# Computional Method and C++ Assigment

Vincent Richard

02/11/17

# Contents

**1 Introduction** **2**
1.1 Presentation of the different methods used . . . 3
1.1.1 DuFort-Frankel . . . 3
1.1.2 Richardson . . . 3
1.1.3 Laasonen . . . 4
1.1.4 Cranck-Nicholson . . . 4

**2 Methods and Procedures** **5**
2.1 Code Structure . . . 5
2.2 Data Gathering . . . 9

**3 Result** **10**
3.1 Comparasion with the Analytical result . . . 10
3.2 Laasonen study . . . 14

**4 Conclusion** **14**

**5 References** **14**

**6 Appendice** **15**
6.1 Richardson Analysis . . . 15
6.2 Code . . . 16

# List of Figures

1 Representation of the initial problem . . . 3
2 General Architecture . . . 5
3 Explicit Architecture . . . 6
4 Implicit Architecture . . . 7
5 Analytical Result . . . 10
6 DuFort-Frankel Result . . . 10
7 Richardson Result . . . 11
8 Laasonen Result . . . 11
9 Cranck-Nicholson Result . . . 12
10 Absolute Error Calculation . . . 12
11 Relative Error Calculation . . . 13
12 Absolute Error of the Laasonen scheme plotted against the step size . . . 14

**Abstract**

In this assigment we are working on the heat equation, and more precisely on the computation of four schemes in order to find which one of them is the more relevant to use. We are studying two explicits schemes, DuFort-Frankel and Richardson, and two implicits schemes, Laasonen and Crank-Nicholson.

We collected the result of the computation on different time and different time step sizes in order to study the error change of each scheme. We found out that the Laasonen scheme is the more accurate. The longer the time is before collecting the result the smaller the error will be. Also there is a linear relation of $f(x) = \frac{1100}{12}x + 18$ between the size of the time step and the error introduce for the Lasonen scheme. So we can say regarding our result that Lasonnen is the best choice for calculation with the heat equation.

# 1 Introduction

In this assignment we are asked to examine the application of numerical schemes for the solution of partial diferential equations. Compute them and then analyse the accuracy of everyone of them. In order to do so we will consider the following probem.

A wall 1 ft. thick and infinite in other directions has an initial uniform temperature Tin of 100°F. The surface temperatures Tsur at the two sides are suddenly increased and maintained at 300°F. The wall is composed of nickel steel (40% Ni) with a diffusivity of D = 0,1 $ft^2/hr$. The governing equation to be solved is the unsteady one-space dimensional heat conduction equation, which in Cartesian coordinates is:

$$\frac{\partial T}{\partial t} = D\frac{\partial T}{\partial x^2} \tag{1}$$

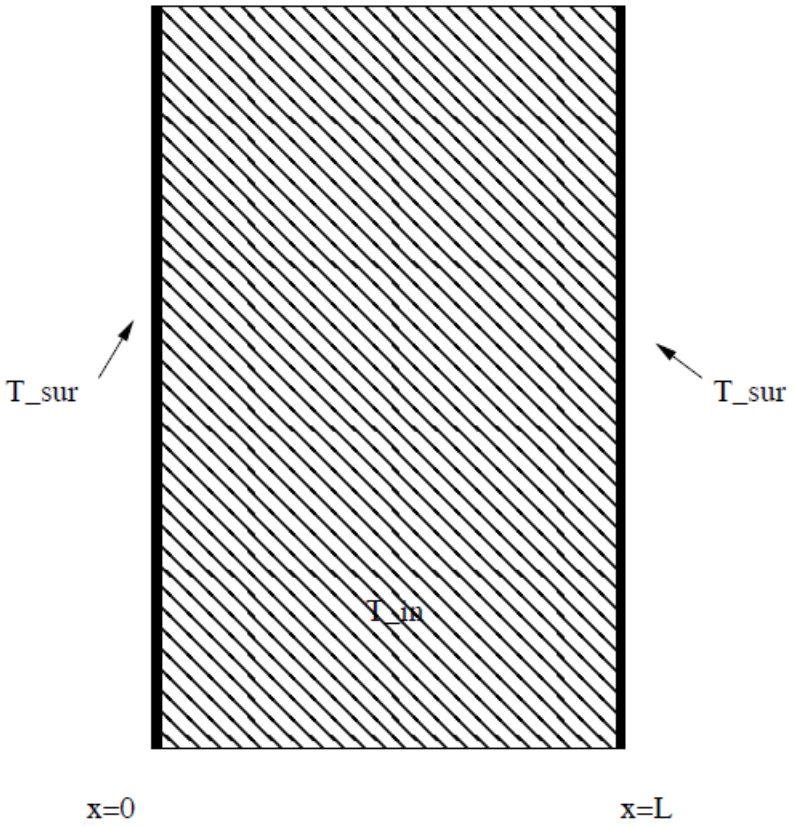


Figure 1: Representation of the initial problem

## 1.1 Presentation of the different methods used

### 1.1.1 DuFort-Frankel

The DuFort-Frankel scheme is an explicit scheme unconditionnaly stable of accuracy $O((\frac{\Delta t}{\Delta x})^2, (\Delta t)^2, (\Delta x)^2)$ the parabolic PDE is:

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

This equation leads to an explicit form which is:

$$T_i^{n+1}(1+2r) = T_i^{n-1} + 2r(T_{i+1}^n - T_i^{n-1} + T_{i-1}^n), r = \frac{D\Delta t}{\Delta x^2} \tag{3}$$

### 1.1.2 Richardson

The Richardson scheme is an explixit scheme, unconditionnaly unstable of accuracy $O((\Delta t)^2, (\Delta x)^2)$:

$$\frac{T_i^{n+1} - T_i^{n-1}}{2\Delta t} = D\frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2} \tag{4}$$

This equation leads to an explicit form which is:

$$T_i^{n+1} = 2r(T_{i+1}^n - 2T_i^n + T_{i-1}^n) + T_i^{n-1}, r = \frac{D\Delta t}{\Delta x^2} \tag{5}$$

### 1.1.3 Laasonen

The Laasonen scheme is an implicit scheme unconditionnaly stable of accuracy $O(\Delta t, (\Delta x)^2)$, that as for equation:

$$\frac{T_i^{n+1} - T_i^{n-1}}{2\Delta t} = D\frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2} \tag{6}$$

This equation leads to a form that result in a system of linear equation:

$$-rT_{i+1}^{n+1} + (1+2r)T_i^{n+1} - rT_{i-1}^{n+1} = T_i^n, r = \frac{D\Delta t}{\Delta x^2} \tag{7}$$

### 1.1.4 Cranck-Nicholson

The Crank-Nicholson scheme is an implicit scheme unconditionnaly stable of accuracy $O((\Delta t)^2, (\Delta x)^2)$, that as for equation:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \frac{D}{2}(\frac{T_{i+1}^{n+1} - 2T_i^{n+1} + T_{i-1}^{n+1}}{\Delta x^2} + \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2}) \tag{8}$$

This equation leads to a form that result in a system of linear equation:

$$-\frac{r}{2}T_{i+1}^{n+1} + (1+r)T_i^{n+1} - \frac{r}{2}T_{i-1}^{n+1} = \frac{r}{2}T_{i+1}^n + (1-r)T_i^n + \frac{r}{2}T_{i-1}^n \tag{9}$$

# 2 Methods and Procedures

## 2.1 Code Structure

To resolve the problem we decided to create an oriented object program base on those figures (for readibility reason I divided the structure in three graphics):

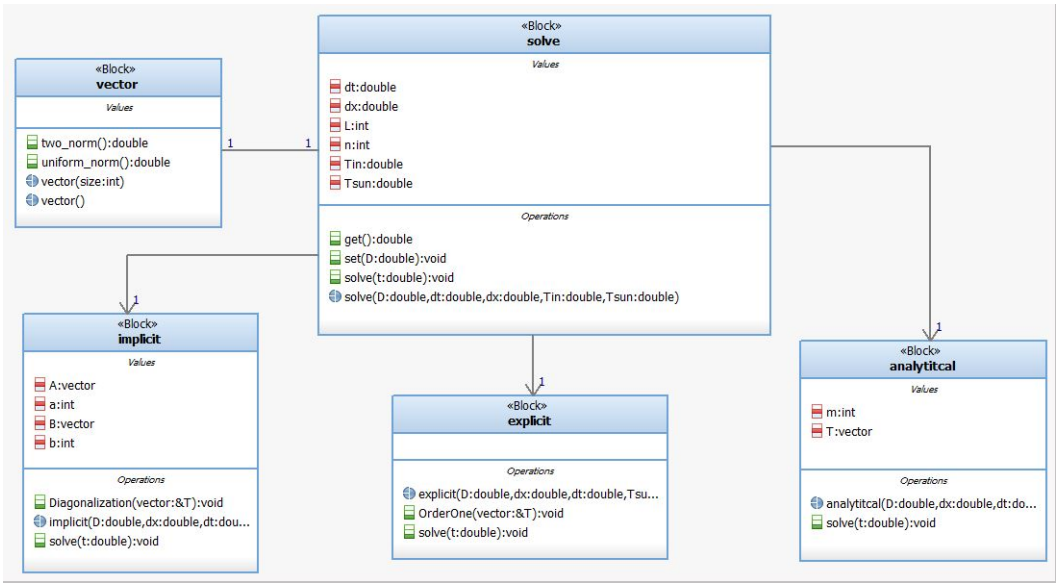


Figure 2: General Architecture

The based class is **Solve**. It has a number of argument:

- D the diffusivity which is in the assigment of $D = 0.1\ ft^2/hr$
- dt the gap in time between n and n+1
- dx the gap in space between i and i+1
- Tin the initial temperature of the wall here $Tin = 100°F$
- Tsun the temperature on the two sides that are maintained at $Tsun = 300°F$
- L the lenght of the wall which is fix to 1 ft. in this exercice
- n the number of possible position x for a fixed t, it calculated with $n = \frac{L}{dx}$

It also has couples of methods:

- Solve(double D, double dx, double dt, double Tsun, double Tin), the only constructor of this class and initialize the different value depending on the user input
- solve(double t) a virtual method that is created to be use in the derived class it will be the function that solve and print the result of the problem

- get() / set() methods, basic accessor methods

The **Solve** class also have three derived class **Implicit**, **Explicit**, and **Analytical**. The **Implicit** and **Explicit** class are further details below. We now will talk about the **Analytical** class.

The class **Analytical** calculate the analytical solution of the input problem. The class provided two new arguments:

- m is an integer used to simplify the expression of T since the anatical value of T is :

$$T = T_{sur} + 2(T_{in} - T_{sur}) \sum_{m=1}^{m=\infty} e^{-D(m\pi/L)^2 t} \frac{1-(-1)^m}{m\pi} sin(\frac{m\pi x}{L}) \quad (10)$$

  We need to get a smaller m for the upper border of the sum. We choosed to fix it at 10 000, the result obtain was an approximation under 0.5% so we decided it was accurate enough to do the simplification.

- T is a vector defined as an argument to to be able to call it from the main program

The **Analytical** class has also two methods:

- The constructor Analytical(...), is based on the constructor of the solve class, it also initialize the value of m and the vector T
- The solve function that is here defined to print the value of the analytical solution T of the problem with the approximation as explain above. It will print the value of T at each $10dt$hr until the argument t is reach

The **Vector** class is associated with the **Solve** class, it is use to defined vector in the derived class. The methods of the vector class:

- one_norm
- two_norm
- uniform_norm

are used to calculate each type of vector norm and will be of use to calculate the accuracy of each scheme.

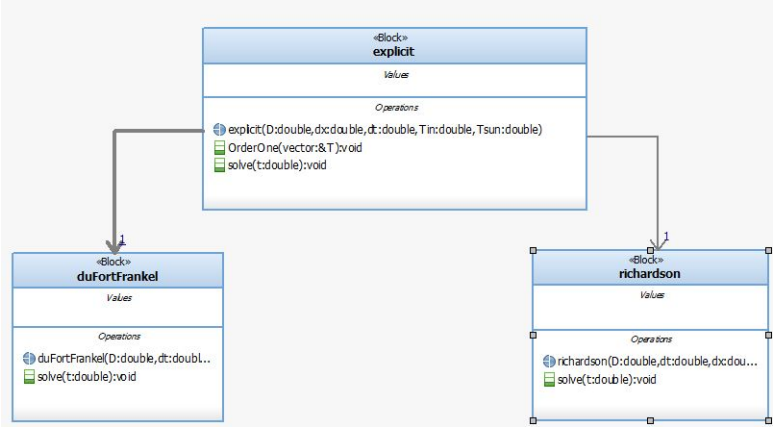

Figure 3: Explicit Architecture

The derived **Explicit** class is also a based class for two other class: the **DuFortFrankel** class and the **Richardson** class. Each of them solving the problem with the scheme of the same name.

The **Explicit** class define 3 new arguments:

- Tpast, T, and Tnext are vector declared in the explicit class because they will be used in both derived class. It is also to be able to use their values from the main class.

The **Explicit** class has the same methods as the **Solving**class. But it does define an aditionnal method.

The **OrderOne** method was created because in the two explicit schemes, DuFort-Frankel and Richardson, we encounter the issue of not being able to start the scheme at $n = 1$. To find the value of the temperature T at a time n we need the value of T at n and n-1 (See equation (3) and (5)). The problem set the values of T at $n = 0$ but we still need to find T at $n = 1$ to use those schemes. So we first need the method **OrderOne** to find T at $n = 1$ with a forward time central space scheme, and then apply the Richardson and DuFort-Frankel scheme to find the other values of T.

Since this method is useful for both derived class, we decided to declare it in the **Explicit** class.

The two derived classes **DuFortFrankel** and **Richardson** work the same way.

They have the same constructor method based on the one defined in the **Explicit** class and they both defined the method **solve** in a very similar way.

Once the initialisation of T at $n = 1$ (with the **OrderOne** method) is done, we use a loop, depending on the scheme, the equation (3) or (5) will be use to iterate the values of T until we reach the value of t wanted by the user.

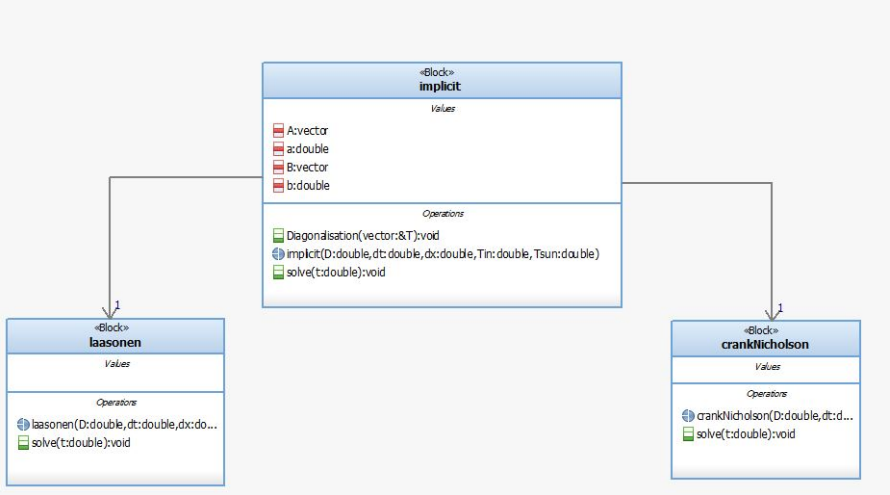


Figure 4: Implicit Architecture

The derived **Implicit** class is also a based class for two other classes: the **Laasonen** class and the **Cranck-Nicholson** class. Each of them solving the problem with the scheme of the same name.

The **Implicit** class just like the **Explicit** as a particular method to simplify the solving of the problem in the two derived classes as well as two new defined vector Tpast and Tnext. To understand what the **Diagonalization** method is used for we need to talk about the resolution of the problem for the two implicit schemes.

The two system of equation that we wrote in equation (7) and (8) can be written like this:

$$\begin{pmatrix} b & a & 0 & \cdots & 0 \\ a & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & a \\ 0 & \cdots & 0 & a & b \end{pmatrix} = \begin{pmatrix} T_0^{n+1} \\ \vdots \\ T_{imax}^{n+1} \end{pmatrix} R \tag{11}$$

Where a, b are doubles and R a vector, all of their values change depending on which scheme we choose. Since the problem is a tri-diagonal matrix and for each scheme, we have (in the special case of the data given by the assigment) $2\|a\| \leq \|b\|$ so we can use Thomas algorithm to solve those equation.

Thomas Algorithm is an efficient way of solving tridiagonal matrix systems. It is based on LU decompositionin which the matrix system $Mx = r$ is rewritten as $LUx = r$ where L is a lower triangular matrix and U is an upper triangular matrix. The system can be efficiently solved by setting $Ux = p$ and then solving first $Lp = r$ for p and then $Ux = p$ for x. The Thomas algorithm consist of two steps. In step one decomposing the matrix into $M = LU$ and solving $Lp = r$ are accomplished in a single downwards sweep, taking us straight from $Mx = r$ to $Ux = p$. In step two the equation $Ux = p$ is solved for x in an upward sweep.

In the **Diagonalization** we are taking care of the first step of the Thomas algorithm. We will have this kind of equation at the end of the method:

$$\begin{pmatrix} 1 & B_0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & B_{imax-1} \\ 0 & \cdots & 0 & 1 \end{pmatrix} = \begin{pmatrix} T_0^{n+1} \\ \vdots \\ T_{imax}^{n+1} \end{pmatrix} P \tag{12}$$

The values of $B_i$ and P will be stored respectively in the B vector and the A vector defined in the **Implicit** class.

The function **solve** in both **Cranck-Nicholson** and **Laasonen** is a loop that initialize the vector R (with the help of the equation (7) and (9), call the method **Diagonalization(R)** and then calculate the second step of the Thomas algorithm. It will print for each $10dt$hr the value of T until the time t in the argument of solve is reached.

To run this program we use a **main.cpp** file that will create an object in each file and depending the input of the user (note that the user can change

only the values of the measuring time t and the step size dt) it will call the function wanted.

We also add a function **accuracy** that will calculate the absolute and relative error of the scheme needed only at the measuring time needed.

We will now see the data collecting procedure.

## 2.2 Data Gathering

In this Assigment there is different type of Data that we are looking for. First of all we want to find all the values of the different scheme for each 0.1hr until 0.5hr. In order to find this data we use the **solve** method explained before, that will print the values of T every $10dt$ until the input time t is reach. So we use as step time size dt = 0.01hr and t = 0.5hr in each scheme to find the Data we need.

Furthermore we want to find the accuracy of each method. To collect this data we defined a function **accuracy**, it will use the value of T for each scheme, this value is the one calculate for the input t, so the T at the end of the each **solve** method. We based the calculation of this function on the absolute error and the relative error. Their equation are respectively:

$$AE = \|T_{scheme} - T_{ana}\|, RE = \frac{\|T_{scheme} - T_{ana}\|}{\|T_{scheme}\|} \tag{13}$$

Since we are in a one base problem we will use the l1-norm that as the following definition:

$$\|x\|_1 = \sum_i \|x_i\| \tag{14}$$

As for the required compution time, we try to implement a timer in the laasonen **solve** method but even in nanoseconds it keeped printing bolean values. So we haven't been able to collect proper data for this part.

# 3 Result

## 3.1 Comparasion with the Analytical result

Those graphics are the result given by the program with $dt = 0.01hr$ and $dx = 0.05ft$, we printed for each scheme the curve of t = 0.1ht to t = 0.5hr:

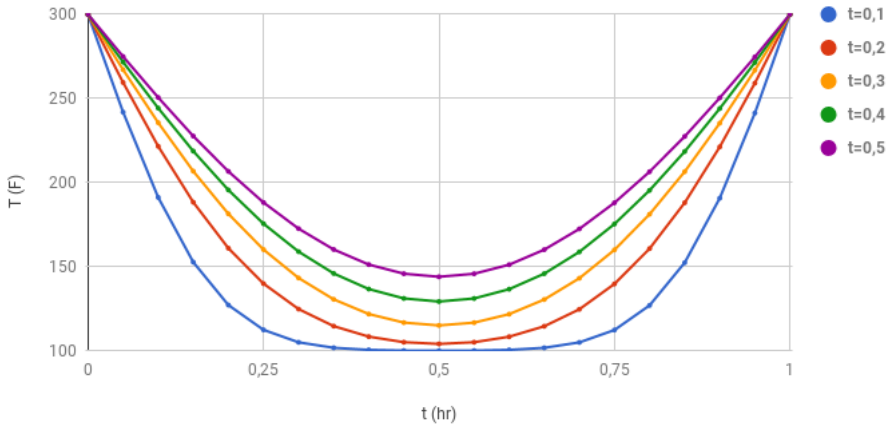


Figure 5: Analytical Result

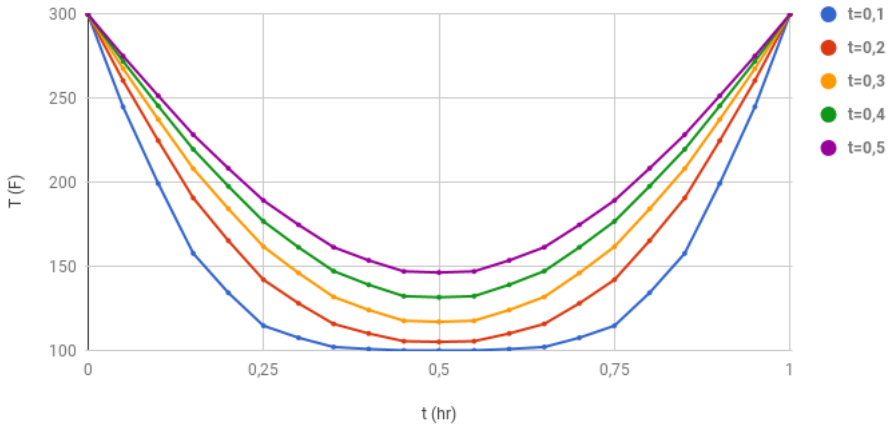


Figure 6: DuFort-Frankel Result

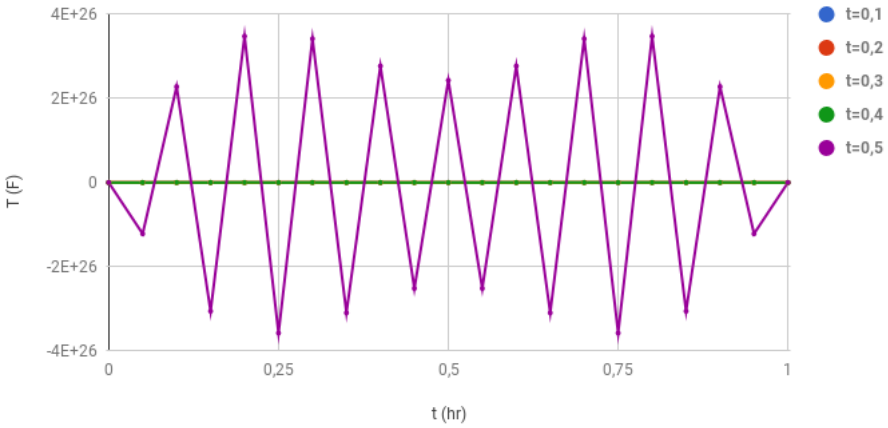


Figure 7: Richardson Result

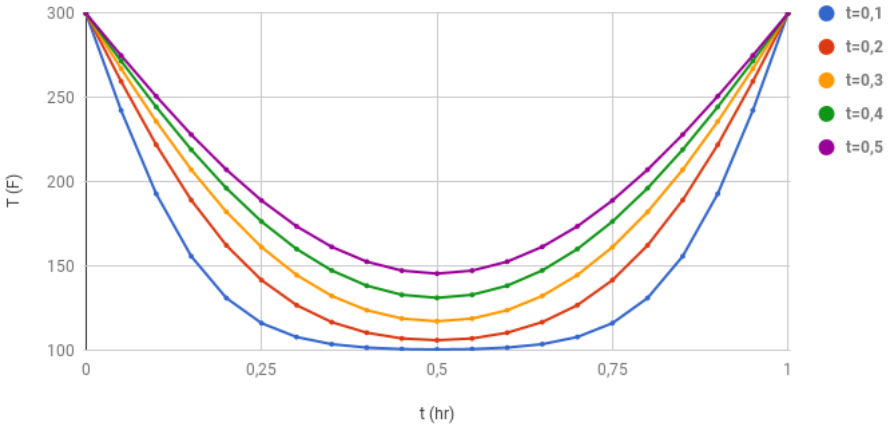


Figure 8: Laasonen Result

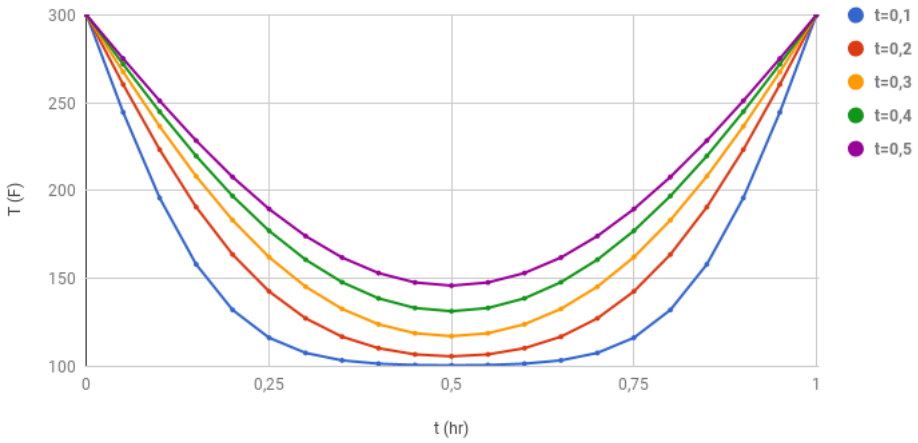


Figure 9: Cranck-Nicholson Result

We can conclude that all the scheme except **Richardson** are stable. It was predictable since Richardson was the only scheme that was unconditionnaly unstable. The other scheme are all unconditionnaly stable and very similar to the Analytical result, it is hard to see which one is the more accurate.

To find out we have calculated the absolute and relative Error (with a fixed $dt = 0.01hr$), we get those graphics:

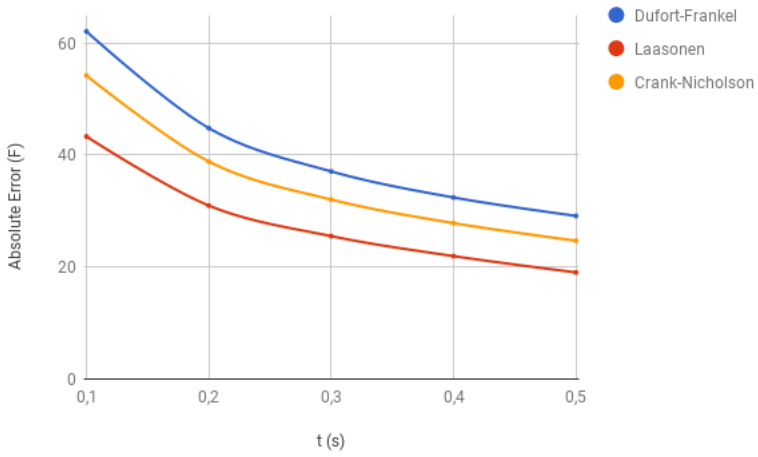


Figure 10: Absolute Error Calculation

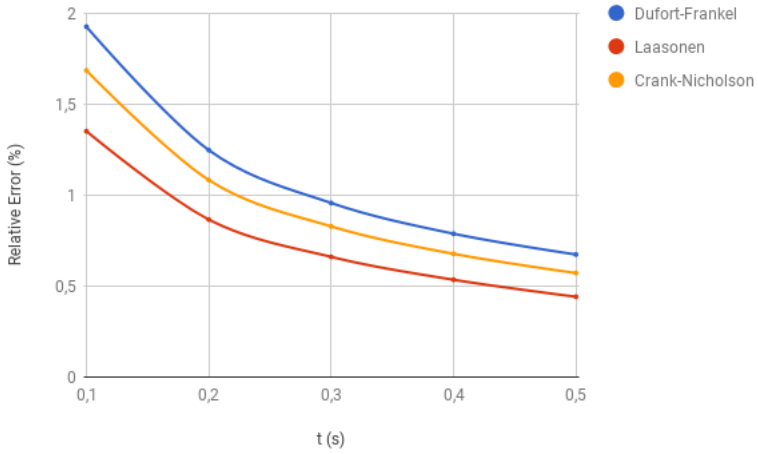


Figure 11: Relative Error Calculation

Those graphics shows that all three schemes get more accurate as the time grows higher. We may be able to explain that the first value are highly inacurate by the fact that the analytical solution is an approximation and even if this one is close to the actual answer it still includes additional error in the result we get with the shemes.

In all cases it is clear that the Laasonen scheme is more accurate than the other since all schemes have the same behaviour and that the initial value of the lasonnen error scheme is smaller. But we can also notice that all of the schemes are pretty accurate since at $t = 0.3hr$ all of them have an error below 1%. We can predict that the error will continue to get smaller until it reach 0. All schemes tend to reach 0, because they are exposed to the continuous surface temperature. To show this behaviour we try to find the error at t=10hrs, we get :

DuFort Frankel Absolute Error: 0.00695182, Relative Error: 1.103e-006
Laasonen Absolute Error: 0.00797042, Relative Error: 1.265e-006
Crank-Nicholson Absolute Error: 0.000457142, Relative Error: 7.256e-008

The error is now much more smaller and it is now Cranck-Nicholson that is the more accurate scheme. It doesn't mean that Crank-Nicholson is more suitable for calculation, because once t = 10 hr is reach the values are all around 0 and we can't get data out of it. So this seconf result is not very intersting, except that it proves that all scheme will reach 0 when t tend to infinity.

### 3.2 Laasonen study

We will now study the effect of the step size on the accuracy on the Laasonen scheme. Once we retrieve all the data we get for an input t = 0.5hr:

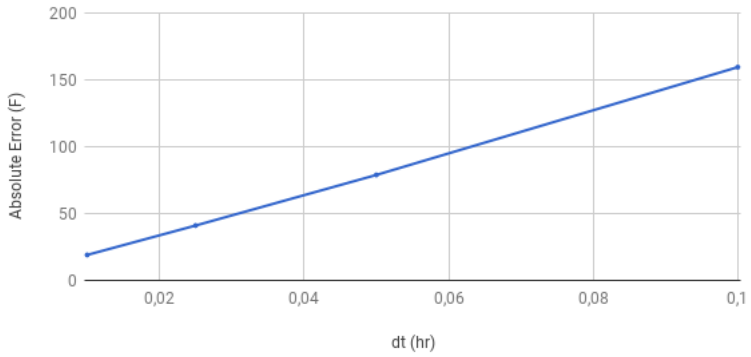


Figure 12: Absolute Error of the Laasonen scheme plotted against the step size

We can observe that the increasing of the step size has a direct effect on the error, the bigger the step size will be the less accurate the scheme will become. Furthermore it seems that there is a linear relationship between those two variables. We can calculate an approximation of this equation:

$f(x) = \frac{1100}{12}x + 18$.

As for the study of the compuuttional time, I wasn't able to implement a working timer so I can't show any result for this question. But since the number of loop to compute to a certain time depend on the value of the step size, we can guess that as the step size increased, the computation time decrease.

## 4 Conclusion

Based on the result we have we can say that the Laasonen scheme has the best accuracy and so that it is more suitable to do calculation. Depending on our need we will have to choose between a total accurate result and a faster program. But since the Laasonen scheme reach a an error $< 1\%$ in less than 0.2hr It is possible to have a good balance in both parameters.

## 5 References

1. http://www.industrial-maths.com
2. http://www.math.pku.edu.cn
3. Von Newmann Stability Analysis, Dr.Johnson 2008

# 6 Appendice

## 6.1 Richardson Analysis

We are gonna study the Richardson scheme accuracy and Stability. In this Assigment we used this expression to calculate Richardson:

$$\frac{T_i^{n+1} - T_i^{n-1}}{2\Delta t} - D\frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2} = 0 \tag{15}$$

To find the accuracy of this scheme we are going to rewrite some term with the Fourrier expansion:

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

$$T_i^{n-1} = T_i^n - (\frac{\partial T}{\partial t})_i^n \Delta t + (\frac{\partial^2 T}{\partial t^2})_i^n \frac{\Delta t^2}{2!} + O(\Delta t^3) \tag{17}$$

$$T_{i+1}^n = T_i^n + (\frac{\partial T}{\partial x})_i^n \Delta x + (\frac{\partial^2 T}{\partial x^2})_i^n \frac{\Delta x^2}{2!} + (\frac{\partial^3 T}{\partial x^3})_i^n \frac{\Delta x^3}{3!} + O(\Delta x^4) \tag{18}$$

$$T_{i-1}^n = T_i^n - (\frac{\partial T}{\partial x})_i^n \Delta x + (\frac{\partial^2 T}{\partial x^2})_i^n \frac{\Delta x^2}{2!} - (\frac{\partial^3 T}{\partial x^3})_i^n \frac{\Delta x^3}{3!} O(\Delta x^4) \tag{19}$$

We then implement those equations in the first Richardson equation, we get:

$$(\frac{\partial T}{\partial t})_i^n - D(\frac{\partial^2 T}{\partial x^2})_i^n + O(\Delta t^2, \Delta x^2) = 0 \tag{20}$$

We can see that we recovered the initial equation. So T is the solution of the continuous error, but also the solution of the discreted problem with some error which is included in $O(\Delta t^2, \Delta x^2)$. The accuracy of the Richardson scheme is $O(\Delta t^2, \Delta x^2)$.

We will now study the stability of this scheme, to do so we will do a Von Newmann Stability Analysis: A numerical scheme is said to be stable if deviation from exact solution decreases in time. So we need to de compose the residual error.

If we consider T the solution of the ortignal problem and $\tau$ the solution of the discreted probem we will have:

$T_i^n = \tau_i^n + r_i^n$

So we have:

$$r_i^{n+1} - r_i^{n-1} = 2\alpha(r_{i+1}^n - 2r_i^n + r_{i-1}^n), \alpha = \frac{D\Delta t}{\Delta x^2} \tag{21}$$

We decompose the error:

$$r_i^n = \sum_{-\infty}^{+\infty} g^n(k) e^{ikx_i} \tag{22}$$

and get the following equation after considering it for only one wave:

$g^{n+1}e^{ikx_i} - g^{n-1}e^{ikx_i} = 2\alpha(g^n e^{ikx_{i+1}} - 2g^n eikx_i + g^n e^{ikx_{i-1}})$
$g^{n-1}e^{ikx_i}(g^2 - 1) = 2\alpha g^n e^{ikx_i}(e^{ik\Delta x} - 2 + e^{-ik\Delta x})$
$g^2 - 1 = 2\alpha g(2cos(k\Delta x) - 2)$, because $2cos(\theta) = e^{i\theta} + e^{-i\theta}$
$g^2 - 1 = -6\alpha g sin^2(\frac{k\Delta x}{2})$, because $2sin^2(x) = 1 - cos(2x)$
So we finaly get: $g^2 + 6\alpha g sin^2(\frac{k\Delta x}{2}) - 1 = 0$

This quadratic equation has two roots, the sum and the product of the two roots are given by:

$g_1 + g_2 = -6\alpha sin^2(\frac{k\Delta x}{2})$, and $g_1 g_2 = -1$

For the scheme to be stable we need $\|g_1\| \leq 1$ and $\|g_2\| \leq 1$. The product of the roots show that if $\|g_1\| < 1$ then $\|g_2\| > 1$ and vice-versa. Also if $g_1 = 1$ and $g_2 = -1$ then we must have $\alpha = 0$.

We can conclude that the Richardson scheme is unconditionnaly unstable.

## 6.2 Code

You will find in this part all my code:

This is my `main.cpp`:

```
#include <iostream>  //generic IO
#include <fstream>   //file IO
#include <stdexcept> //provides exception
#include "Class/Solver/Explicit/explicit.h"
#include "Class/solver/Explicit/DuFort-Frankel/duFortFrankel.h"
#include "Class/solver/Explicit/Richardson/richardson.h"
#include "Class/Solver/Implicit/implicit.h"
#include "Class/Solver/Implicit/Crank-Nicholson/crankNicholson.h"
#include "Class/Solver/Implicit/Laasonen/laasonen.h"
#include "Class/Solver/Analytical/analytical.h"

int scheme;
int error;
double dt;
double t;
void accuracy(DuFortFrankel Df, Richardson Ri, Laasonen La, CrankNicholson
{
    std::cout << "\nError Values for t = " << t << "\n";
    Vector DfError = Vector(Ana.n);
    Vector RiError = Vector(Ana.n);
    Vector LaError = Vector(Ana.n);
    Vector CrError = Vector(Ana.n);
    for (int i = 0; i < Ana.n; i++) DfError[i] = Df.T[i] - Ana.T[i];
    for (int i = 0; i < Ana.n; i++) RiError[i] = Ri.T[i] - Ana.T[i];
    for (int i = 0; i < Ana.n; i++) LaError[i] = La.Tnext[i] - Ana.T[i];
    for (int i = 0; i < Ana.n; i++) CrError[i] = Cr.Tnext[i] - Ana.T[i];
    if(scheme == 1){
        std::cout << "DuFort Frankel Absolute Error: " << DfError.one_norm(
        std::cout << " Relative Error: " << DfError.one_norm() / Df.T.one_n
    }
    if(scheme == 2){
        std::cout << "Richardson Absolute Error: " << RiError.one_norm() <<
```

```
            std::cout << "Relative Error: " << RiError.one_norm() / Ri.T.one_no
        }
        if(scheme == 3){
            std::cout << "Laasonen Absolute Error: " << LaError.one_norm() << "
            std::cout << "Relative Error: " << LaError.one_norm() / La.Tnext.on
        }
        if(scheme == 4){
            std::cout << "Crank-Nicholson Absolute Error: " << CrError.one_norm
            std::cout << "Relative Error: " << CrError.one_norm() / Cr.Tnext.on
        }
        if(scheme == 6){
            std::cout << "DuFort Frankel Absolute Error: " << DfError.one_norm(
            std::cout << " Relative Error: " << DfError.one_norm() / Df.T.one_n
            std::cout << "Richardson Absolute Error: " << RiError.one_norm() <<
            std::cout << "Relative Error: " << RiError.one_norm() / Ri.T.one_no
            std::cout << "Laasonen Absolute Error: " << LaError.one_norm() << "
            std::cout << "Relative Error: " << LaError.one_norm() / La.Tnext.on
            std::cout << "Crank-Nicholson Absolute Error: " << CrError.one_norm
            std::cout << "Relative Error: " << CrError.one_norm() / Cr.Tnext.on
        }
    }

    int main(){
        std::cout << " Which scheme do you want t see the result for?" << "\n"
        std::cout << "1: DuFort Frankel ,\n2: Richardson ,\n3: Laasonen ,\n4: Cran
        std::cin >> scheme;
        std::cout << "Do you want to see the Absolute and Relative error ? (0:
        std::cin >> error;
        std::cout << "This program will print the result considering Diffusity
        std::cout << "Choose now which value of dt you want to choose and then
        std::cin >> dt;
        std::cout << "\n You want to solve until t=";
        std::cin >> t;
        //Initialisation
        Analytical Anaprob = Analytical(0.1, 100, 300, dt, 0.05);
        if(scheme == 5) {
            Anaprob.solve(t);
            std::cout << "\n";
        }
        DuFortFrankel Dufortprob = DuFortFrankel(0.1, 100, 300, dt, 0.05);
        if(scheme == 1){
            Anaprob.solve(t);
            Dufortprob.solve(t);
            std::cout << "\n";
        }
        Richardson Richprob = Richardson(0.1, 100, 300, dt, 0.05);
        if(scheme == 2) {
            Anaprob.solve(t);
            Richprob.solve(t);
            std::cout << "\n";
```

```
    }
    Laasonen Laprob = Laasonen(0.1, 100, 300, dt, 0.05);
    if(scheme == 3) {
        Anaprob.solve(t);
        Laprob.solve(t);
        std::cout << "\n";
    }
    CrankNicholson Crprob = CrankNicholson(0.1, 100, 300, dt, 0.05);
    if(scheme == 4) {
        Anaprob.solve(t);
        Crprob.solve(t);
        std::cout << "\n";
    }
    if(scheme == 6){
        Dufortprob.solve(t);
        std::cout << "\n";
        Richprob.solve(t);
        std::cout << "\n";
        Laprob.solve(t);
        std::cout << "\n";
        Crprob.solve(t);
        std::cout << "\n";
        Anaprob.solve(t);
        std::cout << "\n";
    }
    //Error calculation
    if(error == 0) accuracy(Dufortprob, Richprob, Laprob, Crprob, Anaprob);
    return 0;
}
```

This is my `solve.cpp`:

```
#include "solve.h"

//CONSTRUCTOR
/**
 * Default constructor - Set all the value to 1
 */
Solve::Solve()
{
    this->D = 1;
    this->Tin = 1;
    this->Tsun = 1;
    this->dt = 1;
    this->dx = 1;
    this->r = 1;
    L = 1; //the wall is 1 ft thick
    n = 1; //number of positon in between the two parts of the wall
}
/**
 * Default constructor -
```

```
 * \nSet the value of D, Tin, Tsun, dt, dx and L
and create r and n which are to simplified calculation in the derived class
 */
Solve::Solve(double D, double Tin, double Tsun, double dt, double dx)
{
    //set the data
    this->D = D;
    this->Tin = Tin;
    this->Tsun = Tsun;
    this->dt = dt;
    this->dx = dx;
    this->r = D * dt / (dx * dx);
    L = 1.0; //the wall is 1 ft thick
    n = std::ceil(L / dx) + 1; //number of positon in between the two parts
}

//SOLVING METHODS
/**
 * Virtual method - Not defined here
 */
void Solve::solve(double t)
{
}


//ACCESOR METHODS
/*
* accessor method - get D
*/
double Solve::getD()
{
    return this->D;
}

/*
* accessor method - set D
*/
void Solve::setD(double D)
{
    this->D = D;
}

/*
* accessor method - get Tin
*/
double Solve::getTin()
{
    return this->Tin;
}
```

```
/*
* accessor method - set Tin
*/
void Solve::setTin(double Tin)
{
    this->Tin = Tin;
}

/*
* accessor method - get Tsun
*/
double Solve::getTsun()
{
    return this->Tsun;
}

/*
* accessor method - set Tsun
*/
void Solve::setTsun(double Tsun)
{
    this->Tsun = Tsun;
}

/*
* accessor method - get dt
*/
double Solve::getdt()
{
    return this->dt;
}

/*
* accessor method - set dt
*/
void Solve::setdt(double dt)
{
    this->dt = dt;
}

/*
* accessor method - get dx
*/
double Solve::getdx()
{
    return this->dx;
}

/*
* accessor method - set dx
```

```
*/
void Solve::setdx(double dx)
{
    this->dx = dx;
}
```

This is my vector.cpp:

```
#include "vector.h"

#include <iostream> //Generic IO operations
#include <fstream>  //File IO operations
#include <stdexcept> //provides exceptions
#include <vector>  // std vector upon which our Vector is based
using namespace std;


// CONSTRUCTORS
/*=
* Default constructor (empty vector)
*/
Vector::Vector() : std::vector<double>() {}


/*
* Alternate constructor - creates a vector of a given size
*/
Vector::Vector(int Num) : std::vector<double>()
{
        // set the size
        (*this).resize(Num);

        // initialise with zero
        std::size_t i;
        for (i = 0; i < size(); i++) (*this)[i] = 0.0;
}

/*
* Copy constructor
*/
Vector::Vector(const Vector& copy) : std::vector<double>()
{
        (*this).resize(copy.size());
    // copy the data members (if vector is empty then num==0)
        std::size_t i;
    for (i=0; i<copy.size(); i++) (*this)[i]=copy[i];
}

/*
* accessor method - get the size
*/
```

```
int Vector::getSize() const
{
        return size();
}

// OVERLOADED OPERATORS
/*
* Operator= - assignment
*/
Vector& Vector::operator=(const Vector& copy)
{
        (*this).resize(copy.size());
        std::size_t i;
    for (i=0; i<copy.size(); i++) (*this)[i] = copy[i];
    return *this;
}


// COMPARISON
/*
* Operator== - comparison
*/
bool Vector::operator==(const Vector& v) const
{
        if (size() != v.size()) throw std::invalid_argument("incompatible␣v
        std::size_t i;
        for (i = 0; i<size(); i++)      if (fabs((*this)[i] - v[i]) > 1.e-0
        return true;
}

// NORMS
/*
* 1 norm
*/
double Vector::one_norm() const
{
        int n = size();

        double sum = 0.0;

        for (int i = 0; i < n; i++)
                sum = sum + fabs((*this)[i]);

        return sum;
}

/*
* 2 norm
*/
double Vector::two_norm() const
```

```
{
        int n = size();

        double sum = 0.0;

        for (int i = 0; i < n; i++)
                sum = sum + (*this)[i] * (*this)[i];

        return sqrt(sum);
}

/*
* uniform (infinity) norm
*/
double Vector::uniform_norm() const
{
        int n = size();

        double max = fabs((*this)[0]);

        for (int i = 1; i < n; i++)
                if (fabs((*this)[i]) > max) max = fabs((*this)[i]);

        return max;
}
```

This is my analytical.cpp:

```
#include "analytical.h"
#define PI 3.14

//CONSTRUCTOR
/**
 * Default constructor -
 * \nIs based on the Solve class constructor
 * \nSet a value of m high enough for the approximation to be relevant (her
 * \nSet the solution vector T as null expect for the two bundary condition
 */
Analytical::Analytical(double D, double Tin, double Tsun, double dt, double
{
    T = Vector(n);
    T[0] = T[n - 1] = Tsun;
    m = 10000;
}

//SOLVING METHODS
/**
 * Virtual method -
 * We use two loop in order to calculate the value of T at each time
and each position until the input t is reached.
 * We also print the value of T for every 0.1hrs until t is reach
```

```
 */
void Analytical::solve(double t)
{
    //INITIALISATION OF VARIABLES
    double tmax = t / this->dt;//The amount of time until when the user wan
    double ts = 0;//Counter to keep track of the value of time
    std::cout << "Analytical Result"<<"\n";

    //CALCULATION
    for (int j = 1; j < tmax + 1; j++)
        {
        double xs = 0; //Counter to keep track of the position
        for (int i = 1; i < n - 1; i++)
        {
            double S = 0;//Initialisation of the sum
            xs = xs + dx;
            //CALCULATION OF THE SUM
            for (int k = 1; k < m + 1; k++)
                S = S + exp(-D * std::pow(k * PI / L, 2) * ts) * (1 - std::
            //CALCULATION OF T
            T[i] = Tsun + 2 * (Tin - Tsun) * S;
        }
        ts = ts + dt;

        //Print the result for every 10*dt hr until t is reach
        if (j % 10 == 0)
        {
            std::cout << "*****************************"<< "\n";
            std::cout << "for t = " << j * dt << "\n";
            for (int i = 0; i < n; i++)
            {
                std::cout << T[i] << " ";
            }
            std::cout << "\n";
        }
    }
}
```

This is my `explicit.cpp`:

```
#include "explicit.h"

//CONSTRUCTOR
/**
 * Default constructor - Same as the Solver constructor
 * set all the ele;ents to one
 */
Explicit::Explicit():Solve()
{
}
/**
```

```
 * Same as the Solver constructor
 */
Explicit::Explicit(double D, double Tin, double Tsun, double dt, double dx)
{
    Tpast = Vector(n);
    T = Vector(n);
    Tnext = Vector(n);
    for (int i = 0; i < n; i++) Tpast[i] = Tin;
    Tpast[0] = Tpast[n-1] = Tnext[0] = Tnext[n-1] = Tsun;
}

//SOLVER METHOD
/**
 * Virtual method - Not defined here
 */
void Explicit::solve(double t) {
}

//TOOL METHOD
/**
 * Void methods that initialize T at t=0 and calculate T at t = dt
 */
void Explicit::OrderOne(Vector &T)
{
    //Initialisation
    Vector temp(n);
    for (int i=1; i<n-1; i++) T[i] = 100;
    T[0] = T[n-1] = 300;
        T[n-1] = 300;
    //Calculation
    for (int i = 1; i < n - 1; i++)
    {
        temp[i] = this->r * (T[i + 1] - 2 * T[i] + T[i - 1]) + T[i];
    }
    for (int i = 1; i < n-1; i++)
    {
            T[i] = temp[i];
        }
}
```

This is my `implicit.cpp`:

```
#include "implicit.h"

// CONSTRUCTOR
/**
 * Default constructor - Same as the Solver constructor
 * set all the value to one
 */
Implicit::Implicit() : Solve()
{
```

```
}

/**
 * Constructor - Same as the Soler constructor, we initialise the two vecto
 */
Implicit::Implicit(double D, double Tin, double Tsun, double dt, double dx)
{
    A = Vector(n);
    B = Vector(n);
    Tnext = Vector(n);
    Tpast = Vector(n);
    for (int i = 1; i < n-1; i++) Tpast[i] = Tin;
    Tpast[0] = Tpast[n-1] = Tnext[0] = Tnext[n-1] = Tsun;
}

//SOLVER METHOD
/**
 * Virtual method - Not defined here
 */
void Implicit::solve(double t){}

//TOOL METHOD
/**
 * Void methods that implement the first part of the thomas algorithm
 */
void Implicit::Diagonalization(Vector &T)
{
    this->A[1] = T[1] / b;
    this->B[1] = a / b;
    for (int i = 2; i < n-1; i++)
    {
        this->A[i] = (T[i] - a * A[i-1]) / (b - a * this->B[i-1]);
        this->B[i] = a / (b - a * this->B[i-1]);
    }
}
```

This is my `duFortFrankel.cpp`:

```
#include "duFortFrankel.h"

//CONSTRUCTOR
/**
 * Default constructor - Based on the Solver constructor
 */
DuFortFrankel::DuFortFrankel(double D, double Tin, double Tsun, double dt,
{
}

//SOLVING METHODS
/**
 * Virtual method -
```

```
 * Solve the problem with the DuFort-Frankel scheme
 */
void DuFortFrankel::solve(double t)
{
    //INITIALISATION
    double tmax = t / this->dt;
    std::cout << "DuFort-Frankel Result"<<"\n";
    OrderOne(T);
    //CALCULATION OF T AT N+1
    for (int j = 2; j < tmax+1; j++)
    {
        for (int i = 1; i < n-1; i++) Tnext[i] = (2 * this->r * T[i + 1] +
        for (int i = 0; i < n; i++)
        {
            Tpast[i] = T[i];
            T[i] = Tnext[i];
            }
        //PRINTING THE RESULT FOR EVERY 10*dt hrs
        if (j % 10 == 0)
        {
            std::cout << "*****************************"<< "\n";
            std::cout << "for t = " << j * dt << "\n";
            for (int i = 0; i < n; i++)
            {
                std::cout << T[i] << " ";
            }
            std::cout << "\n";
        }
    }
}
```

This is my `richardson.cpp`:

```
#include "richardson.h"

//CONSTRUCTOR
/**
 * Default constructor - Based on the Solver constructor
 */
Richardson::Richardson(double D, double Tin, double Tsun, double dt, double
{
}

//SOLVING METHODS
/**
 * Virtual method -
 * Solve the problem with the Richardson scheme
 */
void Richardson::solve(double t)
{
    //INITIALISATION
```

```
    double tmax = t / this->dt;
    std::cout << "Richardson Result"<<"\n";
    OrderOne(T);
    //CALCULATION OF T AT N+1
    for (int j = 2; j < tmax+1; j++)
    {
        for (int i = 1; i < n-1; i++) Tnext[i] = Tpast[i] + 2 * this->r * (
        for (int i = 0; i < n; i++)
        {
            Tpast[i] = T[i];
            T[i] = Tnext[i];
            }
        //PRINTING THE RESULT FOR EVERY 10*dt hrs
        if (j % 10 == 0)
        {
            std::cout << "*****************************"<< "\n";
            std::cout << "for t = " << j * dt << "\n";
            for (int i = 0; i < n; i++)
            {
                std::cout << T[i] << " ";
            }
            std::cout << "\n";
        }
    }
}
```

This is my `laasonen.cpp`:

```
#include <chrono>
#include "laasonen.h"

//CONSTRUCTOR
/**
 * Default constructor - Based on the Solver and Implicit constructor
 * We implement the value of a and b use in the Implicit class
 * @see Diagonalization(Vector &T)
 */
Laasonen::Laasonen(double D, double Tin, double Tsun, double dt, double dx)
{
    this->b = 1 + 2 * r;
    this->a = -r;
};

//SOLVING METHODS
/**
 * Virtual method -
 * Solve the problem with the Laasonen scheme
 */
void Laasonen::solve(double t)
{
    //TIMER
```

```
    auto begin = std::chrono::high_resolution_clock::now();
    //INITIALISATION
    double tmax = t / this->dt;
    std::cout << "Laasonen Result"<<"\n";
    //CALCULATION
    for (int j = 1; j < tmax+1; j++)
    {
        //Initialisation of the first and last term of b in Ax = b
        Tpast[1] = Tpast[1] + r * Tsun;
        Tpast[n - 2] = Tpast[n - 2] + r * Tsun;
        //Use Thomas' method
        Diagonalization(Tpast);
        //Resolution of Ux = y
        Tnext[n-2] = A[n-2];
        for (int i = n-3; i > 0; i--)
        {
            Tnext[i] = A[i] - B[i] * Tnext[i+1];
        }
        for (int i = 0; i < n; i++)
        {
            Tpast[i] = Tnext[i];
            }
        // PRINTING THE RESULT FOR EVERY 10*dt hrs
        if (j % 10 == 0)
        {
            std::cout << "*****************************"<< "\n";
            std::cout << "for t = " << j * dt << "\n";
            for (int i = 0; i < n; i++)
            {
                std::cout << Tnext[i] << " ";
            }
            std::cout << "\n";
        }
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::cout << "timer =" << std::chrono::duration_cast<std::chrono::nanos
};
```

This is my `crankNicholson.cpp`:

```
#include "crankNicholson.h"

//CONSTRUCTOR
/**
 * Default constructor - Based on the Solver and Implicit constructor
 * We implement the value of a and b use in the Implicit class
 * @see Diagonalization(Vector &T)
 */
CrankNicholson::CrankNicholson(double D, double Tin, double Tsun, double dt
{
    this->b =1 + r;
```

```
    this->a = -r / 2;
};

//SOLVING METHODS
/**
 * Virtual method -
 * Solve the problem with the Cranck-Nicholson scheme
 */
void CrankNicholson::solve(double t)
{
    //INITIALISATION
    Vector R(n);
    double tmax = t / this->dt;
    std::cout << "Crank-Nicholson Result"<<"\n";
    //CALCULATION
    for (int j = 1; j < tmax+1; j++)
    {
        //Initialisation of b in Ax = b
        R[1] = r / 2 * Tpast[2] + (1 - r) * Tpast[1] + r / 2 * Tpast[0] + r
        R[n-2] = r / 2 * Tpast[n-1] + (1 - r) * Tpast[n-2] + r / 2 * Tpast[
        for (int i = 2; i < n - 2; i++) R[i] =  r / 2 * Tpast[i+1] + (1 - r
        //Use Thomas' method
        Diagonalization(R);
        //Resolution of Ux = y
        Tnext[n-2] = A[n-2];
        for (int i = n-3; i > 0; i--)
        {
            Tnext[i] = A[i] - B[i] * Tnext[i+1];
        }
        for (int i = 0; i < n; i++)
        {
            Tpast[i] = Tnext[i];
            }
        //PRINTING THE RESULT FOR EVERY 10*dt hrs
        if (j % 10 == 0)
        {
            std::cout << "*****************************"<< "\n";
            std::cout << "for t = " << j * dt << "\n";
            for (int i = 0; i < n; i++)
            {
                std::cout << Tnext[i] << " ";
            }
            std::cout << "\n";
        }
    }
};
```