(double _dx, double _dt, double _xmax, double _tmax, double _D, double _tini, double
_tsup) : AbstractSolver(_dx, _dt, _xmax, _tmax, _D, _tini, _tsup) { ; };
CrankNicolson::CrankNicolson(double _dx, double _dt, double _xmax, double _tmax, double _D, double _tini,
double _tsup) : AbstractSolver(_dx, _dt, _xmax, _tmax, _D, _tini, _tsup) { ; };
Laasonen::Laasonen(double _dx, double _dt, double _xmax, double _tmax, double _D, double _tini, double _tsup) :
AbstractSolver(_dx, _dt, _xmax, _tmax, _D, _tini, _tsup) { ; };



// DUFORT-FRANKEL
void DuFortFrankel::solve()
{
        unsigned clock0, clock1;
        clock0 = clock();
        double t = 0;
        double x = 0;
        double r = D * dt / (dx * dx);
        static array<array<double, 10000>, 10000> f;
        int xstep = int(xmax / dx);
        int tstep = int(tmax / dt);

        for (int i = 0; i < xstep+1; i++) {
                f[i][0] = tini;
        }
        for (int i = 1; i < tstep+1; i++) {
                f[0][i] = tsup;
                f[xstep][i] = tsup;
        }

        //for the first time step upwind
        for (int i = 1; i < xstep+1; i++) {
                f[i][1] = f[i][0] + r * (f[i + 1][0] - 2 * f[i][0] + f[i - 1][0]);
                f[xstep][1] = tsup;
        }
        for (int n = 1; n < tstep+1; n++)  // Varying time, constructing from previous solution
        {
                x = 0;
                for (int i = 1; i < xstep + 1; i++) { // Solving for all space indexes in current time
```

```
					f[xstep][n] = tsup;
					f[i][n + 1] = (f[i][n - 1] + 2 * r * (f[i + 1][n] - f[i][n - 1] + f[i - 1][n])) / (1 + 2 * r); // DuFort-
Frankel discretization
					x = x + dx;
				}
		}
		t = dt;
		x = 0;
		ofstream results("Results_DuFortFrankel.txt"); // Output
		ofstream outputcompare("Results_D.txt");
		for (int n = 1; n < tstep+1; n++) {

				x = 0;
				for (int i = 0; i < xstep+1; i++)
				{
					results << f[i][n] << " t= " << t << " x= " << x << " " << endl;
					outputcompare << f[i][n] <<" " /*<< " t= " << t << " x= " << x << " " << endl*/;
					x = x + dx;
				}
				outputcompare << endl;
				t = t + dt;
		}
		results.close();
		outputcompare.close();
		clock1 = clock(); // Computing execution time
		double extim = (double(clock1 -clock0) /1000);
		cout << " DuFort-Frankel time: " << extim << endl;
}

void Richardson::solve()
{
		unsigned clock0, clock1;
		clock0 = clock();
		double t = 0;
		double x = 0;
		double r = D * dt / (dx * dx);
		static array<array<double, 10000>, 10000> f;
		int xstep = int(xmax / dx);
		int tstep = int(tmax / dt);

		for (int i = 0; i < xstep+1; i++) {
				f[i][0] = tini;
		}
		for (int i = 1; i < tstep+1; i++) {
				f[0][i] = tsup;
				f[xstep][i] = tsup;
		}

		//for the first time step upwind
		for (int i = 1; i < xstep+1; i++) {
				f[i][1] = f[i][0] + r * (f[i + 1][0] - 2 * f[i][0] + f[i - 1][0]);
				f[xstep][1] = tsup;
		}

		for (int n = 1; n < tstep+1; n++)  // Varying time, constructing from previous solution
		{
				x = 0;
				for (int i = 1; i <= xstep + 1; i++) { // Solving for all space indexes in current time
					f[xstep][n] = tsup;
					f[i][n + 1] = f[i][n - 1] + 2 * r * (f[i + 1][n] - 2 * f[i][n] + f[i - 1][n]); // Richardson
discretization
					x = x + dx;
				}
		}
		t = dt;
		x = 0;
		ofstream results("Results_Richardson.txt");  // output results in txt file
		ofstream outputcompare("Results_R.txt");
		for (int n = 1; n < tstep; n++)
		{
			x = 0;
			for (int i = 0; i < xstep; i++)
			{
				results << f[i][n] << " t= " << t << " x= " << x << " " << endl;
				outputcompare<< f[i][n] <<" ";
				x = x + dx;
			}
			outputcompare << endl;
			t = t + dt;
		}
		results.close();
		outputcompare.close();
		clock1 = clock();
		double extim = (double(clock1 - clock0) / 1000);
		cout << " Richardson time: " << extim << endl;
}

void CrankNicolson::solve()
{
		double t = 0, x = 0;
		double r = D * dt / (2 * dx * dx);

		int xstep = int(xmax / dx);
		int tstep = int(tmax / dt);

		double maindiag = (1 + 2 * r);
		static array<double, 1000> supdiag;
		static array<double, 1000> infdiag;
		static array<double, 1000> b;
		static array<array<double, 1000>, 1000> sol;

		unsigned clock0, clock1;
		clock0 = clock();


		for (int n = 0; n < tstep + 1; n++) // TIME LOOP
		{
			for (int e = 1; e < xstep + 1; e++)
			{ // VECTOR DEFINITION

				if (n == 0)
				{
					b[e] = 38;
				}
				supdiag[e] = -r;
				infdiag[e + 1] = -r;
			}
			if (n == 0)
			{
				b[1] = tini + r * tsup;
				b[xstep - 1] = tini + r * tsup;
			}
			else
			{
				b[1] = r * sol[2][n - 1] + (1 - 2 * r) * sol[1][n - 1] + 2 * r * tsup;
				b[xstep - 1] = r * sol[xstep - 2][n - 1] + (1 - 2 * r) * sol[xstep - 1][n - 1] + 2 * r * 149;
		}

		// TRANSFORM according to Thomas Algorithm
		supdiag[1] = supdiag[1] / maindiag;
		b[1] = b[1] / maindiag;

		for (int i = 2; i < xstep - 1; i++) // Conversion of tridiagonal matrix into bidiagonal with main
diagonal with 1 value. b also changes.
		{
			supdiag[i] = supdiag[i] / (maindiag - supdiag[i - 1] * infdiag[i]);
			b[i] = (b[i] - b[i - 1] * infdiag[i]) / (maindiag - supdiag[i - 1] * infdiag[i]);
		}

		b[xstep - 1] = (b[xstep - 1] - b[xstep - 2] * infdiag[xstep - 1]) / (maindiag - infdiag[xstep - 1] *
supdiag[xstep - 2]);
		sol[xstep - 1][n] = b[xstep - 1];
		sol[xstep][n] = tsup;
		sol[0][n] = tsup;

		for (int i = xstep - 2; i > 0; i--) // BACKWARD SOLVING
		{
			sol[i][n] = b[i] - supdiag[i] * sol[i + 1][n];
			b[i] = sol[i][n]; // getting ready for next loop
		}
	}
	ofstream results("Results_Crank.txt");     // OUTPUT OF RESULTS
	ofstream outputcompare("Results_C.txt");
	for (int n = 0; n < tstep + 1; n++)
	{
		x = 0;
		for (int i = 0; i < xstep + 1; i++)
		{
			results << sol[i][n] << " t= " << t << " x= " << x << " " << endl;
			outputcompare << sol[i][n] << " ";
			x = x + dx;
		}
		outputcompare << endl;
		t = t + dt;
	}
	results.close();
	outputcompare.close();
	clock1 = clock();
	double extim = (double(clock1 - clock0) / 1000);   // Execution calculation time
	cout << " Crank-Nicolson time: " << extim << endl;
}
void Laasonen::solve()
{
	double t = 0, x = 0; // VAR DEFINITION
	double r = D * dt / (2 * dx * dx);

	int xstep = int(xmax / dx);
	int tstep = int(tmax / dt);
	double maindiag = (1 + 2 * r);
	static array<double, 1000> supdiag;
	static array<double, 1000> infdiag;
	static array<double, 1000> b;
	static array<array<double, 1000>, 1000> sol;
	unsigned clock0, clock1;
	clock0 = clock();


	for (int n = 0; n < tstep + 1; n++)  // TIME LOOP
	{
			for (int e = 1; e < xstep + 1; e++)
			{ // VECTOR DEFINITION

				if (n == 0)
				{
					b[e] = 38;
				}
				supdiag[e] = -r;
				infdiag[e + 1] = -r;
			}
			if (n == 0)
			{
				b[1] = tini + r * tsup;
				b[xstep - 1] = tini + r * tsup;
			}
			else
			{
				b[1] = sol[1][n - 1] + r * tsup;
				b[xstep - 1] = sol[1][n - 1] + r * tsup;
			}

			// TRANSFORM according to Thomas Algorithm
			supdiag[1] = supdiag[1] / maindiag;
			b[1] = b[1] / maindiag;

			for (int i = 2; i < xstep - 1; i++) // Conversion of tridiagonal matrix into bidiagonal with main
diagonal with 1 value. b also changes.
			{
				supdiag[i] = supdiag[i] / (maindiag - supdiag[i - 1] * infdiag[i]);
				b[i] = (b[i] - b[i - 1] * infdiag[i]) / (maindiag - supdiag[i - 1] * infdiag[i]);
			}

			b[xstep - 1] = (b[xstep - 1] - b[xstep - 2] * infdiag[xstep - 1]) / (maindiag - infdiag[xstep - 1] *
supdiag[xstep - 2]);
			sol[xstep - 1][n] = b[xstep - 1];
			sol[xstep][n] = tsup;
			sol[0][n] = tsup;

			for (int i = xstep - 2; i > 0; i--) // BACKWARD SOLVING
			{
				sol[i][n] = b[i] - supdiag[i] * sol[i + 1][n];
				b[i] = sol[i][n]; // getting ready for next loop
			}

		}
		// OUTPUT OF RESULTS
		ofstream results("Results_Laasonen.txt");
		ofstream outputcompare("Results_L.txt");
			for (int n = 0; n < tstep + 1; n++)
			{
				x = 0;
				for (int i = 0; i < xstep + 1; i++)
				{
					results << sol[i][n] << " t= " << t << " x= " << x << " " << endl;
					outputcompare << sol[i][n] << " ";
					x = x + dx;
				}
				outputcompare << endl;
				t = t + dt;
			}
		results.close();
		outputcompare.close();
		clock1 = clock();
		double extim = (double(clock1 - clock0) / 1000);
		cout << " Laasonen time: " << extim << endl;

}

// DESTRUCTORS
DuFortFrankel ::~DuFortFrankel() {}
Richardson ::~Richardson() { }
Laasonen::~Laasonen() {}
CrankNicolson::~CrankNicolson() {}

// compareTo() for each numerical method (inherited class).
void DuFortFrankel::compareTo()
{
		cout << "\n" << "Dufort-Frankel Norms: " << endl;   // Variable  Definition
		double x=0, t = 0;
		static array<array<double, 621>, 621> f;
		static array<array<double, 621>, 621> err;
		int xstep = int(xmax / dx);
		int tstep = int(tmax / dt);
		int i = 0, n=0, j = 0;
		ifstream results;

		results.open("Results_D.txt", ios::in);  //Charging results from txt file
		if (!results) {
				cerr << "unable to open file for reading" << endl;
		}

		for (j= 0; j< tstep; j++)
		{
				x = 0;
				for (i = 0; i < xstep; i++)
				{
						results>> f[i][j];
						err[i][j] = abs(f[i][j] - analytic(D, xmax, x, t, tini, tsup));  // Calculation of error (deviation
from analytic solution)
						x = x + dx;
				}

				t = t + dt;
		}
		results.close();
		norm1(err, xmax, tmax, dt, dx); // Calculating norms - father Abstract Solver void
		norm2(err, xmax, tmax, dt, dx);
		norminf(err, xmax, tmax, dt, dx);

}

void Richardson::compareTo()
{
		cout << "\n" << "Richardson Norms: " << endl;  // Variable  Definition
		double x = 0, t = 0;
		static array<array<double, 621>, 621> f;
		static array<array<double, 621>, 621> err;
		int xstep = int(xmax / dx);
		int tstep = int(tmax / dt);
		int i = 0, n = 0, j = 0;
		ifstream results;

		results.open("Results_R.txt", ios::in); // Charging results from txt file
		if (!results) {
				cerr << "unable to open file for reading" << endl;
		}
		for (j = 0; j < tstep; j++)
		{
			x = 0;
			for (i = 0; i < xstep; i++)
			{
				results >> f[i][j];
				err[i][j] = abs(f[i][j] - analytic(D,xmax,x,t,tini,tsup)); // Calculation of error (deviation 
from analytic solution)
				x = x + dx;
			}

			t = t + dt;
		}
		results.close();
		norm1(err, xmax, tmax, dt, dx); // Calculating norms - father Abstract Solver void
		norm2(err, xmax, tmax, dt, dx);
		norminf(err, xmax, tmax, dt, dx);

}
void CrankNicolson::compareTo()
{
		cout << "\n" << "Crank-Nicolson Norms: " << endl;  // Variable  Definition
		double x = 0, t = 0;
		static array<array<double, 621>, 621> f;
		static array<array<double, 621>, 621> err;
		int xstep = int(xmax / dx);
		int tstep = int(tmax / dt);
		int i = 0, n = 0, j = 0;
		ifstream results;
		results.open("Results_C.txt", ios::in); // Charging results from txt file
		if (!results) {
			cerr << "unable to open file for reading" << endl;
		}

		for (j = 0; j < tstep; j++)
		{
			x = 0;
			for (i = 0; i < xstep; i++)
			{
				results >> f[i][j];
				err[i][j] = abs(f[i][j] - analytic(D, xmax, x, t, tini, tsup)); // Calculation of error (deviation 
from analytic solution)
				x = x + dx;
			}
			t = t + dt;
		}

		results.close();
		norm1(err, xmax, tmax, dt, dx); // Computing norms - father Abstract Solver void
		norm2(err, xmax, tmax, dt, dx);
		norminf(err, xmax, tmax, dt, dx);
}
void Laasonen::compareTo()
{
		cout << "\n" << "Laasonen Norms: " << endl;  // Variable  Definition
		double x = 0, t = 0;
		static array<array<double, 621>, 621> f;
		static array<array<double, 621>, 621> err;
		int xstep = int(xmax / dx);
		int tstep = int(tmax / dt);
		int i = 0, n = 0, j = 0;
		ifstream results;
```

```
	results.open("Results_D.txt", ios::in); // Charging results from txt file
	if (!results) {
		cerr << "unable to open file for reading" << endl;
	}

	for (j = 0; j < tstep; j++)
	{
		x = 0;
		for (i = 0; i < xstep; i++)
		{
			results >> f[i][j];
			err[i][j] = abs(f[i][j] - analytic(D, xmax, x, t, tini, tsup));// Calculation of error (deviation 
from analytic solution)
			x = x + dx;
		}

		t = t + dt;
	}
	results.close();
	norm1( err, xmax, tmax, dt, dx); // Computing norms - father Abstract Solver void
	norm2( err, xmax, tmax, dt, dx);
	norminf(err, xmax, tmax, dt, dx);

}
```

"Main.cpp"

```
#include "main.h"

int main()
{
	char choice;

	//Syntax:: Method  Object (dx, dt,xmax,timemax,Diffusivity, temp_ini,temp_sup)
	DuFortFrankel Sol1(0.05, 0.01, 31, 0.5, 93, 38, 149);
	Sol1.solve();
	CrankNicolson Sol2(0.05, 0.01, 31, 0.5, 93, 38, 149);
	Sol2.solve();
	Richardson Sol3(0.05, 0.01, 31, 0.5, 93, 38, 149.);
	Sol3.solve();
	Laasonen Sol4(0.05, 0.01, 31, 0.5, 93, 38., 149.);
	Sol4.solve();

	cout << "do you want to compute norms? (y/n)" << endl;
	cin >> choice;

	try { // TRY-CATCH EXCEPT FOR RUNTIME ERROR - USER INTRODUCING ERROR
		if (choice != 'y' || choice != 'n')
			throw " You did not enter y/n. ";
	}
	catch (const char* e) { cout << "exception caught. " << e << endl; cout <<"Try 
again."<<"\n"<< "do you want to compute norms? (y/n)" << endl;
	cin >> choice;
	}

	if (choice=='y') {		// Computing norms if user decided to
		Sol1.compareTo();
		Sol2.compareTo();
		Sol3.compareTo();
		Sol4.compareTo();
	}

	system("pause");
	return 0;
}
```

Appendix 2: Matlab Code

This appendix includes the Matlab code developed to plot the images shown in the present report. Two different codes have been developed, the first one for the 3 dimension graphics, and the other one for 2 dimension plots that presents the solution for every X location every 0,1 h step from 0 to 0,5 h.

3D Plots Code

```
% Computational Methods & C++
% Assignment
% Cranfield University
% October 2019
% Marc Barcelo & Julio Maldonado

% This code imports txt files produced by C++ code to plot all the data
in
% 3D plots with color gradient in function of the temperature

% Import txt file
clear all;
filename = 'Results_01.txt';
delimiterIn = '\t';
headerlinesIn = 1;
A = importdata(filename, delimiterIn, headerlinesIn);

% Create the vectors for the variables
N = 3725; % Number of values
t = zeros(N,1); % Time vector
x = zeros(N,1); % Space vector
n = zeros(N,1); % Numerica solution vector
a = zeros(N,1); % Analytical solution vector
e = zeros(N,1); % Errors vector
j=1;

% Fill each vector with the different values of the data imported
for i=1:N

    t(i) = A.data(i, 1);
    x(i) = A.data(i, 2);
    n(i) = A.data(i, 3);
    a(i) = A.data(i, 4);
end

% Norms calculations
% Calculation of the error vector
for i=1:N
    if(t(i) ~= 0)
        e(j) = abs(n(i)-a(i));
    end
    j=j+1;
end

% Norm 1
norm1 = (1/N) * sum(abs(e));

% Norms 2
norm2 = (1/N) * sqrt(sum(e));

% Norm infinity
norminf = max(abs(e));

% Plots 3D
% Plot of the numerical solution
figure
plot3(x, t, n)
hold on
scatter3(x,  t, n, 10, n, 'filled')
colormap default
colorbar
xlim([0 31])
ylim([0 0.51])
zlim([38 148])
xlabel('Space [cm]')
ylabel('Time [h]')
zlabel('Temperature [∫C]')
title('Laasonen Scheme Solution')
grid on

% plot of the analytical solution
figure
plot3(x, t, a)
hold on
scatter3(x, t, a, 10, n, 'filled')
colormap default
colorbar
xlim([0 31])
ylim([0 0.51])
zlim([38 148])
xlabel('Space [cm]')
ylabel('Time [h]')
zlabel('Temperature [∫C]')
title('Analytical Solution')
grid on
```

2D Plots Code

```
% Computational Methods & C++
% Assignment
% Cranfield University
% October 2019
% Marc Barcelo & Julio Maldonado

% This code imports txt files produced by C++ code to plot
% the solution for every X location every 0,1 h step from 0 to 0,5 h

% Import the txt file
clear all;
filename = 'Results.txt';
delimiterIn = '\t';
headerlinesIn = 1;
A = importdata(filename, delimiterIn, headerlinesIn);
N = 13040;
s = 621;

% Create the vectors for the variables
x = zeros(s, 1); % Space vector
zero=zeros(s,1); % Vector for t=0.0
one=zeros(s, 1); % Vector for t=0.1
two=zeros(s, 1); % Vector for t=0.2
three=zeros(s,1 ); % Vector for t=0.3
four=zeros(s,1 ); % Vector for t=0.4
five=zeros(s,1); % Vector for t=0.5
j=1;
k=1;
l=1;
m=1;
n=1;
o=1;
```

```
% Loop for filling the vectors with the corresponding data

for i=1:N

    if A.data(i,1)==0
        x(i) = A.data(i,2);
        zero(o)=A.data(i,3);
        o = o+1;
    end

    if A.data(i,1)==0.1

        one(j)=A.data(i,3);
        j = j+1;

    end

    if A.data(i,1)==0.2

        two(k)=A.data(i,3);
        k = k +1;
    end

    if A.data(i,1)==0.3

        three(l)=A.data(i,3);
        l = l+1;
    end

    if A.data(i,1)==0.4

        four(m)=A.data(i,3);
        m = m+1;
    end

    if A.data(i,1)==0.5

        five(n)=A.data(i,3);
        n = n+1;
    end
end

%% 2D plots

% Plot the 2D graphic solution for every X location every 0,1 h step
% from 0 to 0,5 h

figure
plot(x, zero)
hold on
plot(x, one)
plot(x, two)
plot(x, three)
plot(x, four)
plot(x, five)
hold off
xlim([0 31])
ylim([38 147])
xlabel('Space [cm]')
ylabel('Temperature [∫C]')
title('Laasonen Solution')
grid on
legend('t=0.1','t=0.2','t=0.3','t=0.4','t=0.5')
```

Appendix 3: Richardson Method Study

This appendix includes all the mathematical calculations performed to study the accuracy and the stability of the Richardson method.

Richardson Method Accuracy Study

In this section, the accuracy of the Richardson scheme will be studied using the Taylor series. Starting from the discretisation equation of the heat equation for the Richardson method.

$$\frac{\mathrm{T}_i^{n+1} - T_i^{n-1}}{2\,\Delta t} = D\,\frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2} \tag{14}$$

Now, the Taylor Series will be applied to each element of the equation.

$$T_i^{n+1} = T_i^n + \left(\frac{\partial T}{\partial t}\right)_i^n \Delta t + \frac{\Delta t^2}{2}\left(\frac{\partial^2 T}{\partial t^2}\right)_i^n + O(\Delta t^3) \tag{15}$$

$$T_i^{n-1} = T_i^n - \left(\frac{\partial T}{\partial t}\right)_i^n \Delta t + \frac{\Delta t^2}{2}\left(\frac{\partial^2 T}{\partial t^2}\right)_i^n + O(\Delta t^3) \tag{16}$$

$$T_{i+1}^n = T_i^n + \left(\frac{\partial T}{\partial x}\right)_i^n \Delta x + \frac{\Delta x^2}{2}\left(\frac{\partial^2 T}{\partial x^2}\right)_i^n + O(\Delta x^3) \tag{17}$$

$$T_{i-1}^n = T_i^n - \left(\frac{\partial T}{\partial x}\right)_i^n \Delta x + \frac{\Delta x^2}{2}\left(\frac{\partial^2 T}{\partial x^2}\right)_i^n + O(\Delta x^3) \tag{18}$$

Replacing into the left side of equation (14).

$$\frac{T_i^n + \left(\frac{\partial T}{\partial t}\right)_i^n \Delta t + \frac{\Delta t^2}{2}\left(\frac{\partial^2 T}{\partial t^2}\right)_i^n + O(\Delta t^3) - T_i^n + \left(\frac{\partial T}{\partial t}\right)_i^n \Delta t - \frac{\Delta t^2}{2}\left(\frac{\partial^2 T}{\partial t^2}\right)_i^n + O(\Delta t^3)}{2\,\Delta t} \tag{19}$$

Now, replacing into the right side.

$$D\,\frac{T_i^n + \left(\frac{\partial T}{\partial x}\right)_i^n \Delta x + \frac{\Delta x^2}{2}\left(\frac{\partial^2 T}{\partial x^2}\right)_i^n + O(\Delta x^3) - 2T_i^n + T_i^n - \left(\frac{\partial T}{\partial x}\right)_i^n \Delta x + \frac{\Delta x^2}{2}\left(\frac{\partial^2 T}{\partial x^2}\right)_i^n + O(\Delta x^3)}{\Delta x^2} \tag{20}$$

Simplifying.

$$\left(\frac{\partial T}{\partial x}\right)_i^n - D\left(\frac{\partial^2 T}{\partial x^2}\right)_i^n = O(\Delta t^2, \Delta x^2) \tag{21}$$

The obtained equation is has an accuracy of order *$O(\Delta t^2, \Delta x^2)$.*

Richardson Method Stability Study

In this section, the stability of the Richardson scheme will be studied using the Fourier series. Starting from the discretisation equation of the heat equation for the Richardson method.

$$\frac{\mathrm{T}_i^{n+1} - T_i^{n-1}}{2\,\Delta t} = D\,\frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2} \tag{22}$$

Applying the Von Neumann stability analysis that defines that the numerical solution is equal to the analytical one plus a residual, as defined below.

$$f_i^n = F_i^n + r_i^n \tag{23}$$

Replacing into the discretised equation.

$$\frac{F_i^{n+1} + r_i^{n+1} - F_i^{n-1} - r_i^{n-1}}{2\,\Delta t} = D\,\frac{F_{i+1}^n + r_{i+1}^n - 2(F_i^n + r_i^n) + F_{i-1}^n + r_{i-1}^n}{\Delta x^2} \tag{24}$$

As the analytical solution, is an exact solution of the problem, its equality would be equal to zero, so it can be erased from the equation as shown below.

$$\frac{r_i^{n+1} - r_i^{n-1}}{2\,\Delta t} = D\,\frac{r_{i+1}^n - 2r_i^n + r_{i-1}^n}{\Delta x^2} \tag{25}$$

At this point, knowing that the definition of the Fourier series is the following one.

$$r_i^n = \sum_{-\infty}^{\infty} g^n(k) e^{ikx_i} \tag{26}$$

It can be replaced in equation (12).

$$\begin{aligned}&\frac{\sum_{-\infty}^{\infty} g^{n+1}(k)e^{ikx_i} - \sum_{-\infty}^{\infty} g^{n-1}(k)e^{ikx_i}}{2\,\Delta t}\\ &= D\,\frac{\sum_{-\infty}^{\infty} g^n(k)e^{ikx_{i+1}} - 2\sum_{-\infty}^{\infty} g^n(k)e^{ikx_i} + \sum_{-\infty}^{\infty} g^n(k)e^{ikx_{i-1}}}{\Delta x^2}\end{aligned} \tag{27}$$

Simplifying, and following assumptions below.

$$x_{i+1} = x_i + \Delta x \tag{28}$$
$$x_{i-1} = x_i - \Delta x \tag{29}$$
$$e^{x_{i+1}} = e^{x_i} \cdot e^{\Delta x} \tag{30}$$
$$2cos\emptyset = (e^{i\emptyset} + e^{-i\emptyset}) \tag{31}$$

$$r = \frac{D\,\Delta t}{\Delta x^2} \tag{32}$$

The following equation is achieved.

$$\frac{g^{n+1} - g^{n-1}}{g^n} = 4r[\cos(k\Delta x) - 1] \tag{33}$$

As equation (16) shows, the left side of the equation will be always negative, and by the stability requirements, it is required to be positive, consequently this method will be unconditionally unstable.

Appendix 4: Thomas Algorithm (TDMA)

This algorithm of "Gaussian elimination" will be used for the both implicit methods evaluated: Crank-Nicolson and Laasonen. The reason of having chosen TDMA instead of other algorithm is due to the efficiency and low demand of memory compared to other classical Iterative Methods such as Jacobi, Gauss-Seidel, etc.

Since our matrix of the linear equations to be solved, according to the discretization, is tridiagonal (which subsequently means that it is sparse too), only three vectors are needed to represent it instead of a whole matrix (and an additional for the independent term). In fact, after the transformation made by the algorithm, only 2 vectors will remain necessary for solving the equations' system.

According to: (Lee, n.d.) Thomas Algorithm is a type of LU decomposition (a decomposition into an upper triangular is made) and mainly consists of two steps.
Having our system of linear equations such as:

$$\bar{\bar{A}} \cdot \bar{x} = \bar{b} \tag{7}$$

Having "A" the physical appearance of *Figure 18*:

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \cdot & \\ & & \cdot & \cdot & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \cdot \\ \cdot \\ d_n \end{bmatrix}.$$

Figure 19. Tridiagonal Matrix of 6 space points. Source: https://www.cfd-online.com/Wiki/Tridiagonal_matrix_algorithm_-_TDMA_(Thomas_algorithm)

It is important to notice that the first but also the last row do contain two terms of three as it could be thought by using the discretization. However, in the first but also last space step, the terms of n and 0 are known and Imposed by the boundary conditions, so they are Included in the Independent vector b.

For the first row, we will overwrite the values of A's superior diagonal and independent term by their value divided by the main diagonal. Thus, when performing the next transformation to all the other Matrix's terms:

```
· supdiag[i] = supdiag[i] / (maindiag - supdiag[i - 1] *
infdiag[i]);

· b[i] = (b[i] - b[i - 1] * infdiag[i]) / (maindiag - supdiag[i -
1] * infdiag[i])
```

Being "I" the Index of the space Iteration, a two diagonal (one superior subdiagonal) matrix with the main diagonal full of 1 value will be obtained. It must be considered that since we are dividing the values of each side of the equation by the same value, no alteration of the results is made. Then, to obtain the result of the system of equations (vector x), only a backward substitution from n to 1 will be left to do. The term n, according to the matrix, will be:

```
b[xstep-1] = (b[xstep-1] - b[xstep - 2] *infdiag[xstep-1]) /
(maindiag - infdiag[xstep-1] * supdiag[xstep - 2])

sol[xstep-1][n] = b[xstep-1];
```

From n-1 to 1, this operation must be done to obtain the value regarding the next space-step value (since matrix A has a superior diagonal). Due to this, it is usually said that a backward substitution is performed.

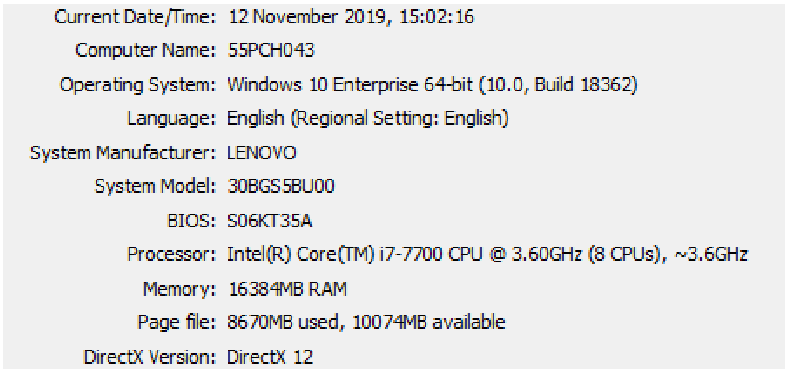
```
sol[i][n] = b[i] - supdiag[i] * sol[i + 1][n];
```

However, as It can be seen, the solution (vector x) is not a vector but a matrix as It represents the evolution of the spatial grid in a temporary lapse.

To obtain the whole matrix, the same process will be done for each time-step regarding that the solution for a time step will constitute the independent (known) term for the next time loop. Therefore, after the space loop, the vector b for the next time-step will be replaced by the "vector x" (actually, the matrix "solutions" with a fixed row of time).

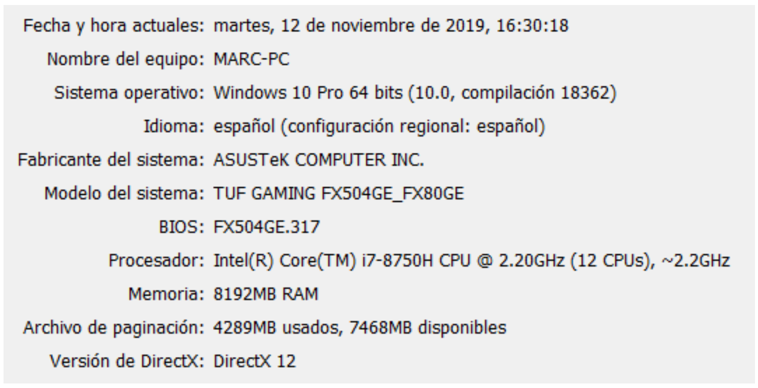
Appendix 5: Computational Time Comparison - Computers Specifications

This appendix includes the specifications of the computers used to make the computational time comparison. It is Interesting to evaluate them in order to see the effect of the properties of a certain computer in the resolution of the methods studied in this report. Both computer technics specifications can be observed below.



Current Date/Time:	12 November 2019, 15:02:16
Computer Name:	55PCH043
Operating System:	Windows 10 Enterprise 64-bit (10.0, Build 18362)
Language:	English (Regional Setting: English)
System Manufacturer:	LENOVO
System Model:	30BGS5BU00
BIOS:	S06KT35A
Processor:	Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz (8 CPUs), ~3.6GHz
Memory:	16384MB RAM
Page file:	8670MB used, 10074MB available
DirectX Version:	DirectX 12

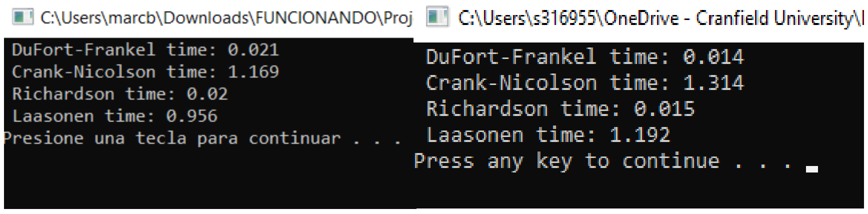
Figure 20.-University computer specifications



Fecha y hora actuales:	martes, 12 de noviembre de 2019, 16:30:18
Nombre del equipo:	MARC-PC
Sistema operativo:	Windows 10 Pro 64 bits (10.0, compilación 18362)
Idioma:	español (configuración regional: español)
Fabricante del sistema:	ASUSTeK COMPUTER INC.
Modelo del sistema:	TUF GAMING FX504GE_FX80GE
BIOS:	FX504GE.317
Procesador:	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz (12 CPUs), ~2.2GHz
Memoria:	8192MB RAM
Archivo de paginación:	4289MB usados, 7468MB disponibles
Versión de DirectX:	DirectX 12

Figure 21.-Personal computer specifications

As can be observed, both computers present similar processors which is the main elements in terms of solving computational problems. Consequently, the personal computer used for this test, which a last generation computer, has more modern CPU with a higher number of cores. This explains why in the test done on this report, the personal computer had lower computational times than the university one.



```
C:\Users\marcb\Downloads\FUNCIONANDO\Proj
DuFort-Frankel time: 0.021
Crank-Nicolson time: 1.169
Richardson time: 0.02
Laasonen time: 0.956
Presione una tecla para continuar . . .
```

```
C:\Users\s316955\OneDrive - Cranfield University\l
DuFort-Frankel time: 0.014
Crank-Nicolson time: 1.314
Richardson time: 0.015
Laasonen time: 1.192
Press any key to continue . . .
```

Figure 22. In the left image, Personal Computer Execution Time vs Execution time from Cranfield University Computer from IT Labs (right image)

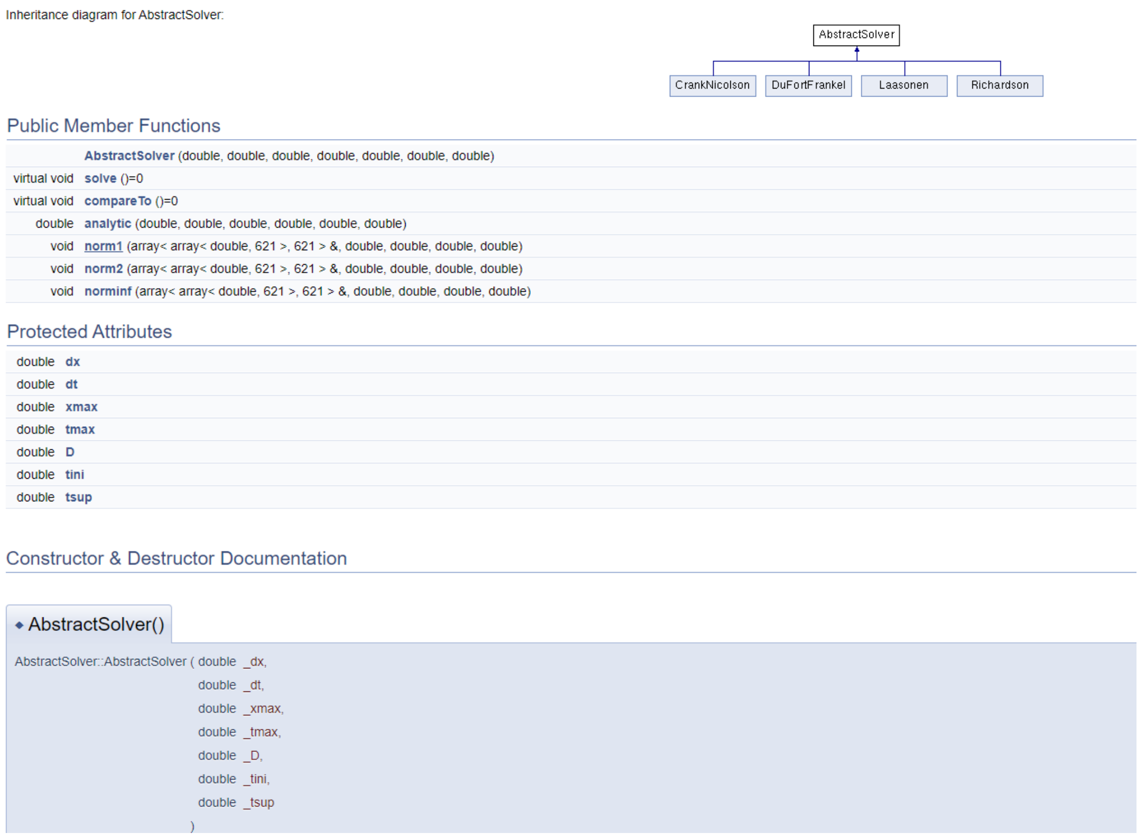
Appendix 6: Doxygen

The Doxygen Documentation has been performed with the app "Doxywizard" and two general folders of documentation files have been extracted: one in local web-based html and the other for LaTeX environment. Here, due to the user-simplicity and clarity, find a compressed file of the HTML generated documentation for our code.

html.7z

Apparently, this program was unable to comment in our cpp files (both *Main* and *Methods*) but completed this task for the header files (*main.h, Numerical Methods.h*). Nevertheless, variables, classes and procedures are defined in the header files and doxygen could analyse them properly.

The html files, as said, brilliantly give direct information of main classes and procedures of our file through menus and a user-friendly interface, with detailed Information about the relationship between all classes, their functions and the variables involved. For example, find attached one of the menus in which inheritance from our classes has been displayed.



Inheritance diagram for AbstractSolver:

Public Member Functions

	AbstractSolver (double, double, double, double, double, double, double)
virtual void	**solve** ()=0
virtual void	**compareTo** ()=0
double	**analytic** (double, double, double, double, double, double)
void	**norm1** (array< array< double, 621 >, 621 > &, double, double, double, double)
void	**norm2** (array< array< double, 621 >, 621 > &, double, double, double, double)
void	**norminf** (array< array< double, 621 >, 621 > &, double, double, double, double)

Protected Attributes

double	**dx**
double	**dt**
double	**xmax**
double	**tmax**
double	**D**
double	**tini**
double	**tsup**

Constructor & Destructor Documentation

◆ AbstractSolver()

```
AbstractSolver::AbstractSolver ( double  _dx,
                                 double  _dt,
                                 double  _xmax,
                                 double  _tmax,
                                 double  _D,
                                 double  _tini,
                                 double  _tsup
                               )
```

Figure 23. Documentation of Doxygen about our program: Classes